Minimizing cost of local variable access for DSP-processors

Seminarvortrag von Mathias Johnson
7.2.2002
Overview

1. Introduction
   - Addressing modes
   - Floating frame pointers

2. Problem description
   - Requirements
   - Modified basic blocks
   - Control flow graphs

3. Algorithm
   - Preparations
   - The control flow line graph (CFLG)
   - Frame pointer values
   - Algorithm

4. Results & summary
1. Introduction

Addressing modes

We begin by going through a few common ways to access local variables in functions. There are other addressing modes, but in the following, the ones relevant to this article.

Post modify

Using the post modify addressing mode, the variable is on the program stack and a pointer to the variable is stored in a register. In a single instruction, the variable can be accessed and the address register can be modified. In many cases, the address register is incremented or decremented by 1 (for example the program counter), but any larger modification is possible. The post modify addressing mode is very common in DSP compilers.

Register + offset

The ‘register + offset’ addressing mode uses a different approach. The start address of the function frame on the stack is stored in a register. The local variable is then accessed using the frame pointer (always pointing at the start address of the function) and a certain offset. This addressing mode is hard to apply on DSP compilers, and is hence very rare.

Immediate address load

Another addressing mode commonly used in DSP architectures is the immediate address load. Here, no pointers are used, but the direct address of the variable is used to access it. This approach implies static variables and is slow in DSP architectures, but sometimes it may be the only available addressing mode.
1. Introduction

Floating frame pointers

The register + offset addressing mode is very rarely available in DSP compilers. Instead, many compilers use static memory allocation for function frames, using the stack only for passing variables. This approach has at least the following disadvantages:

- Recursive functions are not allowed. Although very rare in DSP applications, recursion is a part of the language standard (de facto ANSI-C) and therefore this makes the compiler non-compliant to the standard.
- We are forced to use immediate address loads, which, as stated before, is slow.
- It is also a waste of memory, since we cannot dynamically allocate and deallocate function frames. Function siblings are not active simultaneously.

Instead we introduce another approach: A floating frame pointer. The idea is that the frame pointer does not statically point to the starting address of the function frame, but to the address of the variable that is to be accessed next. After access, the frame pointer is modified using the post modify addressing mode. If we can use post modify instructions at all variable accesses, no overhead is created.

The problem here is how to determine the new value for the frame pointer, so that it is preset before the frame pointer is used the next time.
2. Problem description

Requirements

We now have a clear goal: to preset the value of the frame pointer (FP) for every single instruction in a function. The frame pointer value should be defined and unique at all locations in the function and we use the post modify addressing mode to atomically set it to the value required by the next instruction.

This requires that all addresses in the available memory space can be accessed using a single modify instruction. If this were not possible, we could have to insert lots of “assisting” instructions to get the pointer the rest of the way. Here, we assume all addresses can be reached using one instruction.

Despite that the post modify addressing mode is available, not all assembler instructions can modify the frame pointer. It may therefore be necessary to insert explicit instructions to modify the frame pointer as required. This, of course, is not desired, as it not only makes the code larger but also makes the program execution slower. Our eventual goal is to find an algorithm that minimizes the number of explicit frame pointer modifications.
2. Problem description

**Modified basic blocks**

To start with, we divide the code into basic blocks. Basic blocks are sequences of instructions with one entry and one exit.

There are instructions that...
1. do not need or modify the frame pointer
2. need the frame pointer but cannot modify it
3. need the frame pointer and can modify it

We want to consider not single instructions but basic blocks as a whole, and therefore we classify the blocks in a similar manner. Basic blocks that contain at least one instruction that uses the frame pointer (instruction type 2 or 3) are called FPU blocks (Frame Pointer Use blocks).

All other blocks are non-FPU blocks.
2. Problem description

Control flow graphs (CFG)

A Control Flow Graphs is a directed graph representing program flow. The nodes of the CFG are basic blocks as described above, and the edges illustrate how the blocks can be executed succeeding each other.

Any function can be described using a set of local variables and a CFG.

In addition, each block has a use-estimate. This use-estimate will be used in our algorithm for weighing the importance of alternatives if conflicts arise when determining the frame pointer values. The use-estimate can be determined through experiments or code analysis.

![A sample CFG](image)

**Figure 1.** A sample CFG
3. Algorithm

Preparations

As stated before, in the upcoming algorithm we want to consider the basic blocks and not the individual instructions. To get on a higher of abstraction we perform the following premodification on our code:

For all FPU blocks:
   For all FP-using instructions in block:
      Set post modification value of this instruction to the difference between the current frame pointer value and the value required by the next instruction.
   }
}

If an instruction cannot modify the frame pointer, we insert an explicit instruction. After these modifications, we can consider the basic blocks as independent code segments with in- and out-values for the frame pointer.

