International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

A Survey of Compiler Optimization Techniques

Aman Raghu Malali[1], Ananya Pramod[2], Jugal Wadhwa[3], Sini Anna Alex[4]
[1,2,3]*Student, Department of Computer Science Engineering, Ramaiah Institute of Technology, Bengaluru, India*
[4]*Assistant Professor, Dept. of Computer Science Engg., Ramaiah Institute of Technology, Bengaluru, India*

***Abstract*: This survey paper aims to illustrate the major advancements in techniques for optimization of compilers. Optimizing compilers is a crucial task and often requires extensive work for different types of applications. Compiler suites that employ the latest optimization techniques can offer a plethora of benefits to developers as well as routinely produce code 20-30% faster than standard compilers. Superior optimization would also allow developers to cut cost and improve products without having to fall back on higher speed and higher cost processors. In this paper we discuss the various techniques of obtaining better compiler optimization by using different cache optimization techniques, dynamic optimization techniques as well as optimizing compilers for dynamic languages such as JavaScript. Machine learning techniques that can be employed for better optimization in compilers are also discussed.**

***Keywords*: compiler optimization, cache optimization, dynamic, JavaScript, machine learning, survey.**

1. Introduction

With the rise of various forms of computer programming languages, the multitude of compilers available has also increased. Compiler optimization therefore is a crucial task, allowing code to be run faster as compared to standard compilers without optimizations. Compilers can be optimized by employing a variety of different techniques, such as cache optimization, dynamic optimization etc. The paper is broadly divided into five different categories of compiler optimization, cache optimization, optimization for dynamic languages such as JavaScript, dynamic optimization, machine learning optimization techniques and general optimization techniques. Cache optimization is a technique employed in order to maximize the the reuse of registers and the cache. This is often achieved by using a variety of scheduling algorithms as well as improving data locality in order to reduce the time for memory access. In the case of dynamic languages such as JavaScript, optimization is a crucial yet difficult task. Thus by using the concept of parallelization and removing ambiguity in tasks such as type assignment the performance of dynamic languages can be drastically improved.

Dynamic optimization are techniques that can be used in Just in Time(JIT) compilers in order to improve code performance dynamically and improving the code performance of languages using JIT compilers. With the rise of machine learning and deep learning with the easy availability of GPU's, various machine learning techniques can be used to optimize as well as fine tune compilers. The ease with which machine learning models can solve complex tasks is what makes it a suitable approach to optimize compilers. Also with the use of genetic algorithms, compilers can be further fine-tuned in order to provide a high degree of performance.

2. Literature survey

A. Cache optimization

Kevin Stock et al. in their research developed a framework to enhance register reuse in regular loop computations. This was achieved by utilizing the commutativity and associativity of operations. Optimization frameworks can be used to enhance performance in a particular class of important computations known as stencils. The research further showed how register reuse can be exploited as well as reduce the number of loads/stores by optimizing stencil operations. Higher order stencil operations can be optimized by reordering operations in such a way as to enhance data locality and register reuse. The effectiveness of this transformation framework for a range of higher order stencils was demonstrated by using a multidimensional retiming formalism [1].

Hee-Seok Kim et al. proposed an OpenCL compiler which schedules work-items according to their data locality. By closely monitoring the the memory locations accessed in loops within a kernel, the scheduling technique can schedule work-items to create better memory access patterns. This would lead to a speedup over multiple different types of CPU architectures. The existing work-item scheduling techniques are inherently flawed as they do not take into consideration the impact of work-item scheduling on memory access pattern. In the proposed system the decision of the preferred schedule is made on a per-loop basis. This would mean that multiple loops in the same regions could potentially be assigned different schedules. The proposed technique achieved a speedup of 1.71x over Intel and 3.32x over AMD's implementations on Parboil and Rodinia benchmarks [2].

