Loop Nest Splitting for WCET-Optimization and Predictability Improvement *

Heiko Falk Martin Schwarzer

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

Abstract

This paper presents the influence of the loop nest splitting source code optimization on the worst-case execution time (WCET). Loop nest splitting minimizes the number of executed if-statements in loop nests of embedded multimedia applications. It identifies iterations of a loop nest where all if-statements are satisfied and splits the loop nest such that if-statements are not executed at all for large parts of the loop nest's iteration space.

Especially loops and if-statements of high-level languages are an inherent source of unpredictability and loss of precision for WCET analysis. This is caused by the fact that it is difficult to obtain safe and tight worst-case estimates of an application's flow of control through these high-level constructs. In addition, the corresponding control flow redirections expressed at the assembly level reduce predictability even more due to the complex pipeline and branch prediction behavior of modern embedded processors.

The analysis techniques for loop nest splitting are based on precise mathematical models combined with genetic algorithms. On the one hand, these techniques achieve a significantly more homogeneous structure of the control flow. On the other hand, the precision of our analyses leads to the generation of very accurate high-level flow facts for loops and if-statements. The application of our implemented algorithms to three real-life multimedia benchmarks leads to average speed-ups by 25.0% – 30.1%, while WCET is reduced between 34.0% and 36.3%.

1. Introduction

In contrast to general-purpose systems, embedded systems often have to meet real-time constraints. The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced. Besides the criticality of safeness of real-time systems, the market demands high performance, energy efficient and low cost products. Without knowledge about the worst-case timing of a real-time application, the designer tends to use oversized hardware in order to guarantee the safeness of the real-time system.

In recent years, the real-time behavior of embedded multimedia applications (e. g. medical image processing, video compression) with simultaneous consideration of power efficiency 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. 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

*This work is partially funded by the European IST FP6 Network of Excellence ARTIST2.

```
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)

is implemented using *if*-statements in the innermost loop (see e. g. Figure 1, an MPEG 4 full search motion estimation kernel [9]).

This code fragment has several properties making it suboptimal w. r. t. worst- and average-case execution time *(ACET)*. 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 [13]. This means that the pipeline needs to be stalled for some instruction cycles, so as to prevent the execution of incorrectly prefetched instructions. WCET analysis is faced with the problem to estimate whether a jump is taken or not. The worst-case influence of this decision on pipeline and branch prediction behavior needs to be taken into account. Since it is very difficult to predict these control flow modifications accurately, resulting WCETs tend to become imprecise the more irregular the control flow is.

In addition, the way how conditions are expressed within *if*-statements may also have a negative impact on WCET. If conditions are connected using the logical and / or operators of ANSI-C [10], they are evaluated lazily. For example, expression `e2` of the condition `e1 || e2` is not evaluated if `e1` already evaluates to true. Hence, each occurrence of the `||` and `&&` operators implies hidden control flow modifications having a negative influence on WCET. This source of unpredictability caused by the *if*-statements becomes even more severe if the *if*-statements are located in the heart of a loop nest as depicted in Figure 1. Here, WCET analysis has to multiply the overestimated data computed for the *if*-statements with the possibly also overestimated number of loop iterations, leading to even more imprecise estimates.

Considering the example shown in Figure 1, loop nest splitting is able to detect that

- the conditions `x3<0` and `y3<0` are never true,
- both *if*-statements are true for `x` ≥ 10 or `y` ≥ 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 sources of unpredictability during WCET analysis of a program.

```
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

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.

As shown by this example, our technique is able to generate a very homogeneous control flow in the hot-spots of an application. Furthermore, the algorithms briefly summarized in this paper enable the generation of precise high-level flow facts for WCET analysis. This paper evaluates the effect of loop nest splitting on the WCET of selected real-life benchmarks. Loop nest splitting is done by automatically transforming ANSI-C source codes. These source codes are then compiled for the ARM7 processor. WCET analysis for the resulting executable programs is finally done using the aiT WCET analyzer.

