Control Flow Optimization by Loop Nest Splitting at the Source Code Level

Heiko Falk

Research Report N^{o} 773

University of Dortmund
Dept. of Computer Science 12
D - 44221 Dortmund
Germany

Research Report N^{o} 773

October 18, 2002

Control Flow Optimization by Loop Nest Splitting at the Source Code Level

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

Abstract

In recent years, the application of optimization techniques at the level of program source codes has increasingly attracted interest due to the high effectiveness and the inherent retargetability of such approaches. In this report, a novel source code transformation technique for control flow optimization called loop nest splitting is presented. The goal of this optimization is to reduce runtimes and energy consumption by minimizing the number of if-statements executed in loop nests of typical embedded multimedia applications. Complementary to already known optimizations in this area, we explicitly focus on the optimization of loop-variant if-statements.
The analysis techniques required for performing loop nest splitting are illustrated in detail. They base on precise mathematic models combined with genetic algorithms. The analysis is done statically at compile time and does not rely on profiling.
For a detailed evaluation of the benefits of loop nest splitting, the effects of our optimization with respect to instruction pipeline and cache behavior, runtimes, energy consumption and code sizes are shown. The application of our implemented tools for loop nest splitting to three real-life multimedia benchmarks leads to average reductions of pipeline stalls between 19.7% and 64.8% and an average decrease of instruction cache misses between 8.9% and 45.3%. Measurements on a variety of different programmable processors show average speed-ups between 23.6% and 62.1% of the benchmarks, whereas reductions of energy dissipation between 19.2% and 57.6% are observed.

Contents

List of Figures

List of Tables

1. Introduction

Embedded systems have always been very cost-sensitive. During the earlier periods of embedded system design, the main focus has been on area-efficient designs meeting constraints due to performance and design time. During the last couple of years, power dissipation has become an additional major design measure for many systems next to these traditional measures. More and more applications become portable because this is felt as an added value for the product (e. g. wireless phones, multimedia terminals, ...). The less power they consume, the longer their batteries last and the lighter their batteries can be made. As a consequence, power efficient design has become a crucial issue for a broad class of applications.

Many of these embedded applications turn out to be data-dominated, especially in the multimedia and network protocol domains. Experiments have shown that for these applications, a very large part of the power consumption is due to data transfers from and to memories.

1.1. Control Flow Overhead in Data-dominated Applications

Data-dominated applications such as medical image processing and video compression algorithms typically use a very large amount of data memory. For being able to manipulate its large quantities of data in an effective way, the program code of a data-dominated application often has a certain structure where particular constructs of the programming language are used. Typically, such applications consist of deeply nested *for*-loops. With the help of the index variables of the loops, memory access pointers are calculated that are used for data manipulation. The main algorithmic part is usually located in the innermost loop. Very often, a multimedia algorithm has to treat particular parts of its data in a specialized way, e. g. pixels at the border of an image require a slightly modified algorithm than pixels situated in the center of an image. This boundary checking is implemented with the help of *if*-statements in the innermost loop that check certain values of the index variables. A typical code fragment taken from an MPEG 4 full search motion estimation kernel [5] written in ANSI-C is depicted in figure 1.

Although already written in an optimized manner (common subexpressions are eliminated and loop-invariant code is moved out of loops [23]), this code in its original structure has several properties that make it sub-optimal with respect to runtime and energy consumption. First, the *if*-statements in the innermost loop lead to a very irregular control flow. Any change in the linear control flow of a machine program, i. e. a jump instruction, causes a control hazard for pipelined processor architectures. This means that the pipeline needs to be stalled for a certain number of instruction cycles, so as to prevent execution of incorrectly prefetched instructions. Delay slots of a pipeline can sometimes be filled with useful instructions, but

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {
    x1=4*x;
    for (y=0; y<49; y++) {
      y1=4*y;
      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

for deeply pipelined processors, this is not always the case leading to high jump penalties (Philips TriMedia TM1000: 3 instruction cycles / jump [25]; TI C6201: 5 cycles / jump [30]).

Second, the pipeline performance of a processor is also influenced by data accesses, since pipelines may have to be stalled during the execution of memory transfers. To realize the boundary checking mentioned above, the induction variables are accessed very frequently resulting in pipeline stalls, if these variables can not be kept in processor registers. Degraded pipeline performance not only leads to increased execution times but also to a higher energy consumption due to the forced inactivity of the processor. Additionally, since it has been shown that 50% – 75% of the power consumption in embedded multimedia systems is caused by memory accesses [26, 31, 35], frequently repeated transfers of induction variables across memory hierarchies via system buses also contribute negatively to the total energy balance.

Finally, many instructions are required to evaluate the conditions of the *if*-statements. In the case of the motion estimation kernel shown above, these arithmetic and logical operations are in total as complex as the computations performed in the *then*- and *else*-blocks of the *if*-statements. This also demonstrates that execution time and energy consumption of such multimedia applications are affected

adversely by *if*-statements in loop nests.

1.2. Splitting of Loop Nests for Control Flow Optimization

In this report, a new formalized method for the analysis and transformation of *if*-statements occuring in arbitrarily nested loops 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` of the first *if*-statement are never true,
- both *if*-statements are true for `x` ≥ 10 or `y` ≥ 14.

Information of the first type can be used to detect conditions not having any influence on the control flow of an application. This is a difference compared to conventional dead code, because dead code is defined as those instructions in a program that compute only values that are not used on any executable path leading from the instruction [23]. Since the boolean results computed by the conditions mentioned above are used within an *if*-statement, they will not be removed by a classical dead code elimination. Furthermore, a study of the assembly codes emitted by a large variety of state-of-the-art optimizing compilers (see table 2 on page 28) shows that the compilers do not perform an induction variable analysis to determine that these conditions are unnecessary. Therefore, the removal of this kind of useless code with the help of our techniques leads to reductions of code sizes and computational complexity of a program and thus improves runtimes and energy consumption.

With the help of information of the second kind, the entire loop nest can be rewritten in such a way that the total number of executed *if*-statements is minimized. Therefore, a new *if*-statement (called the *splitting-if*) is inserted in the `y` loop testing the condition `x >= 10 || y >= 14`. The *else*-part of this new *if*-statement is an exact copy of the body of the original `y` loop. Since it is known that 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 their associated *else*-blocks.

To ensure that the splitting-if will not be evaluated repeatedly without need for values of `y` ≥ 14, a second `y` loop is inserted in the *then*-part of the splitting-if counting from the current value of `y` to the upper bound 48. After the execution of this second `y` loop, the index variable `y` has to be decremented by 1 so that `y` contains the correct value of 49 after the execution of the entire loop nest. It is important to see that this second `y` loop modifies the index variable `y` which is also accessed by the original outer `y` loop. Because of this fact, it is assured that the execution of both the new and the original `y` loop leads to the assignment of exactly the same sequence of values to `y` as before the optimization so that the semantical correctness of loop nest splitting is given. At the same time, these twofold `y` loops

```
for (z=0; z<20; z++)
  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++)
            for (l=0; l<9; l++)
              for (i=0; i<4; i++)
                for (j=0; j<4; j++) {
                  then_block_1;
                  then_block_2; }
        y--; }
      else {
        y1=4*y;
        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 (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

lead to the minimization of executions of the splitting-if. The transformed code is illustrated in figure 2.

As can be seen from this example, our technique is able to generate linear control flow in the hot-spots of an application. Furthermore, references to memory are reduced significantly, because a large amount of branching, arithmetic and logical instructions and induction variable accesses are removed from the code. Using these factors, our loop nest splitting technique is able to achieve considerable speed-ups combined with reductions of energy consumption at the expense of slightly increased code size.

The techniques presented in this report are fully implemented and integrated into a tool for the automated splitting of loop nests. Since the analysis of loop characteristics is crucial for the application of our techniques, a high-level intermediate representation preserving loop information is required. The fact that many state-of-the art optimizing compilers use intermediate representations where control flow is modeled only in the form of simple conditional and unconditional branches (e. g. Sun-IR [22], LANCE [15]) leads to the consequence that our optimizations are not integrated into specific compiler frameworks. In contrast, we are focusing on the area of source code transformations where an optimized ANSI-C program is the result of our loop nest splitting tool. Since ANSI-C is standardized [12], this approach has the additional benefit that it is highly retargetable, because the generated C source codes are accepted and compiled by any ANSI compatible C compiler regardless of the actual processor for which a compilation is initiated. This processor independence allows us to perform a very detailed benchmarking on a variety of ten different programmable processors as shown in section 5.

The remainder of this report is organized as follows: A survey of work related to loop optimizations and source code transformations is provided in section 2. Because the techniques for loop nest splitting described here are based on genetic algorithms, section 3 is dedicated to an overview of the main principles of genetic optimization. Section 4 presents the analytical models and algorithms for loop nest splitting. Section 5 contains a description of the applications serving as benchmarks and of the impacts of our optimization on pipelines and caches, runtimes, energy dissipation and code sizes. Section 6 summarizes and concludes this report. Appendix A contains a detailed list of the data gathered during benchmarking.

2. Related Work

Loop restructuring transformations have been described in literature on compiler design for many years (see e. g. [2, 23]) and are often integrated into state-of-the-art optimizing compilers. Conventional *loop splitting* (also called *loop distribution* or *loop fission*) takes a loop that contains multiple statements and splits it into two loops with the same iteration-space traversal, such that the first loop contains some of the statements inside the original loop and the second one contains the others as shown in figure 3.

```
for (i=1; i<n; i++) {          for (i=1; i<n; i++)
  a[i]=a[i]+c;                   a[i]=a[i]+c;
  x[i+1]=x[i]*7+x[i+1]    ⟶   for (i=1; i<n; i++)
              +a[i]; }           x[i+1]=x[i]*7+x[i+1]
                                           +a[i];
```

Figure 3. Conventional Loop Splitting

The main goal of this optimization is to create subloops with fewer data dependencies allowing parts of an original loop to be executed in parallel [2]. Moreover, instruction cache performance may be increased, because locality is improved due to the smaller loop bodies. Additionally, the smaller loop bodies can have the effect that code size constraints of a processor architecture with respect to hardware *do*-loops are now met. As a consequence, control flow modifications after each loop iteration can be performed by the processor hardware making explicit branch instructions in the code unnecessary. If hardware *do*-loops can not be applied to a loop, the control flow becomes more irregular after the application of loop splitting, because a single loop is broken into many loops. Kandemir et al. [11] have shown that loop splitting leads to increased energy consumption of the processor core as well as the memory system. Also, computational complexity during loop execution is not reduced by this optimization, so that this technique is not suitable for solving the problems discussed in section 1.1.

