Control Flow driven Code Hoisting at the Source Code Level

Heiko Falk
University of Dortmund, Computer Science 12, D - 44221 Dortmund, Germany
Heiko.Falk@udo.edu

Abstract

This paper presents a novel source code optimization technique called advanced code hoisting. It aims at moving portions of code from inner loops to outer ones. In contrast to existing code motion techniques, this is done under consideration of control flow aspects. Depending on the conditions of if*-statements, moving an expression can lead to an increased number of executions of this expression. This paper contains formal descriptions of the polyhedral models used for control flow analysis so as to suppress a code motion in such a situation.*

Due to the inherent portability of source code transformations, a very detailed benchmarking using 8 different processors was performed. The application of our implemented techniques to real-life multimedia benchmarks leads to average speed-ups of 25.5%–52% and energy savings of 33.4%–74.5%. Furthermore, advanced code hoisting leads to improved pipeline and cache behavior and smaller code sizes.

1. Introduction

In recent years, the power efficiency of embedded multimedia applications (e. g. medical image processing, video compression) with simultaneous consideration of timing constraints has become a crucial issue driving the need for the generation of highly optimized embedded software. Due to the need for very efficient machine code and the lack of optimizing compilers, assembly-level software development for embedded processors was very common in the past. However, both embedded applications and processors are getting more and more complex, and the development of highly optimizing compilers allows the replacement of assembly level programming. Modern compilers are equipped with a large amount of different optimizations, among which *common subexpression elimination (CSE)* and *loop-invariant code motion (LICM)* have proven to be highly beneficial [17].

This paper presents a new source code optimization called *advanced code hoisting (ACH)*. This technique is an elaborate combination of the already known CSE and LICM optimizations with a formal mathematical criterion steering the application of the mentioned optimizations. Studying advanced code hoisting is motivated by the fact that the application of the conventional CSE and LICM techniques might not be beneficial if the control flow is not considered. This is illustrated by means of a source code fragment extracted from the innovative GSM codebook search depicted in Figure 1.

```
for (j=0; j<l_code; j++)
 for (i=0; i<l_code; i++) {
  if (i<=c1)
   if (i==c1)
    rdm = h2[l_code-1-i];
   else rdm = rr[i*(i-1)/2];
  else rdm = rr[j*(j-1)/2];
  ...mul(rdm, ...);
  if (i<=c2)
   if (i==c2)
    rdm = h2[l_code-1-i];
   else rdm = rr[j*(j-1)/2];
  else rdm = rr[i*(i-1)/2];
  ...mul(rdm, ...); }
```

Figure 1. GSM Codebook Search

For this small example, we assume that the symbolic constants `l_code`, `c1` and `c2` are equal to 10,000, 9,000 and 1,000, respectively. The structure of the *for*-loops and *if*-statements depicted in Figure 1 has the effect that the expression `l_code-1-i` is executed 20,000 times (`i*(i-1)/2`: 179,990,000 times, `j*(j-1)/2`: 19,990,000 times).

The application of the conventional CSE and LICM optimizations has the effect that these three expressions are replaced by accesses to new local variables which are defined in the outermost possible loops. As a consequence, both `l_code-1-i` and `i*(i-1)/2` are moved to the beginning of the `i`-loop, whereas `j*(j-1)/2` is moved to the beginning of the `j`-loop. The resulting code is depicted in Figure 2.

```
for (j=0; j<l_code; j++) {
 int tmp1 = j*(j-1)/2;
 for (i=0; i<l_code; i++) {
  int tmp2 = l_code-1-i,
      tmp3 = i*(i-1)/2;
  if (i<=c1)
   if (i==c1)
    rdm = h2[tmp2];
   else rdm = rr[tmp3];
  else rdm = rr[tmp1];
  ...mul(rdm, ...);
  if (i<=c2)
   if (i==c2)
    rdm = h2[tmp2];
   else rdm = rr[tmp1];
  else rdm = rr[tmp3];
  ...mul(rdm, ...); }}
```

Figure 2: GSM Code after conventional Code Hoisting

For the given values of `l_code`, `c1` and `c2`, this conventional code hoisting leads to an execution frequency of 100,000,000 for `l_code-1-i` and `i*(i-1)/2` (10,000 for

```
for (x=1; x<N-2; x++)
  for (y=1; y<M-2; y++) {
    for (k=-1; k<1; k++) {
      ...
      A[x][y]+=B[x+k][y];
      ...  }
    A[x][y]/=tot; }
```

→

```
for (y=0; y<M+2; y++)
  for (x=0; x<N+2; x++) {
    ...
    if (x>=0 && x<N && y>0 && y<M-1)
      D[x%3]=B[96+(y*N+x%3)%160+
               (y*N+x%3)/160*256];
    ...
    if (x>1 && x<N && y>0 && y<=M-2)
      for (k=-1; k<=1; k++)
        acc+=D[(x-1+k)%3];
    acc/=tot; }
```

Figure 4. A Fragment of the CAVITY Benchmark before and after DTSE

`j*(j-1)/2`). As can be seen, the execution frequency of *`l_code`*`-1-i` increases by a factor of $5,000$.

The reason for the increased execution frequency of *`l_code`*`-1-i` after its motion to the beginning of the `i`-loop are the two *if*-statements surrounding this expression. In the original code depicted in Figure 1, the expression is executed for only two iterations of the `i`-loop. After its motion out of the *if*-statements, it is executed for every iteration of the `i`-loop resulting in the observed increased execution frequency.

The first main contribution of this paper is the novel combination of the well-known compiler optimizations CSE and LICM with a steering criterion. Second, the formulation of this criterion deciding when to apply CSE and LICM is based on control flow aspects. This paper describes formal methods modeling large classes of loop nests and *if*-statements. Polyhedral techniques are applied for the exact computation of execution frequencies of CSEs. Third, all compilers involved during benchmarking are called with their highest levels of optimization enabled. Since they all include a CSE and LICM, it is an important observation that still very large savings can be achieved using ACH. This fact underlines that novel source code optimizations relying on already existing techniques to some extent are still able to outperform existing optimizing compilers.

The remainder of this paper is structured as follows: Section 2 gives a survey of related work. Section 3 presents the analytical models for advanced code hoisting. Section 4 describes the benchmarking results, and Section 5 summarizes and concludes this paper.

2. Related Work

Since CSE and LICM [1, 17] have been known for many years and can be found in any optimizing compiler, the discussion of these simple transformations is omitted here. However, literature clearly states that both CSE and LICM are especially beneficial when applied to addressing code. Since addressing code is generally not written explicitly by human programmers but is generated automatically by a compiler, the programmer is often unaware of the overhead due to memory accesses. The DTSE framework [4] of source code optimizations aims at the optimized exploitation of memory hierarchies and thus has the effect of making addressing code explicit in a program.

