Minimizing cost of local variables access for DSP–processors

Erik Eckstein and Andreas Krall

Atair Software GmbH
Esslinggasse 18
A–1010 Wien
Eckstein@atair.co.at

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A–1040 Wien
andi@complang.tuwien.ac.at

Abstract

Recent work on compilation for DSP-processors deals with optimizing access to local variables of functions. The common way is to use one or more address registers as pointers into the functions stack frame and modify it with post modify addressing modes (which are sometimes the only addressing modes). Additionally to previous work we present an algorithm which assigns frame pointer values over a whole procedure. Our algorithm also deals with basic blocks, which have no accesses to local variables. The algorithm works with a new data structure, the control flow line graph, which is derived from the control flow graph. In our experiments, the algorithm showed improvements to similar algorithms.

1 Introduction

Many DSPs contain addressing units, which can post increment and post decrement address registers after accessing the memory. An addressing mode which can access the memory with a constant (immediate) offset to the address register is very rare in DSP architectures ('register + offset' addressing mode).

A high level language with subroutines like C requires to allocate memory for local variables. These variables are stored in the function frame which usually is allocated on the program stack to minimize the amount of memory needed for local variables and to allow recursive functions. Most C compilers store the frame address in a frame pointer register and access local variables with a 'register + offset' addressing mode. Due to a limited amount of address registers on a DSP and the missing of a 'register + offset' addressing mode this concept is hard to apply on DSP compilers. It would require to waste one register for the frame pointer and to explicitly execute an address computation instruction for every memory access.

Many DSP compilers overcome this problem with static function frames. Instead of allocating the function frames on the program stack they are allocated static. The program stack is only used for passing function arguments. This approach has three significant disadvantages:

- No recursive functions are allowed. Although recursive functions are very rare in DSP applications, it makes the compiler non compliant to the language standard (e.g. ANSI-C).
- Local variables must be accessed with immediate address loads. On most DSP architectures this addressing mode is slower than accessing variables via address registers.
- By static function frame allocation much space is wasted, because frames of function siblings are not active simultaneously. Global call tree analysis combined with sharing of function frames can

avoid this disadvantage.

Our approach is to use a floating frame pointer. A floating frame pointer is not constant inside a function but points to the next memory location to be accessed. After the memory access the frame pointer is modified using post increment/decrement addressing modes. This approach saves one register for the constant frame address and saves the instructions necessary for an explicit address computation at every memory access.

We describe related work in section 2. After presentation of the problem description in section 3, we introduce a new data structure, the control flow line graph (CFLG) in section 4. We show that the floating frame pointer problem can be solved more easily on a CFLG than on the original program and present a concrete algorithm. Section 5 gives an empirical evaluation of our algorithm.

2 Related work

The placement of variables in memory has a significant impact on code size and run time on DSP processors which only support autoincrement/autodecrement addressing modes. The optimization of placement of variables has been first studied by Bartley [Bar92]. He solved the simple offset assignment problem (SOA) where optimal frame offsets of variables within a function are computed using only one address register and only autoincrement and autodecrement addressing modes. Bartley based his algorithm on finding a maximum-weight Hamiltonian path on the access graph.

Liao et al. [LDK+96] showed that the simple offset assignment problem is equivalent to the maximum weighted path covering problem and proved that it is NP-complete. They showed that the solution can be extended to the general offset assignment problem (GOA) which handles a fixed number of address registers and proposed an efficient heuristic to solve the problems. Sudarsanam et al. [SLD97] studied the offset assignment problem with autoincrement/autodecrement values bigger than one and a fixed number of address registers.

Leupers and Marwedel [LM96] extended the work done by Liao et al. by proposing a tiebreaking heuristic and a variable partitioning strategy. They also used modify registers to reduce the the access costs of variables. Leupers and David [LD98] proposed a genetic algorithm to solve the general offset assignment problem for increment/decrement values greater than one.

Rao and Pande [RP99] present techniques to optimize the access sequence of variables by applying algebraic transformations on expression trees to obtain the least cost offset assignment.

In his thesis Liao [LDK+96] also developed heuristics for offset assignment across basic blocks. He extended the simple offset assignment algorithm taking into account the usage counts of basic blocks and control flow edges. A problem with Liaos algorithm is that it does not deal with basic blocks without stack frame memory accesses. On load/store architectures it is quite common that basic blocks do not contain stack frame memory accesses, because all local variables are in registers. But even for these blocks a frame pointer value must be assigned. Liao's algorithm computes an optimal solution for a local region in the control flow graph by evaluating the costs of all possible placements of modify instructions for that region. This leads to an optimal solution for the whole function with the condition that every basic block contains at least one memory access.

