Control Flow driven Splitting of Loop Nests at the Source Code Level

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

Abstract

This paper presents a novel source code transformation for control flow optimization called loop nest splitting which minimizes the number of executed if-statements in loop nests of embedded multimedia applications. The goal of the optimization is to reduce runtimes and energy consumption. The analysis techniques are based on precise mathematical models combined with genetic algorithms. Due to the inherent portability of source code transformations, a very detailed benchmarking using 10 different processors can be performed. The application of our implemented algorithms to three real-life multimedia benchmarks leads to average speed-ups by 23.6% – 62.1% and energy savings by 19.6% – 57.7%. Furthermore, our optimization also leads to advantageous pipeline and cache performance.

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. Many of these applications are data-dominated using large amounts of data memory. Typically, such applications consist of deeply nested *for*-loops. Using the loops' index variables, addresses are calculated for data manipulation. The main algorithm is usually located in the innermost loop. Often, such an algorithm treats particular parts of its data specifically, e. g. an image border requires other manipulations than its center. This boundary checking is implemented using *if*-statements in the innermost loop (see e. g. figure 1, an MPEG 4 full search motion estimation kernel [5]).

This code fragment has several properties making it suboptimal w. r. t. runtime and energy consumption. First, the *if*-statements lead to a very irregular control flow. Any jump instruction in a machine program causes a control hazard for pipelined processors [11]. This means that the pipeline needs to be stalled for some instruction cycles, so as to prevent the execution of incorrectly prefetched instructions.

Second, the pipeline is also influenced by data references, since it can also be stalled during data memory accesses. In loop nests, the index variables are accessed very frequently resulting in pipeline stalls if they can not be kept in processor registers. Since it has been shown that 50% – 75% of the power consumption in embedded multimedia systems is caused by memory accesses [12, 17], frequent transfers of index variables across memory hierarchies contribute negatively to the total energy balance.

```
for (x=0; x<36; x++) { x1=4*x;
 for (y=0; y<49; y++) { y1=4*y;   /* y loop */
  for (k=0; k<9; k++) { x2=x1+k-4;
   for (l=0; l<9; l++) { y2=y1+l-4;
    for (i=0; i<4; i++) { x3=x1+i; x4=x2+i;
     for (j=0; j<4; j++) { y3=y1+j; y4=y2+j;
      if (x3<0 || 35<x3 || y3<0 || 48<y3)
         then_block_1; else else_block_1;
      if (x4<0 || 35<x4 || y4<0 || 48<y4)
         then_block_2; else else_block_2; }}}}}}
```

Figure 1. A typical Loop Nest (from MPEG 4)

Finally, many instructions are required to evaluate the *if*-statements, also leading to higher runtimes and power consumption. For the MPEG 4 code above, all shown operations are in total as complex as the computations performed in the *then*- and *else*-blocks of the *if*-statements.

In this article, a new formalized method for the analysis of *if*-statements occuring in loop nests is presented solving a particular class of the NP-complete problem of the satisfiability of integer linear constraints. Considering the example shown in figure 1, our techniques are able to detect that

- the conditions `x3<0` and `y3<0` are never true,
- both *if*-statements are true for $x \geq 10$ or $y \geq 14$.

Information of the first type is used to detect conditions not having any influence on the control flow of an application. This kind of redundant code (which is not typical dead code, since the results of these conditions are used within the *if*-statement) can be removed from the code, thus reducing code size and computational complexity of a program.

