Addressing Mode Selection

Erik Eckstein
ATAIR Software GmbH,
Vienna, Austria
eckstein@atair.co.at

Bernhard Scholz
Vienna University of Technology
Vienna, Austria
scholz@complang.tuwien.ac.at

Abstract

Many processor architectures provide a set of addressing modes in their address generation units. For example DSPs (digital signal processors) have powerful addressing modes for efficiently implementing numerical algorithms. Typical addressing modes of DSPs are auto post-modification and indexing for address registers. The selection of the optimal addressing modes in the means of minimal code size and minimal execution time depends on many parameters and is NP complete in general.

In this work we present a new approach for solving the addressing mode selection (AMS) problem. We provide a method for modeling the target architecture's addressing modes as cost functions for a partitioned boolean quadratic optimization problem *(PBQP). For solving the PBQP we present an efficient and effect way to implement large matrices for modeling the cost model.*

We have integrated the addressing mode selection with the Atair C-Compiler for the uPD7705x DSP from NEC. In our experiments we show that the addressing mode selection can be optimally solved for almost all benchmark programs and the compile-time overhead of the address mode selection is within acceptable bounds for a production DSP compiler.

1 Introduction

In contrast to RISC architectures, embedded CPU architectures have addressing generation units facilitating a variety of addressing modes. For example digital signal processors (DSP) have powerful addressing modes for efficiently implementing signal processing applications. While there is a shift from assembly code to codes written in a high-level language such as C, it is the task of the compiler to generate highly optimized code that fully exploits addressing mode features of the CPU. In the past only little work has addressed this problem.

In this paper we propose a compiler optimization called *Addressing Mode Selection* (AMS) that selects the optimal addressing mode for addressing registers in the input program. The optimization can be parameterized for different objectives such as execution time and code size.

```
(1)  loop {
(2)    ar0++
       if (c) {
(3)      *ar0
       } else {
(4)      *(ar0 + 1)
       }
(5)    *(ar0 + 2)
(6)  }
```

(a)

$\Longrightarrow$

```
(1)  ar0++
(2)  loop {
       if (c) {
(3)      *(ar0 += 2)
       } else {
(4)      ar0++
(5)      *ar0++
       }
(6)    *ar0--
     }
(7)  ar0--
```

(b)

Figure 1. Running Example

Consider the pseudo code of our running example in Figure 1(a). The goal is to optimize the addressing modes of register `ar0`. The underlying target architecture supports indirect addressing mode `*(ar)`, post modification addressing mode `*(ar0++)`, `*(ar0--)`, `*(ar0+=c)`[1], and indexing addressing mode `*(ar+c)`.

For sake of simplicity we assume that post modification can be executed without additional overhead. The indexing addressing mode (`ar + c`) has a worse pipeline characteristics than post modification and a longer instruction encoding as well. An explicit add instruction for address register `ar0` needs considerable more time than employing addressing modes and should be avoided in general.

In the example of Fig. 1(a) the loop is executed 10 times and the condition `c` is true in 7 iterations and false in 3 iterations. The optimal output program for minimal execution time is shown in the same figure on the right-hand side. The

[1]In contrast to C++ or Java, the += operator has a post modification semantic in this paper.

add instruction can be moved out of the loop and post modification addressing modes can be used instead of indexing addressing modes in line (5) and (6). An explicit add instruction for the address register must be prior to the loop and in line (4), but this takes less cycles than the code given in the original program.

In our experience we have seen that the AMS is a very important problem for embedded system processors. We have integrated AMS with an industrial C-Compiler for the NEC uPD7705x digital signal processor family and used typical digital signal processing applications. The experiments show that code-size reductions up to 50% and speedups of more than 60% are achievable.

As far as the authors are aware of there is only little previous work dedicated to AMS. An approach to the problem is presented in [2] where the AMS is embedded in the problem of live range merging. Live range merging tries to utilize all available address registers to minimize update instructions in a loop. The used technique is called live range growth (LRG) and uses a merge operator which determines the costs of merging. This merge operator is the crucial point in the algorithm and in fact performs addressing mode selection for determining the costs of merging. A ϕ-dependence graph, which is derived from a static single assignment form (SSA) is employed for solving the problem. The limitations of this work is that the algorithm only works for a single loop and heuristics are applied if the ϕ-dependence graph is not a tree. Another approach is presented in [5] that also is limited to the scope of a loop. In the work of [1], the problem of addressing mode selection is solved for the scope of a whole control flow graph. However, the used algorithm imprecisely models the addressing mode selection and may produce sub-optimal results. Moreover, there is no precise model of costs in [2, 5, 1]. In the context of AMS there are related problems that are not addressed in this work and can be seen as orthogonal techniques like live range merging [2] and offset assignment [3, 7, 6]. In the work of Surdarsanam [4] care is taken for assigning offsets for exploiting the addressing modes. Still this technique can be seen as pre-pass of our approach.

The AMS turns out to be a hard problem mainly for two reasons: First, even if we assume a very simple architecture, the AMS is NP complete, which was shown by [2]. Second, in practice the addressing modes of embedded system processors are very manifold and in-homogeneous. To overcome the obstacle of previous work our new approach has following features: (1) a flexible cost model for different optimization goals, e.g. runtime and code-size, (2) a wide range of different addressing modes can be modeled, (3) the scope of the optimization is a whole CFG, not only a loop, and (4) the experiments indicate that AMS can be optimally solved for nearly all programs in the benchmark suite.

The structure of the paper is as follows. In Section 2 the model of cost for the AMS is presented. In Section 3 the algorithm for the optimization is introduced. In Section 4 we present our experiments. In the last section we draw our conclusion.

