ON THE USE OF COMPILERS IN DSP LABORATORY INSTRUCTION

Matthew D. Kleffner[1], *Douglas L. Jones*[1], *Jason D. Hiser*[2], *Prasad Kulkarni*[3],

Julie Parent[2], *Stephen Hines*[3], *David Whalley*[3], *Jack W. Davidson*[2], *Kyle Gallivan*[3]

ABSTRACT

A modern DSP laboratory course should teach students how to quickly develop efficient applications using a mixture of C and assembly instructions. Since typical irregular DSP microprocessor architectures present some challenges to traditional optimizing compilers, we introduced a novel new interactive optimizing compiler, VISTA, in a DSP laboratory course to teach students about optimizing compilers and the trade-offs between code development in C and assembly. Students generally found VISTA educational and gained improved insight into effective DSP software development.

1. INTRODUCTION

A DSP laboratory course should be structured around broadening and deepening students' understanding of DSP theory and its application in the real world. The course lectures and labs should also give students in-depth knowledge of and experience in how real-time, real-world DSP systems are implemented in industrial practice. A lab course followed by an extensive project of the student's choice is an effective way to accomplish these goals [1].

Traditionally, DSP microprocessors have been programmed almost exclusively in assembly language, due to the stringent performance demands and power constraints of most DSP applications. Any course that aspires to teach real-time DSP system engineering should thus include substantial assembly-language programming. However, recent industrial trends (including the introduction of C compilers with reasonable performance for some DSP microprocessors, increased application size and complexity, and increases in the complexity and performance of DSP microprocessors), makes a mixture of assembly and C attractive in an increasing number of situations. The fastest, smallest, and lowest-power system implementations are still generated in hand-coded assembly, but the resulting code is typically difficult to port to other architectures. Furthermore, real-world, hand-coded assembly applications are time-consuming to develop and maintain. While these costs are often more than offset by the resulting savings from lower-cost DSPs in mass production, this is not always the case.

Programming a real-world DSP system in C is less time-consuming, but the resulting code is usually much slower than hand-coded assembly. For these reasons, such systems are now often developed in C, then resource-critical routines are rewritten in hand-coded assembly. We expect that this trend will continue and even intensify. Therefore, a DSP laboratory course should introduce both C and assembly language programming in an integrated manner.

A DSP laboratory course must balance many educational objectives, so each element must be carefully designed so that all objectives are met effectively while making efficient use of students' limited time. We hypothesized that a novel new interactive optimizing compiler with an intuitive graphical interface, called VISTA, would be an effective tool for teaching students both the trade-offs between assembly- and C-based DSP software implementation and the use of optimizing compilers.

2. COMPILERS FOR DSP

DSP microprocessors and applications present some unique difficulties for traditional C compilers, and compiling entire DSP applications in C generally results in very inefficient code relative to the best hand-coded assembly. The C specification conflicts with typical, irregular DSP architectures; therefore some key DSP operations, such as fractional multiplication and saturation on overflow, are not efficiently achievable in C. Also, the C specification interferes with common architectures that have accumulators larger than the memory word width. A compiler may add extra instructions to ensure accumulator results are bit exact to the C specification, but these extra instructions are usually not necessary to achieve the results the developer desires. In some cases it may be impossible to utilize the extra bits available in the accumulator via standard C code. Finally,

[1]University of Illinois at Urbana-Champaign, Electrical and Computer Engineering, Urbana, Illinois, USA

[2]University of Virginia, Department of Computer Science, Charlottesville, Virginia, USA

[3]Florida State University, Department of Computer Science, Tallahassee, Florida, USA

This work was supported in part by the National Science Foundation, Grant no. EIA-0072043.

modern DSP microprocessors support an ever-increasing number of special instructions tailored to specific DSP applications, such as bit-reversing operations for fast Fourier transforms. Some DSP compilers introduce work-arounds to the limits of the C specification through compiler intrinsics (non-standard functions that result in architecture-specific assembly code) for selected operations and inlining assembly directly in C code [5]. Either one of these work-arounds, however, limits the portability of the code and increases the difficulty of software development.