Using the second information, the entire loop nest can be rewritten so that the total number of executed *if*-statements is minimized (see figure 2). In order to achieve this, a new *if*-statement (the *splitting-if*) is inserted in the `y` loop testing the condition `x`$\geq$`10 || y`$\geq$`14`. The *else*-part of this new *if*-statement is an exact copy of the body of the original `y` loop shown in figure 1. Since all *if*-statements are fulfilled when the splitting-if is true, the *then*-part consists of the body of the `y` loop without any *if*-statements and associated *else*-blocks. To minimize executions of the splitting-if for values of $y \geq 14$, a second `y` loop is inserted in the *then*-part counting from the current value of `y` to the upper bound 48. The correctly transformed code is illustrated in figure 2.

```
for (x=0; x<36; x++) { x1=4*x;
 for (y=0; y<49; y++)
  if (x>=10 || y>=14)        /* Splitting-If */
   for (; y<49; y++)        /* Second y loop */
    for (k=0; k<9; k++)
     ...               /* l- & i-loop omitted */
       for (j=0; j<4; j++) {
        then_block_1; then_block_2; }
  else { y1=4*y;
   for (k=0; k<9; k++) { x2=x1+k-4;
    ...               /* l- & i-loop omitted */
      for (j=0; j<4; j++) { y3=y1+j; y4=y2+j;
       if (0 || 35<x3 || 0 || 48<y3)
          then_block_1; else else_block_1;
       if (x4<0 || 35<x4 || y4<0 || 48<y4)
          then_block_2; else else_block_2; }}}}}}
```

Figure 2. Loop Nest after Splitting

As shown by this example, our technique is able to generate linear control flow in the hot-spots of an application. Furthermore, accesses to memory are reduced significantly since a large amount of branching, arithmetic and logical instructions and index variable accesses is removed.

Section 2 of this paper gives a survey of related work. Section 3 presents the analytical models and algorithms for loop nest splitting. Section 4 describes the benchmarking results, and section 5 summarizes and concludes this paper.

2. Related Work

Loop transformations have been described in literature on compiler design for many years (see e. g. [2, 11]) and are often integrated into today's optimizing compilers. Classical *loop splitting* (or *loop distribution* / *fission*) creates several loops out of an original one and distributes the statements of the original loop body among all new loops. The main goal of this optimization is to enable the parallelization of a loop due to fewer data dependencies [2] and to possibly improve I-cache performance due to smaller loop bodies. In [7] it is shown that loop splitting leads to increased energy consumption of the processor and the memory system. Since the computational complexity of a loop is not reduced, this technique does not solve the problems that are due to the properties discussed in section 1.

Loop unswitching is applied to loops containing loop-invariant *if*-statements [11]. The loop is then replicated inside each branch of the *if*-statement, reducing the branching overhead and decreasing code sizes of the loops [2]. The goals of loop unswitching and the way how the optimization is expressed are equivalent to the topics of section 1. But the fact that the *if*-statements must not depend on index variables makes loop unswitching unsuitable for applying it to multimedia programs. It is the contribution of the techniques presented in this paper that we explicitly focus on loop-variant conditions. Since our analysis techniques go far beyond those required for loop splitting or unswitching and have to deal with entire loop nests and sets of index variables, we call our optimization technique *loop nest splitting*.

In [9], classical loop splitting is applied in conjunction with function call insertion at the source code level to improve the I-cache performance. After the application of loop splitting, a large reduction of I-cache misses is reported for one benchmark. All other parameters (instruction and data memory accesses, D-cache misses) are worse after the transformation. All results are generated with cache simulation software which is known to be unprecise, and the runtimes of the benchmark are not considered at all.

Source code transformations are studied in literature for many years. In [6], array and loop transformations for data transfer optimization are presented by means of a medical image processing algorithm [3]. The authors only focus on the illustration of the optimized data flow and thus neglect that the control flow gets very irregular since many additional *if*-statements are inserted. This impaired control flow has not yet been targeted by the authors. As we will show in section 4, loop nest splitting applied as postprocessing stage is able to remove the control flow overhead introduced by [6] with simultaneous further data transfer optimization.

3. Analysis and Optimization Algorithm

This section presents the techniques required for loop nest splitting consisting of four sequential tasks. First, conditions are checked for satisfiability (3.1). Second, an optimized search space for each satisfiable condition is created (3.2). Third, all local search spaces are combined to a global search space (3.3) which has to be explored finally (3.4). Before going into details (cf. also [4] for broader descriptions), some preliminaries are required.