2 Modeling of the AMS problem

The AMS problem can be performed for every address register of the target architecture separately, achieving the same final result. In the following we refer to the currently optimized address register as the *address register*. The input for the AMS algorithm is the control flow graph (CFG) of a program whereby each CFG node represents an instruction. For each node in the CFG, the AMS algorithm decides which addressing mode is the best based on a cost model for each node. This decision cannot be done locally as demonstrated in our running example of Figure 1.

For a better understanding we give some examples of widely used addressing modes and explain how to construct a cost model for a simplistic CPU architecture (in Section 4 we show results of a complex full-blown embedded system processor). An addressing mode is the method of generating an address in the address generation unit of the target architecture. The addressing mode defines how the address is calculated from the address register value and how the address register is modified when generating the address[2].

The basic addressing mode is indirect addressing. The memory is accessed at the address value of the address register and the address register is not modified. We denote the address value as the *access value*. E.g., a load from memory into general purpose register `r1` is given by `r1 = *ar0`.

The input program may contain instructions that use an address register, but do not access memory. For example moving an address value to a general purpose register (`r1 = ar0`). For the AMS model it does not matter if an instruction performs a memory access or not. Therefore, for the model such instructions are equivalent to indirect addressing. The access value is the value of the address register in the input program, although there is no memory access.

Post increment and post decrement addressing modes are basically the same as the indirect addressing mode, except that the address register is incremented or decremented after the memory access, respectively. Usually the increment or decrement value is equal to the access size in memory. These addressing modes are useful for sweeping over an array in a loop. Examples are `r1 = *ar0++` or `r1 = *ar0--`.

A more general form is the post modification addressing mode. In contrast to the post increment/decrement address-

[2]We do not consider addressing modes which do not involve address registers, like direct addressing.

ing modes, the modification value can be specified explicitly within a given range, e.g. `r1 = *(ar0+=2)`. The available range for the modification value depends on the architecture. Usually it is only a subset of the whole address space. The drawback of this addressing mode is that it needs more coding space compared to the post increment/decrement modes.

Some architectures provide an indexed addressing mode. The access value is obtained by adding a constant offset to the address register value, but the value of the address register is not changed, e.g. `r1 = *(ar0+2)`. The indexed addressing mode implies an addition before accessing the memory that may result in pipeline hazards.

In many architectures both the post modification and indexed addressing modes are also available with an offset register instead of an immediate value as shown in the following example: `r1 = *(ar0+=r0)` and `r1 = *(ar0+r0)`. The offset register variants need some special treatment in the cost model by defining pseudo values for the offset registers in the value domain[3].

For the AMS problem, add instructions have a great potential for optimizing the code. First, add instructions might be eliminated by address modes. For example the code sequence `r0=*ar;ar++;` can be replaced by `r0=*ar++`, which saves one instruction. Second, an add instruction might be inserted by the AMS algorithm to optimize the program at another place that is executed more frequently. E.g., an add instruction is inserted at node (4) in Figure 1 for obtaining a better code inside the more frequently executed then-branch of the if-statement. In contrast to addressing modes, add instructions do not have an access value.

Definitions of address register have to be handled by the AMS as well. Definitions, e.g. initializing an address register before a loop, are different from addressing modes and add instructions because the value of the address register is arbitrary before the definition. Note that the access value of a definition is the value that is assigned to the address register.

Finally, nodes in the CFG, which do not contain address registers, can be treated as an add instruction with zero increment. If the AMS algorithm selects a constant different from zero, an explicit add instruction must be inserted.

Let us consider two subsequent statements `stmt1;stmt2`. Both statements use address register `ar` for accessing memory and modify the value of the address register. Now, the principle idea of AMS is to shift the value of `ar` between those two statements. We replace `stmt1` by `stmt1'` and statement `stmt2` by `stmt2'`. The semantics of `stmt1'` and `stmt2'` are given by `stmt1;ar+=`Δ and ar-=Δ;stmt2, respectively. Depending on the offset value Δ, cheaper addressing modes might be employed for both statements. This is

[3]This extension of the model is not further elaborated in this work.

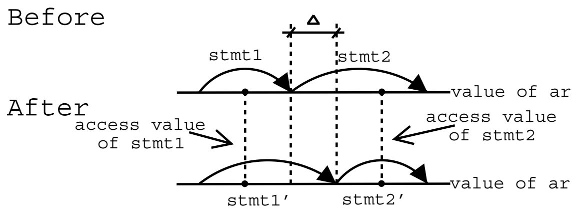


Figure 2. Shift of Address Register

achieved by replacing the increment of the address register and the prior address operation by a cheaper addressing mode (also known as strength reduction).

In Figure 2 the shift of `ar` is illustrated. The first graph depicts the modification and memory access with address register `ar`. The access values of both statements are also given. In the second graph (below the first one) we perform the shift of address register `ar`. In the figure we can see that access values of both statements will not change, albeit the value of `ar` between both statements has changed.

For a statement we have two offset values: One offset relates to the relative address shift before the statement — the other offset relates to the relative address shift after the statement. Based on this observation we introduce the definition of *addressing mode* as follows.

Definition 1 *Addressing mode am is a set of value pairs $\{\langle e_1, x_1\rangle, \ldots, \langle e_n, x_n\rangle\}$. Each pair defines an entry and exit offset for the address register.*

The definition above gives for an addressing mode a set of permitted entry and exit offsets. This means, an original statement `stmt` is rewritten by a new statement `stmt'` that employs an addressing mode. The program semantics of `stmt'` is given by `ar-=`e`;stmt;ar+=`x for a pair $(e, x) \in am$. Note that some of the addressing modes can be parameterized and therefore more than one pair of offset values might be in am. For example an indirect memory access `*(ar)` may be replaced by the post-increment memory access `*(ar++)`. The set am of `*(ar++)` contains only one pair, i.e. $< 0, 1 >$ since the addressing mode is not parameterizable. Contrarily, the post-increment memory access with a constant, i.e. `*(ar+=c)`, induces more than one pair. The number of pairs is given by number of elements in the number domain of `c`.