3 Problem description

The target architecture of our compiler (NEC uPD77016 DSP) is a load/store architecture. The addressing unit of the DSP can address two different memory spaces and supports post modification of address registers with 16 bit immediate values. This makes it possible to reach any data location with a single post modify instruction. Therefore, it is not necessary to perform a storage assignment algorithm, like SOA and GOA, previous to our algorithm. The compiler uses two address registers for stack access, one for each memory space.

The purpose of our algorithm is to calculate the value of the frame pointer for every location (every single instruction) in the function. This implies that the frame pointer value is defined and unique at a given location in the function (i.e. it can not be different in two iterations of a loop at a certain location). The frame pointer value at funtion's entry has to be equal to the one when returning from the function. Assignment of memory spaces and stack frame offsets must be done prior to our algorithm. The algorithm runs for each address register separatly.

At every instruction where the frame pointer is used, the frame pointer has to hold a specific value (i.e. the address of a local variable). At all other instructions in the function, the algorithm is free to select any value for the frame pointer. Values should be selected in a way to minimize the necessary frame pointer modification instructions. Fortunately most DSP architectures provide post increment/decrement addressing modes with which the frame pointer can be modified without any performance losses. Therefore, the algorithm should modify the frame pointer at memory accesses according to the following memory access.

Not all frame pointer modifications can be combined

with memory accesses. There are instructions, which require the frame pointer to hold a specific value, but can not post modify the frame pointer(e.g. function calls). In this case it can be necessary to insert an explicit frame pointer modify instruction. If the frame pointer modification is needed before a memory access, it has to be made explicit, too.

A function consists of a set of local variables and a control flow graph (CFG). The nodes of the CFG are basic blocks, which consist of a list of assembler instructions. There are three types of instructions:

1. instructions, which do not need and do not modify the frame pointer.
2. instructions, which need the frame pointer to hold a specific value, but can not modify the frame pointer.
3. instructions, which need the frame pointer to hold a specific value and can modify the frame pointer. These instructions are all instructions which access memory in the stack frame.

We will call instructions of type 2 or 3 frame pointer instructions. Basic blocks which contain at least one frame pointer instruction are called frame pointer use blocks (FPU blocks), all other blocks are called non-FPU blocks.

Each basic block has an attribute use-estimate, which holds the estimated basic block usage count. This count is either estimated or computed by profiling. For code size optimization the count is set to one.

4 The algorithm

First all FPU blocks are handled, which contain two or more frame pointer instructions, because in the following algorithm we want to concentrate on the frame pointer behavior between blocks and not within a block. The frame pointer instructions within a basic block are located in a linear chain and therefore it is very easy to calculate the increment instructions between them:

The instruction list of the basic block is traversed from the first to the last but one frame pointer instruction. At each frame pointer instruction which can modify the pointer, the modification value is set to the difference to the next frame pointer instruction's value. If the frame pointer instruction can't modify the pointer, an explicit modify instruction must be inserted.

After this first step, all FPU blocks can be seen to require a frame pointer value at the entry of a block (= frame pointer value of the first frame pointer instruction of the block) and to hold a frame pointer value at exit of block exit (= frame pointer value of the last frame pointer instruction of the block).

We define the in-value of a basic block as the frame pointer value at the beginning of the block and the out-value of a basic block as the frame pointer value at the end of the block.

The frame pointer value at the end of a basic block b is propagated to all successor blocks of block b. Therefore the frame pointer in-value of all successor blocks of a block b is the same as the out-value of block b. Similarly the frame pointer out-value of all predecessor blocks of a block b is the same as the in-value of block b. To deal with these constraints the algorithm doesn't work on a control flow graph but on a control flow line graph.

4.1 The Control Flow Line Graph

The control flow line graph (CFLG) is constructed from the control flow graph. The CFG edges are partitioned into edge classes. A relation R over the CFG edges is defined as follows: For CFG edges f and g, f R g if and only if f and g have the same predecessor block or have the same successor block. R* is the transitive closure of the relation R. An edge class contains all edges which are equal corresponding to the equivalence relation R*. For each edge class a CFLG node is created.