```
for (i=1; i<n; i++) {          if (x<7)
  a[i]=a[i]+c;                   for (i=1; i<n; i++) {
  if (x<7)                         a[i]=a[i]+c;
    b[i]=a[i]*c[i];         ⟶     b[i]=a[i]*c[i]; }
  else                           else
    b[i]=a[i-1]*b[i-1]; }          for (i=1; i<n; i++) {
                                     a[i]=a[i]+c;
                                     b[i]=a[i-1]*b[i-1]; }
```

Figure 4. Conventional Loop Unswitching

Classical *loop unswitching* as depicted in figure 4 is applied when a loop contains an *if*-statement with a loop-invariant test condition [23]. The loop is then replicated

inside each branch of the conditional check, saving the overhead of conditional branching inside the loop, reducing the code size of the loop bodies, and possibly enabling the parallelization of one branch of the *if*-statement [2]. The goals of loop unswitching and the way how the optimization is performed on a loop are equivalent to the topics of section 1. The main disadvantage of loop unswitching making it completely unsuitable for applying it to typical multimedia programs is the fact that the *if*-statements must not depend on index variables. It is the contribution of the transformation technique presented in this report that we explicitly focus on loop-variant conditions for loop splitting. Since our analysis techniques go far beyond those required for conventional 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*.

Kim et al. [13] have evaluated the effect of several high-level compiler optimizations (loop unrolling, loop interchange, loop fusion and loop tiling) on memory system energy considering both instruction and data accesses. The authors have observed that these techniques are effective in minimizing the energy consumed due to data accesses, but that the energy consumed due to instruction accesses are increased significantly. They draw the conclusion that techniques are required for the simultaneous optimization of the locality of data and instruction accesses. The results given in this report show that loop nest splitting is able to achieve these aims.

Liveris et al. [18] have applied conventional loop splitting in conjunction with function call insertion as source code transformation for improving the instruction cache performance. After the application of loop splitting, the authors report a large reduction of instruction cache misses for one benchmark. All other parameters (instruction and data memory accesses, data cache misses) are worse after the transformation. Unfortunately, the results are generated with cache simulation software which is known to be unprecise. The effects of the optimizations on the runtime of the benchmark are not reported.

Source code transformation frameworks for data locality and cache performance improvement have been studied recently in literature. In [9], a series of array and loop transformations is illustrated by means of a medical image processing algorithm [3]. The transformation methodology proposed by the authors focuses on the minimization of data transfers between background memories and processing units. In particular, transformations for data reuse, in-place mapping and loop folding and merging are presented. In the code examples, the authors only concentrate on the illustration of the optimized data flow thereby neglecting that the control flow becomes very irregular due to the insertion of a variety of additional *if*-statements by the loop transformations. Furthermore, complex arithmetic for address calculation is introduced into the application. This increased complexity with respect to the control flow and arithmetic expressions leads to the fact that the runtimes of the benchmark increase by a factor of 5 after the application of the

transformation methodology. Additional transformations have been proposed [4] for a reduction of the arithmetical complexity, but an optimization of the control flow has not yet been targeted. As we will demonstrate in section 5, our loop nest splitting is capable of removing the control flow overhead introduced by the transformations proposed in [9].

3. Overview of Genetic Algorithms

Genetic algorithms (*GA*) have proven to solve complex optimization problems by imitating the natural optimization process (see e. g. [1, 8] for an overview). A population of a GA consists of several individuals, each of them representing a potential solution for the optimization problem. The representation of an individual is given by a *chromosome* which is subdivided into *genes*. The genes are used to encode the variables of the optimization problem. This means that finding a suitable combination of *alleles* (concrete values) for the genes is the same as finding good solutions for the optimization problem. By applying genetic operators like *selection*, *mutation* and *crossover* to the members of the population, the fitness of the individuals will increase in the course of the generations. An overview of the main steps of the optimization process of genetic algorithms is given in figure 5.

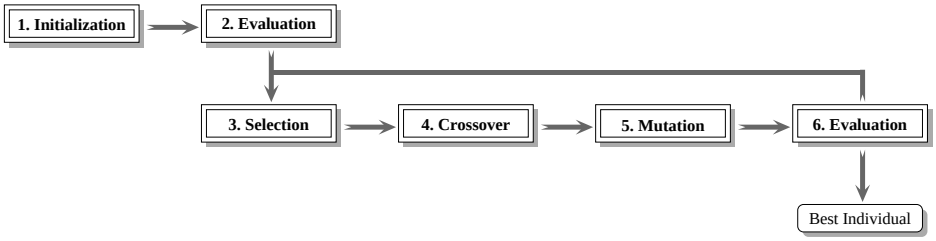

Figure 5. Optimization Steps of a Genetic Algorithm

First, all individuals of the population are initialized (1) and evaluated (2). For the concrete problem of loop nest splitting, a population size of 100 individuals[1] is used. Individuals worth inheriting their genes to the next generation are selected probabilistically in the following step (3). The crossover operator (4) performs a recombination of the genetic information by choosing two individuals and swapping genes between these individuals. Afterwards, mutation (5) creates new gene material by changing alleles. The resulting individuals are evaluated again (6).

The optimization process is iterated until a termination condition is met. Here, the algorithm stops either if the best solution does not change within 100 generations, or if the maximum number of 1,000 generations[1] has been processed. It is an important characteristic of genetic algorithms that suitable gene material is passed to the subsequent generations. The selection ensures that only the 50 fittest individuals are kept and taken into the next iteration. This permits one to revise unfavorable decisions made in a previous optimization phase.

For this reason, genetic algorithms are adequate for solving non-linear optimization problems and have been successfully utilized in the domains of code generation for irregular architectures [16, 19] or hardware synthesis and codesign [14, 24].

[1] This value equals to a default value of the genetic library [17] used for the implementation of loop nest splitting and has proven to be effective w. r. t. optimization quality and runtime.

4. Analysis and Optimization Algorithm

In this section, the analysis and optimization techniques required for loop nest splitting are presented.

The core analysis algorithm (excluding the steps of parsing the input and generation of the output C source code) consists of four sequentially executed tasks (see also figure 6). In the beginning, all conditions in a loop nest are analyzed separatly without considering any interdependencies between them. First, it is detected if conditions ever evaluate to true or not (see section 4.2). Second, all satisfiable conditions are analyzed and an optimized search space for each condition is constructed (4.3). In a third step, all local search spaces are combined to a global search space (4.4) which has to be explored leading to the optimized result for loop nest splitting (4.5).

Before describing the optimization techniques in detail, some general definitions and preconditions are defined in section 4.1.

4.1. Preliminaries

A loop nest is characterized by the following definition.

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 $lb_l \leq i_l \leq ub_l$ ($lb_l, i_l, ub_l \in \mathbb{Z}$).

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

Definition 2

1. For a loop $L_l \in \Lambda$, the values lb'_l and ub'_l define a range of values for the index variable i_l. Therefore, lb'_l and ub'_l must be within the loop bounds of L_l: $lb'_l \geq lb_l$ and $ub'_l \leq ub_l$.
2. All loop-variant *if*-statements in Λ are satisfied for all loops $L_l \in \Lambda$ and all values of their index variables i_l within these ranges: $lb'_l \leq i_l \leq ub'_l$.
3. Loop nest splitting by all values lb'_l and ub'_l leads to the minimization of *if*-statement execution.

All these values lb'_l and ub'_l define maximum ranges of values for all index variables where all *if*-statements are true. These values are used for the construction of

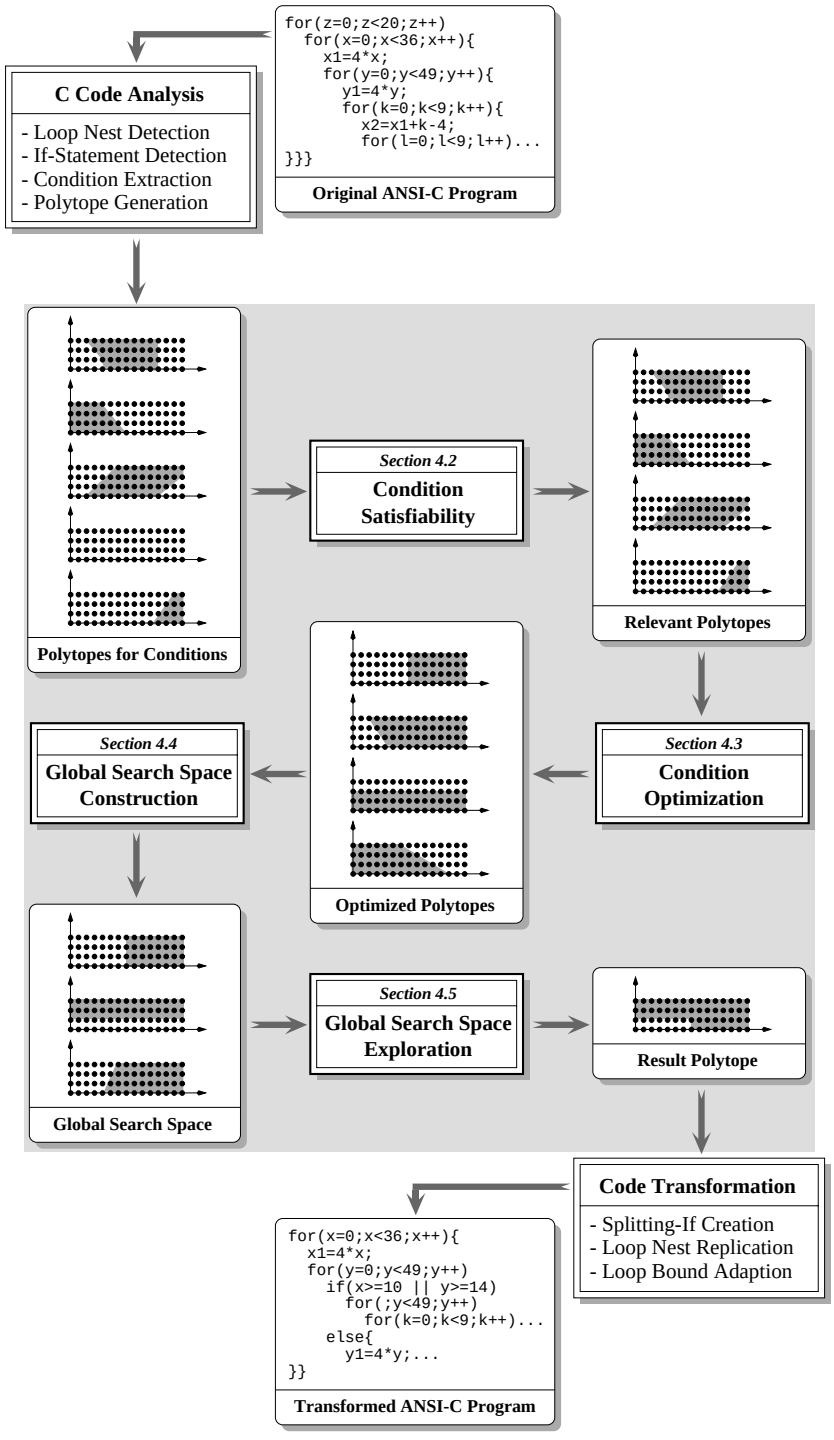

Figure 6. Designflow of Loop Nest Splitting

the condition of the splitting *if*-statement. The techniques described in the following require that some preconditions are met.

Preconditions:

1. All loop bounds lb_l and ub_l are constants.
2. All *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 $\oplus \in \{\texttt{\&\&}, \texttt{||}\}$.
3. Loop-variant conditions C_x are affine expressions of i_l and can have the format $C_x \simeq \sum_{l=1}^{N}(c'_l * i_l) + c' \otimes \sum_{l=1}^{N}(c''_l * i_l) + c''$ for constant values c'_l, c''_l, c', $c'' \in \mathbb{Z}$ and comparators $\otimes \in \{<, \leq, >, \geq\}$. For the sake of simplicity, we only consider the following normalized equivalent format of conditions in the following, where $C_x = \sum_{l=1}^{N}(c_l * i_l) + c \geq 0$ for constants $c_l, c \in \mathbb{Z}$.

Precondition 2 is only due to the current state of implementation of our tools and thus does not impose a general restriction on our optimization approach. By application of *de Morgan's* rule on expressions like `!(`$C_1 \oplus C_2$`)` 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 does not constrain the way how conditions can be combined.

A condition like `(a == b)` can be rewritten as `(a` $\geq$ `b) && (a` $\leq$ `b)` without loss of generality (`(a != b)` analogous), so that the set $\otimes$ of supported comparators defined in precondition 3 is not a restriction either.

By application of *forward substitution* [23] to induction variables not being an index variable i_l of the loop nest Λ (e. g. `x1`, ..., `x4`, `y1`, ..., `y4` in figure 1), it is assured that C_x only depends on the index variables i_l as stated in precondition 3.

Example 1

After having applied the forward substitution to the code shown in figure 1, the if*-statements have the following form:*

