Profile Directed Compilation in DSP Applications

Sanjay Jinturkar[1], Jesse Thilo[1,2], John Glossner[1,2], Dean Batten[1], Paul Darcy[1], Stamatis Vassiliadis [2]

{sjinturkar, jthilo} @lucent.com

[1]Bell Laboratories, Lucent Technologies, 1247 S. Cedar Crest Blvd., Allentown, PA 18103.

[2]Delft University of Technology, Delft, The Netherlands

ABSTRACT

Optimizing compilers play an important role in producing cycle efficient machine code for application programs. These compilers apply a number of very aggressive optimizations. Examples of such optimizations include common subexpression elimination, register allocation, strength reduction, induction variable elimination, loop unrolling, software pipelining, etc. To improve the effectiveness of these optimizations, compilers may use profile information. In this paper, we discuss the effectiveness of profile directed feedback compilation in the Global System for Mobile Communications (GSM) Enhanced Full Rate (EFR) speech coder. Our experiments indicate that the performance improvement for the EFR encoder is approximately 88% when profile information is used to apply standard optimizations along with function inlining. The performance improvement for the decoder is 72%. Performance degradation is less than five percent when profile-based code is executed on inputs not characterized by the original training set.

1 INTRODUCTION

Traditional compiler optimizations consider the constraints imposed by all possible execution paths in a program. These constraints are determined by static analysis techniques such as live variable analysis, dependence analysis, loop analysis, etc. However, these techniques do not distinguish between more and less frequently executed regions of a program. While this approach generates correct code, it may not generate the most efficient code.

To improve efficiency, a compiler may use profile information. In the profile directed feedback compilation methodology, compilation is performed in two steps. First, the compiler applies standard optimizations [Aho86] and instruments the code. The processor then executes the instrumented code on a selected vector (*training vector*). During this process, the profiler collects statistics regarding the execution of the program. These statistics include information about the most frequently executed basic blocks, taken or not taken branches, cache misses, function calls, etc. In the second step, the profile information is fed back into the compiler, and a re-compilation of the application code is performed. The compiler optimizations may now be applied more effectively since the most frequently executed paths through the application are known. The profile information may guide better placement of the object code to avoid cache misses, allows better allocation of variables to registers, and provides better branch prediction. This newly created executable is expected to execute faster on the training vector, since it is optimized for the execution path. Figure 1 shows the steps taken during profile directed compilation.

While profile directed compilation on a training vector is an improvement, it is not sufficient. A better improvement would be if execution of the program provides equally good performance on a variety of other input vectors. The profile information can then be used in a generic fashion to optimize any application.

Typically, whether profile information holds across different input vectors is very application dependent. In general purpose computing workloads, profile information collected on one set of inputs may not aid in producing an executable that runs faster on another set of inputs [Wall91].

2 APPLICATION

In this paper, we focus on a Digital Signal Processing (DSP) application. DSP applications tend to be moderately loop intensive. A basic set of operations are performed repeatedly on a set of inputs. Therefore, they are amenable to compiler optimizations.

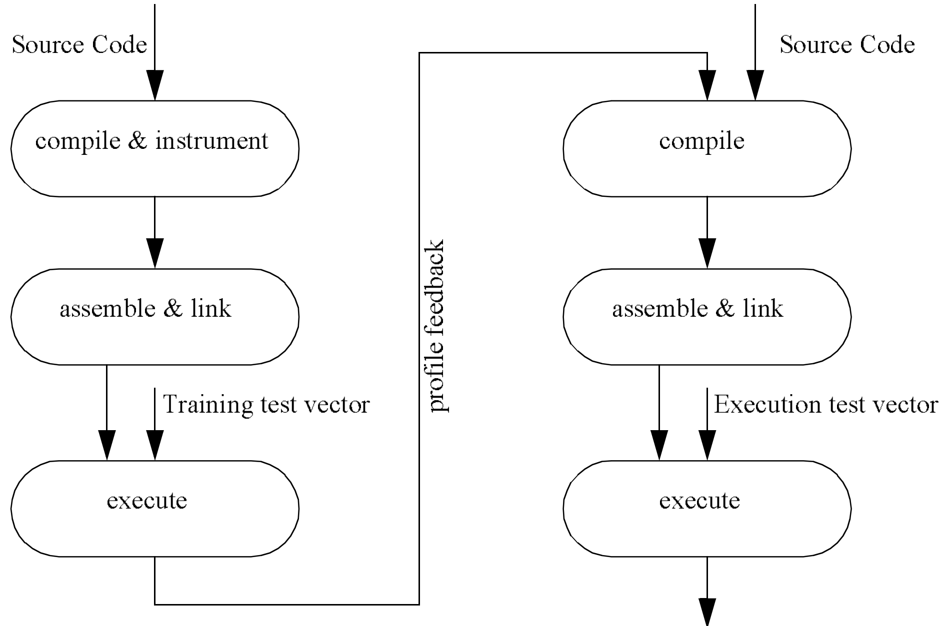

Figure 1: Profile Directed Compilation. In the first step, the application code is compiled with instrumentation, assembled, linked and executed. The execution characteristics are fed back into the compiler. The code is recompiled with profile information and then re-executed.