Definition 1:

1. Let $\Lambda = \{L_1, \ldots, L_N\}$ be a *loop nest* of depth N, where L_l denotes a single loop.
2. Let i_l, lb_l and ub_l be the *index variable*, *lower bound* and *upper bound* of loop $L_l \in \Lambda$ with $lb_l \leq i_l \leq ub_l$.

The optimization goal for loop nest splitting is to determine values lb'_l and ub'_l for every loop $L_l \in \Lambda$ with

- $lb'_l \geq lb_l$ and $ub'_l \leq ub_l$,
- all loop-variant *if*-statements in Λ are satisfied for all values of the index variables i_l with $lb'_l \leq i_l \leq ub'_l$,
- loop nest splitting by all values lb'_l and ub'_l leads to the minimization of *if*-statement execution.

The values lb'_l and ub'_l are used for the construction of the splitting *if*-statement. The techniques described in the following require that some preconditions are met:

1. All loop bounds lb_l and ub_l are constants.
2. *If*-statements have the format `if` $(C_1 \oplus C_2 \oplus \ldots)$ where C_x are loop-variant conditions that are combined with logical operators $\oplus \in \{\&\&, ||\}$.
3. Loop-variant conditions C_x are affine expressions of i_l and can thus be translated to the format $C_x = \sum_{l=1}^{N} (c_l * i_l) + c \geq 0$ for constants $c_l, c \in \mathbb{Z}$.

Precondition 2 is only due to the current state of implementation of our tools. By application of *de Morgan's* rule on an expression `!(`$C_1 \oplus C_2$`)` and inversion of the comparators in C_1 and C_2, the logical *NOT* can also be modeled in *if*-statements. Since all boolean functions can be expressed with `&&`, `||` and `!`, precondition 2 is not a limitation. Without loss of generality, a condition `a==b` can be rewritten as `a`$\geq$`b && b`$\geq$`a` (`a!=b` analogous) so that the required operator $\geq$ of precondition 3 is not a restriction, either.

3.1. Condition Satisfiability

In the first phases of the optimization algorithm, all affine conditions C_x are analyzed separately. Every condition defines a subset of the total iteration space of a loop nest Λ. This total iteration space is an N-dimensional space limited by all loop bounds lb_l and ub_l. An affine condition C_x can thus be modeled as follows by a polytope:

Definition 2:

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

Every condition C_x can be represented by a polytope P_x by generating inequalities for the affine condition C_x itself and for all loop bounds. For this purpose, an improved variant of the Motzkin algorithm [10] is used and combined with some simplifications removing redundant constraints [15].

After that, we can determine in constant time if the number of equalities $Ax = a$ of P_x is equal to the dimension of P_x plus 1. If this is true, P_x is overconstrained and defines the empty set as proven by Wilde [15]. If instead P_x only contains the constraints for the loop bounds, C_x is satisfied for all values of the index variables i_l. Such conditions that are always satisfied or unsatisfied are replaced by their respective truth value in the *if*-statement and are no longer considered during further analysis.

3.2. Condition Optimization

For conditions $C = \sum_{l=1}^{N} (c_l * i_l) + c \geq 0$ that are not eliminated by the previous method, a polytope P_C is created out of values $lb'_{C,l}$ and $ub'_{C,l}$ for all loops $L_l \in \Lambda$ such that C is satisfied for all index variables i_l with $lb'_{C,l} \leq i_l \leq ub'_{C,l}$. These values are chosen so that a loop nest splitting using $lb'_{C,l}$ and $ub'_{C,l}$ minimizes the execution of *if*-statements.

Since affine expressions (see precondition 3) are linear monotone functions, it is unnecessary to deal with two values $lb'_{C,l}$ and $ub'_{C,l}$. If C is true for a value $v \in [lb'_{C,l}, ub'_{C,l}]$ and $c_l > 0$, C must also be true for $v+1, v+2, \ldots$ ($c_l < 0$ analogous). This implies that either $lb'_{C,l} = lb_l$ or $ub'_{C,l} = ub_l$. Thus, our optimization algorithm only computes values $v'_{C,l}$ for C and all loops $L_l \in \Lambda$ with $v'_{C,l} \in [lb_l, ub_l]$. $v'_{C,l}$ designates one of the former values $lb'_{C,l}$ or $ub'_{C,l}$, the other one is set to the correct upper or lower loop bound.