In the following table we list examples of addressing mode sets for the previously introduced addressing modes. We assume that the original statement is a simple indirect memory access, e.g. `r0=*(ar)`.

Indirect addressing:
$am(\texttt{*ar}) = \{\langle 0,0\rangle\}$
Post increment:
$am(\texttt{*ar++}) = \{\langle 0,1\rangle\}$
Post decrement :
$am(\texttt{*ar--}) = \{\langle 0,-1\rangle\}$
Post modification for $c \in \{l,\ldots,u\}$:
$am(\texttt{*(ar+=c)}) = \{\langle 0,l\rangle,\ldots,\langle 0,u\rangle\}$
Indexing for $c \in \{l,\ldots,u\}$:
$am(\texttt{*(ar+c)}) = \{\langle -l,-l\rangle,\ldots,\langle -u,-u\rangle\}$

Note that variables l, u gives a number range for c of the post modification addressing mode and the indexing addressing mode. This number range is normally smaller than the addressing domain of the CPU in order to keep the instruction word small.

The addressing modes of statement `r0=*(ar)` can be easily adopted for other statements. For example if the original statement is a indexing addressing mode, i.e. `r0=*(ar+c)`, the values of the pairs are simply shifted by the constant `c`.

Another class of statements are statements which have no side-effect on the address register and do not access memory with `ar`. As already mentioned this statements can be treated as add instructions with zero increment. The addressing mode is given as follows

$$am(\texttt{ar+=c}) = \{\langle i, i+c\rangle \,|i \in D\}$$

where D is the domain of available values of the address register Usually D contains all values an address register may contain, e.g. $|D| = 2^{16}$ for 16 bit address registers. As the set D is very large, the implementation of the cost model, needs to be represented in a compact form (see Section 3).

For statements, which contain an add instruction $\texttt{ar+=c}_1$, we simply adopt the addressing mode above by shifting the offset values in am.

$$am(\texttt{ar+=c}_2) = \{\langle i-c_1, i+c_2-c_1\rangle \,|i \in D\}$$

Definition of addressing registers allow an arbitrary entry offset and only the exit offset must be zero. Therefore, the addressing mode of an definition is given by $am(\texttt{ar=c}) = \{\langle i,0\rangle \,|i \in D\}$.

Memory accesses and address modification of the input program can be rewritten by several address modes with different costs. Basically, we are interested in choosing address modes for the input programs such that the overall costs are minimal. Note that the selection of an addressing mode is not a local decision because it induces offset constraints for the entry and exit offsets. Preceding and succeeding statements must have matching offset values – otherwise program semantics is destroyed.

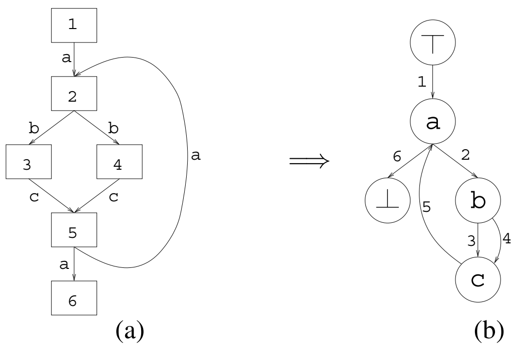


Figure 3. CFG and PBQP graph of example

For a given offset pair of a statement we have several ways to rewrite the associated statement for obtaining the offset shift at the entry and at the exit. However, we want to have the addressing mode with minimal costs for a statement and a given offset pair. Some statements do not allow addressing modes for a given offset pair. In this case we assign infinite costs to the offset pair.

To express the costs of addressing modes, matrices are employed. An offset pair is associated with an element of a cost matrix, i.e. element $C(i,j)$ of matrix C gives the costs for the cheapest addressing mode for the offset pair $< i,j >$. Note that the values of i and j might also be negative. To get positive row and column indices a mapping function is required, e.g. the minimum negative value in D is subtracted, etc.

We can now formulate the cost matrices for our example in Section 1. Figure 3(a) shows the CFG of the input program. We assign costs of 0 for the post modification mode and costs of 0.2 for the indexing mode, because indexing has worse characteristics than post modification (e.g. larger coding). Inserting an add instruction contributes with costs of one. From the assumed loop iteration count of 10 and the condition evaluation of 7 times true we get node weights of $w_1 = 1$, $w_2 = 10$, $w_3 = 7$, $w_4 = 3$, $w_5 = 10$, and $w_6 = 1$. The addressing mode costs must be multiplied with the corresponding node weights. For this example we limit the domain of offset values to set of $\{0,1,2\}$ to keep the matrices small. Of course the real implementation of the algorithm has to take the whole domain of available values into account (see Section 3) and sparse matrix representations are required.

Let us construct the cost matrix of empty node (1). Since the node does not contain a statement which accesses memory or modifies the address register, we treat it is an add instruction with zero increment. For each offset pair in the the domain we find the cheapest addressing mode. Hence, we can only parameterize the constant of the add instruction we have for every add instruction only one choice to select the minimal costs.

$$C_1 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

am	0	1	2
0	`ar+=0`	`ar+=1`	`ar+=2`
1	`ar-=1`	`ar+=0`	`ar+=1`
2	`ar-=2`	`ar-=1`	`ar+=0`

The matrix C_1 represents the costs for all offsets pairs. The table on the right-hand side of the matrix gives the associated addressing modes. Note that the rows relate to the entry offset values and the columns to the exit offset values. Any transition from an entry offset value to an exit different offset value needs an add instruction to be inserted with the costs of 1. If the entry and exit offset values are identical the add instruction can be eliminated since it has a zero increment.