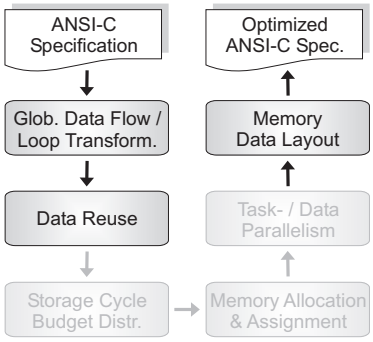


Figure 3: DTSE stages causing Control & Addressing Overhead

Several individual steps of DTSE are responsible for the generation of complex control flow and addressing code (see Figure 3). During *signal substitution*, dependencies on multi-dimensional arrays are removed and redundant array accesses are eliminated. Resolving these data dependencies requires the insertion of *if*-statements depending on the loops' index variables so as to access the appropriate memory locations at every point in time. The main idea of *data reuse exploration* [22] is to insert copies of the most frequently accessed parts of arrays in order to improve temporal locality. This also leads to a degraded control flow due to the insertion of *if*-statements so as to select all relevant non-contiguous parts of an array that will be held in a copy.

In-place mapping [9] aims at reusing physical memory by mapping different array elements that are not alive at the same time to the same memory location. This optimization requires the generation of complex addressing code reflecting the mapping of data elements to positions within an array. The effect of these parts of the DTSE methodology on a source code taken from [2] is illustrated in Figure 4.

Since many different data elements are stored at the same

addresses after in-place mapping, the same kind of address computations is performed several times at various locations in a DTSE optimized code. This is already acknowledged by the address optimization (ADOPT) phase. During *regularity improvement* [13], given address expressions $A = \{a_1, \ldots, a_n\}$ are transformed to $A' = \{a'_1, \ldots, a'_n\}$. Every expression a'_i computes the same value as a_i, but A' is generated such that a maximal reuse of computations using CSEs is achieved. As a consequence, these algebraic transformations open up opportunities for CSE as stated in [11]. For the ADOPT transformations, experiments involving the manual application of a CSE combined with conventional LICM are reported. It is the contribution of this paper that a formal problem definition for ACH is provided for the first time in conjunction with suitable polyhedral algorithms considering control flow issues.

Polyhedral models are frequently used in order to represent memory accesses or iteration spaces of loop nests. An approach for simultaneous generation of optimized data layouts and temporal locality improvement is presented in [16]. In this article, geometric models and algorithms are used to minimize TLB misses. Loop nest splitting as presented in [7, 8] uses polyhedral models in order to represent *if*-statements nested in loops. A complex analysis is performed in order to detect ranges of iterations of the loops such that all *if*-statements are satisfied. Using these iterations ranges, a loop nest is split in order to minimize *if*-statement executions.

Partial Redundancy Elimination (PRE) [12, 10, 3] moves conditionally executed expressions outside their conditional scopes to enable the elimination of partial redundancies along frequently executed paths in a program. The most important disadvantage of all PRE approaches is the lack of a detailed analysis of the control flow. PRE is unable to determine exactly when given conditions evaluate to true or false. Instead, PRE bases on profiling techniques in order to determine the execution frequencies of paths within an application.

Since PRE does not definitely know whether it is safe to move an expression out of a conditional scope, it must insert copies of an expression in all less frequently executed paths. In contrast, the ACH techniques presented in this paper rely on a analytical model of the control flow so that ACH is able to guarantee that it is safe to move an expression out of a conditional scope or not. Hence, ACH does not need to insert extra copies of an expression, leading to significant code size reductions as shown in the results section of this paper. Another disadvantage of PRE is the fact that it can only be applied to certain expressions not causing side-effects. Finally the speed-ups after PRE [3] varying between 0.33% and 4.16% are very small compared to the gains achieved by ACH. It can be concluded that ACH and PRE are complementary techniques.

3. Control Flow Analysis for ACH

As can be seen from the example given in Section 1, the conventional application of CSE and LICM can lead to an increased number of executions of an expression. The goal of advanced code hoisting (ACH) presented in this paper is to avoid such situations by only moving code from inner loops to outer ones as long as this leads to a reduced number of executions of the code. More precisely, *common subexpressions (CSEs)* are considered as candidates for ACH:

Definition 1 *An occurrence of an expression in a program is a common subexpression if there exists another occurrence of the same expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations. [17]*

ACH performs its optimization in a three-step approach:

1. Common subexpressions present in a source code are detected. For all occurrences of a CSE, their total number of executions (also called *execution frequency*) is computed.

2. For every CSE detected in step 1, the outermost loop which can legally contain it is identified. The execution frequency of this loop is also computed.

3. If the execution frequency computed in step 2 is smaller than that of step 1, all occurrences of a CSE are replaced by a new local variable storing the value of the CSE.

The key idea of ACH is the comparison of execution frequencies during step 3 of the optimization. For the GSM codebook search, only the motion of CSEs `i*(i-1)/2` and `j*(j-1)/2` leads to reduced execution frequencies. Hence, only these expressions are eliminated and hoisted. In contrast, `l_code-1-i` is not changed at all. The source code resulting from ACH is shown in Figure 5.

```
for (j=0; j<l_code; j++) {
 int ach_1 = j*(j-1)/2;
 for (i=0; i<l_code; i++) {
  int ach_2 = i*(i-1)/2;
  if (i<=c1)
   if (i==c1)
    rdm = h2[l_code-1-i];
   else rdm = rr[ach_2];
  else rdm = rr[ach_1];
  ...mul(rdm, ...);
  if (i<=c2)
   if (i==c2)
    rdm = h2[l_code-1-i];
   else rdm = rr[ach_1];
  else rdm = rr[ach_2];
  ...mul(rdm, ...); }}
```

Figure 5: GSM Code after Advanced Code Hoisting

Since the details of the conventional CSE and LICM optimizations are well-known and can be found in literature on compiler design [17], large parts of the first two steps of the ACH optimization approach are not explained here.

We thus present the polyhedral model required for computing the execution frequencies of expressions during ACH in this section.

