Literature Review of Leading Research in Compiler Optimizations

Timothy Van Slyke

March 23, 2018

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

TODO: Many sections are currently just lorem ipsum'd and have yet to be written. All other sections mainly consist of talking points.

"Software is getting slower more rapidly than hardware becomes faster."

– Niklaus Wirth

Introduction

The modern software ecosystem is rich with a variety of platforms with distinct needs. With the advent of mobile computing[1], traditional software development practices have seen major disruptions over the past decade. While the x86 architecture continues to dominate the desktop market, mobile platforms have largely adopted RISC[2] architectures; particularly, ARM-based CPUs. Additionally, the widespread and international usage of social media platforms, web search engines, and other large-scale web services has necessitated an expanding usage of centralized data centers, or "server farms". Both of these platforms, mobile devices and servers, despite having distinctly disparate use cases, make nearly identical demands from the software that they use.

To quote Chandler Carruth, technical lead for Google's C++ and LLVM teams, "The only thing [a data center does] is take electricity

[1] Here we use the term "mobile computing/platforms/devices" to primarily refer to so-called "smart devices" such as smart phones and tablets. While traditional mobile computing devices (i.e. laptops) face many of the same issues as smart devices, they share a fair number of similarities with desktop computers and may not appropriately be described by some of the generalizations discussed here.

[2] Reduced instruction set computer.

and turn it into heat. That is its job." [3]. Maintenance of large-scale data centers present some of the largest expenses for firms such as Google, Amazon, Facebook, and other tech giants. Improvements to the power usage of the software that runs on these servers present a major opportunity for engineers to reduce expenditures. Similarly, mobile devices are constrained by the fact that they must employ a depletable power source. Devices which are battery-operated also incentivize low power consumption, by necessity.

To date, the most effective strategy for reducing power consumption by computation has been to increase software speed [3]. CPUs have developed an impressive capacity for reducing energy usage via hibernation capabilities. Modern advice in the software development community for increasing so called "compute per Watt" is to "finish quickly so the device can go to sleep".

Motivations for producing faster software are plentiful even when ignoring the energy consumption argument. It would be hard to conceive of a scenario in which making a computer program slower is desired, and certainly such a task could not be difficult to achieve[3]. In light of this universal need for faster software we seek to answer the following quesion:

[3] It might be argued by some that the task of producing slower programs is a solved problem in modern software development.

> What advancements in the state-of-the-art for compiler optimizations are being pursued in contemprory research?

Background

Modern optimizing compilers have experienced nearly have a maturation. At a bare minimum, any serious optimizing compiler can be expected to be capable of at least the following:

- Callsight-sensitive subroutine inlining.
- Devirtualization.
- Copy elision.
- Loop unrolling.
- Target-specific code generation.
- Common subexpression elimination.
- Constant folding.
- Primitive alias analysis.
- Dead code elimination.
- Register allocation.

Popular open-source compilers are subject to a high degree of scrutiny due to their widespread adoption by both industry professionals and hobbyists alike. The GNU Compiler Collection (GCC - originally the GNU C Compiler) and Clang are two primary targets of innovations in the C and C++ programming communities. The LLVM compiler framework (Clang's parent project) is particularly popular as a platform for experimentation; many of the publications we discuss in this document choose to implement their experiments using LLVM's facilities.

The literature we discuss primarilly references compilation strategies for C, C++, and Java. It is important to note that the compiler ecosystems for C and C++ are largely the same; nearly every modern compiler for either language supports the other[4]. The Java compiler ecosystem is largely a different beast however, due to the fact that Java compilers do no compile to native machine code but instead to virtual machine (VM) bytecode. Additionally, Java and similar managed languages like c#also typically run on a virtual machine with a built-in Just-In-Time (JIT) compiler system.

[4] A minor exception to this rule is MSVC, which no longer actively supports C (but can still compile most standards-compliant C99 code).

It is also worth noting that these compilers must optimize for entirely different object models. The C and C++ ISO standards specify an object model that necessitates data being stored densely and in-place. Java (and c#), on the otherhand, requires exactly the opposite: non-primitive objects are always stored indirectly through an implicit pointer to heap storage. This has major ramifications for performance; dense data storage promotes cache-locality and thus better memory latency while also making alias analysis more managable.