The optimization of the values $v'_{C,l}$ is done by a genetic algorithm (*GA*) [1]. The chromosome length is set to the number of index variables i_l C depends on: $|\{c_l | c_l \neq 0\}|$. For every such variable i_l, a gene on the chromosome represents $v'_{C,l}$. Using the $v'_{C,l}$ values of the fittest individual, the optimized polytope P_C is generated as the result of this phase:

$$P_C = \{(x_1, \ldots, x_N) \in \mathbb{Z}^N \mid lb_l \leq x_l \leq ub_l,\ L_l \in \Lambda,\ x_l \geq v'_{C,l} \text{ if } c_l > 0,\ x_l \leq v'_{C,l} \text{ if } c_l < 0\}$$

The fitness of an individual I is the higher, the fewer *if*-statements are executed when splitting Λ using the values $v'_{C,l}$ encoded in I. Since only the fittest individuals are selected, the GA minimizes the execution of *if*-statements. Consequently, an individual implying that C is not satisfied has a very low fitness. For an individual I, the fitness function computes the number of executed *if*-statements IF_{Tot}. Therefore, the following values are required:

Definition 3:

1. The *total iteration space* (*TS*) of a loop nest Λ is the total number of executions of the body of loop L_N: $TS = \prod_{l=1}^{N} (ub_l - lb_l + 1)$
2. The *constrained iteration space* (*CS*) is the total iteration space reduced to the ranges r_l represented by $v'_{C,l}$: $CS = \prod_{l=1}^{N} r_l$ and $r_l = \begin{cases} ub_l - lb_l + 1 & \text{if } c_l = 0, \\ ub_l - v'_{C,l} + 1 & \text{if } c_l > 0, \\ v'_{C,l} - lb_l + 1 & \text{else} \end{cases}$
3. The *innermost loop* λ is the index of the loop where a loop nest splitting has to be done for a given set of $v'_{C,l}$ values: $\lambda = \max\{l \mid L_l \in \Lambda, r_l \neq ub_l - lb_l + 1\}$

The aggregate number of executed *if*-statements IF_{Tot} is computed as follows:

- $IF_{Tot} = IF_{Orig} + IF_{Split}$ (*if*-statements in the *else*-part of the splitting-if plus the splitting-if itself)
- $IF_{Orig} = TS - CS$ (all iterations of Λ minus the ones where the splitting-if evaluates to true)
- $IF_{Split} = TP_{Split} + EP_{Split}$ (splitting-if is evaluated as often as its *then*- and *else*-parts are executed)
- $TP_{Split} = CS \ / \prod_{l=\lambda+1}^{N} (ub_l - lb_l + 1) * r_\lambda$ (All loop nest iterations where splitting-if is true divided by all loop iterations located in the *then*-part)
- $EP_{Split} = IF_{Orig} \ / \prod_{l=\lambda+1}^{N} (ub_l - lb_l + 1)$ (All loop nest iterations where splitting-if is false divided by all loop iterations located in the *else*-part)

The computation of IF_{Split} is that complex because the duplication of the innermost loop λ in the *then*-part of the splitting-if (e. g. the y loop in figure 2) has to be considered. Since IF_{Tot} does not depend linearly on $v'_{C,l}$, a modeling of this optimization problem using integer linear programming (*ILP*) is impossible, so that we chose to use a GA.

Example: *For a condition C =* `4*x+k+i-40>=0` *and the loop nest of figure 1, our GA can generate the individual* $I = (10,0,0)$. *The numbers encoded in I denote the values* $v'_{C,\mathtt{x}}$, $v'_{C,\mathtt{k}}$ *and* $v'_{C,\mathtt{i}}$ *so that the following intervals are defined:* $\mathtt{x} \in [10,35]$, $\mathtt{k} \in [0,8]$, $\mathtt{i} \in [0,3]$. *Since only variable* `x` *is constrained by I, the* `x`*-loop would be split using the condition* `x>=10`*. The formulas above imply a total execution of 12,701,020 if-statements (*IF_{Tot}*).*

3.3. Global Search Space Construction

After the first GA (see section 3.2), a set of *if*-statements $IF_i = (C_{i,1} \oplus C_{i,2} \oplus \ldots \oplus C_{i,n})$ consisting of affine conditions $C_{i,j}$ and their associated optimized polytopes $P_{i,j}$ are given. For determining index variable values where all *if*-statements in a program are satisfied, a polytope G modeling the global search space has to be created out of all $P_{i,j}$.