The in-value of a block \( b \) should be the out-value of the predecessor block(s), and the out-value of block \( b \) should be the in-value of the successor block(s). The task of determining the in- and out-values (that is, the modification of the frame pointer \textit{between} blocks) would be simple if we knew how the program flows. But as we can see in Figure 1, a simple function with perhaps a while loop and an if-else statement, the program flow is often non-deterministic and because of these complex requirements we introduce a new data structure, which in our case is easier to work with: the Control Flow Line Graph (CFLG).
3. Algorithm

Control flow line graphs (CFLG)

Given CFG edges \( f \) and \( g \), we can say \( f \) and \( g \) are related to each other with a relation \( R \):

\[ f \ R \ g \iff f \text{ and } g \text{ have the same predecessor block or the same successor block.} \]

\( R^* \) is transitive closure of \( R \). In general, transitive closure can be defined as follows:

Given a graph \( G = (V, E) \), transitive closure for \( G \) is \( G^* = (V, E^*) \), such that for all nodes \( v, w \) in \( V \), there is an edge \((v, w)\) in \( E^* \) if and only if there is a non-null path from \( v \) to \( w \) in \( G \). This relation is illustrated by the figure below.

With the constraint that our relation \( R \) only spans one edge at a time, we can define a \textit{CFG edge class} as all edges \( f, g \) for which \( f \ R^* \ g \). For the control flow line graph, we intuitively swap the meaning of nodes and edges in the control flow graph, so that:

- CFLG edge = CFG node
- CFLG node = CFG edge class

![Transitive closure in general](image)

\textbf{Figure 2.} A graphical presentation of transitive closure in general
3. Algorithm

CFG becomes CFLG

Minimizing cost of local variable access for DSP-processors - Mathias Johnson 2002
3. Algorithm

Frame pointer values

We can now say that each CFLG node has at least one in-value and at least one out-value for the frame pointer from the CFLG edges, representing basic blocks. But as the CFLG nodes are just crossroads in our code, we cannot use and pass on all in- and out-values. For each CFLG node, we need to determine from all the suggested values which of them to use, and then modify the in- and out-values of the surrounding edges (basic blocks) as necessary.

With the post-modify addressing mode available, we can usually modify the out-value of a block without overhead. In many cases, we can also modify the frame pointer of a basic block from an arbitrary in-value before the first instruction actually using the frame pointer comes up. Only when post modify is not available, we must add an explicit instruction.

We could set the value of all CFLG nodes to any value - this would just mean that we would have to insert many overhead instructions to adjust the frame pointer as required. The resulting program would still work. The goal of our algorithm is to find the optimal frame pointer value for each CFLG node, thereby minimizing the amount of explicit frame pointer modification instructions, weighted by use estimate of the corresponding blocks. Taking the use-estimates into account when deciding which frame pointer to use means minimizing the cost of modifying the frame pointer.
3. Algorithm

Algorithm

Initial stage

First, all CFLG node values are set to ‘undefined’ with use-estimate zero. Non-FPU edges are assigned in- and out-values ‘undefined’. For FPU edges, the in-value is the frame pointer value required by the first instruction that uses the frame pointer. The out-value is set to ‘undefined’ if the last instruction of the block can modify the frame pointer, otherwise it is set to the last frame pointer value of the block.

Main loop

The main loop visits all CFLG nodes and is repeated until no changes are made. At each node, the value for the frame pointer is calculated. This is done by collecting the out-values of the predecessor edges and the in-values of the successor edges, together with the corresponding use-estimates. If the frame pointer value for two edges is equal, the use-estimates of the two edges are added together. The frame pointer value for the node is then set to the value with the highest use-estimate.

Final stage

Now all we need to do is to modify the frame pointer values according to the CFLG nodes. This is done by adjusting the post modify instructions at the beginnings and ends of the edges (basic blocks). In some cases, we need to insert explicit instructions to modify the pointer, but as we chose the pointer value with the highest estimate, the penalty cost for these overhead instructions is kept to a minimum.
4. Results & summary

Improvement

In the following charts, the improvement of using the algorithm above is illustrated. The implementation was tested using a number of commonly used DSP applications and some other applications. The reference is an algorithm that performs the frame pointer modification within the blocks but does not modify the CFLG node values.

The improvements are higher when the variables are on the stack (direct memory access architecture) than when they are in registers (load store architecture). This is because in the latter case, only few stack accesses are made and the impact of the floating frame pointer algorithm is smaller.
4. Results & summary

**Improvement**

![Bar chart showing execution time, variables on stack for various algorithms and averages.](chart.png)
4. Results & summary

Summary

Our problem was how to access local variables efficiently using dynamic function frame allocation on DSPs? Using static memory is not desired because of several disadvantages, among them that the compiler becomes non-compliant to the language standard.

The presented solution is based on the idea of a floating frame pointer, always pointing to the address of the variable that is to be accessed by the next instruction and defined at all times.

An at least near-optimal algorithm for frame pointer value assignment was found using Control Flow Line Graphs with use-estimates for basic blocks.

This presentation was based on the following article: