An Empirical Study of Retargetable Compilers

Dmitry Boulytchev and Dmitry Lomov

{db,dsl}@tepkom.ru
St.-Petersburg State University, Faculty of Mathematics and Mechanics
Department of System Programming
198504, Russia, St.Petersburg, Bibliotechnaya sq., 2
Tel./Fax: +7(812)428-71-09

Abstract. The paper describes evaluation results of some modern retargetable codegeneration frameworks. The evaluation was performed to estimate applicability of these approaches in hardware-software codesign domain so ease of retargetability and efficiency of generated code were main criteria. Evaluated tools were selected from National Compiler Infrastructure (NCI) project.

1 Introduction

Hardware-software codesign is modern technique aimed to obtain high productivity of real-time and embedded systems. Key feature of this approach is simultaneous development of the program and the target processor or specialization of parameterized processor architecture to match target software application.

Generally, codesign implies iterative development. Each iteration consists of building new hardware description based on previous profiling and efficiency estimations, building (somehow) compiler, debugger, simulator, compiling and possible debugging target application, profiling and estimation of profit/loss. So building set of retargetable tools is basic and very frequent procedure.

Despite a number of retargetability techniques building of compiler still remains matter of art. Since main codegeneration approaches are investigated well the contiguous tasks (supporting of calling and linking conventions, building debugger and profiler etc.) should be solved (semi)–manually. The most crucial problem of building machine–dependent code optimizer also remains open.

Here we describe most recent retargetable codegeneration frameworks that look most preferable for purposes under considerations and briefly present the results of their evaluation (see [4] for details).

2 Retargetability Issues

Compiler's retargetability is usually understood as its ability to be re–targeted to another machine platform "automatically" or "nearly automatically". This implies building of codegenerator from some description. Ideally such a description should be extracted from description of actual hardware but as for now

there is well-known semantic gap between hardware description and codegenerator description. So now transition from hardware to codegenerator is mainly proceeds as follows: first verbal instruction set description is produced, then codegenerator description is written from it.

Starting from the most fundamental results in code generation area [1, 3] main retargetability technique stays tree pattern matching and dynamic programming. A number of ways to exploit this idea are investigated [6, 7, 13, 24, 25]; also there are a number of compilers based on them. These methods often considered as means of *instruction selection* so register allocation and instruction scheduling should be done separately.

Similar attribute-grammar based method described in [14]. Most of heuristic codegenerators use this notion.

Quite different approach suitable for VLIW processors codegeneration is suggested in [15, 20]. This approach is based on covering of so–called *split–node DAG* that reflects possibilities of parallel execution of DAG nodes with primitive instructions — so instruction selection, resigter allocation and scheduling are all performed simultaneously. To provide feasible schedule *binate covering* method is used [17, 20]. Unfortunally there is no compiler built on this technology so there is nothing to evaluate yet.

Finally there are some novel approaches to retargetable codegeneration including automatic building of codegenerator from architecture or instruction set description [19, 23, 31]. However tools presented there are either far from real industrial compilers or not accessible for evaluation.

3 Criteria and Methods

The basic factors to be taken into account are, of course, quality of generated code and ease of retargetability.

To assess quality of generated code, we compare the performance of several benchmarks on architectures that the tools being evaluated are already ported. We use Intel Pentium III and Sun SPARC processors for this purpose.

We used benchmarks developed by Standard Performance Evaluation Corporation (SPEC)[1]. This is an industry-standard set of benchmarks to assess quality of computer systems. However SPEC was not initially designed to be used as tool for compilers' performance comparison. For example it contains benchmarks written in different programming languages (Fortran, C++, C) and moreover utilized some specific compiler-dependent features. So we changed some SPEC benchmarks to make them appropriate for other compilers being evaluated.

Then it turned out that some of compilers were unable to compile some SPEC tests correctly either at whole or with some optimizations turned on. So we provide some auxilliary narrow set of benchmarks beyond basic SPEC set. These benchmarks are:

– *bzip2*: BWT-based data compression utility, by Julian Seward

[1] `http://www.spec.org`

- *gzip*: LZW-based data compressor, by Jean-Loup Gailly
- *ranking*: Implementation of Symbol Ranking text compression algorithm, by Dmitry Lomov

All of these benchmarks were compiled by all of evaluated tools with major optimizations turned on.

In according to reasons mentioned above we evaluated all tools in according with measures listed below:

- *Soundness*: describes how close evaluated tool is to real industry compiler. We express soundness in percents of all passed SPEC benchmarks
- *Selected Performance*: describes peak compiler performance. To evaluate selected performance we compared compilers on narrow set of benchmarks. We express selected performance using formula $K/absolute\ running\ time$, where K - some specially selected constant
- *Overall Performance*: describes performance evaluated on full SPEC suite. In addition we use non-retargetable platform-native compiler for comparison purposes. Overall performance expressed in percents of best performance among all tools

Informally speaking selected performance reflects some expectations about compiler's performance after all bugs eliminated. Note that this estimation is rather optimistic because fast code can probably be generated due to inaccurate analysis during optimizations.