Wei Ding et al. in their research showed how the number of links that off chip data access through the on chip network can be reduced in a chip. By using a compiler-based off-chip data access localization strategy the number of hops required for the off chip data to reach the memory controller can be minimized. Doing so can reduce the network latency of the the off-chip accesses. It would also reduce the network latency of on-chip access. The frameworks is the first compiler based work that targets optimizing on-chip network behaviors of off chip

accesses in both private L2s and shared L2 based multicores. The distance between the requested node to that of the target memory controller is reduced. This allows the off-chip data to travel over shorter distances and further reduces the on-chip access latencies [3].

Steve Carr et al have provided a simple and effective way for cost estimation of loop nest considering with respect to references of cache lines. Attempts have been made to measure inherent data locality of programs and improve them. Fusion, distribution, and reversal have been analysed taking cost model into consideration. Loop Fusion Algorithm and Compound Transformation Algorithms have been proposed with optimizations and tested on cache hit rates. These algorithms have been used to implement cost models, the transformations, and the algorithms in Memoria. The approach presented is unique in the sense that it is the first to combine techniques such as fusion, distribution and reversal to improve data locality [4].

B. Optimization for dynamic languages

Kyle Dewey et al. demonstrated how a JavaScript abstract interpreter could be parallelized. Because of JavaScript's tricky semantics and dynamic behaviour, the static analysis of JavaScript proves to be difficult. The paper presents an alternative model to analyze programs, called STS an alternative to the usual DFA approach. STS splits he analysis into two separate components. An embarrassingly parallel reachability computation on a state transition system, and a method to selectively merge states during reachability computation. The STS framework makes it easy to explore different parallelization strategies. It is also more applicable to languages with difficult control flow as compared to DFA. The results shown were conclusive that the new STS framework by using parallelization provided significant speedups for realistic Java programs [5].

Wonsun Ahn et al. in their research showed how performance in dynamic languages like JavaScript can be improved by examining how data types are assigned to variables. JavaScript does not have a concept of types but still assigns them to objects for ease of code generation. The papers describe how Chrome V8 compiler defines types and the main reasons for the lack of performance improvement. A newer modified chrome V8 compiler is illustrated which removes the requirements (1) the inherited prototype object is part of the current object definition and (2) method bindings are also part of type definition. These requirements are what make type assignment unpredictable. The paper identifies the core problems hampering optimization in type predictability. The optimization methods suggested include function creation, optimizing built-in object creation, partial and complete decoupling of methods from types [6].

Michael R Jantz et al. explore the various single and multilevel (Just In Time) JIT compilation policies for modern machines. Dynamic compilation is important for languages such as Java and C# in order to achieve high performance. In the paper they describe experiments in order to control the compiler aggressiveness and optimization levels in Oracle HotSpot Java VM. By analyzing all the various JIT compilation policies, the most effective policy for any particular application can be identified. It was proven that employing all the free compilation resources aggressively to compiler more program methods eventually reaches a point of diminishing returns. At the same time using free resources to reduce the queue backup significantly increases the performance especially in slower JIT compilers. The paper further shows how prioritizing JIT method is crucial in systems with smaller hardware budgets [7].

C. Dynamic optimization

Byron Hawkins and Brian Demsky et al. Maintaining consistency between the translated application and the source application poses a special performance challenge. Two methods to optimize binary translations are introduced in this paper. This first method is of efficiently annotating the source code, allowing the demarcation of the dynamic regions by the developer and then identifying the code changes in only these regions. Another technique avoids source code requirements and the need for annotation by automatically inferring the presence of a JIT. Then the write instructions are instrumented with translation consistency operations. The techniques in the DynamoRIO showed a performance improvement of 7.3x over the existing state of the art DBT systems on JIT applications [8].

