Interesting compiler projects [closed]

Asked 16 years, 2 months ago Modified 12 years, 3 months ago

Viewed 18k times

26

Closed. This question is off-topic. It is not currently accepting answers.

Want to improve this question? Update the question so it's on-topic for Stack Overflow.

Closed 12 years ago.

Improve this question

I'm currently in the process of choosing a project for a grad-level compiler course to be done over the next 8 weeks. I'd like to do something related to optimization since I haven't worked much in that area before, but anything in the field is fair game.

What was the most interesting compiler-related project you've done? What did you learn the most from?

Edit: Thank you all for your great suggestions. I apologize for not updating this for so long.

The project I ended up doing was a simple autovectorization optimization on LLVM. LLVM has vector

types, but there didn't seem to be any way to take advantage of them without support for the front-end. This optimization converted normal scalar code into vector code.

Since auto-vectorization is a fairly difficult optimization to implement, we limited our scope as much as we could. First, in order to expose instruction level parallelism in the code, we looked for one-block loops that matched our criteria, then unrolled them a specific number of times so they would be conveniently vectorizable. We then implemented the packing algorithm laid out in Exploiting Superword Level Parallelism with Multimedia Instruction Sets by Larsen and Amarasinghe.

Even a simplified version of this optimization is pretty complicated. There are a lot of constraints; for instance, you don't want to vectorize a variable that lives out of the loop, since the rest of the program expects it to be scalar. We put in a lot of hours in the last few weeks. The project was a lot of fun though, and we learned a lot.

compiler-construction

Share

Improve this question

Follow

edited Sep 15, 2012 at 23:19

Bill the Lizard

405k ● 211 ● 572 ● 889

asked Oct 8, 2008 at 17:24

Jay Conrod

29.6k ● 20 ● 99 ● 110

2 So Jay it's been over 8 weeks. Let us know what happened. – Dour High Arch Jan 6, 2009 at 20:05

1 Any news Jay? I will soon start teaching an undergrad compiler course and it will be interesting to know what you did. – Fabio Vinicius Binder May 20, 2009 at 16:20

18 Answers

Sorted by: Highest score (default)

13

With an 8-week timeframe, you're going to need to be careful about "scope creep". That is don't be too ambitious, esp. if this project includes other aspects of compiler construction (lexing/parsing), or if you're still learning the tools (debugger, yacc) and intermediate data structures (DAG).

That said, my first suggestion would be to try some Live Variable Analysis. The algorithms are pretty well established, so you'd pretty much just need to code it up specific to your data structures, etc.

This would let you do a limited form of Dead Code Removal. That is, if you detect that a variable is declared

but never used, don't allocate space for it. If you detect that a value is set but never read, don't generate the set.

Live Variable Analysis can help with Register Allocation too, so you might be able to tackle that too if there's time, and you should be able to re-use some of what you build for Dead Code Removal.

Share Improve this answer

Follow

edited Oct 14, 2009 at 14:57

GEOCHET

21.3k ● 15 ● 77 ● 99

answered Oct 8, 2008 at 18:16

Clayton

918 ● 1 ● 6 ● 13

8

A few years ago, I designed a DSL and wrote the compiler for a product that my company produced. The DSL used an odd combination of declarative rules, event-driven logic, and compositional inheritance. It was a very fun project, and I learned a lot.

It really piqued my interest in parsers & compilers, so I've tried to keep up with interesting new developments in compiler technology.

With respect to optimization, here's a fun article that I read last year:

http://theory.stanford.edu/~aiken/publications/papers/asplos06.pdf

In this paper, the authors describe a technique for automatic discovery of peephole optimizations (surpassing the performance of several popular C++ compiler back-ends) without an expert having to write a bunch of special-case code. Their technique uses unsupervised learning algorithms to discover high-value peephole replacements.

After reading it, it occurred to me that their approach could be augmented by providing the algorithm with a "machine description" listing all of the instructions (with their primary effects and their side-effects) supported by the target processor architecture. Then, rather than using a brute-force approach to finding equivalent instruction sequences, the solver could find those sequences much more easily.

The machine learning algorithm would still use empirical observations to determine the most efficient sequence of instructions (because cache effects, micro-ops, and pipelining almost demand empirical timing data), but the result-equivalency could be predicted using an algebraic theorem-prover, operating on the machine description.

In the paper, they talk about how their optimizers could only discover peephole replacement sequences of two or three instructions (because, otherwise, the brute-force search would take too long and consume too much memory). Putting a smart solver at the right place in the algorithm could enable it to work with longer replacement sequences.

Anyhoo... let me know when you finish with that project! And don't forget to mention me in your "Acknowledgements" section!! ;-)

Share Improve this answer

Follow

answered Oct 8, 2008 at 18:37

benjismith

16.7k ● 9 ● 61 ● 83

7

If you're interested in optimization, Vectorization of loops using SSE and MMX instruction sets could be interesting.