To assess ease of retargetability, each tool evaluated has been ported to a "toy" instruction set, designed for a specific algorithm. Symbol Ranking was choosen as target algorithm. This estimation is also optimistic because it is much simpler to port compiler for special fixed application.

4 Evaluated tools

We selected compilers from *National Compiler Infrastructure (NCI)*[2] project. The project was started under support of DARPA and NSF by major USA Universities (Harvard, Princeton, Stanford, Rice etc.)

On the other hand we have chosen legendary `gcc` compiler [30] as most authoritative industrial optimizing C compiler.

NCI project is aimed at developing interoperable framework for constructing retargetable, optimizing compilers. Combination of these two qualities – *retargetability* and *optimization* – is crucial for hardware-software codesign. Without good retargetability, co-design cycle becomes unbearably long; without optimization, the whole idea of co-design is compromised, as non-optimizing compiler does not employ features of the target architecture to its best. NCI project compilers represent current state-of-the-art in developing easily retargetable, optimizing compilers.

Currently three C compilers are available from NCI: `SUIF/MachSUIF`, `lcc` and `VPO`-based compiler. We evaluated all of them.

[2] http://www.cs.virginia.edu/nci/

SUIF and MachSUIF. `SUIF` *(Stanford University Intermediate Format)* [18] and `MachSUIF` *(Machine SUIF)* [?,29] are developed in Stanford and Harvard Universities correspondingly. Both systems are parts of NCI project. Unfortunately SUIF/MachSUIF compiler is not ported to Sun SPARC so it is not evaluated at that platform.

VPO-based compiler. `VPO` *(Very Portable Optimizer)* is a part of Zephyr[3] project. The project is in turn part of NCI.

lcc compiler. `lcc` compiler was developed in Princeton University, USA, since 1991 and later was also involved into NCI project [9–12].

5 Results and Conclusions

Unfortunately at the time of writing on Sun SPARC platform only selected performance evaluation was completed. The result of the evaluation is shown at figure 1. The other results are to appear at http://oops.tepkom.ru/eval.html in near future.

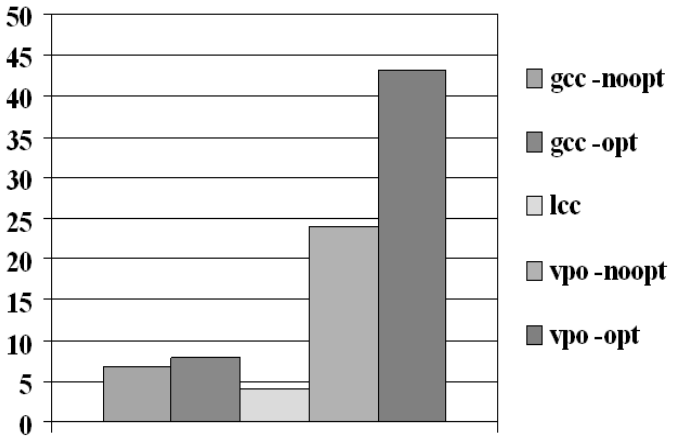

Fig. 1. Selected Performance on Sun SPARC, 1000/absolute time

Results of soudness evaluation on Intel Pentium III platform are shown at figure 2. We can conclude that neither `SUIF` nor `VPO` turned out to be ready-to-use compilers — during the evaluation we encontered lots of bugs that had to be fixed.

Selected and overall performance evaluation results at Intel Pentuim III are shown at figure 3 and figure 4 correspondingly. We have choosen Intel C/C++ compiler (`icc`) as non-retargetable platform-native compiler.