Tiark Rompf et al. Discuss an interesting take on optimizing JIT compilers by converting them into precision tools by adding generic metaprogramming facilities. First, allowing JIT compilation to be invoked explicitly by the programs. It would also enable JIT compiler to report errors and warnings to the program when it is unable to compile a given code in the demanded way. The second metaprogramming ability would be that of allowing JIT compiler to perform compile time computation by being able to call back into the program. This allows the program to itself define the translation strategy for constructs dynamically and enabling "smart" libraries which would supply domain specific compiler optimizations. The framework proposed was Lancet, a JIT compiler framework for Java Bytecode. Lancet allows tight, two-way integration with the running program. Lancet was created by adding abstract interpretation to a simple bytecode compiler using lightweight modular staging [9].

D. Machine learning optimization techniques

Zheng Wang et al. described the use of ensemble machine learning models for compiler optimization. The ensemble model was a mix of different machine learning techniques such as supervised and unsupervised learning. The proposed ensemble model consisted of algorithms such as Decision Trees, Support Vector Machines as well as K-means clustering. The ensemble model was tested and it was shown to produce accurate as well as efficient results [10].

Jay Patel et al. propose a method for code optimization in compilers using artificial neural networks. The method

proposed in the article requires implementing the different optimization techniques in 4Cast-XL. For each compiled method, a feature vector of current method's state is generated along with profiles of programs. Artificial Neural Networks are then used to predict the best optimization to be used. Instead of using the same optimization for the entire program, the prediction helps to select the best ordering of the optimizations for the program. Phase reordering is then performed using genetic algorithms. An automatic feature generation model is also used comprising of training data generation, feature search and machine learning components. As part of data generation, samples of input programs are compiled in different ways and the optimal heuristic value is decided. The feature search component is associated with feature expressions where the feature evaluated on a program is calculated. The machine learning component provides feedback on how good a feature is and selects the best optimization orderings on a per methods basis within a dynamic compiler by including profile of programs [11].

Dmitry Plotnikov et al. Automatic tuning of compilers would provide the expected performance on a particular platform and also suggest which optimizations can be improved to increase the overall performance. For this a genetic algorithm based automatic tuner was created called Tool for Automatic Compiler Tuning (TACT). It works on a genetic algorithm core, which can be used to find optimal parameters to each of the optimizations. It reads the compiler options from a configuration file and then creates the chromosomes which are random set of compiler options. The chromosomes from each of the populations are then evaluated to check for their effectiveness. The best configuration would then move on to the next generation, random mutations and crossbreeding is introduced at every generation. TACT can handle multiple objective tuning as well as error handling. Various performance measures such as error handling capability, tuning speed, convergence and overall tuning result is used in order to evaluate the impact of using TACT [12].

Enyindah and Okon E. Uko have reviewed use of optimization algorithms in new compilers to reduce the actual size of code. The new techniques are contrasted with the conventional optimization methods. The conventional methods like dataflow analysis, local optimization and global optimization are considered for analysis. The newer compilers use mechanisms like dataflow analysis, Leaf optimization functions, cross linking optimization, etc. Data-Flow Analysis brings in fuzziness in data information. It also includes Alias analysis. Reverse Inlining (Procedural Abstraction) aim is to achieve code size reduction. Leaf Function Optimization involves utilizing leaf functions to reduce code length. Leaf functions are the functions that do not directly call functions in a program and form the leaves in a call graph. Cross Linking optimization technique is used to factor out codes so as to reduce the code size. These optimization techniques in new compilers help to utilize memory efficiently, reduce code size and increase program execution speed [13].

3. Discussions

In our survey we identified that cache optimization techniques involve the efficient use of registers and cache memory in order to speed up computation. By exploiting the associativity and commutativity of operations frameworks can reorder important computations ins such as way as to encourage the reuse of memory. Another technique by which cache optimization can be used to optimize compilers is by the use of the principle of data locality. By making sure that frequently accessed data are closer by, the number of load and stores can be greatly reduced. A crucial aspect of cache optimization is the scheduling technique used, by using an appropriate scheduling technique on a per-loop basis can provide significant speedups to the compiler.