Given the limitations of compiled code for DSP microprocessors, it is becoming common practice to develop the full algorithm in C and then rewrite the most performance-critical routines in assembly; the resulting tight coupling between the C and assembly portions makes a mixed programming environment attractive. Since C and assembly are mixed in real-world DSP applications, a visual, interactive optimizing compiler that further unifies the two languages might allow a DSP software developer to produce better code. An added bonus of such a tool is that DSP engineers and particularly students, who may not know much about compilers, can learn interactively about how optimizing compilers obtain their results.

A recent research project has produced an experimental optimizing compiler, VPO Interactive System for Tuning Applications (VISTA), with these characteristics. VISTA has an optimization engine that is based on VPO, the Very Portable Optimizer developed at the University of Virginia [3] [4]. A key advantage of using VPO is it is easily retargeted to new architectures, therefore enabling VISTA to be easily retargeted. VISTA is designed to enable a developer to obtain efficient assembly code from C that is efficiency-competitive with hand-coded assembly. This is possible in part because the developer tunes the application at a low level, in either assembly or VPO's register transfer lists (RTLs). Using VISTA to tune the low-level output of the compiler instead of hand-coding assembly, the developer may get code that is more portable, more robust, and cheaper to develop and maintain than hand-coded assembly. Tuning in VISTA is primarily done through ordering canned optimization phases and manually specifying code transformations. Porting to another architecture merely involves recompiling and retuning in VISTA.

A key advantage VISTA possesses over traditional optimizing compilers is that optimization phases can be specified in different orders, whereas traditional compilers apply optimization in a fixed order. Furthermore, traditional compilers group many optimization phases together into optimization levels, so if one optimization phase causes buggy code, the other phases in that level cannot be used. With VISTA, the performance in cycles and instruction count of an optimization phase is fed back to the developer, which allows the developer to remove buggy or low-performing phases and try others.

VISTA also allows the developer to manually specify code transformations, which enables the developer to more accurately tell the compiler what is desired. These transformations include changing and adding instructions, as well as removing instructions that the compiler sees as necessary for accurate code but the developer's higher-level knowledge indicates is not necessary for the application to properly function. These manual optimizations can lead to better results from subsequent VISTA optimization phases.

VISTA's graphical interface also allows the developer to visualize the control flow of the code and immediately inspect the results of applied optimizations. See [8] for VISTA screenshots. The main VISTA GUI is split into two halves. The left side, when in display mode, displays optimization phases and the resulting reduction in code size and execution time. When in phase entry mode, the left side contains buttons to select optimization phases. Towards the bottom of the left side are navigational buttons that allow the developer to step through each transformation that has been applied. The right side displays the low-level code listing within blocks, with arrows indicating the overall flow and loop structure. This code listing is altered by the navigational buttons and it indicates the current state of the code.

Once the developer has entered some optimization phases, the transformations are listed in the left side along with performance results. The developer can then step through each applied transformation in two steps. Before the transformation is applied the relevant instructions are highlighted in yellow. When the developer steps again, the end of the transformation is reached; the modified, added and/or deleted instructions are highlighted in red.

VISTA's optimization feedback and its low-level indication of the implications of C statements gives the developer or student valuable information on how to rewrite C code and tune it. With VISTA's feedback a student can also gain in-depth knowledge of what optimizing compilers do and how they do it. Also, through entering phase sequences and hand transformations the developer can give the optimizer higher-level knowledge of the application that would otherwise be unavailable.

