Parallel Processing Letters
© World Scientific Publishing Company

OPTIMAL AND NEAR–OPTIMAL SOLUTIONS FOR HARD COMPILATION PROBLEMS

ULRICH KREMER
Department of Computer Science, Rutgers University
Piscataway, New Jersey 08855, U.S.A.

ABSTRACT

An optimizing compiler typically uses multiple program representations at different levels of program and performance abstractions in order to be able to perform transformations that – at least in the majority of cases – will lead to an overall improvement in program performance. The complexities of the program and performance abstractions used to formulate compiler optimization problems have to match the complexities of the high–level programming model and of the underlying target system.

Scalable parallel systems typically have multi–level memory hierarchies and are able to exploit coarse–grain and fine–grain parallelism. Most likely, future systems will have even deeper memory hierarchies and more granularities of parallelism. As a result, future compiler optimizations will have to use more and more complex, multi–level computation and performance models in order to keep up with the complexities of their future target systems. Most of the optimization problems encountered in highly optimizing compilers are already NP–hard, and there is little hope that most newly encountered optimization formulations will not be at least NP–hard as well.

To face this "complexity crisis", new methods are needed to evaluate the benefits of a compiler optimization formulation. A crucial step in this evaluation process is to compute the optimal solution of the formulation. Using ad–hoc methods to compute optimal solutions to NP–complete problems may be prohibitively expensive. Recent improvements in mixed integer and 0–1 integer programming suggest that this technology may provide the key to efficient, optimal and near–optimal solutions to NP–complete compiler optimization problems. In fact, early results indicate that integer programming formulations may be efficient enough to be included in not only evaluation prototypes, but in production programming environments or even production compilers. This paper discusses the potential benefits of integer programming as a tool to deal with NP–complete compiler optimization formulations in compilers and programming environments.

Keywords: compilers, programming environments, hard optimization problems, 0–1 integer programming

1 Introduction

For high–performance scalar and parallel architectures, there is a gap between hardware and system features, and the ability of compilers and programming environments to take advantage of these features. As a result, effort and money spent in developing new system and hardware features may be wasted since the compiler is the main interface between the programmer and the underlying system architecture. In order to close this gap, new and more complex intermediate program representations and performance models are needed to identify when an optimizing transformation is legal and profitable.

In order to capture the complexity of the target system and to make the right tradeoff decisions, compiler optimization problems that have been considered in isolation in the past will have to be combined into a single problem formulation. The resulting compiler optimization problems will most likely be NP–complete. For example, instead of solving register allocation and instruction scheduling in sequence, a combined formulation of the problem avoids the "phase ordering problem" and allows the generation of better code. Another example is compiling for distributed–memory parallel architectures. There are tradeoff decisions between coarse–grain parallelism, fine–grain parallelism, and data locality. Different coarse–grain data and computation mappings may result in different degrees of data locality and instruction–level parallelism that can be exploited for each node processor.

In addition to combining different optimizations, whole program analysis may be required to provide sufficient opportunities for compiler optimizations. Therefore, interprocedural analysis and optimizations will play an important role in future compilers and programming environments.

Following the trend in architecture and system design, future systems will be more complex and therefore will pose even more challenges to compiler and programming environment designers. However, some hardware or system feature can make compilation simpler if they increase the performance predictability of the entire system. For instance, it is easier to predict the performance of a small fully set associative cache than that of a 4-way set associative cache since cache conflict misses are eliminated. If past and current trends in architecture design can be used as an indicator for future developments, the "unpredictable" features will most likely dominate the "predictable" features in future systems.

2 NP–Complete Compilation Problems

For the purpose of this discussion, a compilation problem consists of an intermediate program representation over which an optimization problem has been formulated. A classical example for an optimization problem is register allocation. The program representation is the interference graph and the optimization problem is to find a minimal coloring of the interference graph [1,2].