A CFLG edge corresponds to a CFG node which connects two CFG edge classes. The definition of the CFLG makes it impossible, that a CFLG edge (= CFG node) connects more than two CFLG nodes (= CFG edge classes). Intuitively it can be said, that the CFLG is the CFG with reversed meaning of nodes and edges. An example of an CFG and its corresponding CFLG is shown in figure 1 and 2.

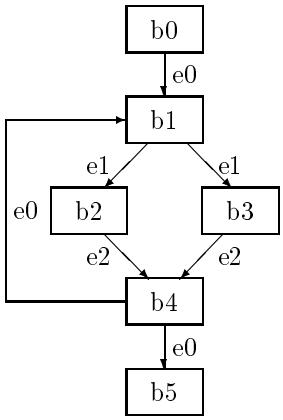

Figure 1: Control flow graph

Building the CFLG is quite simple. The nodes of the CFLG are found by starting at one CFG edge and

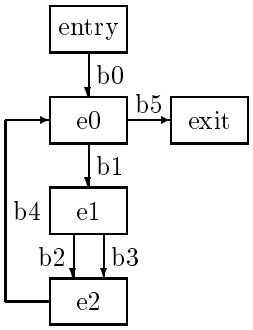

Figure 2: Control flow line graph

recursively collecting all edges of the successor edges of the edge's predecessor block and the predecessor edges of the edge's successor block in the CFG. This procedure is repeated until all edges of the CFG are handled. In a second step for each CFG node a CFLG edge is created, which is connected to the appropriate CFLG nodes (= CFG edge classes). A pseudo code of a procedure BuildCFLG is shown in figure 3.

4.2 Calculating the frame pointer values

A frame pointer value has to be assigned for each CFLG node. The CFG blocks, which correspond to the CFLG edges have to expect the frame pointer value of the predecessor CFLG node as in-value and the frame pointer value of the successor CFLG node as out-value. If the in-value of a block differs from the value of the first instruction of the block, an explicit frame pointer modification instruction must be added at the beginning of the block. The same holds for the end of a block: If the out-value of a block differs with the value of the last instruction of the block, a modification instruction must be added at the end of the block. But if the last instruction can modify the frame pointer with a post-modify addressing mode, no overhead is generated.

The method of constructing a CFLG assumes that it is not possible to create explicit frame pointer modification instructions on critical CFG edges. A critical CFG edge is an edge where the predecessor block has more than one successor and the successor block has more than one predecessor. Inserting an instruction on a critical edge requires to create a new block on this edge. This results in an additional jump instruction in the compiled program, if the critical edge is not an fall-through edge. If the block layout is already known, basic blocks can be inserted in critical fall-through edges without additional jump costs. This makes the fall-through edge non-critical. In the used framework due to other optimizations the final basic block layout is determined at a later stage. Therefore it is not known, which edges are fall-through edges. Experiments showed that inserting additional basic blocks on critical edges is a bad choice. An algorithm, which could insert modification instructions on critical edges, produced results which were not much better or even worse than a naive algorithm performing on a CFLG.

An arbitrary algorithm can be used to assign frame pointer values to CFLG nodes. It should be said, that the correctness of the result does not depend on the selected algorithm. Regardless, which values are assigned to CFLG nodes, the compiled program will always work. The first attempt could be to assign the value 0 to all CFLG nodes. Our goal is to find a near optimal algorithm. Optimal means that the amount of explicit frame pointer modification instructions, weighted by the use-estimates of the container block, will be reduced to a minimum.

4.3 Algorithm

We implemented an iterative algorithm similar to iterative dataflow problem solving algorithms [ASU86].

In the following CFLG edges, which are associated to FPU blocks are called FPU edges, all other edges are called non-FPU edges. Each CFLG node holds a current frame pointer value, in the following called value, and a current use-estimate. First all CFLG nodes are initialized to a pseudo-value 'undefined' with use-estimate zero. Each CFLG edge has an in- and an out-value. For non-FPU edges both values are initialized to 'undefined'. For FPU edges, the in-value is set to the first frame pointer value of the associated block. The out-value is set to 'undefined', if the last instruction of the associated block can modify the frame pointer, else the out-value is set to the last frame pointer value of the block. Intuitively it can be said that all CFLG edges, connected to a node with a defined value, 'require' that value in the node, otherwise an explicit increment instruction must be inserted.

