OPTIMIZATION OF EMBEDDED DSP PROGRAMS USING POST-PASS DATA-FLOW ANALYSIS

Ashok Sudarsanam[1] *Sharad Malik*[1] *Steven Tjiang*[2] *Stan Liao*[2]

[1]Department of Electrical Engineering, Princeton University, Princeton, New Jersey, USA
[2]Advanced Technology Group, Synopsys Inc., Mountain View, California, USA

ABSTRACT

We investigate the problem of code generation for DSP systems on a chip. Such systems devote a limited quantity of silicon to program ROM, so application software must be maximally dense. Additionally, the software must be written so as to meet various high-performance constraints, which may include hard real-time constraints. Unfortunately, current compiler technology is unable to generate dense, high-performance code for DSPs, whose architectures are highly irregular. Consequently, designers often resort to programming application software in assembly – a time-consuming, error-prone, and non-portable task. Thus, DSP compiler technology must be improved substantially. We describe some optimizations that significantly improve the quality of compiler-generated code. Our optimizations are applied globally and even across procedure calls. Additionally, they are applied to the machine-dependent assembly representation of the source program. Our target architecture is the Texas Instruments' TMS320C25 DSP.

1. INTRODUCTION

It is a well-known fact that the quality of compiled code for embedded DSP systems is extremely unsatisfactory, with respect to density and performance [5]. Consequently, most application software for such systems is hand-written – a very time-consuming, error-prone, and non-portable task. In order to increase developer productivity, DSP compiler technology must be improved substantially.

The overall goal of our research is to develop a *retargetable* DSP compiler that generates sufficiently dense, high-performance code, hence obviating the need for assembly programming. Our current objective is to identify all optimizations necessary to generate high-quality code for a single architecture – our target is the *Texas Instruments' TMS320C25* DSP [4], a representative of Texas Instruments' popular fixed-point DSP family. In this paper, we describe some of these optimizations.

Our optimizations are applied *globally* and *interprocedurally* and thus, further improve the quality of *locally*-optimized code. Additionally, our optimizations occur at the *post-pass* level – they are applied to the *machine-dependent assembly* representation of the source program, rather than the *machine-independent intermediate* representation. It is well-understood that post-pass optimizations which exploit the irregular features of DSP architectures are essential for generating code of the highest quality.

This paper is organized as follows: Section 2 gives an overview of the *TMS320C25* architecture. Section 3 describes our optimizations. Section 4 provides experimental results, and finally, Section 5 presents our conclusions.

2. TMS320C25 ARCHITECTURAL OVERVIEW

We first describe the organization of the *TMS320C25* memory subsystem, since it motivates the need for global and inter-procedural post-pass optimizations. The *TMS320C25* does not penalize the use of the *absolute* addressing mode. Consequently, for each procedure that is not involved in recursion, the compiler can statically allocate all automatic variables (those local variables that become active on procedure entry and inactive on procedure exit) and refer to them by their absolute memory address. This eliminates instructions to set up stack frames and modify the frame pointer. In contrast, architectures such as the *Motorola 56000* incur a penalty for each use of the absolute addressing mode – an extra instruction word is required to hold the absolute address. In these architectures, it is advantageous to allocate the automatic variables on the stack frame and access them *indirectly* through one or more address registers.

Main memory is divided into *512* pages, with each page subdivided into *128 16*-bit words. A *9*-bit register **DP** is dedicated to hold a single page number. When the value of **DP** is concatenated with a *7*-bit offset that occurs in the instruction word, a *16*-bit address is formed which represents the absolute address of a word in memory. However, before accessing main memory, one must ensure that **DP** contains the correct page number. This is the purpose of the **LDPK** instruction, which loads **DP** with a *9*-bit constant.