```
if (4*x + i < 0 || 35 < 4*x + i ||
    4*y + j < 0 || 48 < 4*y + j)
if (4*x + k + i - 4 < 0 || 35 < 4*x + k + i - 4 ||
    4*y + l + j - 4 < 0 || 48 < 4*y + l + j - 4)
```

It is obvious that these if*-statements meet precondition 2. For obtaining the normalized format mentioned in precondition 3, the conditions of the* if*-statements are translated to the following appearance which will serve as example throughout the remainder of this report:*

```
if (-4*x - i - 1 >= 0 || 4*x + i - 36 >= 0 ||
    -4*y - j - 1 >= 0 || 4*y + j - 49 >= 0)
```

respectively

```
if (-4*x - k - i + 3 >= 0 || 4*x + k + i - 40 >= 0 ||
    -4*y - l - j + 3 >= 0 || 4*y + l + j - 53 >= 0)
```

4.2. Condition Satisfiability

In the first phases of the optimization algorithm, all affine conditions C_x in a loop nest are analyzed separately. Every single condition defines a particular subset of the total iteration space of a loop nest Λ. This total iteration space forms an N-dimensional geometric space limited by all loop bounds lb_l and ub_l. An affine condition C_x can thus be modeled by a polytope according to the following definition 3.

Definition 3

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

Example 2

The condition `4*x + 3*i - 36 >= 0`*, where the index variables* `x` *and* `i` *iterate through the intervals* $[0, 35]$ *and* $[0, 3]$ *resp., has the following polytope representation:*

$$P = \left\{ x \in \mathbb{Z}^2 \;\middle|\; \begin{pmatrix} 4 & 3 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} x \geq \begin{pmatrix} 36 \\ 0 \\ -35 \\ 0 \\ -3 \end{pmatrix} \right\}$$

The first constraint of P represents the condition itself, whereas the remaining four inequations constrain the polytope to the intervals of `x` *and* `i` *mentioned above. Transformed into a graphical illustration, the condition above defines the shape shown in figure 7.*

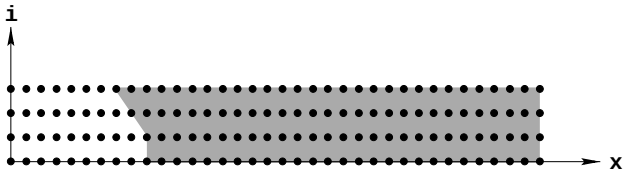

Figure 7. Conditions as Polytopes

Every affine condition C_x in a loop nest is transformed into its polytope representation P_x. For this purpose, an improved variant of the Motzkin algorithm [21] is used in combination with some simplifications in order to remove redundant constraints [33].

Hereafter, we can determine in constant time if the number of equalities $Ax = a$ of P_x (see definition 3.1) is equal to the dimension of P_x plus 1. If this is the case, P_x is overconstrained and thus defines the empty set as proven by Wilde [33]. As a consequence, C_x is shown to be unsatisfiable. If instead, P_x only contains the

constraints modeling the loop bounds lb_l and ub_l, the condition C_x is shown to be satisfied for all values of the index variables i_l.

Such conditions that are always satisfied or unsatisfied are replaced by their according truth value in the *if*-statements and are no longer considered during further analysis.

4.3. Condition Optimization

Satisfiable conditions $C = \sum_{l=1}^{N}(c_l * i_l) + c \geq 0$ are optimized in the next step. C is analyzed in isolation without considering any other conditions in the loop nest Λ. During this stage, it is assumed that Λ contains only one *if*-statement which contains exactly one condition, namely C. The result of the optimization is a polytope P_C which is generated 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 determined in such a way that a loop nest splitting according to ${lb'}_{C,l}$ and ${ub'}_{C,l}$ would minimize the number of executions of C.

Since affine expressions as defined in precondition 3 are linear monotone functions, it is not necessary to deal with pairs ${lb'}_{C,l}$ and ${ub'}_{C,l}$ of values. If C is true for a certain value $v \in [{lb'}_{C,l}, {ub'}_{C,l}]$ and $c_l > 0$, then C must be true also for $v+1$, $v+2$, ... It is impossible that C is true for v and false for $v+1$. The same holds for $c_l < 0$; in this case, C must be true for $v, v-1, v-2, \ldots$ This implies that either ${lb'}_{C,l} = lb_l$ or ${ub'}_{C,l} = ub_l$. For these reasons, the optimization algorithm only calculates values ${v'}_{C,l}$ for a condition C and all loops $L_l \in \Lambda$ with ${v'}_{C,l} \in [lb_l, ub_l]$. This value designates one bound of the former interval $[{lb'}_{C,l}, {ub'}_{C,l}]$, the other bound is set to the the appropriate upper or lower loop bound.

For the optimization of an affine condition C, the chromosome length of our first genetic algorithm (GA) is equal to the number of index variables i_l that C depends on. For every such index variable, a gene on the chromosome is allocated representing ${v'}_{C,l}$. Thus, the chromosome length is equal to $|\{c_l \mid c_l \neq 0\}|$. Using the ${v'}_{C,l}$ values of the fittest individual of the final GA population, the optimized polytope P_C is generated as the result of this phase:

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

Example 3

When considering the condition $C =$ `4*x + k + i - 40 >= 0` *taken from example 1, the genetic algorithm presented in this section can generate the individual* $I = (10, 0, 0)$. *Since* C *only depends on the index variables* `x`, `k` *and* `i`, I *has a chromosome length of 3. The numbers encoded in* I *denote the values* ${v'}_{C,\mathtt{x}}$, ${v'}_{C,\mathtt{k}}$ *and* ${v'}_{C,\mathtt{i}}$*:*

$$v'_{C,\mathtt{x}} = 10 \qquad v'_{C,\mathtt{k}} = 0 \qquad v'_{C,\mathtt{i}} = 0$$

Due to the fact that the index variables x, k *and* i *are multiplied by positive constants in C (i. e. 4, 1 and 1), C is satisfied for all values of these index variables ranging from $v'_{C,l}$ up to the corresponding upper loop bound. For this reason, the intervals $[lb'_{C,l}, ub'_{C,l}]$ for the entire loop nest and condition C are defined as follows:*

$$\mathtt{z} \in [0,19] \;\; \mathtt{x} \in [10,35] \;\; \mathtt{y} \in [0,48] \;\; \mathtt{k} \in [0,8] \;\; \mathtt{l} \in [0,8] \;\; \mathtt{i} \in [0,3] \;\; \mathtt{j} \in [0,3]$$

Since condition C is not influenced by the index variables z, y, l *and* j*, the intervals $[lb'_{C,}, ub'_{C,l}]$ for these four index variables are not computed explicitly by the genetic algorithm, but they are set to the lower and upper loop bounds.*

Out of these particular intervals, the following polytope P_C can be generated:

$$P_C = \left\{ (\mathtt{z},\mathtt{x},\mathtt{y},\mathtt{k},\mathtt{l},\mathtt{i},\mathtt{j}) \in \mathbb{Z}^7 \;\middle|\; \begin{array}{lll} 0 \le \mathtt{z} \le 19, & 0 \le \mathtt{y} \le 48, & \\ 0 \le \mathtt{l} \le 8, & 0 \le \mathtt{j} \le 3, & \\ 10 \le \mathtt{x} \le 35, & 0 \le \mathtt{k} \le 8, & 0 \le \mathtt{i} \le 3 \end{array} \right\}$$

For a given individual I of the genetic algorithm, the corresponding fitness value is the higher, the fewer *if*-statements would be executed when splitting Λ according to the values $v'_{C,l}$ encoded in I. Since the GA only selects those individuals in a population with the highest fitness values, the genetic optimization leads to the minimization of the number of executed *if*-statements. Consequently, an invalid individual whose encoded values imply that C is not satisfied, has a very low fitness value.

Given a condition $C = \sum_{l=1}^{N} (c_l * i_l) + c \ge 0$ and a set of values $v'_{C,l}$ generated by the GA, it is necessary to compute the number of executed *if*-statements IF_{Tot} for the evaluation of the fitness of an individual I. For these computations, the following values are required.

Definition 4

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 intervals represented by $v'_{C,l}$ and represents the size of the polytope P_C:
   $$CS = \prod_{l=1}^{N} r_l \text{ and } r_l = \begin{cases} ub_l - lb_l + 1 & \text{if } c_l = 0, \\ ub_l - v'_{C,l} + 1 & \text{if } c_l > 0, \\ v'_{C,l} - lb_l + 1 & \text{else} \end{cases}$$
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 total number of *if*-statements executed after loop nest splitting is the sum of the executions of the original *if*-statements in the *else*-part of the splitting *if*-statement – if the splitting *if*'s condition is not met – and the executions of the splitting-if itself: $IF_{Tot} = IF_{Orig} + IF_{Split}$ (see e. g. figure 2). The number of loop nest iterations in which an original *if*-statement is executed is equal to the total iteration space without the constrained iteration space: $IF_{Orig} = TS - CS$. The splitting *if*-statement is executed as often as its *then*- and *else*-parts are executed: $IF_{Split} = TP_{Split} + EP_{Split}$.

Since *CS* counts the total number of iterations in which the splitting-if is true, TP_{Split} mainly depends on *CS*. But TP_{Split} is much smaller than *CS*, because *CS* includes all the iterations of the loop nest located inside the *then*-block. In contrast, TP_{Split} counts how often the entire loop nest located inside the *then*-block is executed. By dividing *CS* by all loop nest iterations inside the *then*-block, the correct value for TP_{Split} is obtained: $TP_{Split} = CS \,/ \prod_{l=\lambda+1}^{N} (ub_l - lb_l + 1) * r_\lambda$. With the same kind of argumentation, EP_{Split} can be determined: $EP_{Split} = IF_{Orig} \,/ \prod_{l=\lambda+1}^{N} (ub_l - lb_l + 1)$.

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 genetic algorithm.

Example 4

When considering the intervals $[lb'_{C,l}, ub'_{C,l}]$
$z \in [0,19]$ $x \in [10,35]$ $y \in [0,48]$ $k \in [0,8]$ $l \in [0,8]$ $i \in [0,3]$ $j \in [0,3]$
already used in example 3, the following peace of C code would be generated as a result of loop nest splitting:

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++)
    if (x >= 10) {
      for (; x<36; x++)
        ...
          for (j=0; j<4; j++) { ... }
      x--; }
    else
      for (k=0; k<9; k++)
        ...
          for (j=0; j<4; j++) {
            if (...)  ...  else ...; }
```