The main loop visits all CFLG nodes and continues until no more changes are made on the CFLG. At each node a new value for the node is calculated. This is done by collecting all in-values of outgoing edges and all out-values of incoming edges, which are not undefined. If all such values are undefined, the node value is kept undefined and no further processing is done for this node in this iteration. Otherwise the values are weighted with the use-estimates of the corresponding blocks (use-estimates of equal values are added up). If the maximum use-estimate is greater than the current use estimate of the CFLG node, the value with the maximum use-estimate is taken as the new value for the CFLG node and the use-estimate of the node is set to the new use-estimate. Taking the maximum means,

that the penalty for explicit frame pointer modification instructions, which must be inserted at the node's connected edges, is kept to a minimum.

If the new assigned node value differs from the old node value, the changed flag is set and the new value is propagated along the non-FPU edges. That means that the in-values of all predecessor non-FPU edges and the out-values of all successor non-FPU edges are set to the new node value.

The use-estimate of a node can be at most the sum of the use-estimates of all connected CFLG edges. On the other hand the use-estimate of the node in a successive iteration of the algorithm is higher than in a previous iteration, because a new value is only set if the new use-estimate is greater than the old use-estimate of the node. This means that the use-estimate of the node value is always increasing, but limited to a maximum value. Therefore it is guaranteed, that the algorithm will terminate within a limited number of iterations. In practice only very few iterations are necessary, because in most cases an already set value will not change to another value in successive iterations.

After no more changes occur during the main loop, the frame pointer modification instructions can be inserted. There can be three cases, where explicit modification instructions are necessary:

- for FPU edges, where the in-value differs from the value of the connected predecessor node: A modification instruction must be inserted at the beginning of the edge's associated block. The modification value is the difference between in-value and predecessor node value.
- for FPU edges, where the out-value differs from the value of the connected successor node and the last frame pointer instruction of the associated block can not modify the frame pointer: A modification instruction must be inserted at the end of the edge's associated block. The modification value is the difference between successor node value and out-value.
- for non-FPU edges, where the out-value differs from the in-value: A modification instruction must be inserted in the associated block. The modification value is the difference between the out-value and the in-value.

Figure 4 shows the algorithm in pseudo code form.

5 Results

We tested our algorithm with various benchmark programs. Our benchmarks include important DSP applications: FIR, IIR, Viterby and FFT algorithms. Beside this we took standard benchmark algorithms: the sieve algorithm to compute prime numbers, binary search, quick sort and heap sort. In addition the benchmark includes a set of 911 general ANSI C test files from a validation suite. We made two test cases: first we let the compiler put local variables into registers (as usual), which minimized the stack access frequency. Secondly we modified the compiler, so that it put all local variables onto the stack. With this test we simulated the memory access behavior of a direct memory access architecture, which has a high stack access frequency.

We compared our algorithm, named heuristic2, to two simpler algorithms. The naive algorithm puts the frame pointer to the value 0 at basic block borders. The second algorithm, named heuristic1, calculates correct values at basic block borders, but does not take non-FPU blocks into account. All of the test algorithms perform the floating frame pointer technique within basic blocks.

If we compare our algorithm (heuristic2) to the naive algorithm, the improvements are higher, if all local variables are on stack (corresponds to a direct memory access architecture). When local variables are in registers (load store architecture), only few stack accesses are made, especially in the inner loops. Therefore the impact of an floating frame pointer algorithm is not so high.

On the other hand, if we compare our algorithm (heuristic2) to the heuristic1 algorithm, we observe that the improvements are higher, when the compiler can allocate variables in registers. This is because in the other case - all variables on the stack - nearly every basic block contains at least one stack access and there are hardly no non-FPU blocks. In this case the two algorithms are nearly equivalent.

6 Conclusion

We showed a method of transforming the CFG of a function into a new data-structure, a CFLG. Then we solved the floating frame pointer problem on the CFLG using a heuristic algorithm which solves the problem with an iterative approach.

The concept of the CFLG can be used to solve a class of related problems. These are all problems like placement of mode changes inside a function. With a CFLG it is guaranteed that no mode changes have to be put on CFG edges (which is clearly impossible) rather than on CFG blocks.