Section 2 of this paper gives a survey of related work. Section 3 presents the analyses and optimizations of of 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, 13]) 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. The impact of this optimization on WCET has not yet been studied.

Loop unswitching is applied to loops containing loop-invariant *if*-statements [13]. 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 multimedia programs. The fact that only loop-invariant conditions are considered implies that no valuable flow facts for WCET analysis are generated during this optimization.

In [12], classical loop splitting is applied together with function call insertion at the source code level to improve 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 imprecise, and the WCETs and ACETs of the benchmark are not considered at all.

This survey of related work shows that loop optimizations typically aim at improving temporal or spatial locality of caches and thus focus on ACET reduction. The influence of loop optimizations on WCET has not yet been studied thoroughly. Loop nest splitting was originally presented in [6, 8]. However, these original publications solely concentrated on the optimization of average-case execution time and energy dissipation. The impact of the optimization on WCET was not yet taken into account. Furthermore, all control-flow related data computed during the optimization process was discarded after loop nest splitting and was not used during subsequent optimization or analysis steps.

3. Analysis and Optimization Algorithm

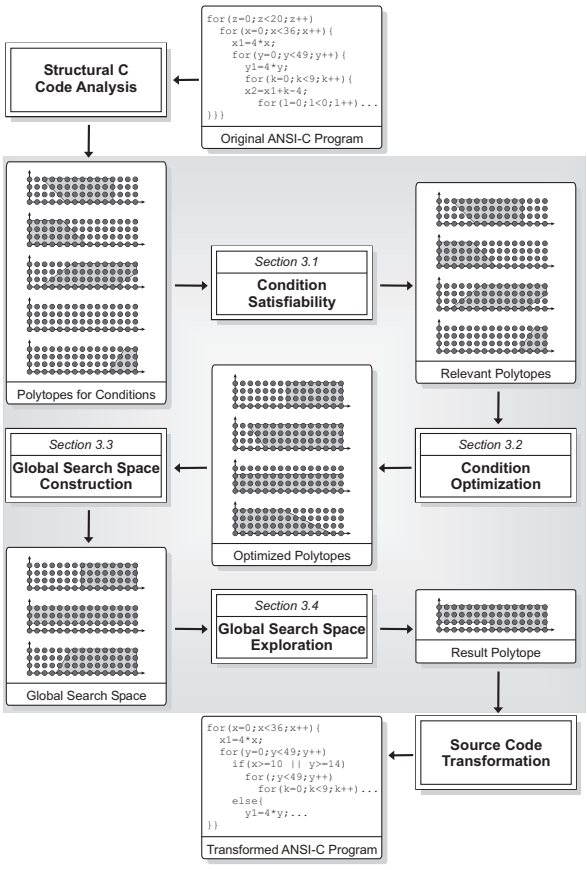

Figure 3. Design Flow of Loop Nest Splitting

Figure 3 gives an overview over the techniques required for loop nest splitting. As can be seen from this figure, loop nest splitting relies on polyhedral models in order to represent loop nests and *if*-statements. Polyhedra and

polytopes are defined as follows:

Definition:

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 $a, b \in \mathbb{Z}^m$ and $m \in \mathbb{N}$.
2. A polyhedron P is called a *polytope*, if $|P| < \infty$.

Since polyhedra are systems of linear inequations, loop nest splitting requires loop bounds and conditions of *if*-statements to be affine expressions depending on the loops' index variables. For a given loop nest $\Lambda = \{L_1, \ldots, L_N\}$ where each loop L_l is characterized by its index variable i_l and lower / upper bounds lb_l and ub_l, loop nest splitting computes values lb'_l and ub'_l for every loop $L_l \in \Lambda$ with

- $lb'_l \geq lb_l$ and $ub'_l \leq ub_l$, i. e. the computed values must lie within the loop bounds,
- 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 individual steps carried out during loop nest splitting as shown in Figure 3 are briefly described in Section 3.1 (cf. also [7] for a more in-depth description). Section 3.2 deals with the automatic generation of high-level flow facts during loop nest splitting.