The matrix for node (2) and the associated addressing modes is given as follows:

$$C_2 = 10 \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

am	0	1	2
0	`ar+=1`	`ar+=2`	`ar+=3`
1	`ar+=0`	`ar+=1`	`ar+=2`
2	`ar-=1`	`ar+=0`	`ar+=1`

Node (2) is executed 10 times and therefore the cost matrix is multiplied by a factor of 10. Moreover, the cost matrix contains two elements whose values are zero. The associated addressing modes of those elements eliminate the add instruction. For all other offset pairs the add instruction remains in the program.

The statement `*(ar)` of node (3) imposes a more complex cost matrix and addressing mode table. For one offset pair there can be more than one choice. For such a case we have to take the addressing mode with the cheapest costs. In addition some offset pairs require additional add instruction to update the value of the address register. The cost matrix of node (3) is given by $C_3 = 7 \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0.2 & 1 \\ 1 & 1 & 0.2 \end{pmatrix}$ where the addressing mode table is listed below

am	0	1	2
0	`*(ar)`	`*(ar++)`	`*(ar+=2)`
1	`ar-=1;*(ar)`	`*(ar+1)`	`ar-=1;*(ar+=2)`
2	`ar-=2;*(ar)`	`ar-=2;*(ar++)`	`*(ar+2)`

The zeros in the first row result from the post modification addressing mode and the elements in the remaining diagonal whose values are 0.2, result from the indexing addressing mode. For all other offset pairs an add instruction must be inserted.

The cost matrices and addressing mode tables of nodes (4) and (5) are constructed akin to previous statments.

$$C_4 = 3 \begin{pmatrix} 0.2 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0.2 \end{pmatrix}, \quad C_5 = 10 \begin{pmatrix} 0.2 & 1 & 1 \\ 1 & 0.2 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

For node (6) we obtain the same cost matrix as already presented for node (1), i.e. $C_1 = C_6$.

The objective of the AMS is to select offset values for the address register for all entries and exits of CFG nodes. The entry- and exit-values define the cost function which has to be optimized.

Definition 2 *For all CFG nodes $n \in V$, let e_n be then entry-value of n, x_n the exit-value of n and C_n the cost matrix of n. Then the cost function for the AMS is given by*

$$f = \sum_{n \in V} C_n(e_n, x_n). \tag{1}$$

Given this definition alone, minimizing the cost function would be trivial. As already shown in Figure 2 the exit and entry offset values of two subsequent statements must match. For a CFG node which may have several successors we can generalize previous observation.

$$x_n = e_m \quad \forall m \in SUCC(n) \tag{2}$$

Offset values are propagated along CFG edges and enforces a consistent offset value between two subsequent statements. The constraint imposes also a partitioning of edges. In a partition, all the entry- and exit-offsets of their associated edges must have the same offset value.

The condition above can only be relaxed if an use of an address register can not be reached in any path starting from the join point, then a consistent offset value is not needed at the join point. An easy way to handle this exception is to perform a liveness analysis prior to the AMS algorithm and exclude all CFG nodes and edges, where the address register is not alive.

For solving the AMS problem we map the original CFG to a new graph. A node of this new graph aggregates all CFG edges which belong to one edge partition and represents this set of edges as a node. An edge in the new graph is a CFG node. Since the new graph is directed, the direction is given by the constraint. This means, the source of a CFG node is associated with the constraint which uses the entry offset – the target of the CFG node is associated with the constraint which uses the exit offset. For the start and end node of a CFG node we would not have source and target nodes and therefore we introduce artificial nodes ($\perp$ and $\top$).

This new graph allows a natural mapping to the PBQP problem and is identical to the PBQP graph as introduced in [8]. The PBQP problem optimizes the cost function by choosing only one offset value for one node in the PBQP graph.

The offset value constraints have a graph property. Edges can be partitioned into edge classes. An edge class is given by following definition.

Definition 3 *The set $L = \{l_1, \ldots, l_n\}$ is the set of CFG edge classes. Two edges $e_i, e_j \in E$ are in the same* *edge class* l_k *iff* $\text{target}(e_i) = \text{target}(e_j)$ *or* $\text{source}(e_i) = \text{source}(e_j)$.

An edge class is the transitive hull of all edges which have common source or target (an edge class is a zig-zag pattern in the CFG). With definition 3 the PBQP graph construction algorithm can be formulated:
(1) group all edges in the CFG into edge classes
(2) generate a PBQP edge for each CFG node n from PBQP node l_i to l_j, where $\forall p \in PRED(n)$: $p \in l_i$ and $\forall s \in SUCC(n)$: $s \in l_j$.
(3) add entry node $\top$ and exit node $\bot$
(4) generate a PBQP edge for the CFG entry node n from PBQP node $\top$ to l_i, where $\forall s \in SUCC(n)$: $s \in l_i$.
(5) generate a PBQP edge for the CFG exit node n from PBQP node l_i to $\bot$, where $\forall p \in PRED(n)$: $p \in l_i$.

Figure 3 shows the CFG and the related PBQP graph of our example. It consists of three edge classes (a, b, c) and the entry and exit classes ($\top$, $\bot$). As there are no register definitions in the program, all CFG nodes and edges are included in the PBQP graph construction process.

3 Algorithm

After generating a PBQP problem from the AMS cost model, the PBQP problem must be solved. This is done by employing the PBQP solver introduced in [8]. In this section we want to give an overview of the solve algorithm.

The solver is based on a dynamic programming approach which works in three phases. In the first phase the PBQP graph is reduced. In each reduction dynamic programming eliminates a node of the graph until only degree zero nodes remain. In the second phase for each node a state with minimal cost is selected. In the third phase the minimal cost states are selected for all reduced nodes by reversing the reduction process.