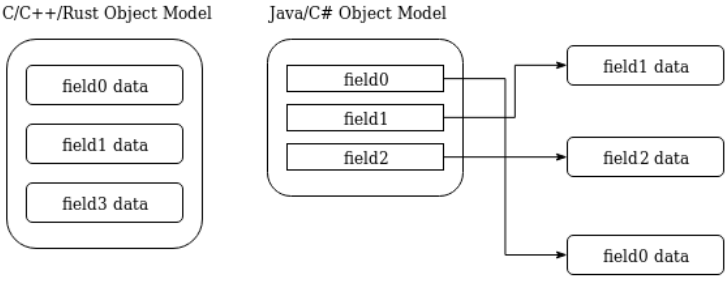

Figure 1: Side-by-side visualization of the C/C++/Rust object model versus the Java/C# object model. The C object model promotes densely-stored data and requires aliasing to be expressed through the type system explicitly (via pointers). This makes compile-time alias analysis tractable and promotes cache-friendly organizaton of data.

The Java object model requires at least one indirection (implicit pointer dereference) to access non-primitive data at runtime and promotes liberal object aliasing through simple pointer copies (reference semantics). This hinders a compiler's ability to perform alias analysis and hurts cache efficiency by storing objects sparsely.

Our discussion will include innovations within the context of both ecosystems. In particular, we explore the following major trends in modern compiler optimization research, as they relate to both ecosystems:

- Non-trivial code generation aimed at adaptive runtime perfor-

mance improvements.

- State-of-the-art in "traditional" optimizations (local code improvements, instruction reordering, static analysis).
- Automatic parallelization.
- Improvements to internal modelling methods and algorithms.

Methods

Optimizing compilers have been the subject of rigorous scrutiny for decades. Many prominent organizations, particularly firms whose income is reliant on providing software services, have a vested interest in producing software that is reliably fast and efficient modern computer hardware. As a consequence, a very rich set of compiler optimizations have largely been "done to death", so to speak. To obtain information on the state-of-the-art in this field, we explore publications from a small number of specialized conferences:

- Symposium on Principles of Programming Languages
- Programming Language Design and Implementation (previously Symposium on Compiler Optimization)
- International Symposium on Code Generation and Optimization

With few exceptions, the publications we analyze provide concrete implementations of their respective proposals; we have have chosen to exclude *most* pure compiler theory research from our review. We analyze optimization intended for traditional compilers, while also exploring specialized techniques like polyhedral compiler systems, the most prominent example of which is PLuTo [2].

Modern Compiler Optimizations

Many of the optimizations made by modern comilers for static languages like C, C++, and Java are speculative and rely on making conservative and provable deductions.

Dynamic Runtime Solutions

A common roadblock to many compiler optimizations is a lack of information at compile-time. Certain information can only be known at runtime, for example:

- The likelihood that a given conditional branch path will be taken.[5]
- Type information contained in dynamic libraries.

[5] Likely branches are typically called "hot pathes".

- Buffer and dynamic array sizes.

This lack of information prevents compilers from making informed decisions about how local optimizations should be structured. Managed languages like Java and c#have a built-in opportunity to leverage this information in the form of state-of-the-art JIT compiler techniques. Languages which require a virtual machine bytecode interpreter typically provide JITs to improve performance by providing facilities to transform VM bytecode to machine code. Compilers for specific VM runtime environments can implement efficient runtime facilities for dynamic code transformations since they "own" both the compiler and the VM. This enables a larger class of optimizations for compilers because they can operate under the assumption that the unknown information can be collected at runtime.

Languages which are typically compiled directly to machine code (prominent examples being C, C++, and Rust) cannot make the same assumptions as managed languages. Implementing such facilities would require inserting code for data collection and the correpsonding conditional code transformations without support from the runtime environment. Additionally, and perhaps more importantly, such aggressive transformations are atypical and violate programmers' expectations about code generation.

Nonetheless, research is being done in this area for both managed and unmanaged programming languages. This research can largely be categorized into one of the following two categories:

- Significant code generation to collect and utilize drastically transform the behavior of the input source code (WHOMST?).
- Extending existing run-time JIT compilation techniques with new methods. (WHOMST'D?).

Innovations in JIT Compilers

Dissenga et al., from Microsoft Research, propose a framework for abstract interpretation in a tracing JIT compiler. Their contribution is largely theoretical, but provides a formal means for developing a JIT which can make more liberal runtime modifications to code without introducing ill-formed ..... . Their framework is the first to successfully describe a JIT that can perform abstract interpretation at runtime.

Rompf et al. contribute a framework and corresponding implementation of an API which exposes control of JIT behavior to user code [9]. Historically, JIT activity has been regarded as an implementation detail of the VM runtime environments. By allowing user code to steer JIT behavior through the use of a macro system, Rompf et al.

were able to observe up to 52× speed improvements over the default. This development shows significant potential for JIT macro systems.

Aggressive Code Generation in Unmanaged Languages

Lifflander and Krishnamoorthy show promising results when applying aggressive transformations to recursive programs in a fairly nonstandard fashion. Proposed is a method for reorgranizing recursive[6] subroutine calls into context switches between a series of lightweight user-level threads; showing comparable speeds with the PLuTo and Pochoir parallelizing compilers. Doesn't scale too well though...