3.1. Workflow of Loop Nest Splitting

Since the analyses of loop nest splitting require that the source code to be optimized meets some preconditions, these requirements are checked in the very beginning. During this phase labeled "Structural C Code Analysis" in Figure 3, only suitable loop nests and *if*-statements having affine bounds and conditions are extracted from the source code. The output of this phase consists of a set of polytopes, each of them representing a single condition occurring in the source code. The core optimization algorithm consists of four sequentially executed tasks that are illustrated as a shaded region in Figure 3. In the beginning, all conditions in a loop nest are analyzed separately without considering any inter-dependencies among them.

First, it is detected if conditions ever evaluate to true or not ("Condition Satisfiability"). For example, the two conditions `x3<0` and `y3<0` are eliminated from the code shown in Figure 1, since they are provably false during each loop nest iteration and are thus represented by empty polytopes.

Second, all satisfiable conditions are analyzed and an optimized search space for each condition is constructed ("Condition Optimization"). This means, that a polytope P representing an original condition C is replaced by an optimized polytope P' modeling a condition C' such that $C' \Rightarrow C$ holds. The goal is to generate P' in such a way that C' is significantly simpler than C. For example, condition optimization detects that $C' =$ `x >= 10` implies $C =$ `4*x + k + i >= 40` for the loops of Figure 1.

In a third step, all polytopes P' generated during condition optimization are combined to form a global search space G ("Global Search Space Construction"). This stage is motivated by the fact that the previous phases only considered single conditions of entire *if*-statements in isolation. In order to determine value ranges of the loop index variables, for which all *if*-statements in a loop nest are true, all P' need to be combined using intersection and union of polytopes according to the structure of all *if*-statements.

Finally, this global search space G has to be explored leading to the optimized result for loop nest splitting ("Global Search Space Exploration"). Basically, this phase selects a subset of constraints defining the global search space G in order to build a final polytope R representing the splitting *if*-statement. For the code shown in Figure 1, the outcome of the global search space exploration is the polytope $R = \{\mathtt{x} \geq 10\} \cup \{\mathtt{y} \geq 14\}$.

The result R of global search space exploration is finally used to rewrite a loop nest ("Source Code Transformation"). For this purpose, the splitting *if*-statement has to be generated and inserted in the loop nest. Its *then*- and *else*-parts are created by replicating parts of the original loop nest.

3.2. Flow Fact Generation

Since the major part of the execution time of a program is spent in loops, the iteration counts play an important role for WCET estimation. Hence, it is crucial to pass precise information about the number of loop iterations to the WCET analyzer in order to obtain safe and accurate WCET bounds. As already stated in the previous section, polytopes are used to model conditions and loops. Since polytopes are represented by linear inequations, the bounds of a loop L_l necessarily have to be affine expressions of the surrounding loops for loop nest splitting. An outermost loop L_1 is not surrounded by any other loop so that its bounds lb_1 and ub_1 are required to be constant. This way, it is ensured that the loop's iterations are allowed to be non-constant but still are fully analyzable at compile time.

During loop nest splitting, polytopes are generated modeling the loop currently under analysis. This is done straightforward by defining affine constraints for the lower and upper bounds of the loop itself and for all surrounding loops. In the resulting polytope, each integral point represents a single execution of the current loop body for one actual assignment of values to the loops' index variables. By counting the number of integral points of these polytopes, the total number of executions of the loop body can be determined exactly. For this purpose, so-called *Ehrhart polynomials* [4] are applied to the polytopes.

Example: *For the* `l`*-loop shown in Figure 1, a polytope having the constraints* `0 <= x < 36`, `0 <= y < 49`, `0 <= k < 9` *and* `0 <= l < 9` *would be generated. This polytope contains 142,884 points. Hence, the body of the* `l`*-loop is executed as many times within the entire loop nest.*

This number of integral points is used to generate flow facts for WCET analysis that exactly specify the number of executions of a loop body compared to the code lying outside the outermost loop L_1. In the case of the aiT WCET analyzer [1] used throughout this work, annotations like `flow 0x40007c / 0x40002e is exactly 142884;` are created, where the given addresses represent the basic blocks lying inside the current loop and outside the outermost loop, respectively.