3. OPTIMIZING COMPILERS IN A DSP LABORATORY

In order to teach the modern DSP development process to students at the University of Illinois, a few years ago we introduced a traditional optimizing compiler in the ECE 420: DSP Laboratory course [6]. The students learn the tradeoffs, especially development and execution time, between code developed in C and assembly. Another educational objective is to teach students a bit about how compilers work so that they can use them more effectively. Students also

learn that the following development process is useful in minimizing development time and maximizing application performance [7]:

1. Develop algorithm on paper
2. Simulate in MATLAB
3. Develop and simulate more efficient implementations
4. Implement algorithm in C
5. Use library routines when available
6. Use optimizing compiler
7. Manually write assembly routines for key routines

Since we utilize TI's TMS320C54X series DSPs, we introduce TI's optimizing C compiler in the course. The first four labs are devoted to familiarizing the students with the hardware, assembly language, and real-time implementation of basic DSP concepts such as FIR and IIR filtering and multirate signal processing [6]. Assembly language is introduced first because it better enables the students to focus on real-time issues and intimately learning DSP architectures. Therefore, we introduce the compiler in the fifth of six labs. In this lab the students learn and experience the trade-offs of C versus assembly by implementing key components of an FFT-based spectrum analyzer in both C and assembly. They also compile and compare efficiencies of two versions of the FFT, one a library routine in hand-coded assembly and the other an FFT routine written in C.

In the sixth lab the students are introduced to the DSP optimization process, which they apply to optimizing a DSP application [7] [8]. This application calculates power spectral density estimates, via calculating some autocorrelation points and computing the FFT, of a pseudo-noise (PN) sequence that has been colored by a feedback-only IIR filter. They are given a reference implementation of the application in C, except for a hand-coded assembly FFT, and a MATLAB implementation. They are asked to optimize, for speed only, the PN-sequence generator, IIR filter, and autocorrelation function. This application illustrates the usefulness of writing the start-up and "housekeeping" code in C while optimizing critical routines by writing assembly by hand. The students are allowed to optimize their code in either C or assembly, and they are strongly encouraged to follow the proper development process.

Students who achieve fast execution times discover that they must rewrite these routines in assembly, but this is not enough to get the fastest code. Although the students have been given a MATLAB and C implementation, the algorithms used in some of the routines are not the most efficient. These algorithms therefore require some thought at the theory and implementation level before the fastest execution times can be obtained. Experience has shown that introducing both C and assembly to DSP students in this manner is effective in conveying the trade-offs of each development method.

In order to introduce the students to interactive methods of optimizing C code, as well as giving the students a more in-depth understanding of what optimizing compilers do and how they do it, the optimization lab was recently modified and VISTA was introduced. Data was also gathered on the perceived educational and efficiency (both code and developer time) value of interactive optimizing compilers in general, and perceptions of the VISTA implementation of an interactive optimizing compiler. Therefore, the students are asked to optimize each routine twice: once using VISTA only and once using any technique desired, such as hand-coding in assembly or modifying the assembly output of a compiler. The students are given a brief overview of optimization techniques in the lecture; in the lab writeup they are given one-sentence descriptions of VISTA's optimizations, and they are shown how to apply an example optimization sequence in VISTA. The students are allowed to modify the reference C implementation when optimizing with VISTA.

Due to time limitations and current limitations in the TI port of VISTA, the students couldn't use hand transformations and were therefore limited to using VISTA's optimization phases in various combinations. Therefore, the primary value of this first VISTA introduction in a DSP laboratory course is in evaluating VISTA's usefulness as a teaching tool.

4. RESULTS

Upon completion of the optimization exercise, the students were "interviewed" regarding their experiences with and perceptions of VISTA and interactive optimizers in general. In these interviews we attempt to ascertain what the students learned. Do the students understand what optimizing compilers do and how they do it? Were they introduced to new instructions or efficient coding techniques that enabled them to more quickly write assembly code by hand? Did interacting with VISTA lead to discovering ways of more efficiently writing C code? Do the students know the proper optimization process? During the interview the students were asked:

1. Was the VISTA optimization exercise attempted first?
2. Did you modify the C code?
3. Did interactive optimization help you identify/modify inefficiencies in the C code?
4. Did interactive optimization help you write efficient hand-coded assembly?