[6] This method allows for indirect recursion of non-trivial depth.

Memory and Cache-Oriented Optimizations

Often the most important factor that determines the speed of software on modern computer systems is the extent to which the software is able to exploit the benefits CPU cache.[7] Much work is being done to speculatively optimize code for optimal memory access patterns and to reduce CPU stall time by rerdering and eliminating memory-bound operations. Software prefetching, explicit instructions to hoist

[7] This factor is relevant on both desktop and mobile platforms. Exceptions to this rule are microcontrollers and other very small systems.

Software prefetching has shown promising results in benchmarks of otherwise cache-unfriendly code. Experiments have shown anywhere from 2.1x to 2.7x performance improvements from compiler-generated prefetch instructions; competing well with "hand rolled" prefetch placement[1]. These promising results complement similar findings by Tran et al. who propose similar generative and transformative methods of cache locality improvements. Tran et al. offer an instruction reordering framework, *Clairvoyance*, which aims to improve access patterns with somewhat more conservative generation of software prefetch instructions[11]. Their methods more heavily rely upon dependency analysis to ensure correctness, but offer opportunities aggressive code transformations.

As noted by the authors, the Clairevoyance framework potentially produces a pessimization through the introduction of excessive register pressure. This is because their framework operates by hoisting memory load instructions to early *access phases* which precede the actual usage of the loaded data in corresponding *execute phases*. This aggregate loading combined with delayed usage results in high register occupation in longer subroutines. However, this shortcoming may combine well with an optimization provided by Stock et al. for reducing register pressure in stencil operation code.

Stencil operations are of particular importance in numerical computation packages. Stencils are characterized by nested read-copy-

update loops where the "update" step typically involves a barrage of multiplications and additions accross multiple dimensions of input arrays. Their contribution consists of a systematic approach to provide a compact set of registers to dedicate to a given stencil. This reduced register usage relieves pressure on the cache by allowing more data to reside in uninhabitted registers. Their experiments shjow up to 3.74× speedup from traditional, unspecialized loop optimizations.

Additional contributions aimed at improving cache performance come from the previously mentioned work by Lifflander and Krishnamoorthy [6]. Their dynamic splicing approach revolves around the assumption that lightweight thread context switches can offer better I-cache[8] retention than traditional function calls. There method reduces call stack usage and redundant function parameter copying by transforming recursive function calls into a coroutine-like usage pattern. For each recursive call site, a user-level thread is spawned in place of pushing a new stack frame. Subsequent invocations at that call site are then implemented as context switches to the already-existing thread. Their method is limited in applicability by potentially intractable read-write patterns to shared data across spawned threads and introduces some overhead in the form of runtime *effect annotations* and context switches.

[8] Modern CPU caches typically segregate data and instructions in separate dedicated caches, respectively called the D-cache and I-cache. Without qualification, the unqualified term "cache" typically refers to just the D-cache.

This method proposed by Lifflander and Krishnamoorthy takes an evidently different approach to those of Tran et al. and Ainsworth et al., most notably in the fact that the former result in significant runtime analysis and code generation while the latter are purely compile-time constructs. Such research into dynamic methods have become more common in recent years and will likely continue to see more attention as memory latency effects continue to hinder software speed.

Automatic transformations of code to produce efficient access patterns and data compaction will likely continue to be a hot topic in compiler research due to its lopsided impact on program performance on modern computer systems.

Model Improvements

Some work has striven to improve compiler performance by considering novel internal representations of programs. Mehta and Yew propose an polyhedral[9] control-flow dependency model and present promising benchmark results in their experimentation. Their contribution can be expressed as modelling data dependencies at block-level, rather than at the granularity of single statements. This results in a model that produces a smaller dependency graph for larger-scale programs without sacrificing optimization oportunities. They imple-

[9] Polyhedral (or polytope) compilation is a name given to a method of modeling nested loops to produce more optimization opportunities than traditional loop optimization techniques offer.

ment their model as an extension to the existing PLuTo polyhedral compiler and provide benchmarks which demonstrate a 1.92× performance gain over the best-performing, non-polyhedral compiler (Intel's ICC). Additionally, their block-level dependency model implementation results in an astonishing 20-168× speedup over the vanilla PLuTo compilation procedure [7].

In experimentation using the LLVM compiler infrastructure tools, Paisante et al. demonstrate drastic improvements in alias analysis techniques over traditional methods. Through the use of novel symbolic methods, a 1.35× improvement in pointer alias disambiguation. The method is able to disambiguate pointers within arbitrarilly-large ranges in linear time (with correlation coefficient $R = 0.982$). Additional modeling improvements are proposed by Ko et al. with a specialized IR[10] for FIFO[11] streams. More precisely, the proposed IR is shown to improve performance over FIFO compile-time stream models with an implementation using the LLVM compiler framework. Their benchmarks with LaminarIR demonstrate an average speedup of 1.56× over the traditional FIFO model, and a 1.34× speedup over the StreamIt compilation framework [5].