Consider the instruction sequence of Figure 1 which adds the contents of statically-allocated variable *var* to the accumulator. Assume that *var* has been allocated to memory page *two* at offset *100*. The first instruction loads **DP** with the number of the page containing *var*. The second instruction first forms the absolute address of *var* by concatenating the value of **DP** with the offset of *var* within page two. It then adds the contents of this memory location to the accumulator. A *basic block* is defined to be a code sequence in which control-flow enters at the beginning and leaves only at the end [1]. Observe that if two successive memory references in a basic block access variables allocated to the same page and no procedure call occurs between these

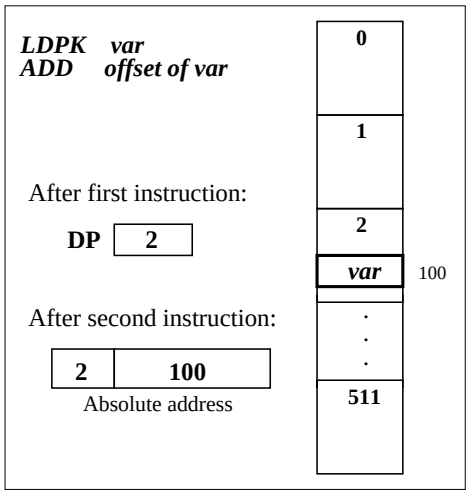

Figure 1. *TMS320C25* **paged memory example.**

references, an **LDPK** instruction before the second reference becomes redundant. Thus, our objective is to take advantage of the static allocation capabilities offered by the *TMS320C25*. However, our efforts will be successful only if the **LDPK** overhead resulting from static allocation is significantly less than the instruction overhead resulting from stack-based allocation. We will now describe how post-pass optimizations may be used to reduce the **LDPK** overhead.

3. POST-PASS OPTIMIZATIONS

3.1. Reducing the LDPK Overhead with Post-pass Data-flow Analysis

Assembly code is traditionally generated on a per-basic block basis. Lin [6] provides a simple post-pass algorithm that removes redundant **LDPK** instructions within basic blocks, assuming that an **LDPK** is generated for each access of a statically-allocated variable. Alternatively, it is possible to suppress generation of many redundant **LDPK**s during code generation itself. Prior to code generation, it is assumed that the value of **DP** is unknown. Before code is generated to access a statically-allocated variable v, the value of **DP** is compared to the number of the page to which v has been allocated. If these values are identical, then generation of an **LDPK** instruction is suppressed, otherwise the value of **DP** is updated and an **LDPK** is generated. It is also assumed that a procedure call destroys the value of **DP**. Hence, for each basic block, an **LDPK** is generated for the very first access of a statically-allocated variable and for each access that is the first to follow a call.

By performing *global post-pass data-flow analysis* [1] for each procedure p, we can determine the value(s) that **DP** assumes upon entry to each basic block of p. We may then remove the very first **LDPK** instruction of each basic block B (assuming it is not preceded by a procedure call) if and only if the value of **DP** upon entry to B is equal to the number of the corresponding page. In order to do this, we must compute three values for each basic block B:

- **IN**(B): value of **DP** upon entry to B
- **OUT**(B): value of **DP** upon exit from B
- **LAST**(B): number of page last accessed in B

LAST(B) is computed by examining the assembly instructions of B in *reverse* order. We still assume that a procedure call destroys the value of **DP**, so if a call is encountered before an **LDPK** in our traversal, **LAST**(B) is assigned the constant **UNKNOWN**. If no **LDPK**s or procedure calls are encountered, **LAST**(B) is assigned the constant **PROPAGATE**. Otherwise, it is assigned an integer corresponding to the page number referenced by the **LDPK**. We then compute **IN**(B) and **OUT**(B) using the equations of Figure 2 – in these equations, a *null* intersection is equivalent to a value of **UNKNOWN**. Once a solution to these equations has been found, we may attempt to remove the very first **LDPK** instruction of each basic block as previously described. Note that due to the global nature of these equations, **IN**(H) is assigned the value **UNKNOWN**, where H is the *entry* basic block of the current procedure. This implies that the very first access in each procedure of a statically-allocated variable will be preceded by an **LDPK**. Additionally, an **LDPK** will still precede each access that is the first to follow a procedure call.

