PERFORMANCE EVALUATION OF REGISTER ALLOCATOR FOR THE ADVANCED DSP OF TMS320C80

Jihong Kim

Department of Computer Science
Seoul National University
Seoul, Korea 151-742
jihong@cs.snu.ac.kr

Graham Short

Texas Instruments
800 Pavilion Drive
Northampton, NN4 7YL, England
Graham.Short@tiuk.ti.com

ABSTRACT

PPCA is an assembly language-level register allocator and instruction compactor for the Advanced DSPs (ADSPs) of the TMS320C80 digital signal processor. It was developed to help the implementation of time-critical ADSP assembly programs which heavily utilize powerful ADSP features optimized for multimedia and image computing applications for maximum efficiency. PPCA takes as an input ADSP assembly operations with symbolic variables. It then allocates the ADSP's physical registers to the symbolic variables and rearranges the operations into a highly-parallelized compact format. In this paper, we have evaluated the performance of a register allocation capability of PPCA using an extensive image computing library for the TMS320C80. We present the basic algorithm of the PPCA's register allocation module and describe the performance evaluation approach used. The result shows that PPCA essentially achieves optimal register allocation for the test cases based on the image computing library functions.

1. INTRODUCTION

High-performance DSPs have several unique architectural features optimized for signal processing such as single-cycle multiplication and addition, multiple operations per cycle and zero-overhead looping. Many newer-generation DSPs also include additional hardware features for supporting multimedia and video compression applications such as bit-field manipulation support or multiple-pixel parallel manipulation capability. The performance-enhancing features of these DSPs, however, often make it more difficult for a compiler to produce efficient codes. The realized performance with high-level language implementations is often just a small fraction of the maximum performance achievable. For example, a 3×3 median filtering operation implemented in C on a multimedia system [6] with a 512×512 8-bit image took 494 milliseconds while the hand-optimized assembly-language implementation took only 10.7 milliseconds. Similar results were also reported for different DSPs [9].

Currently, the only practical approach in getting high performance is to program the DSPs at the assembly-language level accessing the performance-enhancing features directly from assembly-language instructions. Assembly-language programming is, however, largely dependent on an individual programmer's experience and familiarity with the internal architecture, instruction set, and peculiarities of a specific processor without any systematic methodology. In fact, there are few software tools available to support DSP-based assembly-language programming. However, as more processors are architected with many unique features to facilitate multimedia and image computing and they employ a higher degree of instruction-level parallelism, writing an efficient assembly program is becoming a major challenge. The goal of assembly language-level tools is to automate the time-consuming and error-prone parts of assembly programming such as register allocation, instruction compaction and instruction scheduling.

PPCA [8] is one of such assembly language-level tools developed for the Advanced DSPs (ADSPs) of the TMS320-C80 which is highly-optimized for multimedia, video compression, image/signal processing and computer graphics [3]. The main functions of PPCA are automatic register allocation and instruction compaction. PPCA takes as an input ADSP assembly operations with symbolic variables. Using the symbolic variables (instead of the physical registers) allows faster development of an ADSP program. It then allocates the ADSP's physical registers to the symbolic variables and rearranges the operations into a highly-parallelized compact format. Considering that the majority of TMS320C80 software development efforts are spent programming the ADSP, PPCA significantly reduces the overall time-to-market for the TMS320C80 software development.

In this article, we evaluate the performance of a register allocation capability of PPCA using an extensive image computing library for the TMS320C80. Our goal is to understand the performance of PPCA compared to that of an optimal register allocator. The performance evaluation results should be available for programmers' review before such tools are adopted for application development.

The organization of the rest of the article is as follows. Before PPCA is described, a brief description of the ADSP of the TMS320C80 is presented in the next section. In Section 3, PPCA is overviewed, and the algorithm used for the register allocation module of PPCA is described. The comparison results as well as the performance evaluation approach are discussed in Section 4.

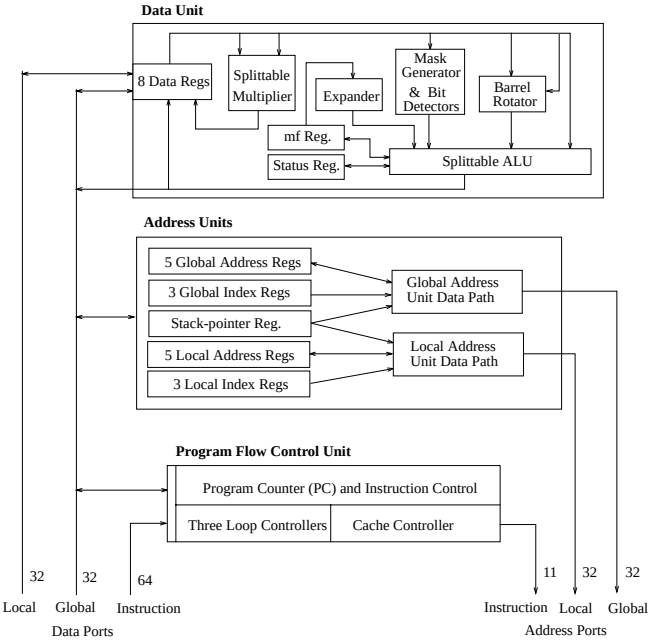

Figure 1: Advanced DSP block diagram.