We focus on the GSM Enhanced Full Rate (EFR) speech coder that has been standardized by the European Telecommunications Standards Institute (ETSI) for the GSM mobile communication system. The ETSI GSM EFR speech transcoder is a specification for encoding and decoding speech signals [ETSI97a, ETSI97b] for wireless applications. The speech encoder first converts 160, 8-bit/A-law input samples to 160, 13 bit uniform Pulse Code Modulated (PCM) samples (20ms of speech data at 8kHz sampling rate). The input PCM speech data frame is encoded using Algebraic Code Excited Linear Prediction (ACELP) algorithm which converts the 160 sample input frame to a 244 bit compressed speech frame. The compressed speech data transmission rate is 12.2 kb/s (244 bits X 50 frames/s). The compressed speech frame is cyclically encoded (8 parity + 8 repetition bits) and the subsequent 260-bit frame is passed to the channel coding function. The channel coder performs convolutional encoding (constraint length 5, rate 1/2). The output from the channel coder consists of 456 bits yielding a gross transmit bit rate of 22.8 kbits/s. The receive direction provides the inverse operations. After successful channel decoding, encoded frames of 244 bits are converted to 160 reconstructed speech samples.

3 EXPERIMENTAL FRAMEWORK

In our experiments, we have used a research C compiler which uses profile information to generate efficient code. The target processor is an in-order RISC superscalar processor with a 4-issue peak capability. The compiler performs a number of state of the art optimizations which give execution times comparable (or better) than well known existing optimizing compilers such as the GNU C compiler [Stal93].

We define a *training test vector* as an input vector to the instrumented application code for the collection of profile information. We define an *execution test vector* as an input vector to the application code for final execution. This may or may not be the same as the training test vector.

For this paper, the experiments were conducted on the EFR encoder and decoder using the 31 test cases supplied with the ETSI GSM EFR reference implementation. All optimized performance results maintain the bit accuracy required by the ETSI C reference implementation [ETSI97c]. Timings were collected using the UNIX */bin/time* program. Each test case was run three times and an average of the three *real* (elapsed) times has been reported. The execution time for the encoder on these test cases is in the range of 5-20 seconds, while the execution time for the decoder is in the range of 0.1-2 seconds. Since the decoder is executed for a very short duration of time, the margin of error (due to the limited precision of the */bin/time* utility) in its observed times is higher than the encoder.