3.1. Supported Model of Control Flow

The code examples given in Section 1 show how control flow issues, namely loops and *if*-statements, influence the execution frequency of an expression. In order to compute execution frequencies, it is thus necessary to formally define the model of control flow supported by advanced code hoisting. For this purpose, definition 2 first specifies which kinds of loops are considered:

Definition 2 *A* for*-loop L_l must fulfill the following requirements:*

1. *The range of the index variable i_l of L_l always lies between the loop's lower and upper bounds lb_l and ub_l resp.: $lb_l \leq i_l \leq ub_l$ $(i_l \in \mathbb{Z})$.*

2. *If L_l is nested in other* for*-loops $L_1,\ldots,L_{l-1}$, its bounds are affine expressions of the surrounding index variables $i_1,\ldots,i_{l-1}$. Hence, i_l iterates between $lb_l = \sum_{j=1}^{l-1}(c'_j * i_j) + c' \leq i_l \leq \sum_{j=1}^{l-1}(c''_j * i_j) + c'' = ub_l$ for constants $c'_j, c', c''_j, c'' \in \mathbb{Z}$.*

3. *After every iteration of L_l, i_l is incremented by a constant integer stride $s_l \in \mathbb{Z}$ with $s_l \neq 0$.*

Definition 2 is mainly based on the fact that advanced code hoisting uses a polytope model representing the control flow. Since polytopes are represented by linear inequations, the loop bounds are required to be affine expressions. An outermost loop L_1 is not not surrounded by any other loop so that definition 2.2 implicitly requires lb_1 and ub_1 to be constant. This way, it is ensured that the loop's iterations are not data dependent and that they are fully analyzable at compile time. Definition 3 determines the shape of *if*-statements:

Definition 3 *Let $L_1,\ldots,L_l$ denote a nest of* for*-loops complying with definition 2 which surrounds an* if*-statement.*

1. *An* if*-statement has the format* `if(`$C_1 \oplus C_2 \oplus \ldots$`)` ... `else` ... *where C_x are affine loop-variant conditions that are combined with logical operators $\oplus \in \{\&\&, ||\}$.*

2. *Loop-variant conditions C_x are affine expressions of the index variables $i_1,\ldots,i_l$ of the surrounding loops. For constants $c_j, c \in \mathbb{Z}$, C_x can be written as $C_x = \sum_{j=1}^{l}(c_j * i_j) \geq c$.*

The choice of the logical *AND* and *OR* operators in definition 3.1 is not a limitation since de Morgan's rules allow to model all boolean combinations of affine conditions this way. For example, the *if*-statement `if(!(`C_1 `&&` C_2`))` is equivalent to `if(!`C_1 `||` `!`C_2`)`. The negation `!`C_x of a single condition can be modeled as follows: `!`C_x = `!`$\left(\sum_{j=1}^{l}(c_j * i_j) \geq c\right)$ = $\sum_{j=1}^{l}(c_j * i_j) < c$ = $\sum_{j=1}^{l}(\text{-}c_j * i_j) \geq c+1$. As can be seen, all situations are captured by definition 3.

An expression *expr* being an occurrence of a CSE can now be arbitrarily enclosed by control flow constructs according to definitions 2 and 3. An example source code structure is depicted in Figure 6. As can be seen, *if*-statements are allowed to be nested in *for*-loops or in other *if*-statements. Within each *for*-loops, an arbitrary number of *if*-statements is allowed to occur. This is denoted by the values *m*, *n* and *o* in Figure 6. Since each loop may contain a different number of *if*-statements, three values *m*, *n* and *o* are shown in this figure. If e. g. the i_1-loop does not contain any *if*-statement, *m* is assumed to be 0.

```
for (i1=c1; i1 ≤ c2; i1+=s1)
 if (C1,1 ⊕ C1,2 ⊕ ...)
  ...
   if (Cm,1 ⊕ Cm,2 ⊕ ...)
    for (i2=lb2; i2 ≤ ub2; i2+=s2)
     if (Cm+1,1 ⊕ Cm+1,2 ⊕ ...)
      ...
       if (Cm+n,1 ⊕ Cm+n,2 ⊕ ...)
        for (iN=lbN; iN ≤ ubN; iN+=sN)
         if (Cm+n+1,1 ⊕ Cm+n+1,2 ⊕ ...)
          ...
           if (Cm+n+o,1 ⊕ Cm+n+o,2 ⊕ ...)
            expr;
```

Figure 6. Structure of Control Flow

Whenever code is executed conditionally by virtue of an *if*-statement, it does not make a difference whether this code is located in its *then*- or *else*-part since both cases are treated by the techniques presented in the following. Hence, all detailed information about *then*- or *else*-parts are omitted in Figure 6 for the sake of simplicity. This structure of control flow supported by advanced code hoisting is formally captured by the following definition:

Definition 4 *Let $\Lambda = \{L_1,\ldots,L_N\}$ be a nest of* for*-loops according to definition 2 and $\Upsilon = \{IF_1,\ldots,IF_O\}$ be a set of* if*-statements in accordance with definition 3. All loops and* if*-statements surround an expression expr.*

A sequence $\Gamma = (\gamma_1,\ldots,\gamma_M,\gamma_{M+1})$ is said to be a nest of control flow structures supported by advanced code hoisting if

1. *the first element of Γ is the outermost loop: $\gamma_1 = L_1$,*

2. *the last element of Γ is the expression whose execution frequency is to be determined: $\gamma_{M+1} = expr$,*

3. *γ_m either is a* for*-loop of Λ or an* if*-statement of Υ for $2 \leq m \leq M$, and*

4. *an* if*-statement* $\gamma_m \in \Upsilon$ *only depends on the index variables* i_l *of surrounding loops* $L_l \in \{\gamma_1,\ldots,\gamma_{m-1}\} \cap \Lambda$.

For an if*-statement* $\gamma_m \in \Gamma$, TP_{γ_m} *denotes its* then*-part and* EP_{γ_m} *its* else*-part, respectively.*

3.2. Polyhedral Representation

In order to compute how many times an expression *expr* is executed under consideration of the code structure shown in Figure 6, a polytope P^Γ is generated iteratively during a first phase. P^Γ is built such that it reflects the control flow given by the nested *for*-loops and *if*-statements. The constraints of P^Γ accurately model all loop bounds and conditions of *if*-statements as will be shown in the following. Polytopes are defined as follows:

Definition 5

1. $P = \{x \in \mathbb{Z}^N \mid Ax = a, Bx \geq b\}$ *is called a polyhedron for* $A,B \in \mathbb{Z}^{m\times N}$, $a,b \in \mathbb{Z}^m$ *and* $m \in \mathbb{N}$.
2. *A polyhedron P is called polytope if* $|P| < \infty$

Since the only variables having influence on the control flow shown in Figure 6 are the index variables of the loops $\Lambda = \{L_1,\ldots,L_N\}$, P^Γ is basically a subset of $\mathbb{Z}^N$:

Definition 6 *Let* $\Gamma = (\gamma_1,\ldots,\gamma_M,\gamma_{M+1})$ *be a nest of control flow structures according to definition 4. The polytope* P^Γ *modeling* Γ *is defined by* $P^\Gamma = (\bigcap_{\gamma_m\in\Lambda} P_m^{FOR}) \cap (\bigcap_{\gamma_m\in\Upsilon} P_m^{IF})$

As can be seen from the following definitions, intersection and union operators for polytopes are used in order to construct P_m^{FOR} and P_m^{IF}. Unfortunately, polytopes are not closed under the union operator. Instead, we use finite unions of polyhedra as proposed in [20] for which the union operator is closed. Hence, P^Γ formally is not a polytope, but a finite union of polytopes. For the sake of simplicity, we keep on using the notion of polytopes instead of their finite unions.