[3] http://www.cs.virginia.edu/zephyr

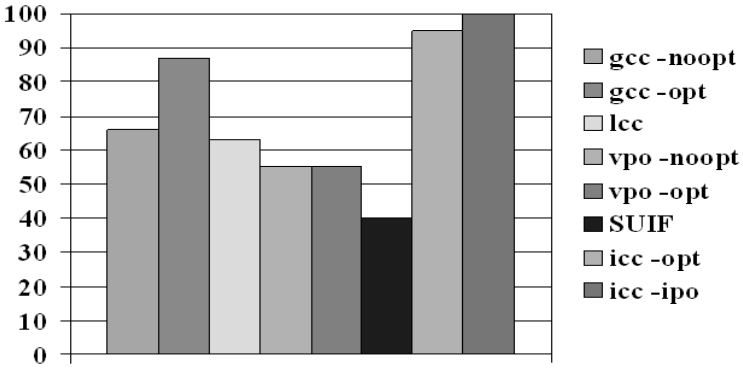

Fig. 2. Soundness on Intel Pentium III, % of passed benchmarks

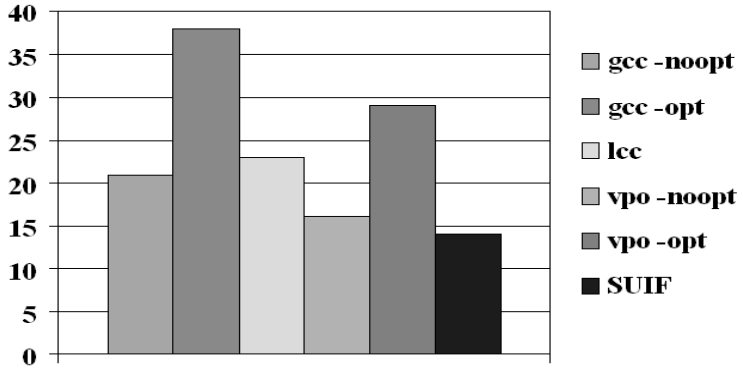

Fig. 3. Selected Performance on Intel Pentium III, 1000/absolute time

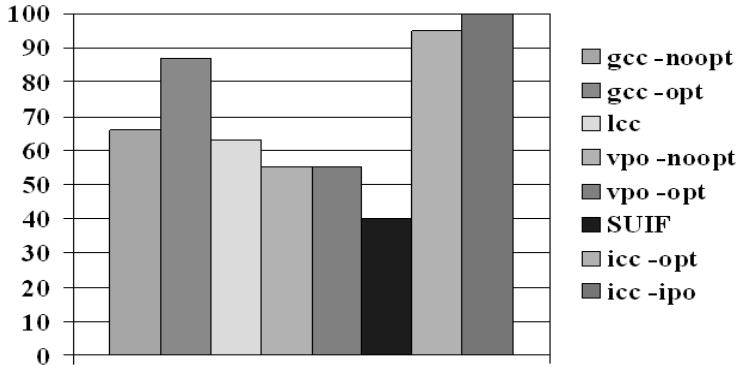

Fig. 4. Overall Performance on Intel Pentium III, % of best performance

Our benchmarks show that SUIF/MachSUIF compiler is competely unapplicable for producing efficient code. This is largerly due to inappropriate instruction selection techniques and lack of optimizations.

Regarding the efficiency of generated code, we saw that generally gcc with optimizations on beats all the other retargetable tools. If optimizations are turned off in all tools, lcc shows best performance. VPO has shown quite irregular performance — on some benchmarks it produces the best code of all, while on others it lose even to non-optimizing lcc compiler.

However as a result of auxilliary testing we discovered "contradictionary" benchmarks that are not fit into conclusion given above:

1. lcc beats all retargetable tools on Objective Caml [4] garbage collector implementation (30% better than gcc) on Intel Pentium III
2. VPO beats all retargetable tools on certain implementation of Symbol Ranking text compression algorithm (5 *times* better than gcc) on Sun SPARC

Finally we can see that platform-specific Intel compiler outperforms all retargetable tools.

As the ease of retargeting, lcc turned out to be the best of all considered tools. gcc and VPO on the whole show same level of retargetability, although gcc is much better documented. SUIF/MachSUIF is less retargetable because it is necessary to rewrite codegenerator manually to retarget it.

We conclude that none of the methods considered allows to build a retargetable code generator that can directly be utilized for co-design purposes.

We also see the importance of instruction selection — lcc, a non-optimizing compiler with good instruction selection algorithm based on BURS [3, 7, 13, 24, 25] shows quite good performance.

However, good instruction selection is not enough for obtaining optimized code. VPO outperforms lcc on majority of tests.

This research shows the directions for further development in co–design and code generation area. Easily retargetable, optimizing compilers are vital for hardware-software co-design, but we see that techniques for building them are yet to be created.

Acknowledgments

We would like to thank Mikhail Smirnov and Eugene Vigdorchik — our colleagues in OOPS team of System Programming Department in Saint Petersburg State Unievrsity — for the invaluable discussions and assistance in obtaining the results presented in this paper.