In the first phase reduction rules are applied to nodes with degree one and two. These reductions do not destroy the optimality of the PBQP. If this is not possible, i.e. at some point of the reduction process there are only nodes with degree three or higher in the graph, a heuristic must be applied. The crucial point is now that for the AMS problem almost all graphs can be reduced without applying the heuristic. So the solution is optimal for almost all graphs. The reason is that the PBQP graphs are derived from the CFGs of the input program, whereas the CFGs of the input program are generated from a structured high level language - in our case C. Almost all control flow which can be formulated in C result in reducible PBQP graphs. There are some exceptions, like the goto statement and program transformations, which produce non reducible graphs.

The directions of the PBQP graph edges indicate the 'directions' of the cost matrices. In other words, changing the direction of an edge means to transpose the cost matrix of the edge.

Definition 4 *Let* $\vec{c}(x)$ *be the cost vector of node* $x \in L$. *Let* C_{xy} *be the cost matrix of edge* $(x, y) \in F$ *or the transposed cost matrix of edge* $(y, x) \in F$.

Figure 4 shows the pseudo code of the reduction algorithm. The algorithm yields optimal solutions as long as only ReduceI and ReduceII reductions are applied. If a ReduceN reduction must be applied the result of the heuristic depends on which node is selected for the ReduceN reduction. But we have seen that ReduceN reductions are so rare that no special effort is taken to a select a node. In our implementation the node with highest degree is selected, because it eliminates most edges when it is reduced.

The reduction procedures ReduceI, ReduceII and ReduceN use the basic operations *vmin* and *mmin* which implement dynamic programming. The *vmin* operation is used in the ReduceI reduction and calculates a cost vector which is added to the reduced node's adjacent node. The *mmin* operation is used in the ReduceII reduction and calculates the cost matrix of the resulting edge.

Definition 5 *Let* $\vec{c}$ *be a vector and* C, C' *be matrices. Function* $\text{vmin}(\vec{c}, C)$ *is defined by vector* $\vec{x}$ *where* $\vec{x}(i) = \min_k \vec{c}(k) + C(k, i)$. *Function* $\text{mmin}(C, \vec{c}, C')$ *is defined by matrix* X *where* $X(i, j) = \min_k C(i, k) + \vec{c}(k) + C'(k, j)$

After reduction only nodes with degree zero remain and the state can be selected by finding the index of the minimum vector element. The state of all nodes can be selected by reconstructing the PBQP graph in the reverse order of reductions. the pseudo code of the state selection phase.

In the sequel we show the reduction process for our example. The reduction steps are depicted in figure 5. First, we normalize the BPQP graph. The only required action in the example graph is to combine edges 3 and 4. The resulting matrix is the sum of matrix C_3 and C_4 which yields

$$C_{34} = \begin{pmatrix} 0.6 & 3 & 3 \\ 7 & 1.4 & 7 \\ 10 & 10 & 2 \end{pmatrix}$$

The first step is the reduction of the degree-one nodes $\top$ and $\bot$ which adds two vectors $(0\ 1\ 1)$ to the node vector a. The resulting node vector is $v_a = (0\ 2\ 2)$. The remaining cycle of three nodes is reduced by reducing (any) node with the ReduceII reduction. In the example we select node b. The new edge gets a matrix of

$$C_{234} = \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ -9.8 & -7 & -7 \\ -3 & -9.8 & -3 \end{pmatrix}$$

In the next step normalization is necessary by combining edges 234 and 5 by adding $C(234)$ and $C(5)^T$.

$$C_{2345} = \begin{pmatrix} 0.4 & 10.2 & 0.4 \\ 0.2 & -6.8 & -7 \\ 7 & 0.2 & -3 \end{pmatrix}.$$

```
**procedure** ReduceI($x$)
**begin**
    $\{y\} := adj\,(x)$
    $\vec{c}_y := \vec{c}_y + vmin(\vec{c}_x, C_{xy})$
    PushVertex(x);
**end**
**procedure** ReduceII($x$)
**begin**
    $\{y, z\} := adj\,(x)$
    **if** $(y, z) \in F$ **then**
        $C_{yz} := C_{yz} + mmin(C_{yx}, \vec{c}_x, C_{xz})$
    **else**
        *add edge* $(y, z)$
        $C_{yz} := mmin(C_{yx}, \vec{c}_x, C_{xz})$
    **endif**
    PushNode(x)
**end**
**procedure** ReduceN($x$)
**begin**
    **forall** nodes $y \in adj\,(x)$ **do**
        $\vec{c}_x := \vec{c}_x + vmin(\vec{c}_y, C_{yx})$
    **endfor**
    $s_x = i_{\mathsf{min}}\,(\vec{c}_x)$
    **forall** nodes $y \in adj\,(x)$ **do**
        $\vec{c}_y := \vec{c}_y + C_{yx}(:, s_x)$
    **endfor**
    PushNode(x)
**end**
**procedure** ReduceGraph
**begin**
    **while** $\exists n : deg(n) > 0$ **do**
        **if** $deg(n) = 1$ **then**
            ReduceI($n$);
        **elsif** $deg(n) = 2$ **then**
            ReduceII($n$);
        **else**
            ReduceN(n);
        **endif**
    **endwhile**
**end**
**procedure** PropagateSolution
**begin**
    **forall** *nodes x where* $deg(x) = 0$ **do**
        $s_x := i_{\mathsf{min}}\,(\vec{c}_x)$
    **endfor**
    **while** *reducible stack not empty* **do**
        *pop node x from reducible stack*
        $\vec{c} := \vec{c}_y;$
        **forall** nodes $y \in adj\,(x)$ **do**
            $\vec{c} := \vec{c} + C_{yx}(:, s_x);$
        **endfor**
        $s(x) = i_{\mathsf{min}}\,(\vec{c});$
    **endwhile**
**end**
```

Figure 4. PBQP Solver

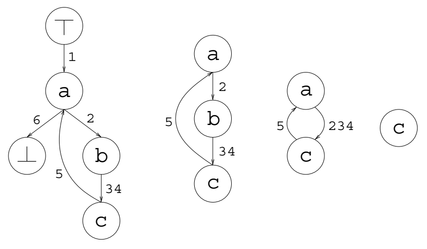


Figure 5. Reduction sequence

The last reduction step is a ReduceI of node a. The only remaining node is c with vector $\{0.4, -4.8, -5\}$ and the state can be selected by taking the index of the minimal element -5: $s_c = 2$. The minimal element -5 represents the total cost of the optimization. It should be negative, because the optimization should bring a benefit, rather than costs. Now the reduction process is reversed and states are selected for all nodes in the order $s_a = 1$, $s_b = 0$, $s_c = 2$, $s_{bot} = 0$, $s_{top} = 0$.