2. ADVANCED DSPS OF TMS320C80

The TMS320C80 can be described as a single-chip, heterogeneous, MIMD multiprocessor connected via a crossbar to multiple on-chip shared memory modules. It combines a RISC processor and four Advanced DSPs as well as an intelligent direct memory access (DMA) controller and two video controllers into a single-chip device. (For the detailed description, see a reference [3].)

Four ADSPs provide most of the TMS320C80's raw performance. Figure 1 shows a block diagram of the ADSP and its four major functional units: the data unit, two address units (LAU and GAU), and the program flow control unit. The data unit consists of the data unit registers, the multiplier, and the ALU data path, and supports many powerful features. New features not found in conventional DSPs include three-operand splittable 32-bit ALU (two 16-bit or four 8-bit units), multiple flags register and expander for multiple arithmetic, splittable 16-bit multiplier (two 8-bit multipliers), and bit-detection logic. Because of four parallel functional units, each ADSP can perform four different operations in a single cycle: multiplication, ALU operation, and two memory accesses.

Each ADSP has 44 programmer-visible registers, and they can be classified into two categories: the general-purpose registers and special-purpose registers. The data unit contains eight general-purpose data registers (`d0-d7`). Each address unit has five general-purpose address registers and three general-purpose index registers: `a0-a4` and `x0-x2` for the LAU, and `a8-a12` and `x8-x10` for the GAU, respectively. The remaining registers, which belong to the special-purpose category, are more specialized ones such as the multiple flags register and status register, or the registers used for the zero-overhead loop control. The current version

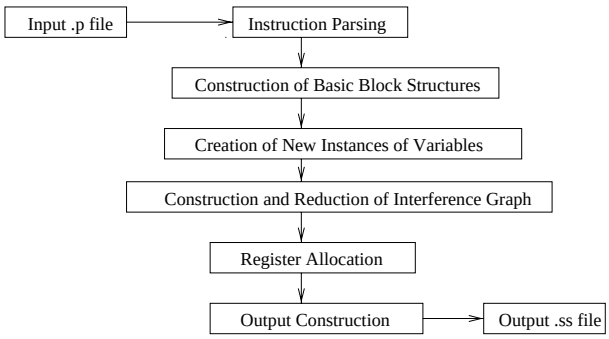

Figure 2: Overall processing steps of the PPCA's register allocation module.

of PPCA supports the register allocation for the general-purpose registers only.

Because of the limited number of operand bits available in specifying operands in an instruction, the ADSP imposes the additional restrictions on the register usage. For example, some operation classes allow both an input operand and a destination operand to be derived from the same operand bits, thus restricting allowable register pairs. Such operand pair are called companion registers. For example, if both operands are data registers, being companion registers means that they must be the same physical register. For a conditional source selection (based on the negative condition code), only successive odd-even pair of data registers are allowed as input operands. PPCA understands these additional restrictions on the register usage and handles them correctly.

3. PPCA: OVERVIEW AND BASIC ALGORITHM

In order to help the register allocation and instruction compaction tasks, PPCA uses some extra assembly-language directives which are added into an ADSP assembly program [8]. Symbolic variables are declared using the **.reg** directive. Program's control-flow information is specified using the control-flow directives (**.entry/.cjump/.ujump/.cexit/.uexit/.entry**). These directives are necessary for correct register allocation but not available from an ADSP assembly program. PPCA then allocates physical registers to symbolic variables and (optionally) reorders the input ADSP operations into a parallel format. The processed ADSP assembly program is written to the output file. The output program contains the legal ADSP assembly operations with symbolic variables replaced by physical registers, and are optionally in a more compact format. The result of register allocation is presented using the **.set** directive. This output file is then assembled by a regular ADSP assembler. The function of an input ADSP program is preserved while the register allocation and instruction compaction tasks are performed.