There are two orthogonal approaches to deal with NP–complete compilation problems. The model can be simplified, resulting in a simpler problem formulation that can be solved more efficiently, or heuristics can be used for the original, complex model with its corresponding NP–complete problem formulation. In other words, you can approximate the model itself by lowering its complexity, or you can approximate the optimal solution for the original model. Most approaches discussed in the literature use heuristics to determine potentially suboptimal solutions.

A compiler and programming environment designer confronted with a potentially NP–complete problem needs to know whether the chosen program representation is precise enough to model the tradeoffs for the desired optimization. Computing the optimal solution to an optimization problem allows:

1. Evaluation of the chosen model by using its optimal solution to verify its performance impact on the final, compiler generated code.
2. Evaluation of different heuristics for the chosen model through comparisons with the optimal solution.

Of course, the ideal situation for a compiler optimization would be an appropriate model with efficient optimal solutions. For some optimizations this ideal situation may be achievable, as shown in the case of a variant of the automatic data layout problem with dynamic remapping [13,14].

In the following section, integer programming is discussed as a tool to formulate and solve NP–complete compilation problems. Integer programming technology has the potential of being a tool to rapid prototype NP–complete compilation problems in order to understand their mutual interaction in an overall system. In addition, integer programming formulations may be efficient enough to provide optimal solutions or a family of heuristics.

3 Integer Programming

Integer programming has been used to solve many real world problems that require the management and efficient use of scarce resources to improve productivity. Examples of such problems are VLSI circuit design, airline crew scheduling, and communication and transportation network design. An instance of an integer programming problem consists of a set of variables, a set of inequality and equality constraints, and an objective function. A solution of the integer programming instance assigns integral values to all variables such that the objective function is maximized or minimized while all constraints are respected. If the integrality restriction is relaxed for some variables, the problem is called a *mixed integer programming problem.*

A *0–1 linear integer programming problem* is a special case of an integer programming problem where variables can only be assigned the integral values 0 or 1, and all constraints are linear functions of the variables. Solving a 0–1 integer programming problem has been shown to be NP–complete. An in-depth discussion of integer programming can be found in [3].

For decades, the integer and combinatorical optimization community has been working on methods to solve integer programming problems fast in practice. The ability to solve integer programming problems has been remarkably improved over the last five to ten years. The basic technique for solving integer programming problems is to apply intelligent branch–and–bound using linear programming at the nodes in the branch–and–bound tree. Important improvements have occurred in three areas. First, linear programming codes are on average approximately two orders of magnitude faster than they were five years ago, particularly for larger problems [4]. Combined with the improvements in computing speed over that same period these codes represent an approximate four to five orders of magnitude improvement in our ability to solve linear programming problems. Further algorithmic improvements are expected in linear programming, in preprocessing, and in branch-and-bound heuristics [5].

The second major development is in so-called cutting-plane technology. Motivated by work of Dantzig, Johnson and Fulkerson in the 50's [6], Padberg, Groetschel and others have shown how cutting-plane techniques could be used to strengthen the linear programming relaxations of many 0–1 integer programming problems [7].

The third major area of improvement has come in the application of parallel processing to handle the branching when cutting planes do not succeed in sufficiently strengthening the linear programming formulation. Parallelism is particularly appropriate for current cutting-plane methods because cuts are computed not only at the root node but at all nodes in the branching tree. The extra computation at the nodes has the effect of making the computations sufficiently coarse grained that communication costs need not be significant. Parallel versions of mixed–integer programming tools have been developed in academia [8] and in industry by Silicon Graphics Incorporation (SGI)[a].

3.1 Prototyping and Model Evaluation

Advanced, experimental compilation systems and programming environments including Stanford's SUIF compiler and the D95 system currently under development at Rice University use aggressive techniques such as integer programming (Omega test [9]) to solve dependence analysis or code generation problems. These aggressive techniques allow the evaluation of the entire system in a best case scenario where no information is lost due to ad–hoc heuristics. Once the optimization problems crucial for the overall performance have been identified through experimentation, some optimization problems may have to be reformulated if they are found to be ineffective. If effective, fast special–purpose techniques may be able to substitute more expensive general–purpose techniques for particular problem instances, allowing faster solutions in practice.