The solution of the PBQP problem yields a state in each node of the PBQP graph. The states are then transferred to the CFG. The entry value e_n of node n is the PBQP state of the predecessor edge class of n, the exit value x_n if the PBQP state of the successor edge class of n. The selection of the addressing mode for an instruction is done by selecting the am with the minimum cost which contains the pair $\langle i, j\rangle$. Add instructions with zero adding constants can be deleted.

In our example the entry and exit values can be obtained from the predecessor and successor edge values respectively: $e_1 = 0$, $x_1 = 1$, $e_2 = 1$, $x_2 = 0$, $e_3 = 0$, $x_3 = 2$, $e_4 = 0$, $x_4 = 2$, $e_5 = 2$, $x_5 = 1$, $e_6 = 1$, $x_6 = 0$. From this entry and exit values the output program, which is already shown in section 1, can be generated.

The dimension of the vectors and matrices used in the PBQP algorithm is the number of available values which an address register may contain, that is $|D|$. In practice this number is very large (e.g. 2^{16} or 2^{32}). Although the number is constant it is not possible to implement such large vectors and matrices as arrays of values. To get a handle on the problem, a sparse representation vector and matrix representation is used.

The vector is expressed by a set of cost regions $\{r_1, \ldots, r_n\}$. A region is an interval which has a lower and upper bound and an associated cost value. A vector element i is defined as

$$\vec{v}(i) = \min_{i \in [lower(r), upper(r)]} cost(r) \tag{3}$$

the cost value of the cost region r with minimal costs whereby i is in the interval of the cost region.

Matrices are also represented as a set of cost regions. For matrices a cost region consists of three intervals: a row interval, a column interval and a main diagonal interval. This three intervals define a six sided region in the matrix. Again a cost value is associated to the region. Similar to vectors, matrix elements of spare matrices are defined as

$$C(i, j) = \min_{\substack{i \in [lowerrow(r), upperrow(r)] \\ j \in [lowercolumn(r), uppercolumn(r)] \\ i+j \in [lowerdiagonal(r), upperdiagonal(r)]}} cost(r) \tag{4}$$

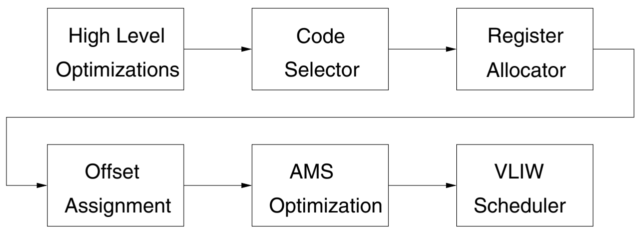


Figure 6. Compiler Phases

All operators between vectors and matrices are implemented to perform on cost regions. The operators include vector addition, matrix addition, matrix transpose, *vmin* and *mmin*. As the operations require to be performed on all regions of the operands, the worst case complexity is high. The highest effort results from the *mmin* operation, which is $m * n * o$ in the worst case, where m and n are the number of regions in the matrices, respectively and o is the number of regions in the vector. To overcome the problem of the high computational effort simplification is performed after each operation. The simplification reduces the number of regions in the region set. This includes removing redundant regions and shrinking or merging partly redundant regions. As we have seen from our experimental results, the number of regions can be reduced significantly with simplification and therefore the real complexity stays within acceptable bounds.

4 Experimental Results

We have integrated the AMS algorithm in the CC7705x C-Compiler for the NEC uPD7705x DSP family. The uPD7705x is a 32 bit fixed point DSP that has VLIW features. Seven functional units (two MAC, two ALUs, two load/store, one system unit) can execute up to four instructions in parallel. In contrast to conventional VLIW architectures, the execution packet length of the uPD7705x is variable. So there is no code size overhead if an execution packet does not include the maximum of four instructions. The load/store units of the uPD7705x facilitates various addressing modes for 8 address registers. Most of the addressing modes of the uPD7705x are discussed in Section 2, e.g. indirect addressing, post increment/decrement, indexing, post modification with offset register. In addition post modification can wrap around a modulo value to implement circular buffers. Furthermore, a bit reverse addressing mode can be selected for efficiently accessing FFT buffers. All of these addressing modes can be modeled by the AMS algorithm whereas the bit reverse addressing mode and the modulo addressing modes require some additional optimization components than presented in this paper.

The structure of the compiler is depicted in Figure 6. Addressing mode related optimizations are performed between register allocation and scheduling on a low-level intermediate representation that is related to the uPD7705x assembly language. The addressing mode selection is performed after assigning the offsets for the function stack frames [7]. Because of the enormous complexity it is not possible to combine all these phases into one overall optimization problem. Therefore register allocation, offset assignment, AMS and scheduling is performed in separate steps.

The AMS algorithm runs for each address register separately.

