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

Optimization of Program Flow using Hybrid Approach of Compiler Optimization and Code Optimization using Machine Learning and ANN

Sini Anna Alex[1], S. Abhishek[2], K. Sidhartha Nambiar[3], K. Keshava Pranath[4]
[1]*Assistant Professor, Dept. of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*
[2,3,4]*Student, Dept. of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*

***Abstract*: Compilers today have large amount of optimization to choose from, and correct one can have significant impact. Also the correct order of applying those optimization has been a long standing problem in compiler research. Traditional system applies same set of optimization in same order to all functions in a program. But, understanding the interactions of optimizations is important in determining good solution to phase ordering problems. Hence, we develop a system that selects good optimization order on per method basis within dynamic compiler automatically. Machine learning based compilation has become mainstream activity in the last decade. Machine learning and compiler optimization can be combined with the main concepts of features, models, training and deployment. Along with the above optimizations, compilers can also be optimized by parallelizing application for symmetric and shared memory multiprocessors. Algorithm considers data locality, parallelism and the granularity of parallelism. Optimization is given by using dependence analysis and a simple cache model. Inter procedural analysis and transformations is used to achieve optimization across procedures.**

***Keywords*: compiler optimization, code optimization**

1. Introduction

There are 2 common requirements to be done for optimizing compiler. One, to minimize the time taken for execution and the other is to minimize the amount of memory occupied. Compiler optimization s generally implemented using a sequence of optimizing transformations, algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources. It has been shown that some code optimization problems are np-complete, or even undecidable. It is generally a very CPU- and memory intensive process.

Compilers translate the code written by humans to binary executable code. Here the correctness is critical and caution is a by-word. Machine learning is an area of AI aimed at detecting and predicting patterns. Hence these can fit and develop into a research domain.

Based on the feedback from compilers, user rewrites the program. Programs rewritten are always independent of any particular vector hardware and is style in which it is written is easily controllable to vectorization. Because of this machine depended vector code were able to generate by the compilers with excellent results. It is very hard to measure the success of parallelized compilers because the presence of parallelism in a dusky seck program is can’t be spotted that easily. There are different kinds of programs mainly of three types where it can be sequential, parallel or in between. The code and algorithmic requirement of the parallelized programs are different from that of the sequential counterparts. These programs are tested with the different fortran collected codes. There are basically two types of code optimization, Machine independent and machine dependent. Optimization regardless of compiler and processor is machine independent optimization. Some important attributes of compiler and processor are to be considered for machine dependent optimization e.g. cycles per element.

One of the important place of optimization is loops. Manly program spend bulk of times in the inner loops. This is a major part where code has to be optimized. By decreasing the number of instruction in the inner loop, running time of the program can be improved. This will not be effected even if the code in the outside of the loop is increased.

Primary jobs of compilers are translation and optimization. The code first has to be converted to binary correctly. After this, the most efficient possible translation is found. With each translation the performance varies significantly, where there are many correct translations. The vast majority of research and engineering practices is focused on this second goal of performance, traditionally misnamed optimization.

The reader is assumed to be familiar with the data dependencies. k represents a hybrid direction/distance vector for a data dependence between two array references. Each entry present in the vector gives the distance or the direction in the loop iteration where location to same location is compared. Dependence vectors are written left to right from the outermost to innermost loop enclosing the references.

2. Algorithm

The way toward applying ML to compilers include:

A. *Feature generation*

The model depends on a lot of quantifiable properties, or highlights, to portray the programs. There are a wide range of features that can be utilized. These incorporate the static data structures that is from the program source code or the compiler transitional portrayal, (for example, the quantity of guidelines or branches), dynamic profiling data, (for example, execution counter qualities) acquired through runtime profiling, or a mix of the both.

Changes that improves the code: Code Motion, Reduction in strength, Common Subexpression Elimination, Loop Unrolling.

3. Procedure

Data Generation, Feature Search

A. Learning model

The second step is to utilize training data to infer a model utilizing a learning calculation. In contrast to different uses of ML, we normally create our very own training data utilizing existing applications or benchmarks. The compiler designer will choose preparing programs which are run of the mill of the application area. For each preparation program, we figure the element esteems, gathering the program with various improvement choices, and running and timing the accumulated pairs to find the best performing choice. This procedure produces, for each preparation program, a preparation case that comprises of the element esteems and the ideal compiler alternative for the program. Mostly three sorts of ANN models are available single layer feed forward system, Multilayer feed forward system and repetitive system.