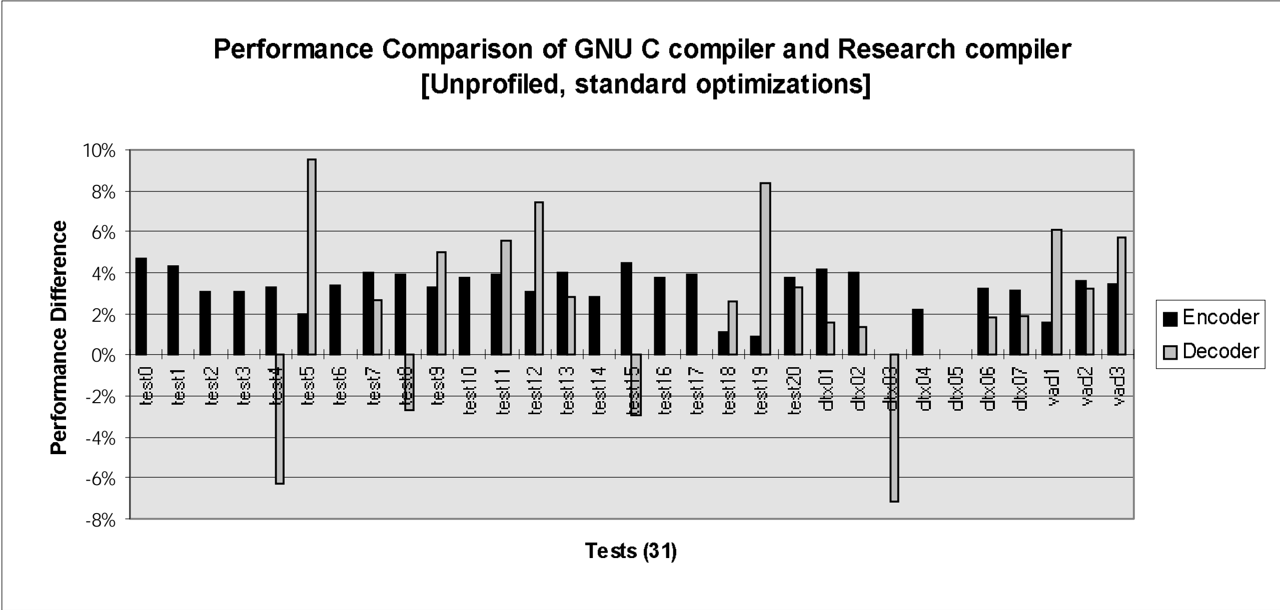

Figure 2: Performance comparison of GNU C Compiler with our Research Compiler indicates that the GNU C compiler performs 3% better than the research compiler for the encoder and 2% better for the decoder.

4 RESULTS

We perform three experiments. First, we compare the performance of the GNU C [Stal93] compiler with our research compiler without the use of profile information.This experiment is performed to show that without profile information, the performance of our research compiler is comparable to the performance of a widely used optimizing compiler. Figure 2 shows that on an average, the performance of GNU C compiler at optimization level -O3 is three percent better than the performance of our research compiler for the encoder and two percent better for the decoder.

In the second experiment, we compare the performance of our research compiler with and without profile information. In this experiment we compile the encoder and decoder, execute it on the training test vector, and collect profile information. This profile information is fed back into the compiler. Using the profile information, the encoder and decoder are re-compiled. In addition to the optimizations applied in the unprofiled case, function inlining is also applied [Baco94]. The code is then re-executed on the training test vector (i.e. execution test vector is the same as training test vector) and timing information is collected. This process is repeated for each of the 31 test cases. The observed timings are compared with the timings observed in the unprofiled case.

Figure 3 shows the result of the comparison. This experiment indicates that when profile information is used and function inlining is applied, the performance increases by about 88% for the encoder and 73% for the decoder.

The third experiment measures the performance degradation when the training test vector and the execution test vector are different. This experiment is similar to the second one except that the profile information is collected for only one training test vector namely *vad3* (*vad3* is chosen because the encoder and the decoder have one of the longest execution times on it). This profile information is fed back into the compiler. Using the profile information, the encoder and decoder are recompiled. Optimized code is then executed on the *each* of the 31 execution test vectors and timing information is collected (i.e. the execution test vector is different than the training test vector in all but *vad3*). These timings are compared to the timings observed in the second experiment, where the encoder and decoder were re-executed on the training test vector itself.

Figure 4 shows the results of this comparison for the encoder. The figure indicates that on average, the performance of the encoder with same training and execution test vectors differs from the performance of the encoder with different training and execution test vectors by less than two percent. The figure also shows