Therefore, integer programming is a promising candidate for "rapid prototyping" of NP–complete compilation problems. Instead of using an ad–hoc method to compute an optimal solution, integer programming formulations can rely on fast solution methods that have been developed in the combinatorial optimization community. In addition, using a common framework for NP–complete problems allows a general classification of the problems according to the characteristics of their integer programming formulations. Compiler and programming environment designers can use such a classification as a guideline to assess the solution complexities of new optimization problems.

In addition to evaluating models for compiler optimizations, optimal integer programming solutions can be used to assess the effectiveness of approximate solutions generated by heuristics. Work along this line has been done by Rutten-

[a] See http://www.sgi.com/Products/hardware/power/operations/perf.html

berg, Gao, Stoutchinin, and Lichtenstein in the context of software pipelining [10], and by Goodwin and Wilken for global register allocation [11]. Ruttenberg, Gao, Stoutchinin, and Lichtenstein showed that the heuristics used in a SGI production compiler are effective and close to optimal. However, they did not compute optimal solutions in all cases due to a time bound imposed on the solution times of their integer programming tool.

3.2 Practical Optimal Solutions and Solution Heuristics

Mixed integer programming and 0–1 integer programming is NP–complete. The main argument against using optimal techniques such as 0–1 integer programming in compilers and programming environments is the expected inefficiency. However, problem instances that occur in practice may not exhibit the worst case behavior. In addition, the notion of "efficiency" depends on the intended application scenario.

For a programming environment such as an automatic data layout tool for High Performance Fortran (HPF), a response time in the order of minutes may be considered acceptable [12,13,14]. For a compiler, an expensive technique can be an important tool if it is applied selectively, i.e., only in cases where the optimal solution is expected to result in a significant performance gain of the compiler generated code. For instance, optimal register allocation and instruction scheduling techniques could be used only for the most performance critical loops in a program.

Although using exact solutions for NP–complete problems is a rather new idea, a few researchers have already recognized the potential benefits of using 0–1 integer programming or general integer programming as part of a compiler or programming environment. Pugh developed a dependence analysis test, called the Omega Test based on an integer programming algorithm [9]. Using integer programming for instruction scheduling under resource constraints for super-scalar machines has been discussed by Feautrier [15], Ning, Govindarajan, Altman and Gao [16,17,18], and Ruttenberg, Gao, Stoutchinin, and Lichtenstein [10]. Goodwin and Wilken used 0–1 integer programming for optimal and near–optimal register allocation [11]. Integer programming techniques in the context of a distributed–memory compiler have been discussed by Phillipsen [19] and Garcia, Ayguadé and Labarta [20,21]. The latter two works have been based on the experience with 0–1 integer programming for efficient solutions of NP–complete problems in an automatic data layout tool as described by Bixby, Kennedy, and Kremer [12,13,14].

It is important to note that the actual formulation of the integer programming problem in terms of the chosen variables and constraints is crucial for the efficiency of the solution process. Typically, there are many possible integer programming formulations for a given compiler optimization problem. However, their solution times may be dramatically different, i.e., up to two orders of magnitude. It is not true that a smaller problem formulation will necessarily lead to faster solution times. A comparison of different 0–1 formulations for a NP–complete problem in the context of automatic layout can be found in [12]. Goodwin and Wilken discuss the impact of the choice of formulation in the context of global register allocation [11]. Another example from outside the compiler domain is described in [22].

In the cases where computing the optimal solution is considered too expensive, integer programming formulations can be used to create families of heuristics that compute near–optimal solutions. Possible heuristics include:

- Return the best feasible solution within a given amount of time. Goodwin and Wilken discuss first results based on this approach [11] and to some extent Ruttenberg et al. as well [10].
- Return the first feasible solution within a specific percentage of optimal.
- Run 0–1 formulation and a "conventional" heuristic in parallel. When the conventional heuristic terminates, compare result with the best feasible solution found for the 0–1 formulation. Return the best of the two solutions.