Share Improve this answer

Follow

answered Oct 8, 2008 at 17:56

shoosh

78.8k ● 57 ● 213 ● 331

I doubt it could be done in 8-week time frame. Automatic Vectorization is a notoriously hard problem, and even GCC doesn't have that. – Calyth Jan 6, 2009 at 20:00

I hate to mark an accepted answer since these are all great suggestions, and there isn't really a true answer to this question. However, this is the project we actually did, and it was a very interesting project so I feel I should award some points here. – Jay Conrod May 24, 2009 at 3:23

@Calyth, You're right, we never could have implemented a full autovectorization optimization in 8 weeks. We limited the scope of our optimization as much as possible without making it completely useless. We still ended up putting more

hours into it than we anticipated. – Jay Conrod May 24, 2009 at 3:24

7

I think writing my own simple embedded scripting language was one of the most interesting compiler related project I have done. It taught me every aspect of the process from design to concept, and since I was doing it from scratch I could make it as simple or complex as I needed which allowed me to understand the concepts with out a lot of noise, that modifying a established project might have.

Share Improve this answer

Follow

answered Oct 8, 2008 at 17:33

Scott Dillman

831 ● 1 ● 9 ● 16

With you totally. I think generating simple languages (or getting existing languages to act like the language you want) is a very basic skill. – Mike Dunlavey Jan 6, 2009 at 22:48

5

Look into helping the Shed Skin project, which compiles Python to C++. I think over the summer, a call for help was made. Determining ways to improve the compilation to C++ would provide phenomenal optimization to python programs!

http://code.google.com/p/shedskin/

Share Improve this answer

edited May 21, 2009 at 1:22

Follow

answered Oct 8, 2008 at 19:06

torial

13.1k ● 9 ● 65 ● 89

1 I think you meant to say "Python to C++". Yes. Good project. – Mike Dunlavey Jan 6, 2009 at 23:05

Even though I'm many months too late, I'm upvoting this in support of Shed Skin. It's what I would have suggested had I found this question at the time it was asked. – John Y May 24, 2009 at 4:21

5

For learning compilers, doing it end-to-end is the best idea. Using a simple backend machine, NOT an x86, rather select some simple machine like a bare-bones MIPS. I did my undergrad compiler project targeting a PDP-11 simulator, which was a great target as it kept things very simple. Thanks to some sample code, we could do a simple imperative language compiler in about four weeks. In C!

Today, with powerful languages like ML, doing a compiler should be much easier.

What you do should really depend on what your interest is in. if it is in optimization, find some kind of simple framework where that is all you need to concern yourself with.

I think the most fruitful area today got to be in compiling for threaded targets or in just-in-time compilation.

Share Improve this answer

Follow

edited Oct 21, 2010 at 21:45

Ferruccio

101k ● 38 ● 229 ● 303

answered Oct 8, 2008 at 17:53

jakobengblom2

5,843 ● 2 ● 28 ● 34

Good answer. I would only add what I said in my own answer about optimization - that it is overrated. – Mike Dunlavey Jan 6, 2009 at 20:29

We used a Motorola 68000 as if it were a stack machine. You can't get much simpler than that. – David Thornley Jan 6, 2009 at 20:46

I often wish IBM had chosen Motorola instead of Intel. Things would be a lot different. – Mike Dunlavey Jan 6, 2009 at 22:43

4

I once wrote a programming language and a virtual machine to run it on. The language was able to interface to function written specifically for in that were contained in a DLL (this was before OLE automation) on 16 bit Windows.

Doing the whole front-to-back thing gave me a great understanding on a language from both ends. I'd read various compiler books at the time (such as the infamous Dragon book) but it never really sunk in until I wrote

something concrete. Now, many years on, the work I did have given me a richer appreciation and understanding of have things like the Java VM and the CLR work.

Share Improve this answer

Follow

answered Oct 8, 2008 at 17:43

Sean

62.4k ● 11 ● 99 ● 138

4

Consider type inference for an existing dynamically-typed language.

Share Improve this answer

Follow

answered Oct 8, 2008 at 18:08

finnw

48.6k ● 24 ● 148 ● 223

4

In addition to the "Dragon Book" suggestions, I also highly recommend "Advanced Compiler Design & Implementation", by Steven S. Muchnick, which focuses on intermediate representations, code-generation, and optimization techniques.

Share Improve this answer

Follow

answered Oct 8, 2008 at 18:51

benjismith

16.7k ● 9 ● 61 ● 83

In my undergrad compilers class, we first wrote a recursive-descent (top-down) parser for a Pascal-like

4

language from scratch: Lexical Analyzer, parser, everything.