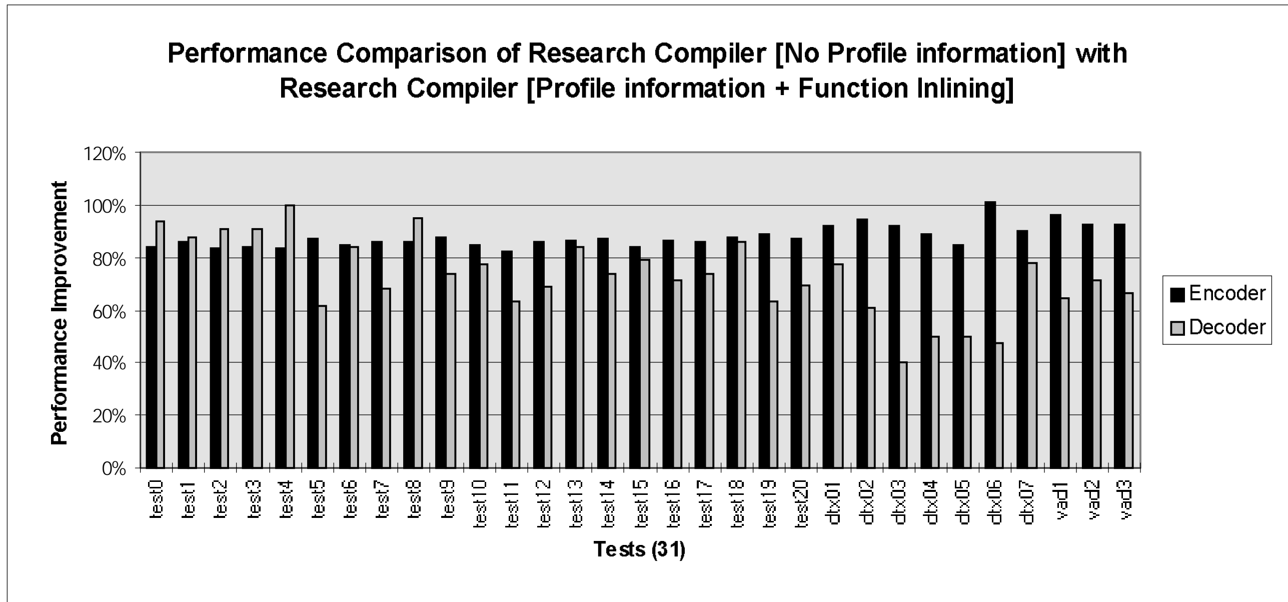

Figure 3 Performance comparison of our Research Compiler on the GSM EFR coder with and without profile information. Results indicate that the performance of the encoder improves by 88% and the performance of the decoder improves by 72%.

the upper and lower bounds of the comparison when a potential error introduced by the */bin/time* utility is taken into account. Figure 5 shows the results for the decoder. The performance difference for the decoder is less than five percent. In both figures, the results for test *dtx05* show a large variation since the short execution times of the encoder and the decoder increases the range of potential error.

We conducted the third experiment with a number of different training sets. We observed that the results of the above experiments remain approximately the same.

5 CONCLUSIONS

Our results indicate that the use of profile directed feedback is of benefit in compiling the GSM EFR speech coder. Our results also show that the profile information collected using a training test vector which is different than the execution test vector is useful in optimizing the GSM EFR coder.

6 BIBLIOGRAPHY

[Aho86] Aho, A., Sethi, R., Ullman, J. “Compilers: Principles, Techniques, and Tools”, Addison-Wesley Publishing Company, CA.

[Baco94] Bacon, D. F., Graham, S. L., and Sharp, O. J., “Compiler Transformations for High-Performance Computing”, ACM Computing Surveys, 26(4), Dec. 1994, pp. 345-420.

[ETSI97a] European Telecommunication Standard (ETSI), "GSM: Digital cellular telecom. system; Enhanced Full Rate (EFR) speech processing functions; General Description", Document GSM 06.51, March, 1997, ETSI,Valbonne, France, pp. 1-13.

[ETSI97b] European Telecommunication Standard (ETSI), "GSM: Digital cellular telecom. system; Enhanced Full Rate (EFR) speech transcoding" Document GSM 06.60, March, 1997, ETSI, Valbonne, France, pp. 1-50.

[ETSI97c] European Telecommunication Standard (ETSI), "GSM: Digital cellular telecom. system; ANSI-C code for the GSM Enhanced Full Rate (EFR) speech codec", Document GSM 06.53, March, 1997, ETSI, Valbonne, France, pp. 1-50.

[Hwu93] Hwu, W., “Super block: An effective technique for VLIW and superscalar compilation”, *Journal of Supersomputing*, Vol 7, pp. 229-248.

[Stal93] Stallman, R., “Using and Porting GNU C”, Free Software Foundation, MA.

[Wall91] Wall, D., “Predicting Program Behavior Using Real or Estimated Profiles”, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, pp. 59-71.