Optimize–Uses loop permutation and tiling on a single nest to exploit data locality and parallelism.

The steps involved in algorithm:

Reference Groups,

Loop Cost in Terms of Cache Lines

Memory Order

Achieving Memory Order

Tiling for Parallelism

Fuser–Incorporating loop fusion and distribution to enable Optimizer on a single nest and to increase the granularity of parallelism across multiple nests. Beginning with the innermost loop ln in a nest, the algorithm divides the statements into strongly connected regions scrs based on the dependences.

B. Intra and Internest Parallelization

We consolidate Optimize and Fuser to upgrade circle settles inside a solitary strategy. We call this algorithm Parallelize. It combines fusion and distribution with Optimize to acquaint powerful parallelism and with improve the granularity of parallelism accomplished. Parallelize–Combines Optimize and Fuser, resulting in an effective intraprocedural parallelization algorithm for loop nests.

C. Loop distribution

If a loop nest cannot be parallelized effectively using Optimize. Distribution algorithm. Beginning with the innermost loop ln in a nest, the algorithm Distribute divides the statements into strongly connected regions scrs based on the dependences.

D. Loop fusion

Loop fusion merges multiple loops with conformable headers into a single loop. Two loop headers are conformable if they have the same number of iterations and are both either sequential or parallel loops. It is safe if it does not reverse any dependencies between candidate loops. We only perform safe fusions. Our goal is to maximize parallelism. Subject to this constraint, we then minimize the number of parallel loops. Fusion does not combine two parallel loops when dependences would force the resulting loop to execute sequentially.

E. Deployment

In the final step, the learned model is embedded into the compiler to anticipate the best streamlining decisions for new projects. To make a prediction, the compiler first extracts the features of the input program, and then feeds the extracted feature values to the learned model to make a prediction.
Enabler: Inter procedural Analysis and Transformation
We introduce two new inter procedural transformations:

1. *Loop embedding*–which pushes a loop header into a procedure called within the loop, and
2. Loop extraction–which extracts an outermost loop from a procedure body into the calling procedure.

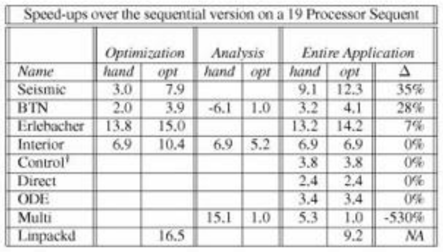
The following subsections first review the inter procedural analysis we need and, then, describe the extensions to the loop transformation, and our use of inter procedural transformations.

For our experimental validation, we take measure our algorithm's ability to match or exceed performance on parallel programs written by programmers who thought and cared about parallel performance, not dusty deck sequential programs. The standard estimation is therefore a hand coded parallelized program. We assembled programs written for a variety of parallel machines. We eliminated all the parallel loops and synchronization to create sequential versions of each program. We then applied our algorithm to these sequential versions. The compiler was required to utilize its examination and calculations to parallelize the program.

4. Results

Experimental results demonstrate that profiles of program can be used for optimization of code.

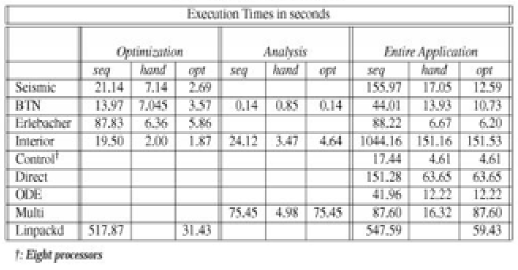
Table 1
Speed ups over sequential program versions



Speed-ups over the sequential version on a 19 Processor Sequent

	Optimization		*Analysis*		*Entire Application*		
Name	*hand*	*opt*	*hand*	*opt*	*hand*	*opt*	Δ
Seismic	3.0	7.9			9.1	12.3	35%
BTN	2.0	3.9	-6.1	1.0	3.2	4.1	28%
Erlebacher	13.8	15.0			13.2	14.2	7%
Interior	6.9	10.4	6.9	5.2	6.9	6.9	0%
Control[†]					3.8	3.8	0%
Direct					2.4	2.4	0%
ODE					3.4	3.4	0%
Multi			15.1	1.0	5.3	1.0	-530%
Linpackd		16.5				9.2	*NA*