About midway through the semester, we flipped to parser/scanner generators like lex/yacc or flex/bison. We built a compiler that would take our Pascal subset and compile it down to assembly for a stack machine we were given (the stack machine is uber simple, but the principles are still the same).

If you're interested in compilers, I cannot recommend highly enough the Dragon Book. It's intended to be used for a 1 semester undergrad course, and the second half as a graduate-level course, and covers every bit of theory and optimization you could ever wish for. Even Joel likes it. =)

Cheers

Share Improve this answer Follow

edited Oct 9, 2008 at 0:25

answered Oct 8, 2008 at 18:04

dmercer

397 ● 5 ● 17

3

Here's another idea... though it's still a little big vague:

Most optimization is done at compile-time (except in JIT compilers, which optimize at runtime). But there are also

a lot of opportunities for optimization at link-time and at install-time.

I'm particularly interested in the idea of a platform where you don't bother linking until install-time. But during that install-time-linking process, you take an aggressively optimization strategy, since you know a lot of detailed information about the machine architecture and can make more subtle and well-informed optimization decisions.

Share Improve this answer

Follow

answered Oct 8, 2008 at 19:24

benjismith

16.7k ● 9 ● 61 ● 83

2

Loop detection and parametrized unrolling should be hard enough to make it interesting. Not very sexy, but getting too sexy in 8 weeks will sink you.

Share Improve this answer

Follow

answered Oct 8, 2008 at 18:00

Paul Nathan

40.2k ● 30 ● 120 ● 215

2

You could write an optimizer for IronScheme, as it currently does sweet blow all, except a few 'intrinsic' functions. `:)`

Share Improve this answer

Follow

answered Oct 8, 2008 at 18:26

community wiki
leppie

2

Other interesting projects might include:

- Add a tail-call optimization to the open-source Sun JVM.
- Add a named return value optimization (NRVO) to the Python or Ruby VM's.
- Add just-in-time regex-to-bytecode compilation, for your favorite target (.Net already has it. I'm pretty sure Java doesn't.)

Share Improve this answer

Follow

answered Oct 8, 2008 at 18:45

benjismith

16.7k ● 9 ● 61 ● 83

I'm not sure tail-calls make sense in an OO language, because you need to statically know which method will get executed for the recursive message send. So that means doing that in a JIT, using runtime type feedback, if I'm not mistaken ? – Damien Pollet Apr 9, 2009 at 0:46

Tail-call optimization does not need to statically know the target. The JVM does present a different obstacle (stack introspection), which can be finessed though (there was a paper by Appel and others). – Darius Bacon May 24, 2009 at 4:05

2

While writing your own language is a fun and traditional academic activity, there are already too many poorly-implemented compiler-related projects out there. Also, 8 weeks is not enough time to do a good job on a complete language implementation.

If you're interested in "something related to optimization", pick an already-existing language or VM and optimize it for performance, size, or both. This will prevent you from having to reinvent the entire language implementation, lets you focus on the optimization problem, will produce a product useful for other people and not just another academic exercise, and can even be useful in getting a job.

I believe the Parrot bytecode interpreter and various .Net Iron-language parsers could benefit from even simple optimizations.

Share Improve this answer Follow

edited Oct 8, 2008 at 19:40

answered Oct 8, 2008 at 18:49

Dour High Arch

21.7k ● 30 ● 77 ● 93

2

I've done this in my own teaching "way back when". I wouldn't give so much emphasis to optimization as to other things, like type inference or JIT or bytecoding or object format / debug support.

There is no harm in concentrating on optimization as long as you also convey that it is a lot less important than people think. It only matters in code that:

- runs in tight loops at the bottom of the call stack (i.e. without calling functions, explicitly or implicitly)
- actually occupies a significant percentage of an app's time (i.e. if code is seldom run, optimizing it won't help much).

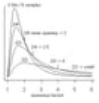
This is a fairly small fraction of all the code that the compiler will see.

Edit: Sadly, I cannot escape from working with Fortran compilers, which scramble the code mercilessly, making it very hard to debug, while having epsilon effect on performance.

Share Improve this answer Follow

edited Jan 6, 2009 at 20:25

answered Jan 6, 2009 at 20:13



Mike Dunlavey

40.6k ● 15 ● 94 ● 138

1

Automatic inline code generation using heuristic tests for size/speed trade offs.

Share Improve this answer Follow

answered Jan 6, 2009 at 19:52

plinth

49.1k ● 11 ● 83 ● 123

The B::CC perl compiler would benefit from adding and analyzing types. I just don't have enough time for that yet.

Had enough time lately.
http://blogs.perl.org/users/rurban/2011/02/use-types.html

0

Share Improve this answer Follow

edited Feb 21, 2011 at 20:58

answered Oct 22, 2010 at 1:35

rurban
4,131 ● 26 ● 28