The total and constrained iteration spaces of this loop nest are computed as follows:

$$TS = 20 * 36 * 49 * 9 * 9 * 4 * 4 = 45,722,880$$
$$CS = 20 * 26 * 49 * 9 * 9 * 4 * 4 = 33,022,080$$

Since only the iteration space of the index variable x *is constrained by the above*

intervals, the innermost loop λ refers to the x *loop.*

The formulas given in this section indicate that the two original if*-statements which can be found in the* else*-part of the splitting-if are executed*

$$IF_{Orig} = 45,722,880 - 33,022,080 = 12,700,800$$

times. The calculation of executions of then- *and* else*-parts of the splitting-if are performed as follows*

$$TP_{Split} = 33,022,080 \;/\; (49 * 9 * 9 * 4 * 4 * 26) = 20$$
$$EP_{Split} = 12,700,800 \;/\; (49 * 9 * 9 * 4 * 4) = 200$$

leading to the following values for the number of executions of the splitting-if and of all if*-statements in the code fragment shown above:*

$$IF_{Split} = 20 + 200 = 220$$
$$IF_{Tot} = 12,700,800 + 220 = 12,701,020$$

4.4. Global Search Space Construction

After the step described in section 4.3, a set of *if*-statements $IF_i = (C_{i,1} \oplus C_{i,2} \oplus \ldots \oplus C_{i,n})$ consisting of affine conditions $C_{i,j}$ together with their associated optimized polytopes $P_{i,j}$ is given. In order to determine index variable values for which all *if*-statements in a program are satisfied, a polytope G modeling the global search space has to be constructed out of $P_{i,j}$.

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 ||. For initialization, P_i is set to the optimized polytope of the first condition: $P_i = P_{i,\pi(1)}$. 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. If two conditions are connected with the logical *AND* in an *if*-statement, the intersection of polytopes is used. In the case of the logical *OR*, unions of polytopes are built: $\forall j \in \{2,\ldots,n\} : P_i = P_i \uplus P_{i,\pi(j)}$ with $\uplus = \begin{cases} \cap & \text{if } C_{i,\pi(j-1)} \texttt{ \&\& } C_{i,\pi(j)} \\ \cup & \text{if } C_{i,\pi(j-1)} \texttt{ || } C_{i,\pi(j)} \end{cases}$

The polytopes P_i now model the ranges of values of the index variables for which a single *if*-statement i is satisfied. Since for loop nest splitting, all *if*-statements need to be satisfied, the global search space is constructed by intersecting all P_i: $G = \bigcap P_i$.

Example 5

In this example, we consider the two if*-statements already given in example 1:*

```
IF1 = -4*x - i - 1 >= 0 || 4*x + i - 36 >= 0 ||
      -4*y - j - 1 >= 0 || 4*y + j - 49 >= 0
IF2 = -4*x - k - i + 3 >= 0 || 4*x + k + i - 40 >= 0 ||
      -4*y - l - j + 3 >= 0 || 4*y + l + j - 53 >= 0
```

After the first two steps of our optimization algorithm, the following polytopes are associated with each condition:

$$P_{1,1} = \emptyset,\ P_{1,2} = \{\mathsf{x} \geq 9\},\ P_{1,3} = \emptyset,\ P_{1,4} = \{\mathsf{y} \geq 13\}$$
$$P_{2,1} = \{\mathsf{x} = 0 \ \wedge\ \mathsf{k} = 0\},\ P_{2,2} = \{\mathsf{x} \geq 10\},$$
$$P_{2,3} = \{\mathsf{y} = 0 \ \wedge\ \mathsf{l} = 0\},\ P_{2,4} = \{\mathsf{y} \geq 14\}$$

Since polyhedra are not closed under the union operator, the geometrical objects P_i defined above are – strictly spoken – no polytopes. Instead, an implementation for finite unions of polyhedra is used for which all operations on sets, especially the union, are closed [33]. For these reasons, P_1 and P_2 for the example are shown explicitly as a union of polytopes:

$$P_1 \ = \ \{\mathsf{x} \geq 9\} \cup \{\mathsf{y} \geq 13\}$$
$$P_2 \ = \ \{\mathsf{x} = 0 \wedge \mathsf{k} = 0\} \cup \{\mathsf{x} \geq 10\} \cup \{\mathsf{y} = 0 \wedge \mathsf{l} = 0\} \cup \{\mathsf{y} \geq 14\}$$

After logical simplification, the intersection of P_1 and P_2 modeling the global search space looks as follows:

$$G \ = \ \{\mathsf{x} = 0 \wedge \mathsf{k} = 0 \wedge \mathsf{y} \geq 13\} \cup \{\mathsf{x} \geq 10\} \cup \{\mathsf{y} = 0 \wedge \mathsf{l} = 0 \wedge \mathsf{x} \geq 9\} \cup \{\mathsf{y} \geq 14\}$$

4.5. Global Search Space Exploration

After the construction of G as described in the previous section, the global search space G has the following structure:

Definition 5

1. G generally has the structure of a finite union of M polytopes R_r with $1 \leq r \leq M$: $G = R_1 \cup R_2 \cup \ldots \cup R_M$.
2. Each polytope R_r of G defines a region of the total iteration space where all *if*-statements in a loop nest Λ are satisfied.

The final stage of the loop nest split optimization (compare figure 6 on page 11) consists of the selection of appropriate regions R_r of G such that once again the total number of executed *if*-statements is minimized after loop nest splitting.

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

In analogy to section 4.3, the index λ of the innermost loop where the loop nest would be split has to be defined for each individual I of the genetic algorithm. In

$IF_I = 0;$
$\forall i_1 \in [lb_1, ub_1]$
$\quad \ldots$
$\quad\quad \forall i_\lambda \in [lb_\lambda, ub_\lambda]$
$\quad\quad\quad IF_I = IF_I + 1;$
$\quad\quad\quad$ if $(G_I =$ true for $(i_1, \ldots, i_\lambda))$
$\quad\quad\quad\quad i_\lambda = ub_\lambda;$
$\quad\quad\quad$ else
$\quad\quad\quad\quad IF_I = IF_I + IF_{\lambda+1};$

Figure 8. Global If-Statement Counter

order to be able to count how many *if*-statements are executed when splitting a loop nest using the regions selected by I, some more definitions are necessary:

Definition 6

1. For an individual I, G_I is the global search space G reduced to only those regions selected by I:
   $G_I = \bigcup R_r$ with $I_r = 1$

2. λ_r denotes the index of the innermost loop referenced by region R_r of G. For the example given in the previous section, the λ_r values point to the `k`, `x`, `l` and `y` loops respectively.

3. The *innermost loop* λ for a given individual I is the index of the loop where the loop nest has to be split when considering all regions selected by I:
   $\lambda = \max\{\lambda_r \mid r \in [1, M], I_r = 1\}$

4. ι_l denotes the number of *if*-statements which are located in the body of loop L_l but not in any other loop L_l' which is nested in L_l. For the code of figure 1, $\iota_{\mathtt{j}}$ is equal to 2, all other values ι_l are zero.

5. IF_l denotes the number of *if*-statements that are evaluated 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 genetic algorithm described in this section uses the same parameters (i. e. population size, number of generations, stopping criteria, ...) as the one used for condition optimization (see section 3 for concrete values). The fitness of an individual I represents the total number IF_I of *if*-statements that are executed when splitting Λ and considering the regions R_r selected by I. For the fitness evaluation, IF_I is incremented by one for every execution of the splitting-if. If the splitting-if evaluates to true, the counter retains this value. In the other case, IF_I has to be incremented by the number of executed original *if*-statements as depicted in figure 8.

After the termination of the genetic algorithm, the best place for inserting the splitting-if in the loop nest is determined using the innermost loop λ of the fittest individual. 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.

5. Benchmarking Results

The techniques presented in section 4 have been fully implemented and integrated into a single tool for source code transformation. The well-known class hierarchy SUIF [34] is used as intermediate representation for ANSI-C programs. The polyhedral and genetic algorithms are based on the Polylib [33] and PGAPack [17] libraries.

Our loop nest splitting tool was applied to the source codes of three multimedia applications from the image processing domain. The first one is a medical image processing application (*CAVITY* [3]) which extracts contours from images to help physicians detect brain tumors. Since this application has served as test driver for the so called DTSE transformations described in [9], we use it as benchmark in this report to show that loop nest splitting is able to remove the overhead introduced by DTSE. The cavity detector consists of two nested loops scanning an image in its x and y dimensions. The index variables iterate from 0 to 1,002 (y) and 0 to 1,282 (x) leading to a total amount of 1,286,849 loop iterations. During each iteration, 13 *if*-statements suitable for loop nest splitting are executed containing 40 index variable accesses and 71 arithmetical logical ANSI-C operations.