In addition, more annotations concerning the splitting-if generated after global search space exploration can be provided to aiT. As already mentioned in Section 3.1, the final solution of loop nest splitting is a polytope R which is used to generate the splitting-if. The computation of the size of R using Ehrhart polynomials leads to the actual number of

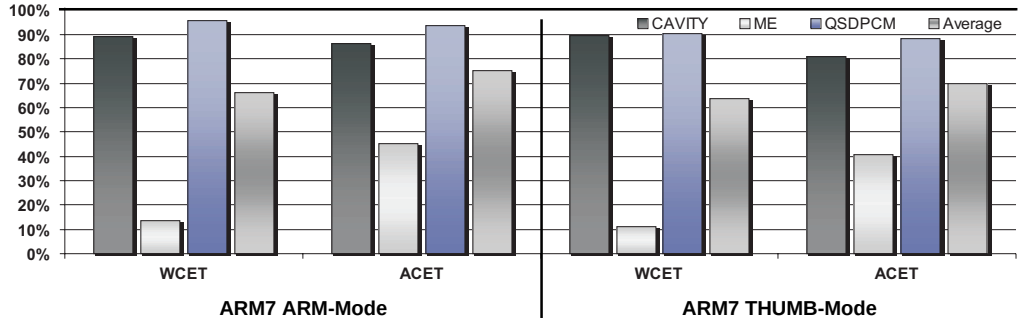

Figure 4. Relative WCETs and ACETs after Loop Nest Splitting

loop iterations for which the splitting-if provably is true. Since all loops are fully analyzable at compile time, the number of iterations for which the splitting-if is not true can also be computed. These two values are used to generate additional `flow` annotations for aiT precisely modeling the control flow structures resulting from loop nest splitting.

4. Evaluation

In this section, the impact of loop nest splitting on WCET is evaluated. For this purpose, the benchmarking workflow is presented in Section 4.1. Benchmarking results are given in Section 4.2.

4.1. Benchmarking Methodology

The techniques presented in Section 3 are fully implemented using the SUIF [16], Polylib [15] and PGAPack [11] libraries. Both GAs use the default parameters provided by [11] (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]) is used. The second benchmark is an MPEG 4 full search motion estimation (*ME* [9], see Section 1), and the QSDPCM algorithm [14] for scene adaptive coding serves as third benchmark.

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, only a maximum of 1.58 CPU seconds (CAVITY) are required for optimization on an AMD Athlon running at 1.3 GHz.

In order to quantify the influence of loop nest splitting on the WCET of the benchmarks, we considered an ARM7 based processor architecture. The ARM7 is a dual instruction set CPU having a 32-bit wide ARM instruction set and a 16-bit THUMB instruction set. For both instruction sets, the native ARM compilers *armcc* and *tcc* were used to generate executable code from the benchmark's source codes. Both compilers are always invoked with all optimizations enabled so that highly optimized code is generated.

In a first step, the source codes were compiled for both instruction sets without loop nest splitting being applied. The resulting executables were passed to AbsInt's WCET analyzer aiT [1] for the ARM7 to obtain the WCETs before our optimization. In addition to the binary executable, a specification file containing the exact number of loop iterations is also provided to aiT. In parallel, the same executables were processed by the cycle-true native ARM simulator. These simulations used typical input data for the benchmarks and the resulting cycle counts are considered as the ACETs of the benchmarks in the following.

In a second step, the source codes were optimized using our tool for loop nest splitting. The resulting optimized source codes were processed in the same way as described in the previous paragraph, leading to the corresponding WCETs and ACETs after loop nest splitting. For WCET analysis of the optimized codes, a specification file containing the `flow` annotations for loop bodies and the splitting-if (cf. Section 3.2) is also provided.