In summary, the formulation of a NP–complete optimization problem as an integer programming formulation has the advantage that it can provide optimal and near-optimal solutions based on given solution time limits. Once an efficient integer programming formulation has been found, its solution can directly benefit from future algorithmic improvements in integer programming technology without any further effort.

For example, solving a 0–1 integer programming instance with 14950 variables and 4663 constraints for an automatic data layout problem took 357 seconds on a Sparc–10 using *CPLEXv3.0*[b][14]. *CPLEX* is a state–of–the–art linear integer programming tool and library, partly developed by Robert Bixby at Rice University [23]. *CPLEX* includes an implementation of a general–purpose branch–and–bound code for mixed integer programming. Being general purpose, this code does not a priori exploit the structural properties of a particular 0–1 problem. The same problem instance can now be solved on an UltraSparc–1 in 29 seconds using the new version *CPLEXv4.0*. Approximately 50% of this improvement can be attributed to algorithmic improvements in the presolver step of the tool [24]. Choosing the *CPLEX* flag that selects a dual instead of the default primal simplex for the initial LP relaxation results in a further reduction of the solution time to 3.4 seconds. It is important to note that these dramatic speedups were achieved without modifications to the 0–1 integer programming problem instance.

4 Conclusion

Mixed–integer and 0–1 integer programming technology has improved dramatically over the last few years. We have reached a point where this technology can be used in optimizing compilers and advanced programming environments to formulate and solve NP–complete optimization problems. Mixed–integer programming formulations of NP–complete problems allow the evaluation of prototype systems and the generation of optimal or families of near–optimal solutions in production compilers and programming environments. The further assessment of the benefits of integer programming technology for compiler and environment designers will be an exciting future research field.

[b] *CPLEX* is a trademark of CPLEX Optimization, Inc.

5 References

1. G. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982.
2. P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.
3. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.
4. R. Bixby. Progress in linear programming. *ORSA Journal on Computing*, 6(1), 1994.
5. G. L. Nemhauser. The age of optimization: Solving large-scale real-world problems. *Operations Research*, 42(1):5–13, January–February 1994.
6. G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large scale traveling salesman problem. *Operations Research*, 7:58–66, 1954.
7. M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.
8. R. Bixby, W. Cook, A. Cox, and E. K. Lee. Computational experience with parallel mixed integer programming in a distributed environment. Technical Report CRPC-TR95-554, Center for Research on Parallel Computation, Rice University, June 1995.
9. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.
10. J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.
11. D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocation using 0–1 integer programming. *Software—Practice and Experience*, 26(8):929–965, August 1996.
12. R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0–1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, August 1994.
13. K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.
14. U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, October 1995. Available as CRPC-TR95-559-S.
15. P. Feautrier. Fine-grain scheduling under resource constraints. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.
16. Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.
17. R. Govindarajan E. R. Altman and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, December 1994.

18. E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

19. M. Philippsen. Automatic alignment of array data and processes to reduce communication time on DMPPs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

20. J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95)*, Houston, TX, April 1995.

21. J. Garcia. *Automatic Data Distribution for Massively Parallel Processors.* PhD thesis, Universitat Politècnica de Catalunya, Barcelona, April 1997.

22. C. Barnhart, E. L. Johnson, G. L. Nemhauser, B. Sigismondi, and P. Vance. Formulating a mixed integer programming problem to improve solvability. *Operations Research*, 41(6):1013–1019, November–December 1993.

23. R. Bixby. Implementing the Simplex method: The initial basis. *ORSA Journal on Computing*, 4(3), 1992.

24. Irv Lustig. Director of numerical optimization, CPLEX Optimization, Inc., private communication, August 1996.