By performing *inter-procedural* post-pass data-flow analysis, we can determine the value(s) that **DP** assumes after each procedure call, as well as upon entry to each procedure. This information allows us to potentially remove a subset of the remaining **LDPK** instructions. The former can be derived by assigning **DP** the value of **OUT**(T) after each procedure call, where T is the *exit* basic block of the called procedure. The latter can be computed, for each procedure p, by examining the value of **DP** prior to each procedure call to p in the program, and then taking the intersection of these values. Note that **DP** is assumed to be unknown before program execution begins. Thus, the very first access in the program of a statically-allocated variable will always be preceded by an **LDPK**.

```
while ( OUT(B)'s have changed ) {
   for each basic block B {
      IN(B) = ∩ OUT(p);
              p in predecessor(B)
      if ( LAST(B) == PROPAGATE )
         OUT(B) = IN(B);
      else OUT(B) = LAST(B);
   }
}
```

Figure 2. Global data-flow analysis equations used for LDPK and RSXM/SSXM removal.

3.2. Reducing the Data-Memory Overhead by Coloring Automatic Variables

The primary disadvantage of statically allocating automatic variables is that the run-time data-memory requirements of the program increase considerably – statically-allocated

variables are assumed to exist for the entire duration of the program and hence, may not be dynamically deallocated. Reducing the data-memory overhead of the application software is essential in the design of embedded systems.

By definition, automatic variables are required to exist during the time period between procedure entry and procedure exit. This allows us to perform a *graph coloring* of all statically-allocated automatic variables (henceforth referred to as *eligible variables*), such that identically-colored eligible variables are allocated to the *same* data-memory location. In the first step of the coloring process, *live-variable analysis* [1] is performed on each procedure's eligible variables so as to determine their *live ranges*. The live range of a variable v specifies those regions of the program where v is active. We enforce liveness relationships among eligible variables of different procedures using the following heuristic: at each call instruction s in procedure p, the live ranges of all eligible variables of p active at s are assumed to overlap with the live ranges of all eligible variables of (i) the called procedure c and (ii) those procedures directly and transitively called by c. In the second step, an *inter-procedural interference graph* is constructed in which a vertex exists for each eligible variable in the program, and edge $e(i, j)$ exists if and only if the live ranges of variables i and j overlap. This edge specifies that i and j must be colored differently, or analogously, they must be allocated to different data-memory locations. Otherwise, a definition of i will overwrite the value of j, or vice versa, hence leading to incorrect code.

We now use a binary search to determine a small number of colors that satisfies the graph-coloring constraints imposed by this interference graph (call it G). At each step of the search, we apply Briggs' coloring heuristic [3] to G. Experimental results demonstrate that this optimization significantly reduces the total quantity of data-memory consumed by the program. Additionally, if the **LDPK** optimization is applied after coloring, we find that we are able to further reduce the **LDPK** overhead – since fewer pages of memory are consumed, **DP** now assumes fewer values.

3.3. Reducing the RSXM/SSXM Overhead with Post-pass Data-flow Analysis

The post-pass data-flow analysis techniques of Section 3.1 may be used to eliminate other redundant assembly instructions. The *TMS320C25* permits the programmer to set and reset the *sign-extension (s-e) mode* using the **SSXM** and **RSXM** instructions, respectively. The former instruction causes future accumulator loads to be automatically sign-extended, while the latter suppresses sign-extension. Let us redefine **IN**(B), **OUT**(B), and **LAST**(B) for each block B:

- **IN**(B): value of s-e mode upon entry to B
- **OUT**(B): value of s-e mode upon exit from B
- **LAST**(B): last value of s-e mode in B

LAST(B) is again computed by examining the assembly instructions of B in reverse order: if a procedure call is encountered before an **RSXM/SSXM** instruction in our traversal, **LAST**(B) is assigned the value **UNKNOWN**. If no **RSXM/SSXM** instructions or procedure calls are encountered in the traversal, **LAST**(B) is assigned the constant **PROPAGATE**. Otherwise, it is assigned the constant **RESET** or **SET** depending on whether an **RSXM** or **SSXM**, respectively, was encountered first.