The overall processing steps taken by the PPCA's register allocation module are summarized in Figure 2. The control-flow directives are used to split the list of instructions parsed into basic blocks as found in a conventional

compiler design. A basic block is a straight line sequence of instructions without any branches. For each variable x of a basic block B, the lifetime information is computed for basic blocks. Once the lifetime information is computed, different variable names are assigned to different live ranges of the same variable for more efficient register allocation. Based on the updated lifetime information, an interference graph is constructed. An interference graph nicely models register-conflict situations. Based on a coloring of the interference graph, actual register allocation is made and an output is saved to a file. The PPCA-related directives are included as comments in the output file. In this paper, we describe only two steps in detail, the construction and reduction of the interference graph step and the register allocation step. (For a complete description of the PPCA's register allocation module, see a reference [4].)

3.0.1. Construction and Reduction of Interference Graph

As described by Chaitin [2] and Briggs [1], the problem of register allocation is nicely abstracted to that of graph coloring. Based on the lifetime information of a basic block as well as a basic block itself, exact lifetime information for all the variables (including new instances) can be determined. Using this information, an interference graph is constructed. An interference graph G of n variables is represented by a bit matrix of $n \times n$. The $G[x, y]$ element is set to 1 if the lifetimes of two variables x and y can be overlapped. In this case, x and y cannot be allocated to the same physical register. If the lifetimes of two variables x and y do not overlap, the $G[x, y]$ is set to 0.

Once the interference graph is built, the reduction step is started. For the reduction step, the degree $d(v)$ of each vertex v (i.e., variable) is first computed. This is the number of variables that interfere with a variable v. Then, vertices are grouped into an interference array D where $D[i]$ points to a list of vertices whose degree is i, as done with in Briggs [1].

This interference array is successively reduced by searching for the first non-empty list from the beginning. The head of the selected list is removed from the interference array and all its interfering nodes are moved down one position in the list. The variable representing the vertex is pushed into the coloring stack. This process iterates until all vertices are removed.

3.0.2. Register Allocation

Once all the vertices are pushed into the coloring stack, actual register allocation is started by deleting the top element of the coloring stack at a time. For each deleted variable, it is checked whether a free register (which was not taken by interfering variables) is available or not. During this test, the interference between a variable and pre-allocated registers are also considered. If no physical register is available, an extra physical register is allocated instead of inserting spill code as typically done in a high-level language compiler. When an extra physical register is allocated, an error message is printed to a programmer requesting to restructure the program. The rational behind this approach was that (1) PPCA would be mainly used for time-critical applications where extra spill code should be avoided as much as possible, and (2) assembly programmers would do a better job of restructuring their programs to avoid unnecessary spill code.

4. RESULTS

In order to evaluate how well the PPCA's register allocation module performs for image computing applications, we have compared the PPCA's performance with that of an optimal register allocator. As described in the previous section, the PPCA's register allocator is based on a graph coloring, and its performance is determined by the performance of a graph coloring algorithm used. The method used in PPCA is commonly called a *smallest-last coloring* method because the nodes of the *smallest* degree are removed first from an interference graph and they are colored *last* [7].

For performance evaluation, PPCA was modified to produce an interference graph of an input .p file as an intermediate result. This interference graph is fed into an optimal graph coloring program which we have developed. The register allocation results are compared to the graph coloring results from an optimal graph coloring program.

In order to focus on image computing applications, we have used as test cases the hand-optimized ADSP assembly routines from an image computing library for the TMS320C 80, the University of Washington Image Computing Library (UWICL)[5]. ADSP assembly programs were all manually converted into the PPCA's input format by inserting appropriate directives. Manual conversion was inevitable because the control-flow directives required the understanding of overall program execution.

An interference graph generated from PPCA included interference information among symbolic variables only, missing several interference relations existing in an input ADSP program. For example, the interference between pre-allocated registers and symbolic variables are not represented in an interference graph. In addition, the special restrictions such as companion registers and a pair of registers are not expressed as well. The special restrictions were not considered because of the difficulties in representing them in an interference graph. (In PPCA, these restrictions are represented separately from an interference graph and extra tests are performed to satisfy them.)

An optimal coloring program is based on two heuristics and a direct enumeration algorithm as shown in Figure 3. The main difference of this algorithm from other optimal graph coloring algorithms (based on a direct enumeration method) is in how to compute a lower bound $l_{\gamma(G)}$ on the smallest number $\gamma(G)$ of colors necessary for a graph G. A lower bound is computed using the fact that the vertices colored with the same color in a complementary graph $\overline{G}$ form a clique in G. Since a clique of n vertices in G requires n colors, the maximum clique provides a lower bound on $\gamma(G)$. The complementary graph $\overline{G}$ is colored using a *largest-last* coloring method and the maximum number of vertices sharing the same color in this coloring is set to $l_{\gamma(G)}$. (In a largest-last coloring method, the nodes of the *largest* degree are removed first from a graph and they are colored *last*.) An upper bound $l_{\gamma(G)}$ on $\gamma(G)$ is computed using a smallest-last coloring method as done in PPCA. Once both bounds are computed, a recursive enumeration algorithm

Input: an undirected graph G
Output: the smallest number $\gamma(G)$ of colors necessary for G.