In a first step, a polytope P_i is built for every *if*-statement IF_i. Therefore, the conditions of IF_i are traversed in their natural execution order π which is defined by the associativity and precedence rules of the operators && and ||. P_i is initialized with $P_{i,\pi(1)}$. While traversing the conditions of *if*-statement i, P_i and $P_{i,\pi(j)}$ are connected either with the intersection or union operators for polytopes: $\forall j \in \{2,\ldots,n\}$:

$$P_i = P_i \uplus P_{i,\pi(j)} \text{ with } \uplus = \begin{cases} \cap & \text{if } C_{i,\pi(j-1)} \;\&\&\; C_{i,\pi(j)} \\ \cup & \text{if } C_{i,\pi(j-1)} \;||\; C_{i,\pi(j)} \end{cases}$$

P_i models those ranges of the index variables where one *if*-statement i is satisfied. Since all *if*-statements need to be satisfied, the global search space G is built by intersecting all P_i: $G = \bigcap P_i$. Since polyhedra are not closed under the union operator, the P_i defined above are no real polytopes. Instead, we use finite unions of polyhedra for which the union operator is closed [15].

3.4. Global Search Space Exploration

Since all P_i are finite unions of polytopes, the global search space G also is a finite union of polytopes. Each polytope of G defines a region where all *if*-statements in a loop nest are satisfied. After the construction of G, appropriate regions of G have to be selected so that once again the total number of executed *if*-statements is minimized after loop nest splitting.

Since unions of polytopes (i. e. logical *OR* of constraints) can not be modeled using ILP, a second GA is used here. For a given global search space $G = R_1 \cup R_2 \cup \ldots \cup R_M$, each individual I consists of a bit-vector where bit I_r determines whether region R_r of G is selected or not: $I = (I_1, I_2, \ldots, I_M)$ with $I_r = \begin{cases} 1 & \text{if region } R_r \text{ is selected,} \\ 0 & \text{else} \end{cases}$

Definition 4:

1. For an individual I, G_I is the global search space G reduced to those regions selected by I: $G_I = \bigcup R_r$ with $I_r = 1$
2. The *innermost loop* λ is the index of the loop where the loop nest has to be split when considering G_I: $\lambda = \max\{l \mid L_l \in \Lambda, i_l \text{ is used in } G_I\}$
3. ι_l denotes the number of *if*-statements located in the body of loop L_l but not in any other loop L'_l nested in L_l. For figure 1, ι_j is equal to 2, all other ι_l are zero.
4. IF_l denotes the number of executed *if*-statements when the loop nest $\Lambda' = \{L_l, \ldots, L_N\}$ would be executed:

$$\begin{aligned} IF_l &= (ub_l - lb_l + 1) * (IF_{l+1} + \iota_l) \\ IF_{N+1} &= 0 \end{aligned}$$

The fitness of an individual I represents the number IF_I of *if*-statements that are executed when splitting Λ using the regions R_r selected by I. IF_I is incremented by one for every execution of the splitting-if. If the splitting-if is true, the counter remains unchanged. If not, IF_I is incremented by the number of executed original *if*-statements (see figure 3).

```
IF_I = 0;
∀i_1 ∈ [lb_1, ub_1]
  ...
    ∀i_λ ∈ [lb_λ, ub_λ]
      IF_I = IF_I + 1;
      if (G_I = true for (i_1, ..., i_λ))
        i_λ = ub_λ;
      else
        IF_I = IF_I + IF_{λ+1};
```

Figure 3. Global If-Statement Counter

After the GA has terminated, the innermost loop λ of the best individual defines where to insert the splitting-if. The regions R_r selected by this individual serve for the generation of the conditions of the splitting-if and lead to the minimization of *if*-statement executions.