In order to construct P^Γ, polytopes associated with the individual elements γ_m of Γ need to be intersected. Depending on whether γ_m is a *for*-loop or an *if*-statement, P_m^{FOR} or P_m^{IF} has to be taken. After its construction, P^Γ contains exactly all those values of the index variables $i_1,\ldots,i_N$ of Λ leading to the execution of expression γ_{M+1}. For a given loop $\gamma_m \in \Lambda$, the following polyhedron P_m^{FOR} is generated:

Definition 7 *Let* $\Gamma = (\gamma_1,\ldots,\gamma_M,\gamma_{M+1})$ *be a nest of control flow structures with* Λ *denoting its nested loops.*

For a loop $\gamma_m \in \Lambda$, *a polyhedron* P_m^{FOR} *is created using the following constraints:*

1. $i_m \geq \sum_{j=1}^{m-1}(c'_j * i_j) + c'$ for lb_m
2. $i_m \leq \sum_{j=1}^{m-1}(c''_j * i_j) + c''$ for ub_m
3. $i_m = \begin{cases} \sum_{j=1}^{m-1}(c'_j * i_j) + c' + (i'_m * s_m) & \text{if } s_m > 1, \\ \sum_{j=1}^{m-1}(c''_j * i_j) + c'' + (i'_m * s_m) & \text{if } s_m < -1 \end{cases}$

Obviously, the constraints of definition 7.1 and 7.2 ensure that γ_m only contains points within the loop bounds lb_m and ub_m. i_m can take every integer value between lb_m and ub_m so that a loop with stride 1 or -1 is modeled. For loops with a stride s_m other than 1 or -1, it is necessary to add a constraint restricting i_m to only those integer values between lb_m and ub_m that can be reached using s_m. For $s_m > 1$, the constraint $i_m = lb_m + (i'_m * s_m)$ is added to P_m^{FOR}. Here, i'_m is an auxiliary integer variable for loop γ_m. This constraint ensures that i_m can take any value being a multiple of stride s_m which is added to the lower bound lb_m. Starting with lb_m itself, only every s_m-th integer value is assigned to i_m (analogously for $s_m < -1$).

EXAMPLE 1

The application of definition 7 to the loop `for (i=3; i<19; i+=4)` *leads to the polytope*

$P_{\mathtt{i}}^{FOR} = \{\ \mathtt{i} \in \mathbb{Z} \mid (\mathtt{i} \geq 3) \wedge (\mathtt{i} \leq 18) \wedge (\mathtt{i} = 3 + (\mathtt{i}' * 4))\ \}$

`i` *is only allowed to take those values which are a multiple of 4 added to the lower bound of 3, and which lie between 3 and 18. By means of this example, it can be seen that constraints for* $\mathtt{i}'$ *are not necessary.* $\mathtt{i}'$ *is unable to hold a negative value since the third constraint would lead to the situation that* `i` *is assigned a value resulting from the addition of a negative number to the lower bound. This violates the first constraint ensuring that* `i` *is greater than or equal to the lower loop bound. On the other hand,* $\mathtt{i}'$ *can not be greater than 3 since this would result in the assignment of a value greater than or equal to 19 to* `i` *violating the second constraint* $\mathtt{i} \leq 18$.

In their combination, the constraints of $P_{\mathtt{i}}^{FOR}$ *ensure that* `i` *can only take the values 3, 7, 11 and 15 which represents exactly the iterations of the above* for*-loop. Assuming that the stride of the above loop is changed to a negative value* `for (j=18; j>=3; j-=4)` *the associated polytope* $P_{\mathtt{j}}^{FOR}$ *differs from* $P_{\mathtt{i}}^{FOR}$ *only with regard to the third constraint:*

$P_{\mathtt{j}}^{FOR} = \{\ \mathtt{j} \in \mathbb{Z} \mid (\mathtt{j} \geq 3) \wedge (\mathtt{j} \leq 18) \wedge (\mathtt{j} = 18 + (\mathtt{j}' * \text{-}4))\ \}$

Here also, $P_{\mathtt{j}}^{FOR}$ *only consists of those values assigned to* `j` *by the loop, namely 18, 14, 10 and 6.*

Theorem 1 *For a given nest* Γ *of control flow structures, let* $\gamma_m \in \Lambda$ *be a* for*-loop in compliance with definition 2. Let* P_m^{FOR} *be the corresponding polyhedron according to definition 7.*

P_m^{FOR} contains exactly those values of the index variable i_m for which γ_{m+1} is executed.

Using complete induction, theorem 1 can be proven. Due to the lack of space, the proof – as well as the ones for the following theorems – is not given in this paper. Instead, they can be found in [6]. For a given *if*-statement $\gamma_m \in \Upsilon$, the following polytope P_m^{IF} is generated:

Definition 8 *Let $\Gamma = (\gamma_1, \ldots, \gamma_M, \gamma_{M+1})$ be a nest of control flow structures composed of a loop nest Λ and a set of* if*-statements Υ.*

For an if*-statement $\gamma_m = (C_1 \oplus C_2 \oplus \ldots \oplus C_n) \in \Upsilon$ surrounded by loops $L_1, \ldots, L_l$, π denotes the permutation of $\{1, \ldots, n\}$ representing the natural execution order of the conditions C_x, i. e. the right order to evaluate all C_x which is defined by the precedence rules of the the logical operators (&&, ||) and by possible brackets.*