Step 1: Set $u_{\gamma(G)}$ to be the number of colors necessary when the graph G is colored using a smallest-last coloring method.
Step 2: Construct a complementary graph $\overline{G}$ and color $\overline{G}$ using a largest-last coloring method. Set $l_{\gamma(G)}$ to be the largest number of vertices which were assigned to the same color.
Step 3: Starting from $l_{\gamma(G)}$, find a minimum n ($l_{\gamma(G)} \leq n \leq u_{\gamma(G)}$) such that G can be colored using n colors. A (recursive) direct implicit enumeration algorithm is used to check n-colorability.

Figure 3: An optimal register coloring algorithm.

is used sequentially, starting from $l_{\gamma(G)}$ colors until a legal coloring is found.

Using a largest-last coloring method with $\overline{G}$ produced a very tight lower bound for both our test cases and other interference graphs based on real programs. For example, for the contributed interference graphs, lower bounds based on a largest-last coloring method consistently outperformed ones based on a smallest-last coloring method by a large margin (up to 19 colors for 30- to 65-colorable graphs). Furthermore, for 12 out of 14 graphs, a largest-last method found an optimal value as a lower bound, making the step 3 of Figure 3, the most time consuming step of an optimal coloring algorithm, unnecessary. A smallest-last based method did not find a single lower bound which is equal to an optimal value.

We have tested 125 ADSP assembly routines from the UWICL library. Each routine contains up to five different interference graphs corresponding to five different register types (`d`, `la`, `ga`, `lx` and `gx`). The total number of interference graphs tested were 193, excluding trivial cases involving one or two symbolic variables. Out of 193 test cases, PPCA achieved optimal register allocation for 164 cases. Among the remaining 29 cases, the PPCA's results were off by one for 22 cases, by two for 6 cases, and by three for 1 case, from optimal numbers. All of non-optimal test cases were related to unaccounted restrictions of ADSP programs. For example, many non-optimal PPCA allocations occurred when pre-allocated registers were used in ADSP assembly routines such as the use of `d0` register for EALU operations [3]. Since our optimal graph coloring program did not consider the extra restrictions, its coloring results were less than the PPCA's results by one or two. When an input ADSP program involved only symbolic variables without these extra restrictions (164 cases), PPCA essentially achieved optimal performance.

5. CONCLUSION

We have described the assembly language-level register allocator and instruction compactor, PPCA, for the Advanced DSPs of the TMS320C80, and evaluated the performance of its register allocation module using image computing library routines for the TMS320C80. Our goal has been to understand how well PPCA (which is based on a simple heuristic, a smallest-last coloring method) performs the register allocation task for image computing applications. We compared the result from PPCA to that of an optimal register allocator. For our test cases, the PPCA's register allocation module essentially achieved optimal performance.

We believe that assembly language-level tools such as PPCA will be more and more important to achieve potential maximum efficiency from high-performance DSPs and multimedia chips as they incorporate more sophisticated special hardware features in their architectures. As presented in this paper, the performance evaluation results of such tools, particularly ones based on the commonly-used functions in the intended application areas of DSPs/chips, would provide a useful guideline for application programmers in selecting the most appropriate tools for developing time-critical applications.

6. REFERENCES

[1] P. Briggs, K. D. Cooper, K. Kennedy and L. Torczon, "Coloring heuristics for register allocation," in *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989, pp. 275-284.

[2] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, pp. 47-57, 1981.

[3] K. Guttag, R. J. Gove, and J. R. Van Aken, "A single-chip multiprocessor for multimedia: The MVP," *IEEE Computer Graphics & Applications*, vol. 12, no. 6, pp. 53-64, 1992.

[4] J. Kim and G. Short, "Performance evaluation of assembly-level register allocator for advanced DSP of TMS320C80," *Texas Instruments Technical Journal*, vol. 14, no. 3, pp. 76-89, May-June 1997.

[5] J. Kim and Y. Kim, "UWICL: a multi-layered image computing library for single-chip multiprocessor-based time-critical systems," *Real-Time Imaging*, vol. 2, no. 3, pp. 187-199, 1996.

[6] W. Lee, Y. Kim, R. J. Gove and C. J. Read, "MediaStation 5000: integrating video and audio," *IEEE MultiMedia,* vol. 1, no. 2, pp. 50-61, 1994.

[7] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of ACM*, vol. 30, no. 3, pp. 417-427, 1983.

[8] Texas Instruments, *PPCA User's Guide,* 1995.

[9] V. Živojnović, "Compilers for digital signal processors: the hard way from marketing- to production-tool," *DSP & Multimedia Technology Magazine,* vol. 4, no. 5, pp. 27-45, 1995.