Table 2
Program execution times



Execution Times in seconds									
	Optimization			*Analysis*			*Entire Application*		
	seq	*hand*	*opt*	*seq*	*hand*	*opt*	*seq*	*hand*	*opt*
Seismic	21.14	7.14	2.69				155.97	17.05	12.59
BTN	13.97	7.045	3.57	0.14	0.85	0.14	44.01	13.93	10.73
Erlebacher	87.83	6.36	5.86				88.22	6.67	6.20
Interior	19.50	2.00	1.87	24.12	3.47	4.64	1044.16	151.16	151.53
Control†							17.44	4.61	4.61
Direct							151.28	63.65	63.65
ODE							41.96	12.22	12.22
Multi				75.45	4.98	75.45	87.60	16.32	87.60
Linpackd	517.87		31.43				547.59		59.43

†: Eight processors

5. Conclusion

In this paper we introduced a hybrid approach for optimization of code which includes the machine learning based compilation using an evidence approach for an optimized compilation, parallelization algorithm for balancing the parallelism factor and data locality, and optimizing the program code using neural networks.

Machine learning-based aggregation is presently a standard compiler look into zone and, throughout the most recent decade or somewhere in the vicinity, has produced a lot of scholarly premium and papers. We utilize a powerful procedure to present area, abuse parallelism, and expand the granularity of parallelism. Interprocedural segment investigation is a significant segment of our triumphs. We assessed the parallelization calculation against hand-parallelized programs with promising outcomes. The calculation improves execution over hand-parallelized programs at whatever point it connected advancements, essentially improving execution in three of the nine projects.

References

[1] McKinley, Kathryn S. "A compiler optimization algorithm for shared-memory multiprocessors." IEEE Transactions on Parallel and Distributed Systems 9, no. 8 (1998): 769-787.
[2] Patel, Jay, and Mahesh Panchal. "Code Optimization in Compilers using ANN." (2014): 557-561.
[3] Wang, Zheng, and Michael O'Boyle. "Machine learning in compiler optimization." Proceedings of the IEEE 99 (2018): 1-23.
[4] F. Allen, M. Burke, P. Charles, J. Ferrante, W. Hsieh, and V. Sarkar, "A Framework for Detecting Useful Parallelism," Proc. Second Int'l Conf. Supercomputing, St. Malo, France, July 1988
[5] J.R. Allen, D. Callahan, and K. Kennedy, "Automatic Decomposition of Scientific Programs for Parallel Execution," Proc. 14th Ann.
[6] ACM Symp. Principles of Programming Languages, Munich, Germany, Jan. 1987.
[7] J.R. Allen and K. Kennedy, "Automatic Loop Interchange," Proc. SIGPLAN '84 Symp. Compiler Construction, Montreal, Canada, June 1984.
[8] J.R. Allen and K. Kennedy, "Automatic Translation of FortranPrograms to Vector Form," ACM Trans. Programming Languages and Systems, vol. 9, no. 4, pp. 491–542, Oct. 1987.
[9] J. Anderson, S.P. Amarasinghe, and M. Lam, "Data and Computation Transformations for Multiprocessors," Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, Santa Barbara, Calif., July 1995
[10] B. Appelbe, S. Doddapaneni, and C. Hardnett, "A New Algorithm for Global Optimization for Parallelism and Locality," Proc. Sixth Workshop Languages and Compilers for Parallel Computing, Portland, Ore., Aug. 1993.
[11] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques & Tools, Pearson Publication.
[12] S. Rajasekaran, G. A. Vijayalakshmi Pai, Neural Networks, Fuzzy Logic and Genetic Algorithm: Synthesis and Application, PHI learning Pvt. Ltd.
[13] Sameer Kulkarni, John Cavazos, Mitigating the Compiler Optimization Phase Ordering Problem Using Machine Learning, 14th April 2012.
[14] J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith, "A mathematical language compiler," in Proc. 11th ACM Nat. Meeting, 1956, pp. 114–117.
[15] P. B. Sheridan, "The arithmetic translatorcompiler of the IBM FORTRAN automatic coding system," Commun. ACM, vol. 2, no. 2, pp. 9–21, 1959.
[16] M. D. McIlroy, "Macro instruction extensions of compiler languages," Commun. ACM, vol. 3, no. 4, pp. 214–220, 1960.