4. Benchmarking Results

The techniques presented in section 3 are fully implemented using the SUIF [16], Polylib [15] and PGAPack [8] libraries. Both GA's use the default parameters provided by [8] (population size 100, replacement fraction 50%, 1,000 iterations). Our tool was applied to three multimedia programs. First, a medical tomography image processor (*CAVITY* [3]) having passed the so called DTSE transformations [6] is used. We apply loop nest splitting to this transformed application for showing that we are able to remove the overhead introduced by DTSE. The second benchmark is an MPEG 4 full search motion estimation (*ME* [5], see section 1), and the QSDPCM algorithm [14] for scene adaptive coding serves as third test driver.

Since all polyhedral operations used [15] have exponential worst case complexity, loop nest splitting as a whole also has exponential complexity. Nevertheless, the effective runtimes of our tool are very low, from 0.41 CPU seconds (QSDPCM) up to 1.58 seconds (CAVITY) are required for optimization on an AMD Athlon running at 1.3 GHz. For obtaining the results presented in the following, the benchmarks are compiled and executed before and after loop nest splitting. Compilers are always invoked with all optimizations enabled so that highly optimized code is generated.

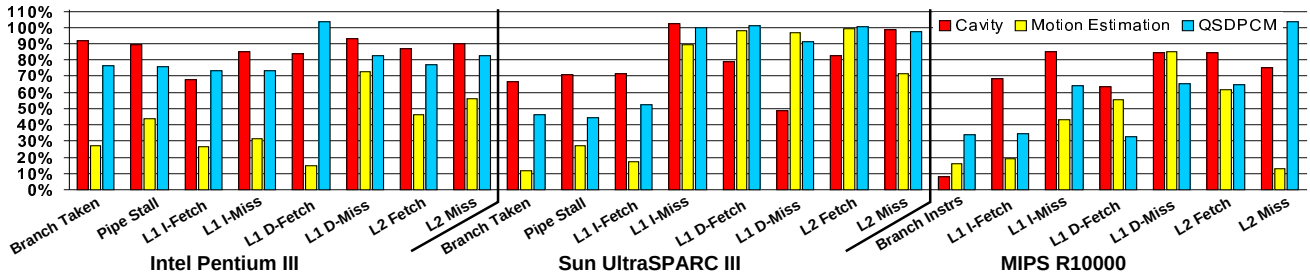


Figure 4. Pipeline and Cache Behavior after Loop Nest Splitting

4.1. Pipeline and Cache Behavior

Figure 4 shows the effects of loop nest splitting 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 the columns Branch Taken and Pipe Stall, we are able to generate a more regular control flow for all benchmarks. The number of taken branch instructions is reduced between 8.1% (CAVITY Pentium) and 88.3% (ME Sun) consequently leading to similar reductions of pipeline stalls (10.4% – 73.1%). For the MIPS, a reduction of executed branch instructions between 66.3% (QSDPCM) and 91.8% (CAVITY) were observed. The very high gains for the Sun CPU are due to its complex pipeline consisting of 14 stages which is very sensitive to stalls.

The hardware counters also clearly show that the behavior of the L1 I-cache is improved significantly. The number of I-fetches is reduced by 26.7% (QSDPCM Pentium) – 82.7% (ME Sun), large improvements of I-cache misses are reported for the Pentium and MIPS (14.7% – 68.5%). For the Sun, this parameter remains almost unchanged. Due to the removal of index variable accesses, the L1 D-caches also benefit in several cases. Fetches from the D-cache are reduced by 1.7% (ME Sun) resp. 85.4% (ME Pentium); only for the QSDPCM benchmark, data fetches increase up to 3.9% due to the insertion of spill code. D-cache misses drop by 2.9% (ME Sun) – 51.4% (CAVITY Sun). The very large register file of the Sun UltraSPARC III (160 integer registers) is the reason for the slight improvements of the L1 D-cache behavior for ME and QSDPCM. Since these benchmarks only use very few local variables, they can be stored entirely in registers even before loop nest splitting.