4.2. Benchmarking Results

WCET and ACET

Figure 4 shows the effects of loop nest splitting on the WCET and ACET of the benchmarks for the ARM7 using both the ARM and the THUMB instruction sets. The figure shows the corresponding values for the optimized benchmarks as a percentage of the unoptimized versions denoted as 100%.

As can be seen from this figure, loop nest splitting is able to reduce both ACET and WCET significantly. Concerning ACET, improvements between 6.4% (QSDPCM) and 54.8% (ME) were measured for the ARM instruction set. Similarly, ACET is reduced between 11.5% (QSDPCM) and up to 59.4% (ME) using THUMB instructions. On average for all considered benchmarks, ACET is reduced between 25.0% (ARM) and 30.1% (THUMB). These numbers clearly demonstrate that the generation of a homogeneous control flow within loop nests leads to increased average performance since a large amount of code located in the innermost loops before our optimization is eliminated.

However, Figure 4 also shows that the WCET reductions achieved after loop nest splitting have a similar order of magnitude. Here, the gains reach from 4.4% (QSDPCM) up to 86.5% (ME) when using 32-bit wide instructions. For the 16-bit THUMB instruction set, reductions of WCET between 9.6% (QSDPCM) and even 89.0% (ME) were reported by aiT. On average over all benchmarks, the reductions of WCET achieved by loop nest splitting are significantly larger than the corresponding ACET reductions. In terms of WCET, average improvements of 34.0% (ARM) and 36.3% (THUMB) can be reported.

Despite the fact that WCET is reduced more than ACET, Figure 4 does not show an invalid WCET underestimation. This is due to the fact that Figure 4 presents all results just as a percentage of the WCETs and ACETs of the unoptimized benchmarks. This way, it is legal to reduce WCET by 89% and ACET by just 59%. For all results presented in this paper, the estimated absolute WCETs are correct and safe and are larger than the corresponding absolute ACETs.

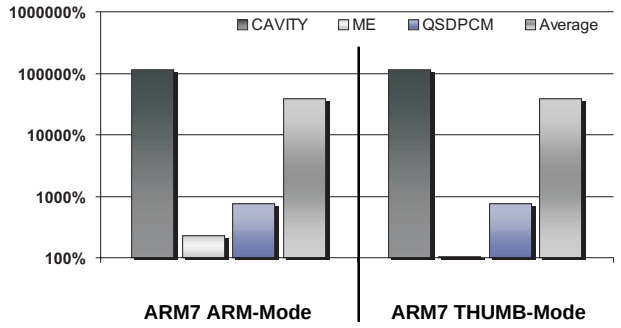

Figure 5. Relative WCETs after Loop Nest Splitting without Flow Annotations

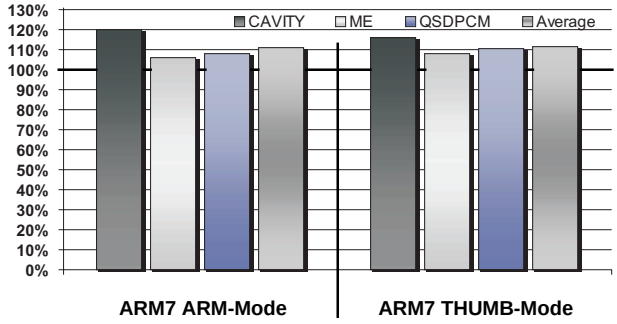

Figure 6. Relative Code Sizes after Loop Nest Splitting

Of course, these differences in the average WCET and ACET values are caused by the enormous improvements of WCET for the ME benchmark. For the two other benchmarks (CAVITY and QSDPCM), the WCET reduction scales with the corresponding ACET improvements. This behavior shows that the achievable gains in terms of WCET also depend on the overall structure of the unoptimized benchmark's code.