The second benchmark is the MPEG 4 full search motion estimation kernel [5] (*ME*) which already served as example throughout this report. From figure 1 it can be seen that the sevenfold loop nest defines an iteration space of size 45,722,880. In the innermost loop of the ME benchmark, there are two *if*-statements with a total of eight variable accesses and 14 C operations.

Finally, the QSDPCM algorithm [29] for scene adaptive coding serves as third test driver. Like the previous benchmark, its loop nest has a depth of seven so that 12,830,400 iterations of the innermost loop are executed. In the hot-spot of this application, one *if*-statement containing 12 variable references and 19 computational operations can be found.

The runtimes of our loop nest splitting tool are very low. For the optimization of CAVITY including the polyhedral and genetic algorithms and the transformation of the code on an AMD Athlon (1.3 GHz), 1.58 CPU seconds are required (ME: 0.84 s, QSDPCM: 0.41 s). The main characteristics of the benchmarks are summarized in table 1.

In section 5.1, the impacts of loop nest splitting on CPU pipeline and cache behavior are illustrated. Section 5.2 shows how the execution times and code sizes of the benchmarks are affected by loop nest splitting on a variety of different processors. Section 5.3 demonstrates that the techniques explained in this report are able to reduce the energy consumption of the benchmarks considerably. Measurements based on a precise energy model for the ARM7TDMI embedded RISC core are presented here.

	CAVITY	ME	QSDPCM
Nest depth	2	7	7
# Iterations	1.286.849	45.722.880	12.830.400
# If-statements	13	2	1
# Variable accesses	40	8	12
# Operations	71	14	19
Optimization runtime	1.58 s	0.84 s	0.41 s

Table 1. Characteristical Properties of Benchmarks

5.1. Pipeline and Cache Behavior

In this section, the effects of loop nest splitting on several cache and pipeline parameters of three different processors are demonstrated. Because of their easy to access on-chip debugging and profiling facilities, the results reported in this section have been measured with an Intel Pentium III processor (section 5.1.1), a Sun UltraSPARC III CPU (section 5.1.2) and a MIPS R10000 (section 5.1.3).

5.1.1. Intel Pentium III

Following the definition given by Hennessy and Patterson [7], the Intel Pentium III processor is a typical Harvard architecture having a single main memory but separate instruction and data caches as illustrated in figure 9. Internally, the processor communicates with its level 1 caches via separate data and instruction buses. The on-chip level 1 data and instruction caches each have a size of 16 kB and use a 4-way set-associative mapping with 32 bytes per cache line. The off-chip level 2 cache is a unified cache storing both data and instructions. It has a size of 256 kB with 32 bytes per line using an 8-way set-associative mapping.

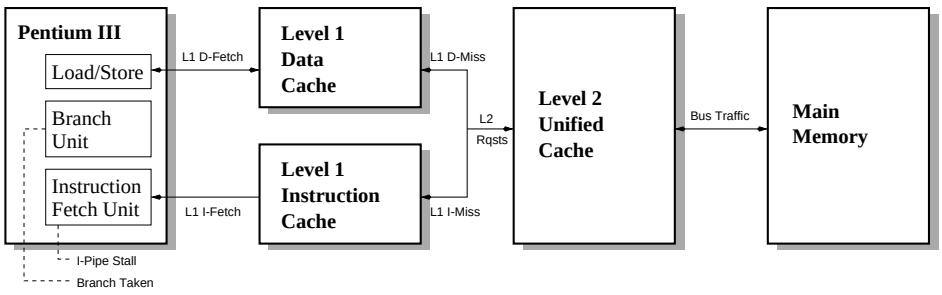

Figure 9. Structure of the Intel Pentium III architecture

Figure 10 shows how the instruction pipeline, the caches and the transfers of data and instructions on the system buses are affected by loop nest splitting. For obtaining these results, the benchmarks were compiled and linked with a performance measuring library [6] that makes use of hardware performance-monitoring coun-

ters of the Pentium processor [10]. This way, reliable values can be generated by profiling without using erroneous cache simulation software. The probe-points where the data shown in figure 10 was measured within the Pentium III architecture are indicated in figure 9. The results for the optimized benchmarks after loop nest splitting are given as a percentage of the unoptimized versions denoted as 100%.

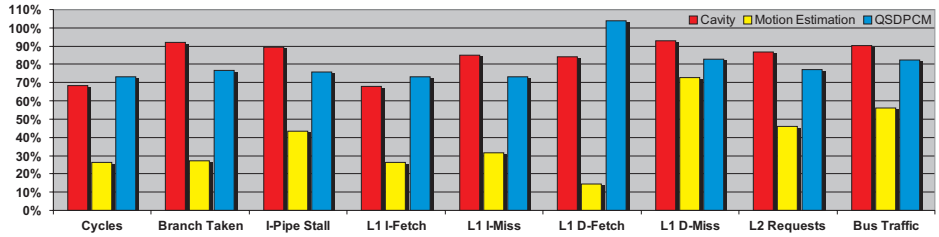

Figure 10. Pipeline and Cache Behavior for Intel Pentium III

As can be seen from the columns Branch Taken and I-Pipe Stall, loop nest splitting is able to generate a more regular control flow for all three benchmarks. The number of taken branch instructions is reduced between 8.1% (CAVITY) and 72.9% (ME) consequently leading to similar reductions of pipeline stalls (10.4% – 56.5%).

The fact that loop nest splitting removes control flow overheads and thereby significantly improves the runtime behavior of caches is reflected by the columns L1 I-Fetch and L1 I-Miss of figure 10. As can be seen, the number of instruction fetches is reduced by 26.7% (QSDPCM) resp. 73.6% (ME), and the total amount of instruction cache misses is improved between 14.8% (CAVITY) and 68.5% (ME). Due to the removal of conditions and the involved reduction of index variable accesses, the L1 data cache also benefits from the optimization. Data fetches from the cache are reduced by 16% (CAVITY) resp. 85.4% (ME); only in the case of the QSDPCM benchmark does the amount of data fetches increase by 3.9% due to the insertion of spill code by the compiler. Furthermore, figure 10 shows that the absolute amount of data cache misses drops by 7.2% (CAVITY) up to 27.2% (ME).

Transfers of instructions and data between L1 caches, the unified L2 cache and main memory are optimized likewise. As can be seen from column L2 Requests from figure 10, the L2 cache is referenced 13.1% (CAVITY) up to 53.8% (ME) less than before the optimization. When focusing on the system bus connecting the Pentium with the main memory, between 9.9% (CAVITY) and 43.8% (ME) less traffic can be observed.

All in all, the factors mentioned above lead to a speed-up of our benchmarks (see column Cycles) between 26.7% (QSDPCM) and 73.5% (ME). Given that a considerable amount of the total energy consumption results from memory accesses, loop nest splitting leads to significant energy savings, because activities on system

buses between functional units, L1 and L2 caches and main memory are reduced by 25.9% (CAVITY), 77.8% (ME) and 18.7% (QSDPCM, average values for columns L1 I-Fetch, L1 D-Fetch, L2 Requests and Bus Traffic).

5.1.2. Sun UltraSPARC III

In principle, the architecture of the Sun UltraSPARC III processor [32] is comparable to the one of the Pentium III, since it also consists of separate level 1 data and instruction caches and a unified level 2 cache (compare figure 11). As in the case of the Pentium, this CPU allows processor event counting for performance measurements.

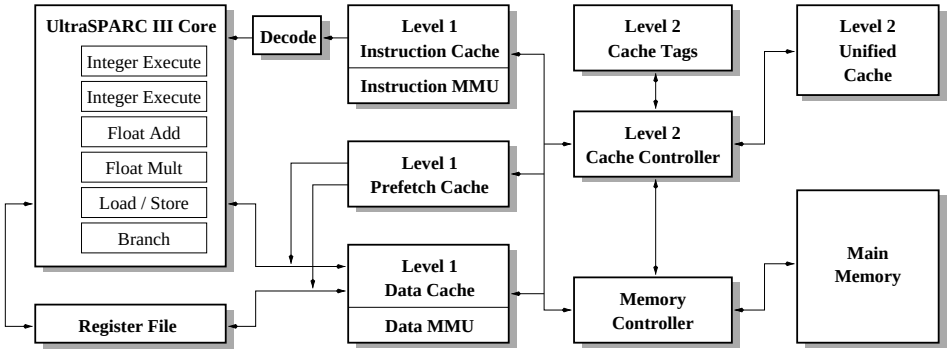

Figure 11. Structure of the Sun UltraSPARC III architecture

The main difference between these processors is the size of the various memories. The on-chip level 1 instruction cache of the UltraSPARC III has a size of 32 kB, the level 1 data cache is 64 kB large. Both caches use a 4-way set associative mapping with 32 bytes per cache line. The external level 2 cache has a size of 8 MB with 512 bytes per line using a 2-way set-associative mapping. The Sun processor has a very large register file containing 160 general purpose integer registers and up to 32 floating point registers. The floating point registers can be loaded automatically by using the prefetch cache and its associated speculative hardware controller[2].

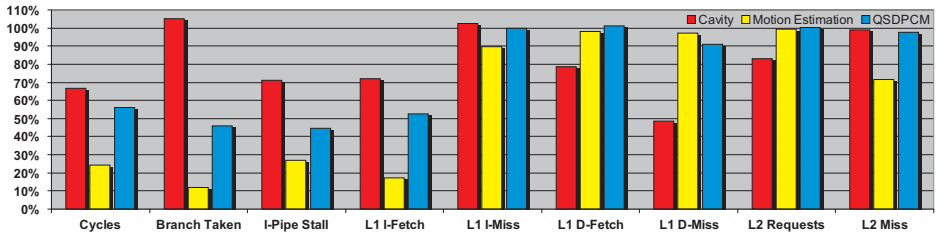

Figure 12. Pipeline and Cache Behavior for Sun UltraSPARC III

[2] Since floating point computations are not performed by the applications considered in this section, the UltraSPARC III prefetch cache is not taken into account during benchmarking.

Figure 12 shows the effects of loop nest splitting on the instruction pipeline and the caches of the UltraSPARC III processor. This processor has a very long instruction pipeline consisting of fourteen stages which is very sensitive to stalls due to its complexity. For this reason, loop nest splitting leads to very high gains for the SPARC processor as can be seen from the columns Branch Taken and I-Pipe Stall. The total number of pipeline stalls is reduced by 29.1% in the case of the CAVITY benchmark. For the QSDPCM application, improvements of 55.5% have been observed, and the optimization of the ME benchmark leads to 73.1% less pipeline stalls.