5. How are interactive optimizing compilers able or unable to assist you in obtaining efficient code?

6. What are the limitations of VISTA, and how can it be improved?

7. Did you find any code in VISTA that you wanted to change by hand?

8. What are some sequences that you have found useful?

While reactions to VISTA were mixed, more than half of the students found inherent value in using optimizing compilers as a teaching tool, and many believed that interactive optimizing compilers could be valuable in real-world development.

Students felt they didn't have enough information on what the optimization phases meant or did before using VISTA, which is understandable given the limited time to implement the many course objectives. Many students resorted to randomly ordering optimization phases they found useful. A few came up with orderings they believed would improve the code, but other random orderings resulted in better code. However, upon interactively optimizing the code some students were able to learn how optimizations were applied to the code, and these students believe they achieved a better understanding of what many optimizations were.

Of those who attempted to complete the VISTA optimization exercise first, some found new hand-coding optimization ideas by looking at the assembly output of VISTA or the TI compiler. However, a few did find new hand-assembly optimization ideas through VISTA interaction.

Students enjoyed seeing the block and loop structure of the code in VISTA. Since the code consisted of just a few loops, typically the students were able to relate this structure to the original C. Although students generally found it hard to understand how VISTA's assembly corresponded to individual statements in the C code, some commented that they also found the TI compiler's output hard to understand; interacting with VISTA enabled many of these students to better understand how optimizing compilers obtain their final assembly output. One student even commented that VISTA seems to be a good tool for designing compilers!

While students generally found some educational value in using VISTA, most students believed they needed more experience with the tool to get code that is comparable in efficiency to hand-coded assembly. However, a few were able to identify instructions in VISTA that they wanted to change or delete by hand to improve efficiency.

Finally, a significant number of students believed that VISTA would be useful in complex real-world applications. Some of these students thought sufficiently optimal code could be achieved with VISTA, while others thought VISTA would be a good start in writing optimal code.

5. CONCLUSION

Optimizing C compilers should be introduced in a modern DSP laboratory course, and experience has shown that traditional C compilers can be effectively used in a course in which assembly language programming is heavily emphasized. Interactive optimizing compilers have strong potential in teaching DSP students about optimization techniques and optimizing compilers. VISTA is a promising interactive optimizing compiler, and students tended to learn more about optimization than they would have with the TI compiler. Students were unable to generate compiled code that was more efficient than that produced by the standard fixed sequence, but the routines assigned to optimize may have been too small and simple to exploit the performance-enhancing potential of interactive optimization.

6. REFERENCES

[1] D.L. Jones, "Designing effective DSP laboratory courses," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 5, 7-11 May 2001, pp. 2701–2704.

[2] M.L. Kramer, M. Haun, S. Appadwedula, D.G. Sachs, and D.L. Jones, "Effective Use of Projects in DSP Laboratory Instruction" *Proceedings of the First IEEE Signal Processing Education Workshop*, Hunt, Texas, October 15–18, 2000.

[3] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, K. Gallivan, and D. Jones "VISTA: A System for Interactive Code Improvement" *Proceedings of the ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems*, June 2002, pages 155–164.

[4] M.E. Benitez and J. W. Davidson, "A portable global optimizer and linker" *Proceedings of the SIGPLAN'88 conference on Programming Language design and Implementation*, ACM Press, 329–338.

[5] Texas Instruments, "Optimizing C/C++ Compiler User's Guide" `http://www-s.ti.com/sc/psheets/spru103g/spru103g.pdf`

[6] D.L. Jones, "Digital Signal Processing Laboratory (ECE 420)" `http://cnx.rice.edu/content/col10236/1.13/`

[7] M.D. Kleffner and D.L. Jones, "DSP Optimization Techniques" `http://cnx.rice.edu/content/m12380/1.2/`

[8] M.D. Kleffner, "Spectrum Analyzer: VPO/VISTA Optimization Exercise" `http://cnx.rice.edu/content/m12393/1.2/`