Benchmark	# of graphs	max nodes	avg nodes
mp3	419	2009	134.66
gsm	900	1930	143.07
aac	487	1509	85.84
trcbk	6	20	15.33
cfirc	16	61	37.62
firc	44	58	31.32
iirc	6	25	16.33
iirbiqc	13	65	35.00
lmsc	15	70	34.73
matmult	8	26	18.62
vadd	6	15	9.67
vdot	4	8	6.50
vmin	4	13	8.25
vmult	6	15	9.67
vnorm	3	8	6.00

Table 1. Problem Size of Benchmarks

Two of the address registers are used as “floating-frame-pointers”. They are used to access two stack frames (one address register per stack frame). As a result of the AMS optimization, the frame pointers do not point to the beginning of the stack frames, but may point to any location within the execution of a function to utilize the addressing modes in the most optimal way.

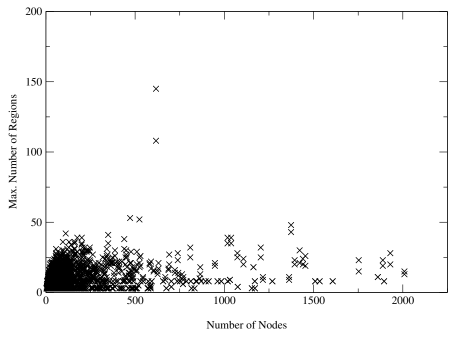
We have conducted experiments with a number of DSP benchmarks. The first group of benchmarks contains three complete DSP applications: AAC (advanced audio coder), MPEG, and GSM (gsm half rate). All three benchmarks are real-world applications that contain some large PBQP graphs. The second group of benchmarks are DSP-related algorithms of small size. These kind of benchmarks allow the detailed analysis of the AMS algorithm for typical loop kernels of DSP applications. All benchmarks are compiled ’out-of-the-box’[4].

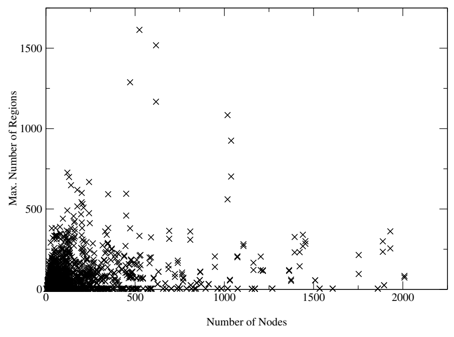
The benchmarks and the problem size of the benchmarks are listed in Table 1. The first column *# of graphs* shows the number of PBQP graphs that are solved for the optimizations. The number of graphs is determined by the number of functions in a program and the number of used address registers. Note that for the uPD7705x architecture there is an upper limit of 8 address registers. The computational complexity of AMS mainly depends on the number of nodes in a PBQP graph. The second column *max nodes* of Table 1 gives the number of nodes for the largest graph in the benchmark suite. In the last column *avg nodes* shows the average number of nodes for a benchmark.

In the following we give some performance details of the PBQP solver. Almost all PBQP graphs can be solved optimally. The most frequent reductions are ReduceI (60.9%), followed by ReduceII (25.4%), and 13.6% of all nodes have degree zero. Only 8 reductions out of 230481 reductions are ReduceN reductions which are solved by heuristics. One of the most important observation of our experiments is that the result of the address mode selection is optimal in almost all benchmarks.

Our PBQP solver employs sparse representation techniques of

[4]This means that the benchmark source codes are not rewritten and tuned for the CC7705x compiler.





(b) Matrix Regions

Figure 7. Max. Number of Regions

Benchmark	percent
mp3	8.90
gsm	4.77
aac	7.57
trcbk	4.65
cfirc	4.24
firc	4.90
iirc	3.69
iirbiqc	6.59
lmsc	5.14
matmult	5.69
vadd	6.09
vdot	5.76
vmin	6.54
vmult	6.92
vnorm	5.80

Table 2. Compile time overhead of AMS

large vectors and matrices. Figure 7 illustrates the relation between number of nodes in a PBQP graph and the maximum number of regions occurring in a vector and matrix respectively. As in the graph depicted the maximum number of regions does not correlate with the number of nodes. In practice the maximum number of regions is bounded and does not grow exponentially which is essential for AMS. Table 2 shows the compile-time overhead of AMS compared to the overall compile time in percent. It ranges from 4 to 9%. This is within acceptable bounds for a production DSP compiler, taking the high code improvements into account. The table also shows that the overhead for the large applications is not significant higher than for the small benchmarks. This indicates that the AMS scales almost linear with the problem size.

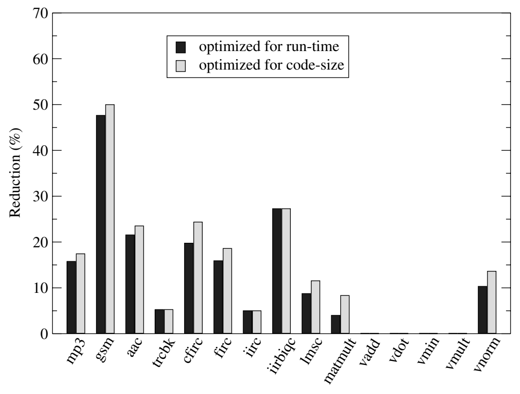
In the sequel we show the performance results of the AMS optimization in comparison with not applied AMS optimization. For the benchmark we evaluated the achieved code reduction and runtime improvements for different parameterizations of the compiler. The baseline is a code which is generated after strength reduction. The address register is set up only before the first access, but access and update instructions are not combined. All compiler optimizations are performed in the baseline, except AMS.

In the best case our AMS algorithm achieves code-size reductions up to 50% and speedups of more than 60%. Since we have a VLIW architecture, where add instructions can be scheduled without the penalty of additional execution cycles, we measured the effect of AMS by emitting VLIW code and by linear code. Moreover, we conducted experiments with two different cost models. The first cost model minimizes execution time and the second cost model minimizes code size. In Figure 8(a)-(b) the code reductions of the benchmark programs with linear and VLIW code are depicted whereas Figure 8(c)-(d) shows runtime improvement achieved by AMS.