When considering the number of taken branches as an indicator of the linearity of the control flow, similar improvements can been observed. After loop nest splitting, the execution of the QSDPCM benchmark leads to 54% less branches, in the case of the MPEG-4 motion estimation, a reduction by 88.3% can be reported. The fact that the branching behavior of the CAVITY detector becomes worse by 5.3% after the optimization is remarkable. Further experiments and measurements have shown that this effect is caused by the current version (6.2) of the Sun Workshop compiler used and that this is not a misconduct of loop nest splitting itself.

In the code of the CAVITY benchmark, a sequence of eight *if*-statements checking data-dependent conditions can be found. These *if*-statements are not suitable for loop nest splitting. Therefore, they are duplicated by our transformation – one time in the *then*-part of the splitting-if, and one time in the *else*-part. It has turned out that these *if*-statements are the reason for the increase of taken branches when using the Sun compiler. When commenting out these data-dependent *if*-statements, the CAVITY detector only contains *if*-statements that can be processed by loop nest splitting. Under these circumstances, the branching behavior of the benchmark is improved as expected by 28.4%. This number shows that the gains achieved by our optimization of loop-variant *if*-statements are compensated by some particular transformations of this Sun compiler applied to the data-dependent *if*-statements. The presumption that this current compiler version is the reason for the behavior observed can be validated when using the previous version 6.1 instead. In this case, a reduction of 33.7% of taken branches is observed.

As in the case of the Pentium processor, the number of accesses to the L1 instruction cache is reduced significantly. The improvements vary between 28.2% (CAVITY) and 82.7% (ME). In contrast, the amount of I-cache misses is only reduced for the ME benchmark (10.4% improvement). For the other benchmarks, the number of cache misses remains nearly unchanged (less than 0.1% of improvement for QSDPCM and 2.6% degradation for CAVITY).

When focusing on the level 1 data cache, it can be seen that the number of accesses to it is only reduced after the optimization of the CAVITY benchmark. In this case, 21.2% less data transfers concerning this cache are performed. For the other benchmarks, the number of data fetches stays constant. The reason for this behavior is

the large register file of the UltraSPARC processor. Since the ME and QSDPCM benchmarks only use very few local variables, the compiler is able to store them entirely in the processor registers even before loop nest splitting. The number of L1 data cache misses is reduced between 2.9% (ME) and 51.4% (CAVITY).

As can be seen from the columns L2 Requests and L2 Miss, the level 2 unified cache also benefits from loop nest splitting. In the case of the CAVITY benchmark, the number of accesses to this cache is reduced by 17.2%, whereas no changes have been observed for the other benchmarks. L2 cache misses are only reduced for the ME benchmark (improvement of 28.6%), the transformation of the other applications leads only to marginal improvements (1.1% for CAVITY, 2.3% for QSDPCM).

In their combination, the improvements of cache and pipeline behavior lead to large speed-ups of all benchmarks for the Sun UltraSPARC III processor. The column Cycles denotes the number of clock cycles required for the execution of the benchmarks. It can be seen that the speed-ups range from 33.3% (CAVITY) up to 75.8% (ME).

5.1.3. MIPS R10000

With respect to the first and second level caches, the architecture of the MIPS R10000 processor is equivalent to the one of the Intel Pentium III. In the case of the MIPS, the sizes of the level 1 instruction and data caches are 32 kB. Both caches are 2-way set-associative, the block size of the instruction cache is 16 words, the data cache uses 8 words per block. The secondary unified cache has a size of 1 MB and is also 2-way set-associative.

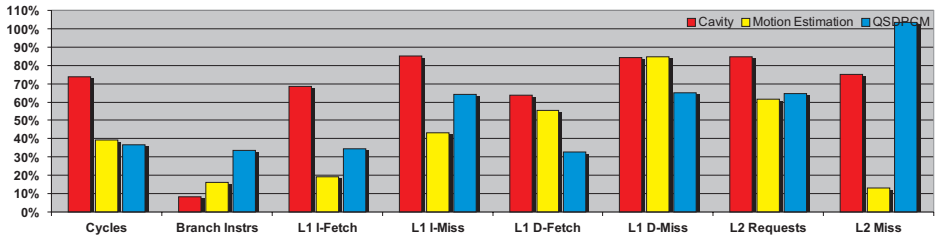

Figure 13. Pipeline and Cache Behavior for MIPS R10000

Since the MIPS performance counters [20] are not able to profile the performance of the instruction pipeline, no detailed data with respect to pipeline stalls and branching behavior can be given. Only the total number of executed branch instructions can be measured. The column Branch Instrs of figure 13 denotes all branch instructions – taken branches as well as untaken jumps – evaluated during the execution of the benchmarks. This column shows that reductions of executed branch instructions between 66.3% (QSDPCM) and 91.8% (CAVITY) have been achieved by loop nest splitting.

The MIPS caches benefit from our optimization for a large extent. As can be seen in figure 13, loop nest splitting is able to reduce the number of accesses to the level one caches significantly. Instruction fetches are decreased between 31.5% (CAVITY) and 81% (ME), whereas data cache accesses are alleviated between 36.4% (CAVITY) and 67.3% (QSDPCM). Due to these improvements of cache accesses, the number of cache misses is reduced also. The number of level 1 instruction cache misses is diminished between 14.7% (CAVITY) and 56.9% (ME), data cache misses occur between 15.2% (ME) and 34.8% (QSDPCM) less frequent after the optimization.

Improvements related to the level 2 unified cache are observed likewise. The number of read/write requests to this cache is reduced between 15.4% (CAVITY) and 38.3% (ME). Level 2 cache misses are slightly increased by 3.3% for the QSDPCM benchmark and are decreased by 25% (CAVITY) respectively 87% (ME).

Accumulated, the enhancements reported in detail for the MIPS R10000 processor lead to an acceleration of the benchmarks between 26.1% (CAVITY) and 63.4% (QSDPCM).

5.2. Execution Times and Code Sizes

In this section, we show that the improvements reported in section 5.1 are not restricted to the platforms presented there. A large collection of different processors including embedded RISC architectures, VLIW machines, DSPs and RISC workstations was used to measure the runtimes and code sizes of the benchmarks.

For generating the data presented in this section, the source codes of the benchmarks before and after loop nest splitting were compiled for a Sun UltraSparc III, Intel Pentium Pro MMX, HP-PA 9000, MIPS R10000, PowerPC G3, DEC Alpha EV4, Philips TriMedia TM-1000, Texas Instruments TMS320C62 and an ARM7 TDMI core, the latter both in 16-bit thumb-mode and 32-bit arm-mode. Since it is known that optimizations in state-of-the art compilers are hardly ever able to cross control flow boundaries, loop nest splitting has an enabling effect on the compiler optimizations due to the explicit elimination of such boundaries in the hot-spots of an application. For exploring this implicit optimization potential of our transformation, compilers were always invoked using the highest degree of optimization. The compilers used for the different processors are listed in table 2 together with their corresponding command line options.

Runtimes are measured by executing and profiling the compiled programs on existing hardware using either available workstations or evaluation boards. Since a detailed profiling as done in section 5.1 including cache statistics can not be done for most of these processors due to the lacking hardware profiling support, only the runtimes of the benchmarks are reported here.

Processor	Compiler	Version	Command line options
Sun UltraSparc III	Sun WorkShop	6.2	CC -fast -xtarget=ultra3 -xarch=v9b
Intel Pentium Pro	Sun WorkShop	6.2	CC -fast -xtarget=pentium_pro -xarch=pentium_pro
HP-PA 9000/785	HP Softbench	A.01.18	aCC +O4
MIPS R10000	MIPSpro	7.2.1	CC -mips4 -r10000 -O2
PowerPC G3	GNU gcc	2.95.4	gcc -O7
DEC Alpha EV4	GNU gcc	2.95.4	gcc -O7
TriMedia-1000	Philips TriMedia SDE	V5.3.4	tmcc -O3 -host WinNT
TI TMS320C62	TI Code Generation Tools	3.01	cl6x -o3
ARM7 TDMI thumb	ARM SDT	2.50	tcc -O2 -Otime
ARM7 TDMI arm	ARM SDT	2.50	armcc -O2 -Otime

Table 2. Compilers and Optimization Levels for Runtime Measurements

The code sizes of the benchmarks are determined by compilation of the C source codes in the way described above and generation of assembly listings instead of executable binaries. The total amount of assembly instructions occurring in the generated files is then taken as a measure for the code sizes.

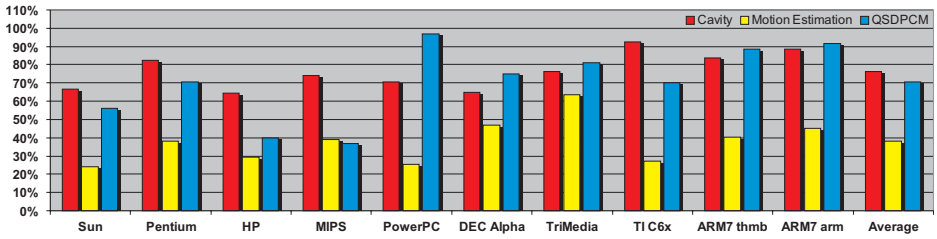

Figure 14. Runtimes after Loop Nest Splitting

As can be seen in figure 14, the runtimes of the CAVITY benchmark are improved between 7.7% (TI C6x) and 35.7% (HP). On the average over all processors, a speed-up of 23.6% has been 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 for this benchmark. This application is accelerated by 62.1% in average. The minimum speed-up amounts to 36.5% (TriMedia), whereas the Sun platform honors the optimization with an acceleration of 75.8%. For the QSDPCM benchmark, the improvements range from 3% (PowerPC) up to 63.4% (MIPS). On the average, loop nest splitting leads to an acceleration of 29.3% for QSDPCM.

As for every code replicating optimization, the large improvements of loop nest splitting with respect to execution times entail increases in code sizes. In the case of the CAVITY benchmark, the average speed-up of 23.6% leads to an average increase of code sizes of 60.9%. The assembly code generated by the MIPS compiler

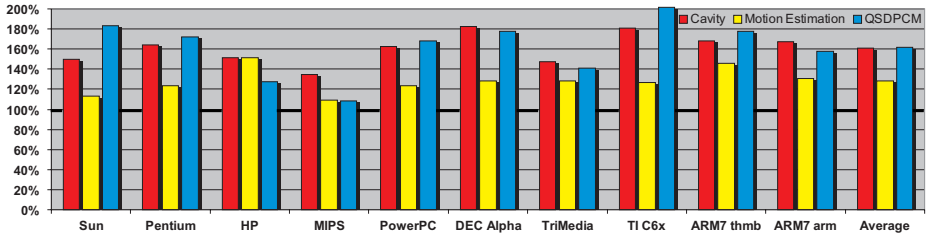

Figure 15. Code Sizes after Loop Nest Splitting

after loop nest splitting is the most condensed one, since it is only 34.7% larger than the original code. In the case of the DEC Alpha, an increase of 82.8% of code size has been measured. Among all benchmarks considered in this report, it is the ME application for which loop nest splitting leads to the highest accelerations. In addition, figure 15 shows that our optimization also leads to the smallest increases in code size in this case. In the average, an increase of only 28% has been observed whereby the minimum and maximum growths amount to 9.2% (MIPS) and 51.4% (HP). Finally, the increases of code size for the QSDPCM application vary between 8.7% (MIPS) and 101.6% (TI C6x) leading to an average augmentation of 61.6%.