Furthermore, the columns L2 Fetch and L2 Miss show that the unified L2 caches also benefit significantly, since reductions of accesses (0.2% – 53.8%) and misses (1.1% – 86.9%) are reported in most cases.

4.2. Execution Times

All in all, the factors mentioned above lead to speed-ups between 17.5% (CAVITY Pentium) and 75.8% (ME Sun) for the processors considered in section 4.1 (see figure 5a). To demonstrate that these improvements not only occur on these CPUs, additional runtime measurements were performed for an HP-9000, PowerPC G3, DEC Alpha, TriMedia TM-1000, TI C6x and an ARM7TDMI, the latter both in 16-bit thumb- and 32-bit arm-mode.

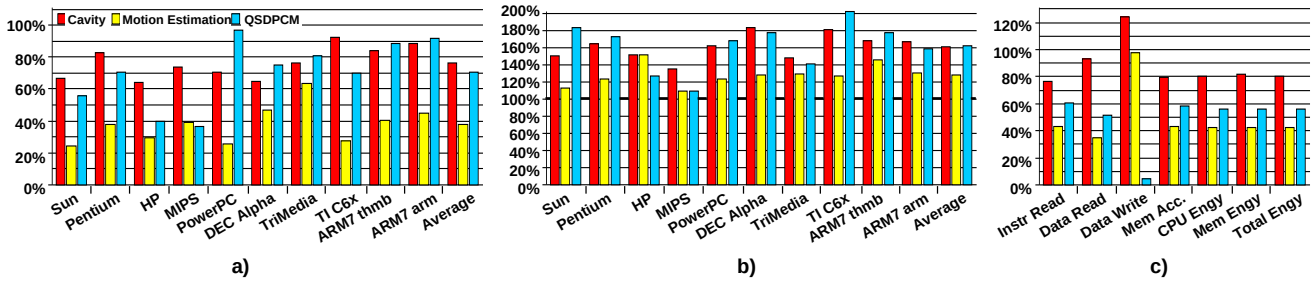
Figure 5a shows that all benchmarks benefit from loop nest splitting. The runtimes of CAVITY are improved between 7.7% (TI C6x) and 35.7% (HP). On the average over all processors, a speed-up of 23.6% was measured. The fact that loop nest splitting is able to generate a very regular control flow in the innermost loop of the ME benchmark leads to very high gains in this case. The benchmark is accelerated by 62.1% on average. The minimum speed-up amounts to 36.5% (TriMedia), whereas the Sun CPU honors the optimization with an acceleration of 75.8%. For QSDPCM, the improvements range from 3% (PowerPC) up to 63.4% (MIPS) leading to an average speed-up of 29.3%.

The variations among different CPUs depend on several factors. As already stated in section 4.1, the complexity of register files and pipelines are important parameters. Additionally, runtimes are influenced by different compiler optimizations and register allocation algorithms. Due to lack of space, a detailed study can not be given here.

4.3. Code Sizes and Energy Consumption

Since code is replicated, loop nest splitting entails an increase in code size (see figure 5b). On average, the CAVITY benchmark's code size increases by 60.9%, with minimum and maximum increases of 34.7% (MIPS) and 82.8% (DEC). Although the ME benchmark is accelerated most, its code enlarges least. Increases between 9.2% (MIPS) and 51.4% (HP) lead to an average growth of only 28%. Finally, the code of QSDPCM enlarges between 8.7% (MIPS) – 101.6% (C6x) leading to an average increase of 61.6%.

These increases by a few hundred instructions are not a serious drawback, since the added energy required for storing these instructions is compensated by the savings achieved by loop nest splitting. Figure 5c shows the effects of loop nest splitting on memory accesses and energy



a) b) c)

Figure 5. a) Execution Times b) Code Sizes c) Energy Consumption after Loop Nest Splitting

consumption using an instruction-level energy model [13] for the ARM7 core considering bit-toggles and offchip-memories and having an accuracy of 1.7%.