Dynamic languages like JavaScript pose an interesting challenge when it comes to compiler optimization. But by parallelizing compilation and by analyzing the programs in separate components can improve the performance of dynamic language compilers. The STS framework leverages various parallelization strategies in order to select the most optimal one. Another technique to improve performance in dynamic languages is by simplifying how type assignment works. By making this step less ambiguous, the programs are shown to have a significant speedup.

Dynamic Optimization, is vastly improves performances for Just In Time(JIT) compilers. Optimizing binary translations is one of the ways in order to improve performance of JIT compilers. By annotating the dynamic aspects of the source code and by only looking for code changes in these regions, the compiler efficiency can be improved. Another technique involves converting the JIT compiler into a precision tool by adding metaprogramming facilities. The metaprogramming abilities discussed are allow programs to explicitly invoke the JIT and another being allowing JIT to perform compile time computation by being able to call back into the program

Machine learning techniques are also being used for the overall performance improvement as well as fine tuning of compilers. Ensemble methods of models can be used in order to improve the performance of the compiler. This ensemble method consisted of K-means, Support vector machines as well as decision trees. Deep learning techniques with the use of Artificial Neural Networks can also be used in order to improve compiler performance, by treating all parameters of compilers as features and iterating through all of them till the most efficient of those is found.

4. Conclusion

A survey of the current research being carried out on compiler optimization techniques have shown how various different subsystems are being worked on in order to improve compiler efficiency. We have discussed various techniques under the categories of cache optimization, dynamic optimization, optimization in dynamic languages as well as the use of machine learning techniques in order to improve

compiler performance. Several different techniques of improving these subsystems in current literature as well as their overall impact on compiler performance was discussed. The survey paper is meant to be an aggregation of current research being conducted in the field of compiler optimization. It is meant to serve as a reference as well as a benchmark for researchers working on new techniques to improve compiler efficiency.

References

[1] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Nol Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. “A framework for enhancing data reuse via associative reordering,” in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), June 2014.

[2] Hee-Seok Kim, Izzat El Hajj, John Stratton, Steven Lumetta and Wen-mei Hwu. “Locality Centric Thread Scheduling for Bulk-synchronous Programming Models on CPU Architectures,” in Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO ’15), February 2015.

[3] Wei Ding, Xulong Tang, Mahmut Taylan Kandemir, Yuanrui Zhang, and Emre Kultursay. “Optimizing Off-Chip Accesses in Multicores,” in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’15), June 2015.

[4] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality. SIGPLAN Not. 29, 11 (November 1994), pp. 252-262.

[5] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. “A Parallel Abstract Interpreter for JavaScript,” in Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO’15), February 2015.

[6] Wonsun Ahn, Jiho Choi, Thomas Shull, Mara J. Garzarn, and Josep Torrellas. “Improving JavaScript performance by deconstructing the type system,” in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), June 2014.

[7] Igor Costa, Pericles Alves, Henrique Nazare Santos, Fernando Magno Quintao Pereira. “Justin-Time Value Specialization,” in Proceedings of the 2013 International Symposium on Code Generation and Optimization (CGO ’13), February 2013.

[8] Byron Hawkins, Brian Demsky, Derek Bruening and Qin Zhao. “Optimizing Binary Translation of Dynamically Generated Code,” in Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO ’15), February 2015.

[9] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, Hyouk Joong Lee, Hassan Chafi and Kunle Olukotun. “Surgical precision JIT compilers,” in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), June 2014.

[10] Wang, Zheng, and Michael O'Boyle. "Machine learning in compiler optimization." Proceedings of the IEEE 99 (2018): 1-23.

[11] Jay Patel and Mahesh Panchal, International Journal of Computer Science and Mobile Computing, Vol.3 Issue.5, May- 2014, pp. 557-561

[12] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov and Je-Hyung Lee (2013). Automatic Tuning of Compiler Optimizations and Analysis of their Impact. Procedia Computer Science, 18, pp. 1312–1321.

[13] Enyindah P., Okon E. Uko "The New Trends in Compiler Analysis and Optimizations". International Journal of Computer Trends and Technology (IJCTT) V46(2):95-99, April 2017.