5.3. Energy Consumption of an ARM7TDMI Core

The increases of code sizes by a few hundred instructions (see the previous section 5.2) are not a serious drawback of our optimization, since the added energy required for storing these instructions is compensated by the savings achieved by loop nest splitting. The results given in this section demonstrate that loop nest splitting is a powerful technique for reducing the energy dissipation of embedded multimedia applications. Using the instruction-level energy model for an ARM7TDMI embedded RISC core presented in [27], we are able to compute the energy consumption of our benchmarks before and after loop nest splitting with an accuracy of 1.7%.

The energy measurements have been performed by firstly compiling the source code of the benchmarks using the energy aware compiler *encc* [28]. The generated assembly code is then processed by assembler and linker to form a binary executable that is executed using a simulator. A particular energy profiler takes as input the information generated by the simulator, which includes information on number and kind of executed instructions as well as different kinds of accesses to memories. Additionally, the profiler also reads from a database that contains information about the energy consumption of each individual instruction as well as memory accesses. The individual values are summed up, considering inter-instruction effects, and the final result, the performance statistics of the complete program, is generated. These statistics include – among others – the number of

executed instructions, number of memory accesses and the energy consumption.

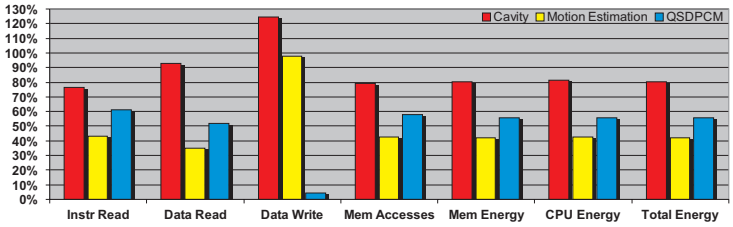

Figure 16. Energy Consumption after Loop Nest Splitting

The first four columns of figure 16 confirm the observations already presented in section 5.1 that loop nest splitting leads to a higher locality of instructions and data thereby reducing memory bandwidth and bus demand. In the case of the ARM7, between 23.5% (CAVITY) and 56.9% (ME) fewer instructions are fetched from the memory. Accesses to data stored in the memory are likewise optimized. From column Data Read, it can be seen that data transfers from the memory to the ARM7 core are reduced between 7.1% (CAVITY) and 65.3% (ME). Since data is read much more frequently than it is written to memory, the fact that 24.5% more memory stores occur in the case of CAVITY is not significant. This increase is due to the fact that the compiler needs to insert some spill code during register allocation. The contrary holds for the QSDPCM benchmark, where the elimination of spill code leads to a reduction of memory stores by 95.4%. When focusing on the total energy dissipation of the entire memory system, only the absolute number of memory accesses is important. Column Mem Accesses clearly shows that loop nest splitting leads to large reductions, ranging from 20.8% up to 57.2%.

These improvements consequently lead to large diminuitions of the energy dissipation of the memory system (see column Mem Energy). The energy profiling shows that we are able to achieve savings between 19.6% (CAVITY) and 57.7% (ME). In addition, the benefits of loop nest splitting are not limited to the memory system. As can be seen from column CPU Energy, the amount of energy consumed by the ARM7 core itself is reduced by the same order of magnitude; gains reaching from 18.4% (CAVITY) up to 57.4% (ME) have been observed. Accumulated, the total energy consumption of the processor and its memory is reduced between 19.2% (CAVITY) and 57.6% (ME). These results demonstrate that loop nest splitting is capable of optimizing the locality of instruction and data accesses simultaneously as desired by Kim et al. [13].

6. Conclusions

In this report, a novel source code transformation called loop nest splitting is presented which is able to remove redundancies in the control flow of embedded multimedia applications. With the help of polyhedral and genetic algorithms, *if*-statements occurring in loop nests are analyzed. Conditions having no effect on the control flow of an application are removed. Ranges of values of the index variables are identified for which all *if*-statements are provably satisfied. With the help of these ranges of values, the source code of an application is transformed in such a way that the total number of *if*-statements evaluated during the execution of a loop nest is minimized.

Detailed benchmarking of three complex benchmarks shows that loop nest splitting is able to improve the branching and pipeline behavior significantly. For the three processors supporting profiling of the instruction pipeline (Intel Pentium III, Sun UltraSPARC III, MIPS R10000), average reductions of pipeline stalls in a range between 19.7% and 64.8% are measured. Furthermore, the caches also benefit from our optimization because the average number of I-cache misses is decreased by 8.9% – 45.3% (D-cache misses: 15.1% – 24.7%). Since the number of instruction and data transfers via system buses is reduced to a large extent, loop nest splitting entails a higher locality of instructions and data. This less bus demand observed during benchmarking consequently leads to large power savings. Concrete reductions of energy dissipation are measured on an ARM7TDMI embedded RISC processor. For this environment, it has been shown that our optimizations lead to improvements between 19.2% and 57.6%. The measurements have shown that these total energy savings attribute from savings both of the instruction and data memory. An extended benchmarking using a set of ten different programmable processors shows that we are able to speed up the runtimes of the benchmarks between 23.6% and 62.1% on the average.

The selection of the benchmarks used in this report demonstrates that our optimization is a very general and powerful technique. It is not only able to improve the code of typical real-life multimedia applications. But in addition, loop nest splitting can be used to eliminate the negative effects of other source code transformation frameworks introducing a large control flow overhead into an application.

In the future, efforts will be made for generalizing 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. Furthermore, if the global search space introduced in section 4.4 defines the empty set due to *if*-statements which can not be satisfied at the same time, our algorithm terminates without having performed any optimization. Our future work will also include extensions of the methods presented in section 4 so that only a promising subset of all *if*-statements will be considered in that case.

A. Benchmarking Data

In this appendix, all data measured during the benchmarking of loop nest splitting is listed. Out of the material presented in the following sections, all results given in section 5 can be reproduced. In section A.1, the data for rating the behavior of the instruction pipelines and caches is given. Section A.2 lists the exact runtimes of the benchmarks for all processors considered in section 5.2 whereas the material concerning the evolution of code sizes for these processors is specified in section A.3. Finally, data related to the energy dissipation of an ARM7TDMI core is presented in section A.4.

A.1. Values of Performance-Monitoring Counters

A.1.1. Intel Pentium III

This section contains all pipeline and cache related counter values determined with the help of the PMC library [6] for the Intel Pentium III processor. In the following tables, the absolute numbers of events recorded by this library during the execution of the benchmarks are listed. To avoid side-effects of other running processes being measuered during the execution of the benchmarks, all periodically active processes were terminated beforehand. Furthermore, each measurement was done twenty times. After that, the minimum and maximum values measured were discarded and average values over the remaining eighteen measurements were computed.

Table 3 shows the average values for the original versions of the benchmarks, whereas the results after loop nest splitting are given in table 4. Out of these values, the diagram shown in figure 10 (see page 23) was generated.

Original	CAVITY	ME	QSDPCM
Cycles	457,882,361	1,057,369,204	416,172,096
Branch Taken	15,503,809	209,133,207	35,145,643
I-Pipe Stall	3,724,967	4,269,815	5,256,488
L1 I-Fetch	453,220,737	1,051,974,908	406,400,954
L1 I-Miss	10,122	17,424	39,100
L1 D-Fetch	277,196,342	597,853,395	144,444,058
L1 D-Miss	136,056	43,693	46,332
L2 Requests	113,894	86,109	157,424
Bus Traffic	148,941	69,129	58,746

Table 3: PMC Values for Original Benchmark Versions

Loop Nest Split	CAVITY	ME	QSDPCM
Cycles	312,217,838	279,831,507	304,873,722
Branch Taken	14,248,396	56,755,097	26,928,781
I-Pipe Stall	3,338,489	1,858,695	3,994,489
L1 I-Fetch	308,599,829	277,828,106	297,909,549
L1 I-Miss	8,626	5,494	28,699
L1 D-Fetch	232,823,980	87,489,875	150,062,180
L1 D-Miss	126,279	31,809	38,419
L2 Requests	99,009	39,770	121,331
Bus Traffic	134,209	38,857	48,418

Table 4: PMC Values for Benchmarks after Loop Nest Splitting

A.1.2. Sun UltraSPARC III

The pipeline and cache related material presented in section 5.1.2 for the Sun UltraSPARC III processor was collected with the help of the performance instrumentation counters (*PIC*) [32] of the processor and the `cputrack` utility of the Solaris operating system. Each measurement was done twenty times. Hereafter, the minumum and maximum values measured were discarded and average values over the remaining eighteen measurements were computed.

Table 5 shows the average values for the original versions of the benchmarks, whereas the results after loop nest splitting are given in table 6. Out of these values, the diagram shown in figure 12 (see page 25) was generated.

Original	CAVITY	ME	QSDPCM
Cycles	393,472,375	868,531,375	237,383,891
Branch Taken	26,211,162	147,832,045	27,915,617
I-Pipe Stall	13,157,318	34,530,345	6,123,507
L1 I-Fetch	147,889,880	412,710,424	128,183,341
L1 I-Miss	18,293	15,786	11,767
L1 D-Fetch	89,690,758	6,007,608	24,754,265
L1 D-Miss	259,081	168,838	140,497
L2 Requests	66,330,936	5,812,883	24,571,642
L2 Miss	25,121	10,179	3,558

Table 5: PIC Values for Original Benchmark Versions

Loop Nest Split	CAVITY	ME	QSDPCM
Cycles	262,503,992	210,301,284	133,130,265
Branch Taken	27,598,854	17,314,895	12,836,237
I-Pipe Stall	9,335,217	9,300,382	2,727,359

Loop Nest Split	CAVITY	ME	QSDPCM
L1 I-Fetch	106,189,973	71,526,680	67,382,877
L1 I-Miss	18,778	14,139	11,760
L1 D-Fetch	70,652,809	5,903,200	25,050,442
L1 D-Miss	126,011	163,925	128,130
L2 Requests	54,952,441	5,764,959	24,655,932
L2 Miss	24,850	7,265	3,476

Table 6: PIC Values for Benchmarks after Loop Nest Splitting

A.1.3. MIPS R10000

For accessing the MIPS R10000 performance counters [20] (*PC*), the `perfex` utility was used. As in the section before, every measurement was performed twenty times for obtaining stable values. The following tables 7 and 8 show the average results of all measurements.