Using the equations of Figure 2, we now compute **IN**(B) and **OUT**(B) for each basic block B. Assume that the first sign-extension-mode-related instruction i in block B is not preceded by a procedure call. Then if i is equivalent to an **RSXM** (**SSXM**) instruction, we may safely remove it if and only if **IN**(B) has a value of **RESET** (**SET**). Inter-procedural post-pass data-flow analysis may be used to remove other redundant **RSXM/SSXM** instructions.

3.4. Post-pass Optimizations in a Retargetable Environment

The routines responsible for reducing the **LDPK** and **RSXM/SSXM** overhead are virtually identical – the only difference occurs in the routines responsible for computing **LAST**. Rather than implementing these optimizations with two sets of nearly-identical code, we have instead developed *generic post-pass data-flow analysis interfaces* which these and other optimizations may use. In particular, we have developed generic interfaces for post-pass *reaching definitions analysis*, *live-variable analysis*, and *available expressions analysis* [1]. Consequently, these optimizations have been implemented with very little code duplication.

In order to make use of these data-flow analysis routines, one must provide two optimization-specific functions to the appropriate interface. Each of these functions takes an assembly instruction as input and outputs a bit-vector. For reaching definitions and available expressions analysis, these functions are *gen* and *kill* – *gen(i)* and *kill(i)* represent the set of definitions *generated* and *killed*, respectively, by instruction i. For live-variable analysis, these functions are *use* and *def* – *use(i)* and *def(i)* represent the set of data *used* and *defined*, respectively, by instruction i. We have used the available expressions interface to implement the optimizations of Sections 3.1 and 3.3. As an example of the use of the *gen* and *kill* functions, assume variable v has been statically allocated to data-memory page *5*. Then,

- *gen(* **LDPK**(v) *)* = < 5 >
- *kill(* **LDPK**(v) *)* = < 0, 1, 2, 3, 4, 6, 7, ..., 511 >

We have also developed a *generic graph coloring interface* that performs live-variable analysis and graph coloring of optimization-specific data. We have implemented the optimization of Section 3.2 by providing this interface with the set of eligible variables of each procedure in the application program. We have also used this interface to allocate *address registers* to the pointer variables of a program[2].

4. EXPERIMENTAL RESULTS

We have implemented our optimizations in the **SPAM** compiler [5] – a joint project of Synopsys, Inc., Princeton University, University of Aachen, and M.I.T. – which is a retargetable code generation framework for embedded DSP processors. Table 1 shows the results of applying our post-pass optimizations to benchmarks from the *DSPstone* benchmark suite [7] – *adpcm* is a large speech-encoding algorithm, while *complex update*, *fft*, *fir*, *iir biquad*, *lms* and *matrix multiply* are small DSP kernels. The benchmarks are listed in the first column of Table 1. The next column specifies the

DSPstone Benchmark	LDPK Count (After Coloring)			RSXM/SSXM Count		
	Initial	Global	InterProc	Initial	Global	InterProc
adpcm	692 (461)	385 (37)	364 (1)	105	59	49
complex update	4 (4)	4 (4)	1 (1)	1	1	1
fft	55 (55)	17 (17)	1 (1)	25	5	3
fir	40 (40)	5 (5)	1 (1)	16	2	2
iir biquad	13 (13)	6 (6)	1 (1)	5	3	2
lms	12 (12)	5 (5)	1 (1)	6	2	2
matrix multiply	19 (19)	4 (4)	1 (1)	7	2	2

Table 1. Results of post-pass optimizations.