1. *For an affine condition $C_x = \sum_{j=1}^{l} (c_j * i_j) \geq c$ of γ_m, the following polyhedra P_x and $\overline{P_x}$ are required:*

$$P_x = \{\ (i_1, \ldots, i_l) \in \mathbb{Z}^l \mid c_1 * i_1 + \ \ldots\ + c_l * i_l \geq c\ \}$$
$$\overline{P_x} = \{\ (i_1, \ldots, i_l) \in \mathbb{Z}^l \mid \text{-}c_1 * i_1 \text{-}\ \ldots\ \text{-}c_l * i_l \geq \text{-}c + 1\ \}$$

2. *In order to represent the first condition $C_{\pi(1)}$ of γ_m, the polyhedron P_x is used if the next element γ_{m+1} of Γ is located in the* then*-part of γ_m ($\overline{P_x}$ for the* else*-part, respectively):* $P_{m,1}^{IF} = \begin{cases} P_{\pi(1)} & \text{for } \gamma_{m+1} \in TP_{\gamma_m} \\ \overline{P_{\pi(1)}} & \text{for } \gamma_{m+1} \in EP_{\gamma_m} \end{cases}$

3. *For the remaining conditions $C_{\pi(x)}$ ($2 \leq x \leq n$), the corresponding polytope $P_{m,x}^{IF}$ on the one hand depends on the logical operator used to combine $C_{\pi(x)}$ with $C_{\pi(x-1)}$. On the other hand, the position of γ_{m+1} in the* then- *or* else*-part of γ_m has to be considered, too:*

$$P_{m,x}^{IF} = \begin{cases} P_{m,x-1}^{IF} \cap P_{\pi(x)} \text{ if } \gamma_{m+1} \in TP_{\gamma_m},\ C_{\pi(x-1)} \&\& C_{\pi(x)} \\ P_{m,x-1}^{IF} \cup P_{\pi(x)} \text{ if } \gamma_{m+1} \in TP_{\gamma_m},\ C_{\pi(x-1)} \,||\, C_{\pi(x)} \\ P_{m,x-1}^{IF} \cup \overline{P_{\pi(x)}} \text{ if } \gamma_{m+1} \in EP_{\gamma_m},\ C_{\pi(x-1)} \&\& C_{\pi(x)} \\ P_{m,x-1}^{IF} \cap \overline{P_{\pi(x)}} \text{ if } \gamma_{m+1} \in EP_{\gamma_m},\ C_{\pi(x-1)} \,||\, C_{\pi(x)} \end{cases}$$

4. *The polytope P_m^{IF} representing the entire* if*-statement γ_m is defined as $P_m^{IF} = P_{m,n}^{IF}$*

As can be seen from definition 8, P_m^{IF} is generated iteratively. In order to build P_m^{IF}, the polyhedral representation of every condition C_x is required (cf. definition 8.1). For every condition of γ_m, the polyhedra P_x and $\overline{P_x}$ representing C_x and its inverse $\overline{C_x}$ are needed.

Definition 8.2 shows that $P_{m,1}^{IF}$ is equal to the polytope representing the first condition of γ_m according to the execution order π. Here, the position of the next element $\gamma_{m+1} \in \Gamma$ in relation to the *if*-statement γ_m has to be considered. If γ_{m+1} is located in the *then*-part of γ_m, it is executed only if γ_m is satisfied so that $P_{\pi(1)}$ has to be taken ($\overline{P_{\pi(1)}}$ for γ_{m+1} in the *else*-part, resp.).

After this, all remaining conditions C_x of γ_m are examined using the ordering given by π. For C_x, the polytope $P_{m,x\text{-}1}^{IF}$ generated so far is connected with the appropriate polytope representing C_x. In analogy to the previous paragraph, $P_{\pi(x)}$ is used if γ_{m+1} is located in the *then*-part of γ_m (first two cases of definition 8.3), or $\overline{P_{\pi(x)}}$ for $\gamma_{m+1} \in EP_{\gamma_m}$ (last two cases), resp.

If $P_{\pi(x)}$ is taken, the intersection of polyhedra is applied whenever conditions $C_{\pi(x)}$ and $C_{\pi(x\text{-}1)}$ are connected using the logical *AND*. For the logical *OR*, the union of polyhedra is used. For γ_{m+1} being located in the *else*-part of γ_m, the negation of the conditions combined with de Morgan's rules has the effect that the union is used for the logical *AND* and vice versa.

EXAMPLE 2

Let C_1 denote the condition `4*i + j >= 12` *and C_2 represent the condition* `i + 5*j <= 28` *for some index variables* `i` *and* `j`*. The application of definition 8.1 to C_1 leads to the polyhedra*

$$P_1 = \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid 4*\mathtt{i} + \mathtt{j} \geq 12\ \}$$
$$\overline{P_1} = \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid \text{-}4*\mathtt{i} - \mathtt{j} \geq \text{-}11\ \}$$

where P_1 obviously represents C_1 and $\overline{P_1}$ denotes the negated condition $\overline{C_1}$. Analogously, the following polyhedra are defined for condition C_2:

$$P_2 = \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid \text{-}\mathtt{i} - 5*\mathtt{j} \geq \text{-}28\ \}$$
$$\overline{P_2} = \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid \mathtt{i} + 5*\mathtt{j} \geq 29\ \}$$

When assuming the if*-statement* `if (`C_1 `&&` C_2`) expr;` *definition 8 leads to the generation of*

$$\begin{aligned} P^{IF} &= P_1 \cap P_2 \\ &= \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid (4*\mathtt{i} + \mathtt{j} \geq 12) \wedge \\ &\qquad\qquad (\text{-}\mathtt{i} - 5*\mathtt{j} \geq \text{-}28)\ \} \end{aligned}$$

which is the straightforward case since the polyhedra representing the conditions simply need to be intersected in order to model the logical AND*. Similarly, the union operator has to be used for* `if (`C_1 `||` C_2`) expr;` *since in this case, all values of* `i` *and* `j` *have to be captured by P^{IF} which belong either to P_1 or P_2:*

$$\begin{aligned} P^{IF} &= P_1 \cup P_2 \\ &= \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid 4*\mathtt{i} + \mathtt{j} \geq 12\ \} \cup \\ &\quad\ \{\ (\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid \text{-}\mathtt{i} - 5*\mathtt{j} \geq \text{-}28\ \} \end{aligned}$$

Obviously, the if*-statement* `if (`C_1 `&&` C_2`) ... else expr;` *is equivalent to* `if !(`C_1 `&&` C_2`) expr; else ...` *De Morgan's rule applied to the conditions of the* if*-statement thus leads to* `if (!`C_1 `||` `!`C_2`) expr; else ...`