For example, both ME and QSDPCM have a similar structure like that shown in Figure 1. The difference between both benchmarks leading to the varying WCET reductions is the structure of the code blocks executed conditionally by virtue of the *if*-statements. For ME, `then_block_1` and `then_block_2` just contain the assignment of a constant to a variable, whereas both `else`-blocks contain very complex array accesses and address computations. Since these address computations invoke integer divisions and modulo computations, this code leads to the generation of calls to runtime libraries. For QSDPCM, the situation is vice versa – here, the *then*-blocks are more complex than the *else*-blocks.

This slight difference has the effect that for the unoptimized ME, the WCET path passes through the *else*-parts of the *if*-statements, whereas it lies on the *then*-parts for QSDPCM. After loop nest splitting, the new WCET path traverses the *then*-part of the splitting-if for both benchmarks. For ME, the innermost loop of this *then*-part now just contains the assignments of constants. Thus, the new WCET path of ME after loop nest splitting does no longer contain the costly address computations mentioned above, leading to the very high gains reported in this section. In the case of QSDPCM, the innermost loop of the *then*-part of the splitting-if still contains the complex address computations after loop nest splitting. As a consequence, this complex code still lies on the WCET path so that the gains in terms of WCET are not as high as compared to ME.

Impact of Flow Facts on WCET

The benefits of the flow facts extracted during loop nest splitting for WCET analysis are depicted in Figure 5. This diagram shows the WCETs resulting from the analysis of the benchmarks after loop nest splitting, but without providing aiT with the flow annotations precisely describing the splitting-if (cf. Section 3.2). Results are presented in a relative way such that the 100% baseline represents the WCETs before loop nest splitting.

As can be seen, the flow facts computed during global search space exploration are essential for a successful WCET minimization after loop nest splitting. Without this information, aiT is unable to compute precise WCETs from the optimized control flow structures. For all benchmarks, the WCETs without flow facts are worse than WCETs before any optimization being applied. For the ME benchmark, the degradations of WCET range between 4% (THUMB) and 231% (ARM). For QSDPCM, WCET results without flow facts are even worse. Here, increases between 743% and 767% compared to the WCETs of the unoptimized benchmarks were measured. The highest WCETs were computed for CAVITY. For this benchmark, WCETs increase between 113,031% (ARM) and 113,953% (THUMB) if aiT is not provided with the flow facts generated by loop nest splitting.

Code Size

Since code is replicated, loop nest splitting obviously entails a certain increase in code size that we do not want to neglect. However, Figure 6 shows that these increases are within small bounds. In order to measure code sizes, the size of the text sections in bytes was extracted from the ELF binaries of the benchmarks before and after loop nest splitting. For CAVITY, code size increases range between 19.9% (ARM) and 15.9% (THUMB). Although the ME benchmark is accelerated most, its code enlarges least. Increases of just 5.8% (ARM) and 8.1% (THUMB) were measured. Finally, the code of QSDPCM enlarges between 7.9% (ARM) and 10.5% (THUMB). On average over all benchmarks, code size increases of just 11.2% (ARM) and 11.5% (THUMB) were measured.

For fine tuned embedded systems with hard constraints on both worst-case execution time and code size, code size increases might potentially be a severe drawback. However, loop nest splitting offers inherent opportunities for solving this problem since it is perfectly suited for trading off WCET with code size increases.

As depicted in Figure 2, loop nest splitting generates a splitting-if like `if (x >= 10 || y >= 14)` and places it in the `y`-loop, since this is the innermost loop of the entire loop nest the splitting-if directly depends on. Within the splitting-if, the remaining loop nest consisting of the `k`-, `l`-, `i`- and `j`-loop can be found. Since the splitting-if does not depend on index variables of this remaining loop nest by definition, it is always legal to place the splitting-if in any of these loops. This way, the portions of code replicated during loop nest splitting become smaller on the one hand.