total number of **LDPK** instructions present in the assembly code after basic block code generation has been performed. These numbers serve as base measurements against which we will compare the results of our optimizations. The next two columns specify the total number of **LDPK**s present after global and inter-procedural data-flow analysis, respectively, have been performed. The numbers in parentheses specify the total number of **LDPK** instructions present after variable coloring and post-pass data-flow analysis have been performed in succession. The next column specifies the total number of **RSXM/SSXM** instructions present in the assembly code after basic block code generation has been performed. Again, these values serve as base measurements. The final two columns specify the total number of **RSXM/SSXM**s present after global and inter-procedural data-flow analysis, respectively, have been performed.

In our experiments, we assumed the compiler allocated eligible variables to data-memory in order of *decreasing static frequency*. It is evident that inter-procedural post-pass data-flow analysis, combined with coloring of automatic variables, essentially eliminated the **LDPK** overhead in each of these benchmarks. For instance, for the *adpcm* benchmark, global data-flow analysis combined with coloring resulted in an **LDPK** instruction count of *37*, a *94.6%* reduction from the base measurement of *692*, while inter-procedural analysis combined with coloring resulted in an **LDPK** overhead of just *one* instruction. These results can be attributed to the fact that after coloring, only *one* page of memory was required for static storage of eligible variables. For the smaller benchmarks, inter-procedural data-flow analysis resulted in the same **LDPK** overhead, regardless of whether or not coloring was performed. For example, for the *fft* benchmark, global data-flow analysis combined with coloring resulted in an **LDPK** instruction count of *17*, a *69%* reduction from the base measurement of *55*. Inter-procedural analysis with or without coloring reduced the **LDPK** count to just *one*. These results can be attributed to the small number of eligible variables present in these programs – the compiler was able to allocate them to one page of memory even without coloring.

Finally, it is evident that our optimizations were quite successful at reducing the **RSXM/SSXM** overhead. For instance, for the *adpcm* benchmark, global data-flow analysis resulted in an instruction count of *59*, a *43.8%* reduction from the base measurement of *105*. Inter-procedural analysis reduced the overhead further by *16.9%*. For the *fft* benchmark, our optimizations reduced the overhead from the base measurement of *25* to *3*, an *88%* reduction.

5. CONCLUSIONS

We have described some post-pass optimizations that significantly improve the quality of compiled code for embedded DSP systems. In particular, we have shown how redundant **LDPK** and **RSXM/SSXM** instructions can be eliminated by performing *global* and *inter-procedural* data-flow analysis on the assembly code. We have also shown how coloring of all statically-allocated automatic variables can significantly reduce the data-memory requirements of the application software, as well as further reduce the **LDPK** overhead. Finally, we have described our implementation of *generic interfaces* for post-pass data-flow analysis and graph coloring which can be used to implement these, as well as other machine-dependent optimizations.

6. ACKNOWLEDGEMENTS

This research was supported by an NSF NYI award (grant MIP 9457396).

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] G. Araujo, A. Sudarsanam, and S. Malik. Instruction Set Design and Optimizations for Address Computation in DSP Architectures. In *Proceedings of 1996 International Symposium on System Synthesis*, 1996.

[3] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. ACM *Trans. on Prog. Lang. and Sys.*, 16(3):428–455, 1994.

[4] Texas Instruments. *TMS320C2x User's Guide*. January 1993. Revision C.

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Živojnović, and H. Meyr. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 165–186. Kluwer Academic Publishers, 1996. Proceedings of the NATO Advanced Study Institute on Hardware/Software Co-Design.

[6] W. Lin. An Optimizing Compiler for the TMS320C25 DSP Processor. Master's thesis, University of Toronto, 1995.

[7] V. Živojnović, J. Martínez Velarde, and C. Schläger. DSPstone: A DSP-oriented Benchmarking Methodology. In *Proceedings of the 5th Int'l Conference on Signal Processing Applications and Technology*, October 1994.