As a consequence, it is obvious that P^{IF} consists of the negated polyhedra $\overline{P_1}$ and $\overline{P_2}$. Furthermore, the use of the union

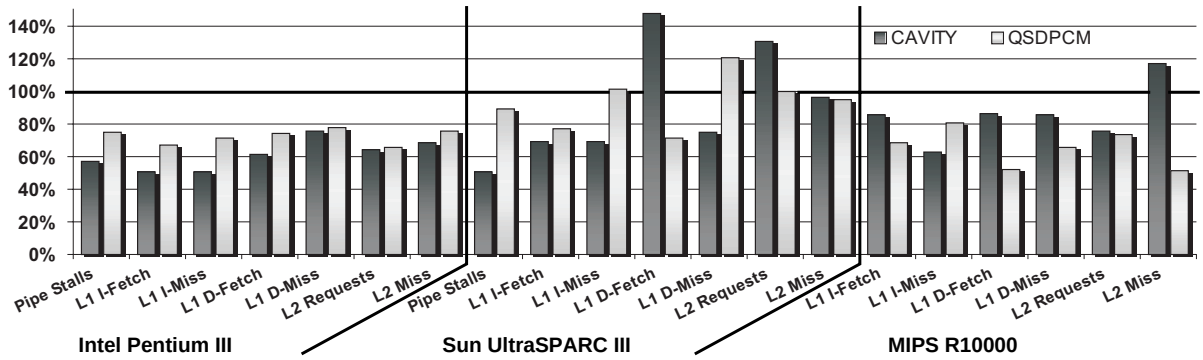


Figure 7. Relative Cache and Pipeline Behavior after Advanced Code Hoisting

operator for polyhedra in this situation can be explained with the switching of the logical operator from AND *to* OR *due to de Morgan's rule. All in all, the following polyhedron corresponds to the above* if*-statement:*

$$\begin{aligned} P^{IF} &= \overline{P_1} \cup \overline{P_2} \\ &= \{\,(\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid \text{-}4*\mathtt{i}-\mathtt{j} \geq \text{-}11\,\} \cup \\ &\quad \{\,(\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid \mathtt{i}+5*\mathtt{j} \geq 29\,\} \end{aligned}$$

Analogously, the if*-statement* `if (`C_1 `||` C_2`) ... else` *expr*`;` *is represented correctly by*

$$\begin{aligned} P^{IF} &= \overline{P_1} \cap \overline{P_2} \\ &= \{\,(\mathtt{i},\mathtt{j}) \in \mathbb{Z}^2 \mid (\text{-}4*\mathtt{i}-\mathtt{j} \geq \text{-}11) \wedge \\ &\qquad\qquad (\mathtt{i}+5*\mathtt{j} \geq 29)\,\} \end{aligned}$$

Theorem 2 *For a given nest* Γ *of control flow structures, let* $\gamma_m = (C_1 \oplus \ldots \oplus C_n) \in \Upsilon$ *be an* if*-statement in compliance with definition 3. Let* P_m^{IF} *be the corresponding polyhedron according to definition 8.*

P_m^{IF} contains exactly those values of the index variables of Λ *for which* γ_{m+1} *is executed.*

Using theorems 1 and 2 expressing the correctness of the polytopes for *for*-loops and *if*-statements, it can be proven that the polytope P^Γ (cf. definition 6) reflects Γ accurately.

Theorem 3 *Let* $\Gamma = (\gamma_1, \ldots, \gamma_M, \gamma_{M+1})$ *be a sequence of control flow structures composed of a loop nest* Λ *and a set of* if*-statements* Υ *(see definition 4). Furthermore, let* P^Γ *be the polytope associated to* Γ *according to definition 6.*

P^Γ contains exactly those values of the index variables of Λ *for which the expression expr* $= \gamma_{M+1}$ *is executed.*

Theorem 3 is helpful for the computation of the execution frequency of an expression *expr* since it states that P^Γ contains exactly one point $i \in \mathbb{Z}^N$ for every time *expr* is executed. As a consequence, the execution frequency of *expr* is equal to the size of P^Γ:

Corollary 1 *Let* $\Gamma = (\gamma_1, \ldots, \gamma_M, \gamma_{M+1})$ *be a sequence of control flow structures complying with definition 4 and* P^Γ *be the corresponding polytope according to definition 6.*

The execution frequency of the expression expr $= \gamma_{M+1}$ *is equal to the size of* P^Γ*: #expr* $= |P^\Gamma|$

The computation of the number of points included in a polytope is *#P*-complete [14] in terms of its number of linear (in-) equations and their dimensions. In order to determine the execution frequency of an expression *expr*, the techniques described in [5] for the computation of a polytope's size are applied to P^Γ. A detailed description of these techniques is omitted here since it is beyond the scope of this paper. In short, the parametric vertices are computed based on the linear constraints of P^Γ in a first step. Using these vertices, the so called Ehrhart polynomial is determined which is a parametric representation of the number of integer points of P^Γ. For more details, the interested reader is referred to [5].

Due to the *#P*-completeness of the techniques mentioned above, the computation of the execution frequency of an expression *expr* does not have a polynomial complexity. Instead, the worst-case complexity of the techniques described in this section is exponential. However, as will be shown in the following Section 4, the use of this methodology for real-life applications leads to feasibly short runtimes of only a few CPU seconds.

4. Benchmarking Results

The techniques presented in the previous section are fully automated using the SUIF intermediate format [21] and the polyhedral library Polylib [15]. ACH is applied to the source codes of two representative benchmarks having passed the DTSE transformations (cf. Section 2). The CAVITY benchmark is a medical tomography image processor [2], and the QSDPCM application [19] performs scene adaptive coding. The efficiency of the polyhedral analysis employed for ACH is apparent by virtue of the low runtimes required for the optimization of these benchmarks. Using a Pentium 4 based host machine (2.6 GHz), ACH applied to

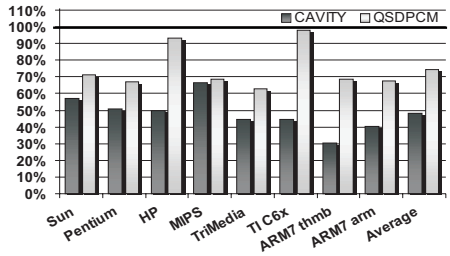
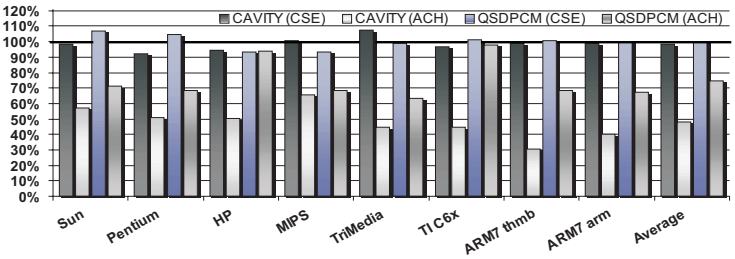


Figure 8. a) Execution Times



b) Runtime Comparison ACH / CSE

CAVITY requires only 0.44 CPU seconds. In the case of QSDPCM, 29.9 seconds are necessary. For obtaining the results presented in the following, the benchmarks are compiled and executed before and after ACH. Compilers are always invoked with all optimizations enabled so that highly optimized code is generated. Table 1 lists the compilers used for machine code generation for all processors considered in this paper.