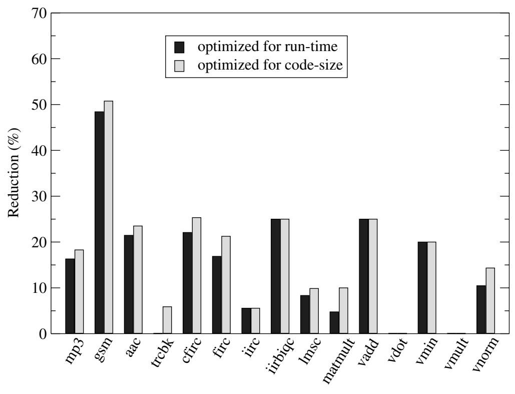
We used different models for execution time and code size optimizations. The execution time model reflects the execution cycles and delay cycles of the target instructions. The costs are weighted with estimated execution counts of the basic blocks. The execution count estimation is based on the loop structure of the function. It turned out that the accuracy of the estimation is sufficient for our purpose. As the AMS algorithm is performed for each function separately, recursive functions are not considered in the execution count estimation. The cost model for code size optimization is derived from the instruction code length of the target hardware. The costs of addressing modes directly correspond to the addressing modes code size. In the code size model the costs are not weighted with the execution count of the basic blocks.

The execution time improvements are significant larger for small benchmarks than for bigger applications. Nevertheless, there are impressive code size reductions for bigger applications, e.g. GSM. The reason is that small benchmarks mainly contain kernels of typical DSP algorithms. The execution time improvements, which are achieved in the kernel loops, directly effects the overall improvement. For larger application more "control code" (e.g. function calls) is executed, which gives less opportunity for runtime improvements. However, as shown in Figure 8(a)-(b) the code-size of larger applications can be significantly reduced. For some small benchmarks, e.g. trcbk, there is no improvement at all since there is no potential to optimize the addressing mode selection in this cases.

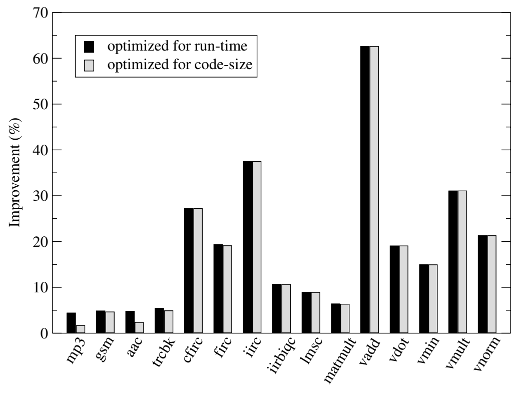
For our target architecture the compiler is able to schedule add instructions without the penalty of additional execution cycles. Even if the AMS optimization is disabled, the scheduler might find a free VLIW slot for placing add instruction of address register. In order to simulate a architecture without VLIW capabilities we conducted performance experiments on linear code (no VLIW code is generated). The code size improvements are roughly the same as with VLIW code generation but the execution time improvements are considerable larger. Another major contribution



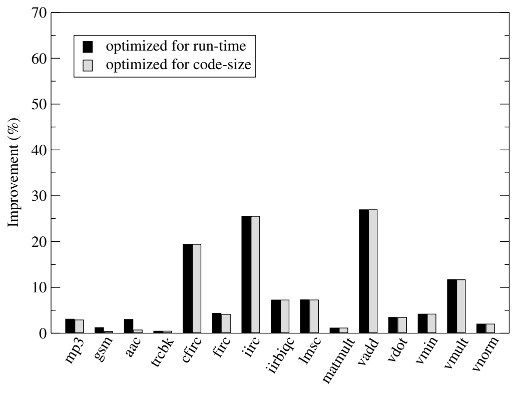
(a) Code-Size Reduction (linear code)



(b) Code-Size Reduction (VLIW code)



(c) Run-time Improvement (linear code)



(d) Run-time Improvement (VLIW code)

Figure 8. Run-time Improvement and code-size reduction

of our experiments is that in VLIW architectures the performance gain of addressing modes deteriorates.

5 Conclusion

Conventional optimization techniques can not handle the addressing mode features of modern DSP architectures. This work presents a novel approach for exploiting addressing modes of DSPs. Even complex addressing modes can be utilized by a compiler and, therefore, it is a further step towards implementing critical DSP applications in a high level language rather than in assembly code.

Embedded CPU architecture design often involves a trade-off between "application specific" and "compiler friendly". With the presented algorithm a wide range of addressing mode implementations can be modeled. Therefore, the addressing mode design can be targeted to the application area without loosing the opportunity for best support of the compiler. Changes in the addressing mode design can be brought in the compiler by simply changing the cost model of the AMS algorithm.

As the runtime improvements are up to 60% and code-size reductions up to 50%, we can state that the optimization is of vital importance for architectures which provide complex addressing mode features.

References

[1] E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In *Proceedings of LCTES'99*, 1999.

[2] G. Ottoni et al. Optimal live range merge for address register allocation in embedded programs. In *Proceedings of CC'01*, 2001.

[3] S. Liao et al. Storage assignment to decrease code size. *ACM TOPLAS*, 18(3):235–253, May 1996.

[4] Sudarsanam et al. Analysis and evaluation of address arithmetic capabilities in custom dsp. *Proceedings of Design and Automation of Embedded Systems*, 4(1), Jan 1999.

[5] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in DSP programs. In *Proceedings of Asia and South Pacific Design Automation Conference*, 1998.

[6] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings of ISSS*, 1998.

[7] Stan Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, 1996.

[8] B. Scholz and E. Eckstein. Register allocation for irregular architecture. In *Proceedings of LCTES/SCOPES'02*, 2002.