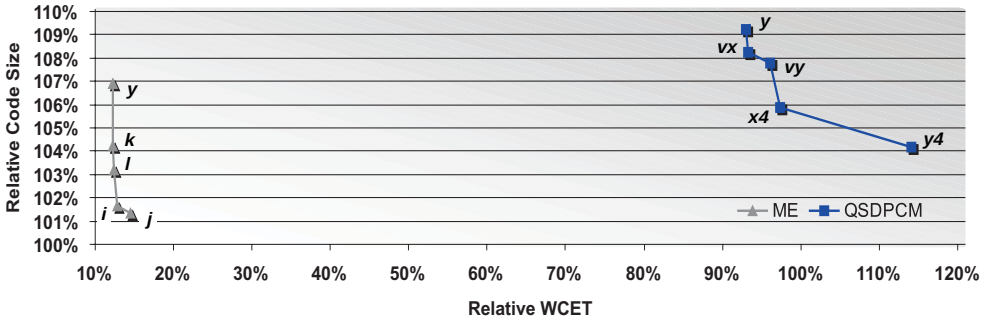

Figure 7. Possible WCET / Code Size Trade-Offs for Loop Nest Splitting

On the other hand, it can be expected that less improvements of WCET will be achieved since more *if*-statements are executed, leading to the mentioned trade-off.

Figure 7 shows the corresponding Pareto curves for the ME and QSDPCM benchmarks. The x-axis denotes the relative WCETs of the benchmarks, whereas the y-axis shows the corresponding relative code sizes (100% = unoptimized code version). Each point is labeled with the loop in which the splitting-if is placed. The code versions used to generate all previously presented results are marked with `y` (ME and QSDPCM). As expected, they lead to the lowest WCETs and entail the highest code size increases. In contrast, code versions `j` (ME) and `y4` (QSDPCM) are the slowest but smallest ones. In between these two extremal points, other interesting solutions for loop nest splitting can be found.

These experiments show that it is worthwhile to study possible trade-offs when applying loop nest splitting under tight code size constraints. A more systematic study than that presented here resulting in an automated approach to explore WCET / size trade-offs is part of the future work.

5. Conclusions

This paper puts the previously presented source code optimization loop nest splitting in the context of WCET. Loop nest splitting 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.

It has turned out that loop nest splitting is highly beneficial for WCET optimization. This is due to the fact that the quality of WCET analysis inherently depends on a precise description of the control flow of an application under analysis. On the one hand, precise high-level flow facts representing e. g. loop iterations have to be provided. On the other hand, assembly-level jumps modifying the control flow are hard to analyze since the conditions under which a jump is taken or not are difficult to analyze resulting in imprecise worst-case assumptions.

The benefits of loop nest splitting on WCET are twofold. First, the optimization by itself produces a very linear and homogeneous control flow in the hot-spots of an application. As a consequence, the potential for applying the imprecise worst-case assumptions mentioned above during WCET analysis of the time-critical parts of a code is heavily reduced. Second, loop nest splitting inherently computes execution frequencies of all relevant control flow constructs during its analyses. These execution frequencies can directly be used to formulate precise loop and flow annotations for the WCET analyzer.

The results presented in this paper underline the effectiveness of loop nest splitting. In terms of average-case execution times, it achieves improvements between 25.0% – 30.1%. However, even larger average gains are reported in terms of WCET. Here, reductions between 34.0% and 36.3% were measured for an ARM7 based processor.

In the future, we intend to integrate loop nest splitting into our WCET-aware C compiler [5]. Due to its multi-objective capabilities, it is perfectly suited to systematically explore the WCET / size trade-offs of loop nest splitting.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework.

References

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers. *http://www.absint.com/ait*, 2005.

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

[5] H. Falk and P. Lokuciejewski. Design of a WCET-Aware C Compiler. In *Proc. of "6th Intl. Workshop on WCET Analysis" (WCET)*, Dresden, July 2006.

[6] H. Falk and P. Marwedel. Control Flow driven Splitting of Loop Nests at the Source Code Level. In *Proc. of DATE*, Munich, Mar. 2003.

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

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

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

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

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

[12] 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.

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

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