Processor	*Compiler*	*Version*
Sun UltraSPARC III	Sun WorkShop	9.0
Intel Pentium III	GNU gcc	2.95.3
HP-PA 9000	HP Softbench	A.01.18
MIPS R10000	MIPSpro	7.2.1
TriMedia-1000	Philips SDE	V5.7.1
TI TMS320C62	TI CCS	3.01
ARM7TDMI thumb	ARM SDT	2.50
ARM7TDMI arm	ARM SDT	2.50

Table 1. Compilers used for Code Generation

4.1. Cache and Pipeline Behavior

Figure 7 shows the effects of advanced code hoisting on the caches and pipelines of an Intel Pentium III, Sun UltraSPARC III and a MIPS R10000 processor. To obtain these results, the benchmarks were compiled and executed on the processors while monitoring performance-measuring counters available in the CPU hardware. This way, reliable values can be generated without using erroneous cache simulation software. The figure shows the performance values for the optimized benchmarks as a percentage of the unoptimized versions denoted as 100%.

As can be seen from this figure, ACH is very beneficial for the Pentium. The increased reuse of computed results leads to improved behavior of pipeline and all caches. The measured reductions of instruction fetches (L1 I-Fetch) by 32.8%–49.1% implies reduced pipeline stalls (24.7%–42.5% improvement) and significantly less I-cache misses (28.5%–49.1% reduction). These factors imply an improved behavior of both the L1 D-cache and the unified L2 cache as can be seen from Figure 7. For these caches, improvements in the order of magnitude of 22%–38.5% were measured.

The results for the Sun CPU also show a significantly improved locality of instruction fetches after ACH. Columns L1 I-Fetch indicate reductions of instruction executions by 22.7%–30.3%. As a consequence, the amount of pipeline stalls decreases by 10.6%–48.9%. For QSDPCM, I-cache misses remain constant after the optimization, but an improvement of 30.7% was achieved for CAVITY. Column L1 D-Fetch shows for CAVITY that ACH increases the number of data fetches by 47.9%. This is due to the insertion of spill code by the Sun compiler for the 27 new local variables holding CSEs after the optimization. In contrast, ACH leads to reductions of D-fetches by 28.4% for QSDPCM. Column L1 D-Miss shows reductions of L1 D-cache misses by 24.8% for CAVITY, whereas a growth by 20.7% was measured for QSDPCM. Due to the worse data locality of CAVITY, the amount of pipeline stalls increases by 55.6%. Concerning the L2 cache, an increase of cache accesses by 30.6% for CAVITY was measured, whereas this parameter remains unchanged for QSDPCM. L2 cache misses are slightly reduced by 3.1%–4.8%.

Figure 7 shows that ACH applied to the benchmarks leads to improvements for almost all caches of the MIPS CPU. These improvements range from 13.6% (L1 D-fetch for CAVITY) up to 48.6% (L2 Miss for QSDPCM). For CAVITY, L2 cache misses increase by 17.7% which is the only category where a degradation of performance was observed.

4.2. Execution Times and Code Sizes

Figure 8 shows the impact of advanced code hoisting on the runtimes of the benchmarks using a large set of different processors. Amongst the three processors considered in the previous section, runtimes were also measured for a HP-9000, TriMedia TM-1000, TI C6x and an ARM7TDMI processor, the latter both in 16-bit thumb- and 32-bit arm-mode. Figure 8a) shows that all processors benefit from ACH. For CAVITY, overall speed-ups ranging from 33.4% (MIPS) up to 69.6% (ARM thumb) were measured. For QSDPCM, a maximum acceleration of 36.9% was observed (TriMedia). In contrast, ACH only leads to a marginal speed-up of 2.3% for the TI DSP. On average over all eight processors, the source code optimization presented in this paper leads to a high gains between 25.5%–52%.

As can be seen from this figure, ACH also leads to large

speed-ups for the Sun UltraSPARC processor even though its L1 data cache is affected adversely by ACH (see previous section). The actual runtime savings reported here are due to the fact that ACH eliminated dozens of modulo computations and integer divisions. These operations are extremely costly for the Sun CPU so that the elimination of a modulo / division still leads to speed-ups even if the new local variable needs to be spilled to and from the main memory.

In order to highlight the benefits achieved by the computation of execution frequencies during ACH, a direct comparison of runtimes after a conventional CSE applied whenever possible and after ACH is depicted in Figure 8b). This diagram clearly shows that the application of a CSE to the benchmarks' source codes does not have a significant impact on the runtimes. In some situations, CSE leads to speed-ups of up to 11.3% (CAVITY / Sun), whereas runtime degradations of up to 7.3% (CAVITY / TriMedia) were measured in other situations. On average for all considered CPUs, a maximum improvement of 2.8% was observed after CSE. Figure 8b) leads to the conclusion that the improvements achieved by ACH clearly originate from the novel steering criterion based on the global computation of execution frequencies which is used to control CSE and LICM.

In addition to the runtime savings, ACH also leads to significant code size reductions (cf. Figure 9). This is due to the fact that the elimination and reuse of frequently used CSEs explicitly removes code from an application. Implicitly, some code can be added by the compiler since ACH augments register pressure and might lead to the generation of spill code. As can be seen from the figure, the average code size reductions after ACH for all processors amount to 10.5%–20%.

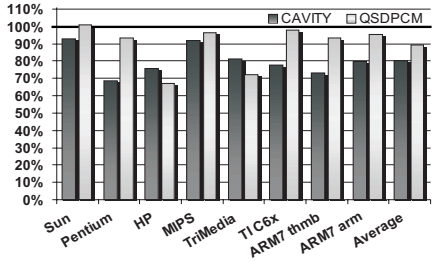


Figure 9. Code Sizes after ACH

4.3. Energy Consumption

Figure 10 shows the effects of ACH on memory accesses and energy consumption using an instruction-level energy model [18] for the ARM7 core considering bit-toggles and offchip-memories and having an accuracy of 1.7%. The first four columns of Figure 10 depict the relative number of various kinds of memory accesses, whereas the last three columns show the changes in energy consumption. As usual, the 100% base line denotes the values of the unoptimized benchmarks.