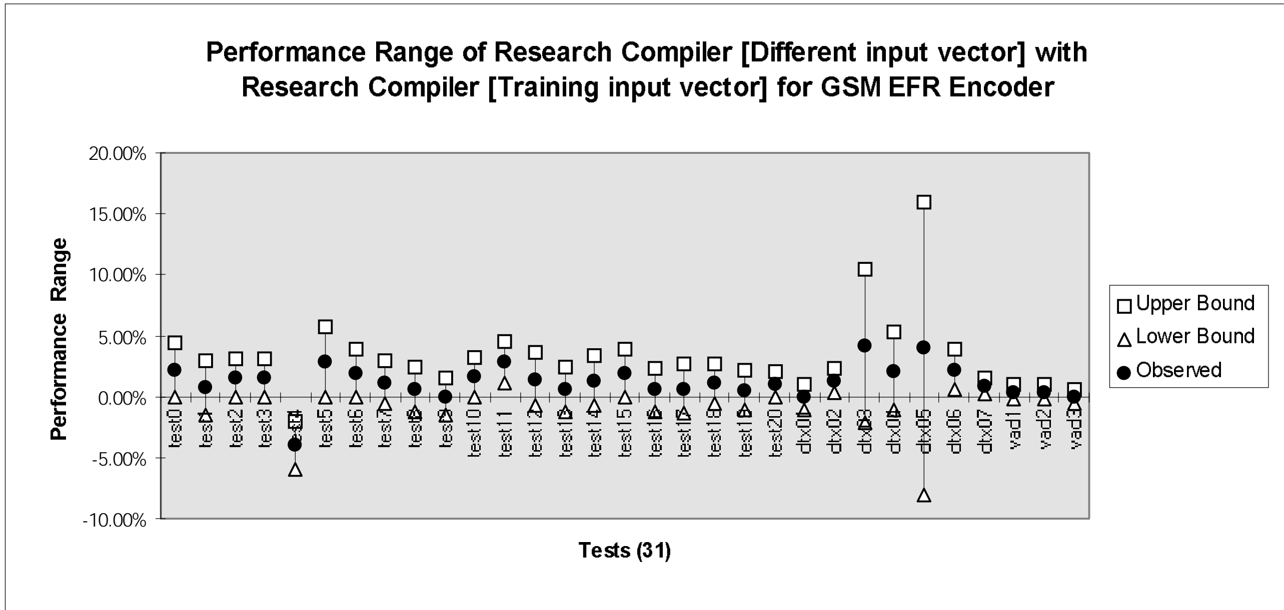

Figure 4 Performance comparison of our Research Compiler when the GSM EFR encoder is executed with same training test vector and execution test vector vs. when executed with a training test vector different than execution test vector. Results indicate that on average, the observed variation is less than 2% when the training test vector differs from the execution test vector for the encoder.

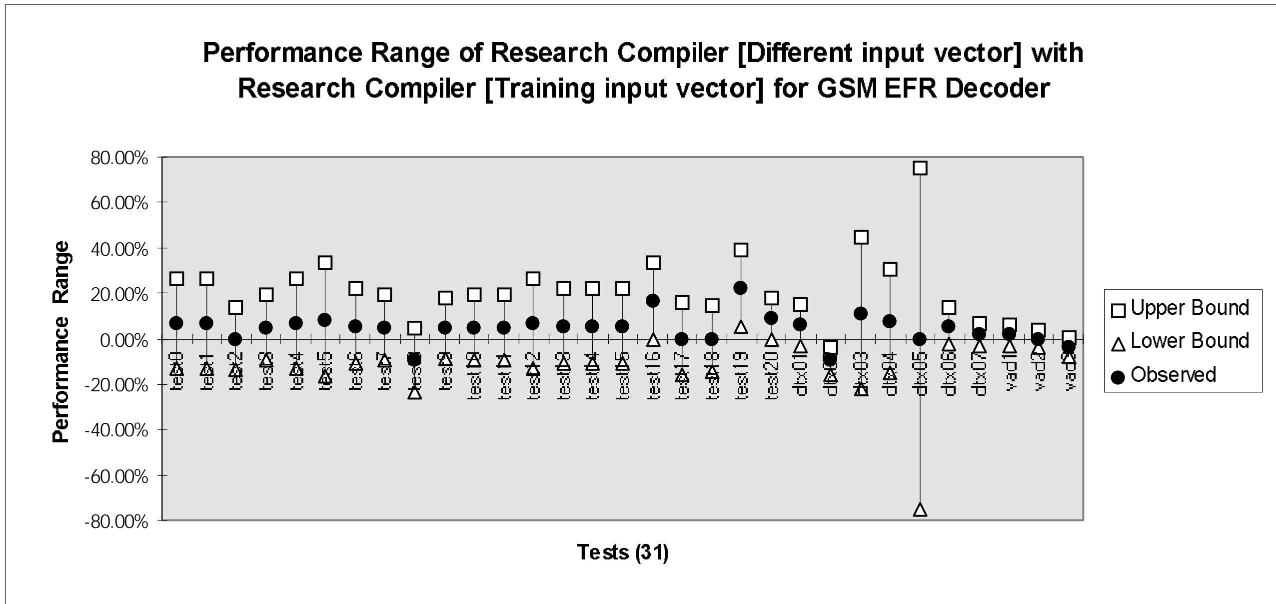

Figure 5 Performance comparison of our Research Compiler when the GSM EFR decoder is executed with same training test vector and execution test vector vs. when executed with a training test vector different than execution test vector. Results indicate that on average, the observed variation is less than 5% when the training test vector differs from the execution test vector for the decoder.