Code compilation

State of the art

Ramon Brullo | it211505

Abstract

Since their invention in the 50s and today code compilers have undergone a long history of development. This paper will give a brief summary on how compilers work and an overview of the current state of research. From there topics of interest and future research directions will be shown.

Contents

Abstract 2
Introduction 4
Research Efforts 5
  Parallelism 5
  Machine learning 5
  JIT compilation 5
  Code compaction 6
  Compilation in small steps 6
  Hardware Compatibility 6
Conclusion 7
References 8

Introduction

Computers naturally can only execute sequences of instructions in order to accomplish tasks and compute results. However, specifying algorithms and programs as an imperative sequence in this way is not intuitive for humans and thus required a lot of effort on the part of software-developers.

A solution to this problem is to write source-code in a more human-friendly format and use an automated process to convert this textual program into the instructions the computer can execute. This process is called compilation and can be performed by a separate program which is thus called a compiler. (Wirth, 2005, p. 6)

The first such compiler was developed by Corrado Böhm for their PhD dissertation which featured a custom programming language as well as the compiler written in that same language. (Böhm, 1954) This compiler was an example of a self-hosting compiler, meaning that it was written in the same language it was designed to compile.

In general, program compilation can be split into the following parts:

1. **Lexical analysis**: The source-code is read and translated into a sequence of meaningful symbols which make up the text. (Wirth, 2005, p. 6) In the next step, this sequence of symbols or tokens will be used instead of the "raw" text.
2. **Syntax analysis**: The symbol-sequence is translated (parsed) into a data-form, usually a tree-structure, which more accurately represents the syntactic structure of the program. (Wirth, 2005, pp. 6–7)
3. **Verification and optimization**: The syntax-tree is now verified to ensure that no detectable errors end up in the compiled code. (Wirth, 2005, p. 6) It can also be optimized in this stage, meaning, its performance or memory-usage improved. (Wirth, 2005, p. 100)
4. **Code generation**: The verified and optimized source-tree must finally be converted to the desired output-format, usually binary or an assembly-language. (Wirth, 2005, pp. 6–7) The output of the generation phase, may also be source-code of a different programming language, in which case the process is called transpiling instead of compiling. (Parr, 2021)

Research Efforts

There are currently several fields of research in order to improve compilers. The following chapters will give a broad overview.

Parallelism

One of the driving forces in increasing computer performance over the past decades has been increasing the number of transistors which can fit into CPUs. However, since transistors are physical components, adhering to physical rules, there is a minimum size transistors can be built in and thus also a natural limit to performance from transistor-count. (Sutter, 2005)

Increasingly, running multiple CPUs or cores in parallel, will be used in order to increase performance. (Gepner & Kowalik, 2006) It has thus become a goal of compilers to optimize code by generating instructions which can be run in parallel on multiple processors. For this the compiler must analyze the program to detect which calculations are independent and can run detached from one another. (Midkiff, 2012)

Machine learning

An originally niche field which has moved into the mainstream over the past decades is using machine learning algorithms in aiding code optimization. Machine learning may be used to evaluate different optimization heuristics and determine how to best parallelize processes. (Ashouri et al., 2019; Wang & O'Boyle, 2018)

JIT compilation

JIT or "just in time" compilation is a technique of compilation in which the source-code is compiled on the target machine right before being executed, similarly to interpretation. While JIT compilation is not a new concept as detailed by Aycock (Aycock, 2003), there could still be untapped advantages in the technique.

Code compaction

An important field of study is the optimization of generated instructions in minimizing storage space. This is especially important when compiling for devices which only support limited storage capacity. (Debray et al., 2000)

Compilation in small steps

In their book “Functional reactive programming”, Blackheath and Jones propose that compilers of the future should be able to handle more high-level concepts such as, dependency-graphs better in order to allow programmers to work on these higher levels without concerning themselves with the underlying implementation. (Blackheath & Jones, 2016)

Many compilers compile directly from the generated syntax-tree to the desired output-language, which makes it difficult for compiler-developers to smoothly translate these high-level concepts directly to low-level instructions.

There also exist compilers which insert a number of intermediary languages, which aim to “lower” the code down to the low-level language in small steps in order to maximize the possibilities of optimization at each level. (Lattner et al., 2020)

Hardware Compatibility

Many target-devices feature processors with their own proprietary instruction-sets. In order to be able to target different devices, the code-generation algorithms of the compiler must take the target architecture into account. It is therefore an active research field to develop compilers which can target multiple different architectures while not sacrificing compilation or code performance. (Baghdadi et al., 2018)

Conclusion

This short paper gave a brief overview of current research fields regarding code compilation. Important fields include:

- Performance optimization of generated code
- Memory optimization of generated code
- Compatibility
- Better support for high-level concepts
- Incorporation of machine-learning into compilation

References

Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., & Silvano, C. (2019). A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys*, *51*(5), 1–42. https://doi.org/10.1145/3197978

Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys*, *35*(2), 97–113. https://doi.org/10.1145/857076.857077

Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., & Amarasinghe, S. (2018). Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *ArXiv:1804.10694 [Cs]*. http://arxiv.org/abs/1804.10694

Blackheath, S., & Jones, A. (2016). *Functional Reactive Programming*. Manning Publications.

Böhm, C. (1954). Calculatrices digitales. Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme. *Annali di Matematica Pura ed Applicata*, *37*(1), 175–217. https://doi.org/10.1007/BF02415099

Debray, S. K., Evans, W., Muth, R., & De Sutter, B. (2000). Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, *22*(2), 378–415. https://doi.org/10.1145/349214.349233

Gepner, P., & Kowalik, M. F. (2006). Multi-Core Processors: New Way to Achieve High System Performance. *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, 9–13. https://doi.org/10.1109/PARELEC.2006.54

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., & Zinenko, O. (2020). MLIR: A Compiler Infrastructure for the End of Moore's Law. *ArXiv:2002.11054 [Cs]*. http://arxiv.org/abs/2002.11054

Midkiff, S. P. (2012). Automatic Parallelization: An Overview of Fundamental Compiler Techniques. *Synthesis Lectures on Computer Architecture*, *7*(1), 1–169. https://doi.org/10.2200/S00340ED1V01Y201201CAC019

Parr, K. (2021, January 10). *Compiling vs Transpiling*. DEV Community. https://dev.to/kealanparr/compiling-vs-transpiling-3h9i

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, *30*(3). http://www.gotw.ca/publications/concurrency-ddj.htm

Wang, Z., & O'Boyle, M. (2018). Machine Learning in Compiler Optimisation. *ArXiv:1805.03441 [Cs]*. http://arxiv.org/abs/1805.03441

Wirth, N. (2005). *Compiler Construction*. Addison-Wesley.