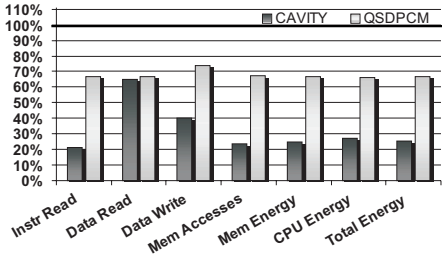


Figure 10. Energy Consumption

Column Instr Read of Figure 10 shows that ACH is able to reduce the number of instructions fetched by the ARM7 significantly from 33.1% up to 78.5%. The simultaneous elimination and hoisting of code performed by ACH reduces the number of read accesses to the data memory (Data Read) by 33.3%–35.2%. The large reductions of writing memory accesses by 26.4%–59.9% – cf. column Data Write – originate from a better register allocation after ACH. Overall, a diminution of total memory accesses between 32.8%–76.4% was achieved.

The measured savings in terms of energy consumption of both the memory and the ARM7 core are of the same order of magnitude as the total reductions of memory accesses. As can be seen, the energy consumption of both components drops between 33.4% and 75.1%. These factors lead to total energy savings of 33.4%–74.5% for the combination of memory and processor (see column Total Energy of Figure 10).

5. Summary and Conclusions

This paper presented a new source code optimization called advanced code hoisting (ACH). This technique is an elaborate combination of already known compiler optimizations (CSE and LICM) with a formal mathematical criterion steering the application of the mentioned optimizations.

It turned out that the elimination of CSEs might not be beneficial if the control flow surrounding an expression is not considered. As a consequence, ACH is based on a formal model in order to compute the execution frequencies of a CSE. The control flow given by nested *for*-loops and *if*-statements is represented by a polytope model which is then used to compute how many times a CSE surrounded by all these loops and *if*-statements is executed.

The results provided in this paper demonstrate that this new optimization is highly beneficial. First, significant speed-ups of the studied benchmark applications by 25.5%–52% were achieved. These accelerations are mainly due to the reduced number of instruction executions because of the high reuse of already computed results. Second, the fact that complex expressions are eliminated has the effect that the code sizes of the benchmarks decrease by 10.5%–20% after ACH. Third, considerable energy savings by 33.4%–74.5% were measured which are due to the reduced amounts of main memory accesses for fetching instructions and data.

Classical literature on compiler construction suggests not to perform a CSE without considering compiler and processor internal information about register pressure and the size of the register file since this may degrade a program's performance under particular circumstances. In the context of ACH, an obvious runtime degradation was not observed

for any processor. In every case, large speed-ups and energy savings were measured, justifying the approach presented in this paper which only focuses on a program's structure instead of actual processor architectures. However, for one actual processor, ACH causes a significant degradation of D-cache performance. This observation leads to the conclusion that a more sophisticated mechanism taking register pressure and spill code generation into account can probably achieve even higher gains due to a better D-cache behavior. This is part of our future work.

References

[1] D. F. Bacon, S. L. Graham et al. Compiler Transformations for High-Performance Computing. *ACM Computing Surv.*, 26(4), 1994.

[2] M. Bister, Y. Taeymans et al. Automatic Segmentation of Cardiac MR Images. *IEEE Journ. on Computers in Cardiology*, 1989.

[3] Q. Cai and J. Xue. Optimal and Efficient Speculation-Based Partial Redundancy Elimination. In *Proc. of CGO*, San Francisco, 2003.

[4] F. Catthoor, K. Danckaert et al. *Data Access and Storage Management for embedded programmable Processors*. Kluwer, Boston, 2002.

[5] P. Clauss and V. Loechner. Parametric Analysis of polyhedral Iteration Spaces. *Journ. of VLSI Signal Processing*, 19(2), 1998.

[6] H. Falk and P. Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer, Boston, 2004.

[7] H. Falk, P. Marwedel and F. Catthoor. *Control Flow driven Splitting of Loop Nests at the Source Code Level*, volume Embedded Software for SOC. Kluwer, Boston, 2003.

[8] H. Falk and M. Verma. Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization. In *Proc. of SCOPES*, Amsterdam, 2004.

[9] E. d. Greef, F. Catthoor et al. Memory Size Reduction through Storage Order Optimization for embedded parallel Multimedia Applications. *Parallel Computing*, 23(12), 1997.

[10] R. Gupta, D. A. Berson et al. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *Proc. of ICCL*, Chicago, 1998.

[11] S. Gupta, M. Miranda et al. Analysis of high-level Address Code Transformations for programmable Processors. In *Proc. of DATE*, Paris, 2000.

[12] R. N. Horspool and H. C. Ho. Partial Redundancy Elimination Driven by a Cost-Benefit Analysis. In *Proc. of ICCSSE*, Herzliya, 1997.

[13] M. Janssen. *Word-level Algebraic Optimisation Techniques for Accelerator Data-Paths and custom Address Generators*. PhD thesis, Katholieke Universiteit Leuven, Leuven, Jan. 2000.

[14] V. Kaibel and M. E. Pfetsch. *Some algorithmic Problems in Polytope Theory*, volume Algebra, Geometry and Software Systems. Springer, Berlin, 2003.

[15] V. Loechner. PolyLib: A Library for manipulating parameterized Polyhedra. *http://icps.u-strasbg.fr/polylib/*, 1999.

[16] V. Loechner, B. Meister et al. Precise Data Locality Optimization of nested Loops. *The Journal of Supercomputing*, 21, 2002.

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[18] S. Steinke, M. Knauer et al. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc. of PATMOS*, Yverdon-Les-Bains, 2001.

[19] P. Strobach. A new technique in scene adaptive coding. In *Proc. of EUSIPCO*, Grenoble, 1988.

[20] D. K. Wilde. A Library for doing polyhedral Operations. Technical Report 785, IRISA Rennes, France, Dec. 1993.

[21] R. Wilson, R. French et al. An Overview of the SUIF Compiler System. *http://suif.stanford.edu/suif/suif1*, 1995.

[22] S. Wuytack, J.-P. Diguet et al. Formalized Methodology for Data Reuse Exploration for Low-Power hierarchical Memory Mappings. *IEEE Trans. VLSI*, 6(4), 1998.