The algorithm both showed good test results and is very easy to implement.

```
procedure BuildCFLG (CFG cfg)
    e.node := nil   ∀ edges e ∈ cfg
    for each edge e ∈ cfg do
        if e.node = nil then
            create CFLG-node node
            AppendSourceEdges(e, node)
            AppendTargetEdges(e, node)
    for each blocks b ∈ cfg do
        create CFLG edge e between node of first predecessor edge of b
        and node of first successor edge of b
        e.block := b

procedure AppendSourceEdges(e, node)
    e.node := node
    for each successor edges se ∈ predecessor block of edge e do
        if se.node = nil then
            AppendTargetEdges(se, node)

procedure AppendTargetEdges(e, node)
    e.node := node
    for each predecessor edges pe ∈ successor block of edge e do
        if pe.node = nil then
            AppendSourceEdges(pe, node)
```

Figure 3: Pseudocode for control flow line graph construction

	naive	heuristic1	heuristic2	gain(1/naive)	gain(2/1)
FIR	3546	3546	3546	0%	0%
IIR	5077	5076	4954	2.42%	2.40%
Viterby	766	771	747	2.48%	3.11%
FFT	39873	39098	38845	2.58%	0.65%
sieve	313643	313643	313642	0%	0%
binsearch	6683	6719	6683	0%	0.54%
qsort	677032	690598	666341	1.58%	3.51%
hsort	839058	828435	808611	3.63%	2.39%
suite	908625647	932556639	896404241	1.35%	3.88%
average				1.56%	1.94%

Table 1: Execution time in cycles, local variables in registers

```
procedure AssignFramePointerValues(CFLG ceg)
   for all edges e do:
      if e.block is FPU then
         e.invalue = first value of e
         if can modify last value of e.block then
            e.outvalue = undefined
         else
            e.outvalue = last value of e.block
         endif
      else
         e.invalue = undefined
         e.outvalue = undefined
      endif
   endfor
   for all nodes n do:
      n.value = undefined
      n.useestimate = 0
   endfor
   changed = true
   while changed do:
      changed = false
      for all nodes n do:
         initialize vector V to all elements zero
         for all successor edges se of n do:
            if se.invalue is not undefined then add se.useestimate to element value of V
         endfor
         for all predecessor edges pe of n do:
            if pe.outvalue is not undefined then add pe.useestimate to element value of V
         endfor
         find element (maxvalue) with maximum useestimate (maxuseestimate) in V
         if maxvalue != n.value and maxuseestimate > n.useestimate then
            changed = true
            n.value = maxvalue
            n.useestimate = maxuseestimate
            for all successor edges se of n do:
               if se.block = non-FPU then se.invalue = maxvalue
            endfor
            for all predecessor edges pe of n do:
               if pe.block = non-FPU then pe.outvalue = maxvalue
            endfor
         endif
      endfor
   endwhile
```

Figure 4: Pseudocode for frame pointer value assignment

	naive	heuristic1	heuristic2	gain(1/naive)	gain(2/1)
FIR	62084	57485	57485	7.41%	0%
IIR	21800	20449	20449	6.20%	0%
Viterby	1224	1151	1127	7.92%	2.09%
FFT	53419	52838	52838	1.09%	0%
sieve	536969	487304	487303	9.25%	0%
binsearch	7316	7236	7236	1.09%	0%
qsort	927001	880805	876237	5.48%	0.52%
hsort	1231951	1164716	1159392	5.89%	0.46%
suite	1010759169	998438385	962288465	4.80%	3.62%
average				5.49%	0.74%

Table 2: Execution time in cycles, local variables on stack

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bar92] D. H. Bartley. Storage assignment to decrease code size. *Software – Practice & Experience*, 22(2):101–110, 1992.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[LD98] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In *ISSS '98*, December 1998.

[LDK+96] Stan Liao, Srinivas Devadas, Kurt Keutzer, Stevent Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, 1996.

[Lia96] Stan Liao. *Code Generation and Optimization for embedded Digital Signal Processors.* PhD thesis, Massachusetts Institut of Technology, 1996.

[LM96] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in dsp code generation. In *International Conference on Computer-Aided Design (ICCAD)*, November 1996.

[RP99] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *1999 SIGPLAN Conference on Programming Language Design and Implementation.* ACM, June 1999.

[SLD97] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. Analysis and evaluation of address arithmetic capabilities in custom dsp architectures. In *Design Automation Conference*, pages 287–292. ACM/IEEE, 1997.