References

1. Alfred V.Aho, S.C.Johnson. Optimal Code Generation for Expression Trees. Journal of the ACM, Vol. 23, No. 3, July 1976, pp. 488–501

[4] http://caml.inria.fr/index-eng.html

2. Alfred V.Aho, Ravi Sethi. Compilers: Principles, Techniques and Tools. Addison-Wesley Pub Co., Nov. 1985
3. Alfred V.Aho, Steven W.K.Tjiang. Code Generation Using Tree Matching and Dynamic Programming. ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, Oct. 1989, pp. 491–516
4. Dmitry Boulytchev, Eugene Vigdorchik, Dmitry Lomov, Mikhail Smirnov. Retargetable Tools for Efficient Code Generation, Technical Report, St.Petersburg State University, January 2001, http://oops.tepkom.ru/eval.html
5. Hubert Comon, Max Dauchet et al. Tree Automata Techniques and Applications. http://l3ux02.univ-lille3.fr/~tommasi/TATAHTML/main.html
6. H. Emmelmann, F.W.Schröer, R.Landwehr. BEG — a Generator for Efficient Back Ends. Proceedings of the SIGPLAN'89 Conference on Programming Languages Design and Implementation, 1989, pp. 227–237
7. M. Anton Ertl. Optimal Code Selection in DAGs. Proceedings of the 26th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 1999, pp. 242–249
8. Jack W. Davidson, Steve G.Losen, Norman Ramsey. VPO Code-Generation Interfaces. Department of Computer Sciences University of Virginia, 1998, http://www.cs.virginia.edu/zephyr/vpoi
9. Christopher W.Fraser, David R.Hanson. A Retargetable C Compiler: Design and Implementation. Addison-Wesley Pub Co., Jan. 1995
10. Christopher W.Fraser, David R.Hanson. A Retargetable Compiler for ANSI C. ACM SIGPLAN Notices, Vol. 26, No. 10, Oct. 1991, pp. 29–43
11. Christopher W.Fraser, David R.Hanson. A Code Generation Interface for ANSI C. Software — Practice and Experience Vol. 21, No. 9, Sept. 1991, pp. 963–988
12. Christopher W.Fraser, David R.Hanson. Simple register spilling in retargetable compiler. Software — Practice and Experience, Vol. 22, No. 1, Jan. 1992, pp. 85–99
13. Christopher W.Fraser, David R.Hanson, Todd A.Proebsting. Engeneering a simple, efficient code generator generator. ACM Letters on Programming Languages and Systems, Vol. 1, No. 3, Sep. 1992, pp. 213–226
14. Mahadevan Ganapathi, Charles N. Fischer. Affix grammar driven code generation. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, Oct. 1985, pp. 560–599
15. Silvina Hanono, Srinivas Devadas. Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator. Proceedings of the 35th ACM/IEEE Annual Conference on Design Automation, 1998, pp. 510–515
16. David R. Hanson. Early Expirience with ASDL in lcc. http://www.cs.princeton.edu/software/lcc/doc
17. Seh-Woong Jeong, Fabio Somenzi. A New Algorithm for the Binate Covering Problem and Its Application to the Minimization of Boolean Relations. Proceedings of the 1992 IEEE/ACM International Conference on Computer–Aided Design, 1992, pp. 417–420
18. Monika Lam et al., An Overview of the SUIF2 Compiler Infrastructure. Computer Systems Laboratory, Stanford University, 2000, http://suif.stanford.edu/suif/suif2
19. Rainer Leupers, Peter Marwedel. Retargetable Generation of Code Selectors from HDL Processor Models. Proceedings of the 1997 European Design and Test Conference
20. Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. Proceedings of 1995 IEEE/ACM International Conference on Computer–Aided Design, 1995, pp. 393–399

21. Robert Morgan. Building an optimizaing Compiler. Digital Press, Feb. 1998
22. Steven Muchnik. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, July 1997
23. Carsten Müllr. Code Selection from Directed Acyclic Graphs in the Context of Domain Specific Digital Signal Processors. Technical Report, Humboldt-Universität zu Berlin, August 10, 1994
24. Eduardo Pelegrí-Llopart, Susan L.Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. Proceedings of the conference on Principles of programming languages, 1988, 294–308
25. Todd A.Proebsting. BURS Automata Generation. ACM Transactions on Programming Languages and Systems, Vol. 17, No. 3, May 1995, pp. 461–486
26. Norman Ramsey, Mary F. Fernandez. Specifying Representation of Machine Instructions. ACM Transactions on Programming Languages and Systems. Vol. 19, No. 3, May 1997, pp. 492–524
27. Norman Ramsey, Jack W. Davidson. Specifying Instructions' Semantics Using λ-RTL (Interim Report). University of Virginia, July 11, 1999, http://www.cs.virginia/edu/zephyr/csdl/lrtlindex.html
28. Michael D. Smith, Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries and Optimizations. Division of Engineering and Applied Sciences, Harvard University, 2000, http://www.eecs.harvard.edu/hube/research/machsuif.html
29. Michael D. Smith, Glenn Holloway. A User's Guide to the Optimization Programming Interfaces. Division of Engeneering and Applied Sciences, Harvard University, 2000, http://www.eecs.harvard.edu/hube/research/machsuif.html
30. Using and Porting GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc_toc.html
31. Bert-Steffen Visser. A Framework for Retargetable Code Generation Using Simulated Annealing. Proceedings of the 25th Euromicro Conference, 1999