The column Instr Read shows that the number of instruction memory accesses is reduced by 23.5% (CAVITY) – 56.9% (ME). Furthermore, our control flow optimization also leads to a significant reduction of data memory accesses. Data reads are reduced up to 65.3% (ME). For QSDPCM, the removal of spill code reduces data writes by 95.4%. In contrast, the compiler inserts spill code for CAVITY so that an increase of 24.5% was observed. The total amount of all memory accesses (Mem Acc) is reduced by 20.8% (CAVITY) – 57.2% (ME).

Our optimization leads to large energy savings both of the CPU and its memory. The energy consumed by the ARM core is reduced by 18.4% (CAVITY) – 57.4% (ME), the memory consumes between 19.6% and 57.7% less energy. Total energy savings by 19.6% – 57.7% are measured.

Anyhow, if code size increases (up to a rough theoretical bound of 100%) are critical, it is easy to change our algorithms so that the splitting-if is not placed in the outermost possible loop. This way, code duplication is reduced at the expense of lower speed-ups, so that trade-offs between code sizes and savings in runtimes can be realized.

5. Conclusions

We present a novel source code optimization called loop nest splitting which removes redundancies in the control flow of embedded multimedia applications. Using polytope models, conditions having no effect on the control flow are removed. Genetic algorithms identify ranges of the iteration space where all *if*-statements are provably satisfied. The source code of an application is rewritten in such a way that the total number of executed *if*-statements is minimized.

A detailed study of 3 benchmarks shows that the branching and pipeline behavior is improved significantly. Furthermore, caches also benefit from our optimization since I- and D-cache misses are reduced heavily (up to 68.5%). Since accesses to instruction and data memory are reduced to a large extent, loop nest splitting consequently leads to large power savings (19.6% – 57.7%). An extended benchmarking using 10 different CPUs shows that we are able to speed-up the benchmarks by 23.6% – 62.1% on average.

The selection of the benchmarks used in this paper demonstrates that our optimization is a very general and powerful technique. It is not only able to improve the code of typical real-life applications, but in addition, it can be used to eliminate the negative effects of other source code transformation frameworks introducing a very large control flow overhead into an application. In the future, we will generalize our analytical models so that more classes of loop nests can be treated. In particular, extensions to loops not having constant bounds will be developed.

References

[1] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

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

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

[4] H. Falk. *Control Flow Optimization by Loop Nest Splitting at the Source Code Level*. Research Report 773, University of Dortmund, Germany, Oct. 2003.

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

[6] Y. H. Hu, editor. *Data transfer and storage (DTS) architecture issues and exploration in multimedia processors*, volume Programmable Digital Signal Processors – Architecture, Programming and Applications. Marcel Dekker Inc., New York, 2001.

[7] M. Kandemir, N. Vijaykrishnan et al. Influence of compiler optimizations on system power. In *Proc. of DAC*, Los Angeles, 2000.

[8] D. Levine. *Users Guide to the PGAPack Parallel Genetic Algorithm Library*. Tech. Rep. ANL-95/18, Argonne National Lab., 1996.

[9] N. Liveris, N. D. Zervas et al. A Code Transformation-Based Methodology for Improving I-Cache Performance of DSP Applications. In *Proc. of DATE*, Paris, 2002.

[10] T. S. Motzkin, H. Raiffa et al. The double description method. *Theodore S. Motzkin: Selected Papers*, 1953.

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

[12] M. R. Stan and W. P. Burleson. Bus-Invert Coding for Low-Power I/O. *IEEE Transactions on VLSI Systems*, 3(1), 1995.

[13] S. Steinke, M. Knauer, L. Wehmeyer and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc. of PATMOS*, Yverdon-Les-Bains, 2001.

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

[15] D. K. Wilde. *A Library for doing polyhedral Operations*. Tech. Rep. 785, IRISA Rennes, France, 1993.

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

[17] S. Wuytack, F. Catthoor et al. Power Exploration for Data Dominated Video Applications. In *Proc. of ISLPED*, Monterey, 1996.