[10] Intermediate representation; a language-agnostic representation of source code that optimizers apply transformations to. IRs are converted to machine code in the final stages of compilation.

[11] First in, first out.

The drastic gains demonstrated by ongoing research in internal compiler models show promise for specialized compilation techniques.

Conclusions

Contemporary work on optimizations would appear to be trending towards higher degrees of specialization. Additionally, the ubiquity of issues with memory latency is evident in the amount of research being directed towards reducing CPU cache and register usage. Such optimizations have proven to be fruitful, though the problem of memory latency is unlikely to be completely solved without major innovations in hardware design.

References

[1] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 305–317. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049865

Proposes method of generating code for software-level cache prefetching in contexts where hardware prefetching is deemed insufficient. Focuses on reducing memory latency in contects where indirect memory accesses typically yield poor cache efficiency (linked-lists, trees, graphs, etc). All code generation is done through static analysis at compile-time (no JIT). Implemented in LLVM - benchmarked agains manually-inserted prefetches and shows comparable performance. Results are very promising, 2.1x - 2.7x speedups.

[2] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "A practical and fully automatic polyhedral program optimization system," 2007.

[3] C. Carruth, "Efficiency with algorithms, performance with data structures," 2014, CppCon 2014. [Online]. Available: https://cppcon2014.sched.com/event/1lYBpr1?iframe=no

[4] S. Dissegna, F. Logozzo, and F. Ranzato, "Tracing compilation by abstract interpretation," *SIGPLAN Not.*, vol. 49, no. 1, pp. 47–59, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535866

Largely theoretical and abstract. Focuses on optimizing dynamic languages through the usage of a tracing JIT.

[5] Y. Ko, B. Burgstaller, and B. Scholz, "Laminarir: Compile-time queues for structured streams," *SIGPLAN Not.*, vol. 50, no. 6, pp. 121–130, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737994

[6] J. Lifflander and S. Krishnamoorthy, "Cache locality optimization for recursive programs," *SIGPLAN Not.*, vol. 52, no. 6, pp. 1–16, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062385

Describes a method for optimizing non-trivial recursive function execution with automatic generation of parallel code. The proposed method involves statically annotating function definitions with desciptions of data access patterns. With these annotations, recursive function execution is transformed into (user-level) threaded code where each thread handles the execution of function call.

[7] S. Mehta and P.-C. Yew, "Improving compiler scalability: Optimizing large programs at small price," *SIGPLAN Not.*, vol. 50, no. 6, pp. 143–152, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737954

Proposes alternative to typical compiler dependency model; blocks of statements, rather than individual statements themselves, are viewed in aggregate and modelled as single nodes in a dependency graph. This

[8] V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira, "Symbolic range analysis of pointers," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 171–181. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854050

Proposes a method which employs abstract interpretation to conduct global and local alias analysis on pointers. Particularly, the method focuses on deducing whether a subsection of a given pointed-to range aliases a subsection of another. Using symbolic methods, while-program alias analysis can be conducted in linear time.

[9] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun, "Surgical precision jit compilers," *SIGPLAN Not.*, vol. 49, no. 6, pp. 41–52, June 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594316

Addresses the problem of unreliability in modern JIT compilers, like the HotSpot Server compiler (JVM). Proposed is a programmer-visible API to the respective managed language JIT compiler. Lancet, the corresponding implementation of the proposed JIT compiler, makes use of Abstract Interpretation.

[10] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," *SIGPLAN Not.*, vol. 49, no. 6, pp. 65–76, June 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594342

Proposes optimization techniques for stencil operations (loop -> compute -> update) with a focus on minimizing usage of register real estate. Leverages associativity and commutativity of addition and multiplication to transform operations in nested loop to develop efficient access patterns that use a minimal number of registers.

[11] K. A. Tran, T. E. Carlson, K. Koukos, M. SjÃd'lander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead

compile-time scheduling," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2017, pp. 171–184.

Proposes a method for compile-time reordering of memory-bound instructions to improve performance on non-complex out-of-order CPUs. Method consists of splitting instruction sequences in loop bodies into multiple "Access Phases" and "Execute Phases". Memory-bound instructions are hoisted up into access phases in a safe, dependency-aware manner while following execute phases are simplified to reduce register pressure. Benchmark results show mostly small improvements across different C library recompilations, with a few large improvements and occasional pessimisation.