Original	CAVITY	ME	QSDPCM
Cycles	364,843,119	729,839,198	124,772,234
Branch Instrs	55,231,492	249,296,813	53,268,372
L1 I-Fetch	576,065,664	1,115,530,132	282,093,449
L1 I-Miss	61,616	51,843	2,947
L1 D-Fetch	199,546,246	39,341,594	24,406,526
L1 D-Miss	155,691	41,550	7,396
L2 Requests	217,277	93,376	10,347
L2 Miss	13,922	11,879	436

Table 7: PC Values for Original Benchmark Versions

Loop Nest Split	CAVITY	ME	QSDPCM
Cycles	269,520,370	285,561,094	45,706,635
Branch Instrs	4,532,214	40,149,316	17,973,082
L1 I-Fetch	394,699,884	211,966,736	97,632,479
L1 I-Miss	52,555	22,365	1,887
L1 D-Fetch	126,904,539	21,892,341	7,985,215
L1 D-Miss	131,260	35,238	4,819
L2 Requests	183,898	57,647	6,707
L2 Miss	10,442	1,558	451

Table 8: PC Values for Benchmarks after Loop Nest Splitting

A.2. Execution Times

In this section, the absolute runtimes of the benchmarks on the ten processors considered in this report are shown. The values given in the tables below for the Pentium, HP, PowerPC, DEC and TriMedia processor denote the number of CPU seconds required for the execution of a benchmark. For the Sun, MIPS, TI and ARM processor, the total amount of clock cycles is given. Table 9 shows the runtimes before the optimization. The speed-ups reported in figure 14 (see page 28) have been generated by comparing this data with the runtimes after loop nest splitting which can be found in table 10.

Original	CAVITY	ME	QSDPCM
Sun	393,472,375	868,531,375	237,383,891
Pentium	0.35	1.10	0.29
HP	0.56	1.87	0.30
MIPS	364,843,119	729,839,198	124,772,234
PowerPC G3	0.82	2.41	0.48
DEC Alpha	4.61	12.76	3.46
TriMedia	5.10	21.92	3.38
TI C6x	1,264,823,138	2,681,116,485	1,237,341,543
ARM7 thumb	146,308,938	8,230,155,008	2,137,855,898
ARM7 arm	172,938,892	9,663,200,190	2,005,827,480

Table 9: Runtimes of Original Benchmark Versions

Loop Nest Split	CAVITY	ME	QSDPCM
Sun	262,503,992	210,301,284	133,130,265
Pentium	0.29	0.34	0.23
HP	0.36	0.55	0.12
MIPS	269,520,370	285,561,094	45,706,635
PowerPC G3	0.58	0.61	0.46
DEC Alpha	2.99	6.01	2.60
TriMedia	3.89	13.93	2.74
TI C6x	1,167,015,404	733,300,141	867,214,454
ARM7 thumb	122,616,256	3,305,386,900	1,891,857,950
ARM7 arm	152,914,152	4,367,474,972	1,840,264,050

Table 10: Runtimes of Benchmarks after Loop Nest Splitting

A.3. Code Sizes

In the following tables 11 and 12, the code sizes before and after loop nest splitting for a variety of processors are given. The values listed in this section denote the number of assembly instructions emitted by the compilers used.

Original	CAVITY	ME	QSDPCM
Sun	634	369	165
Pentium	482	241	138
HP	444	492	176
MIPS	691	456	242
PowerPC G3	594	264	160
DEC Alpha	615	302	143
TriMedia	1,750	960	585
TI C6x	577	295	184
ARM7 thumb	601	300	273
ARM7 arm	469	216	136

Table 11: Code Sizes of Original Benchmark Versions

Loop Nest Split	CAVITY	ME	QSDPCM
Sun	951	418	303
Pentium	791	297	238
HP	673	745	224
MIPS	931	498	263
PowerPC G3	964	325	269
DEC Alpha	1,124	387	254
TriMedia	2,580	1,235	825
TI C6x	1,042	373	371
ARM7 thumb	1,010	437	485
ARM7 arm	784	282	215

Table 12: Code Sizes of Benchmarks after Loop Nest Splitting

A.4. Energy Consumption of an ARM7TDMI Core

This section contains the material gathered when using an accurate instruction-level energy model [27] for an ARM7 embedded RISC core during benchmarking. The first four rows of the following tables denote the absolute number of different kinds of memory accesses. In the lower part of these tables, the energy dissipation of the benchmarks is given in μJ. Table 13 shows the data of the energy model

before loop nest splitting, whereas table 14 gives the values after our optimization.

Original	CAVITY	ME	QSDPCM
Instruction Read	1,488,952,642	4,187,759,099	1,436,042,507
Data Read	171,633,720	800,612,554	397,077,933
Data Write	36,299,715	95,364,299	29,922,359
Total Memory Accesses	1,696,886,077	5,083,735,952	1,863,042,799
CPU Core Energy Consumption	19,996,772.345	63,456,704.577	24,112,184.733
Memory Energy Consumption	44,051,984.515	143,911,901.100	54,457,644.234
Total Energy Consumption	64,048,756.860	207,368,605.677	78,569,828.967

Table 13: Energy Consumption of Original Benchmark Versions

Loop Nest Split	CAVITY	ME	QSDPCM
Instruction Read	1,138,610,481	1,804,375,600	874,713,006
Data Read	159,407,055	278,139,796	205,511,633
Data Write	45,190,087	93,331,459	1,369,659
Total Memory Accesses	1,343,207,623	2,175,846,855	1,081,594,298
CPU Core Energy Consumption	16,320,808.977	27,015,207.376	13,489,391.884
Memory Energy Consumption	35,406,147.932	60,858,791.538	30,364,170.438
Total Energy Consumption	51,726,956.909	87,873,998.914	43,853,562.322

Table 14: Energy Consumption of Benchmarks after Loop Nest Splitting

References

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

[2] D. F. Bacon, S. L. Graham and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), Dec. 1994.

[3] M. Bister, Y. Taeymans and J. Cornelis. Automatic Segmentation of Cardiac MR Images. *IEEE Journal on Computers in Cardiology*, 1989.

[4] C. Ghez, M. Miranda, A. Vandecappelle et al. Systematic high-level Address Code Transformations for Piece-wise Linear Indexing. In *Proceedings of "IEEE Workshop on Signal Processing Systems" (SIPS)*, Lafayette, Oct. 2000.

[5] S. Gupta, M. Miranda, F. Catthoor et al. Analysis of High-level Address Code Transformations for Programmable Processors. In *Proceedings of "Design, Automation and Test in Europe" (DATE)*, Paris, Mar. 2000.

[6] D. Heller. Performance-Monitoring Counters Library for Intel / AMD Processors and Linux. *http://www.scl.ameslab.gov/Projects/Rabbit/*, Oct. 2000.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers, San Mateo, 1990.

[8] J. H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1992.

[9] 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, chapter 1, pages 1–39. Marcel Dekker Inc., New York, 2001.

[10] *IA-32 Intel Architecture Software Developer's Manual*, volume 3. Intel Corp., Santa Clara, 2002.

[11] M. Kandemir, N. Vijaykrishnan, M. J. Irwin and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of "Design Automation Conference" (DAC)*, Los Angeles, June 2000.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[13] H. S. Kim, M. J. Irwin, N. Vijaykrishnan and M. Kandemir. Effect of compiler optimizations on memory energy. In *Proceedings of "IEEE Workshop on Signal Processing Systems" (SIPS)*, Lafayette, Oct. 2000.

[14] B. Landwehr. A Genetic Algorithm based Approach for Multi-Objective Data-Flow Graph Optimization. In *Proceedings of "Asia South Pacific Design Automation Conference" (ASP-DAC)*, Hong Kong, Jan. 1999.

[15] R. Leupers. *Code Optimization Techniques for Embedded Processors – Methods, Algorithms and Tools*. Kluwer Academic Publishers, Boston, 2000.

[16] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings of "International Symposium on System Synthesis" (ISSS)*, Dec. 1998.

[17] D. Levine. Users Guide to the PGAPack Parallel Genetic Algorithm Library. Technical Report ANL-95/18, Argonne National Laboratory, Jan. 1996.

[18] N. Liveris, N. D. Zervas, D. Soudris and C. E. Goutis. A Code Transformation-Based Methodology for Improving I-Cache Performance of DSP Applications. In *Proceedings of "Design, Automation and Test in Europe" (DATE)*, Paris, Mar. 2002.

[19] M. Lorenz, R. Leupers, P. Marwedel, T. Dräger and G. Fettweis. Low-Energy DSP Code Generation Using a Genetic Algorithm. In *Proceedings of "International Conference on Computer Design" (ICCD)*, Austin, Sept. 2001.

[20] *MIPS R10000 Microprocessor User's Manual*. MIPS Technologies, Inc., Mountain View, Oct. 1996.

[21] T. S. Motzkin, H. Raiffa, G. L. Thompson and R. M. Thrall. The double description method. *Contributions to the Theory of Games – Annals of Mathematics Studies*, (28), 1953.

[22] S. S. Muchnick. Optimizing Compilers for SPARC. *SunTechnology*, 1(3), 1988.

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

[24] R. Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, Boston, 1998.

[25] *TriMedia TM1000 Preliminary Data Book*. Philips Electronics North America Corp., Oct. 1997.

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

[27] S. Steinke, M. Knauer, L. Wehmeyer and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proceedings of "International Workshop on Power And Timing Modeling, Optimization and Simulation" (PATMOS)*, Yverdon-Les-Bains, 2001.

[28] S. Steinke, L. Wehmeyer et al. The *encc* energy aware C compiler homepage. *http://ls12-www.cs.uni-dortmund.de/research/encc/*, 2002.

[29] P. Strobach. A new technique in scene adaptive coding. In *Proceedings of "European Signal Processing Conference" (EUSIPCO)*, Grenoble, Sept. 1988.

[30] *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments Inc., Literature Number SPRU189D, Mar. 1999.

[31] M. Theokharidis. Energiemessung von ARM7TDMI Prozessor-Instruktionen. Master's thesis, University of Dortmund, Dortmund, Nov. 2000.

[32] *UltraSPARC III Cu User's Manual*. Sun Microsystems Inc., Palo Alto, May 2002.

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

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

[35] S. Wuytack, F. Catthoor, L. Nachtergaele et al. Power Exploration for Data Dominated Video Applications. In *Proceedings of "International Symposium on Low Power Electronics and Design" (ISLPED)*, Monterey, Aug. 1996.