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 effect of the loop nest splitting source code optimization on worst-case execution time (WCET). Loop nest splitting minimizes the number of executed if-statements in loop nests of multimedia applications. It identifies iterations 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 are an inherent source of unpredictability and loss of precision for WCET analysis. This is caused by the difficulty to obtain safe and tight worst-case estimates of an application's high-level control flow. In addition, assembly-level control flow redirections reduce predictability even more due to complex processor pipelines and branch prediction units.

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

1. Introduction

Embedded systems often have to meet real-time constraints. The correctness of such a system depends not only on the result of the computation, but also on the time at which the results are produced. Besides safe real-time systems, the market demands high performance, energy efficient and low cost products. Without knowledge about the worst-case timing, designers tends to oversize hardware to guarantee the safeness of the real-time system.

In the past, the real-time behavior of embedded multimedia applications 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, they 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 is implemented using *if*-statements in the innermost loop (cf. Figure 1, an MPEG 4 full search motion estimation [8]).

```
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)

This code has several properties making it sub-optimal 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 [11]. 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 needs to estimate whether a jump is taken or not. The worst-case influence of this decision on pipeline and branch prediction needs to be taken into account. Since it is 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 of an *if*-statement are expressed also has a negative impact on WCET. If conditions are connected using the logical and / or operators of ANSI-C [9], 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 `||` and `&&` implies hidden control flow modifications with a negative influence on WCET. This unpredictability due to *if*-statements becomes even more severe if the *if*-statements are located in deeply nested loops (cf. Figure 1). Here, WCET analysis multiplies the overestimated data of 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

*Partially funded by the European IST FP6 NoE ARTIST2.

```
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

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 improving predictability during WCET analysis of a program.

Using the second information, the loop nest can be rewritten to minimize the total number of executed *if*-statements (cf. Figure 2). Here, a new *if*-statement (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 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.

As shown by this example, our technique generates a very homogeneous control flow in the hot-spots of an application. The algorithms briefly outlined 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 optimizing ANSI-C source codes. These source codes are then compiled for the ARM7 processor. WCET analysis of the resulting executables 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 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 [2, 11] and are often integrated into today's compilers. Classical *loop splitting* creates several loops out of an original one and distributes the statements of the original loop body among all new loops. This optimization enables the parallelization of a loop due to fewer data dependencies [2] and possibly improves 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 [11]. The loop is then replicated inside each branch of the *if*-statement, reducing the branching overhead and decreasing code sizes of the loops [2]. Goals and code transformation of loop unswitching are equivalent to the topics of Section 1. But since *if*-statements must not depend on index variables, loop unswitching is 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 [10], 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 effect of loop optimizations on WCET has not yet been studied thoroughly. Loop nest splitting was originally presented in [5, 7]. These original publications solely considered the optimization of ACET and energy dissipation. The impact of the optimization on WCET was not taken into account. Furthermore, all control-flow related data computed during loop nest splitting was discarded and was not used during subsequent optimization or analysis steps.

3. Analysis and Optimization Algorithm

Figure 3 depicts the workflow of loop nest splitting. As can be seen, the optimization relies on polyhedral models representing 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 sets of linear inequations, loop nest splitting requires loop bounds and conditions of *if*-statements to be affine expressions of the loops' index variables. For a given loop nest $\Lambda = \{L_1, \ldots, L_N\}$ where each loop L_l is represented 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 $lb'_l \leq i_l \leq ub'_l$,

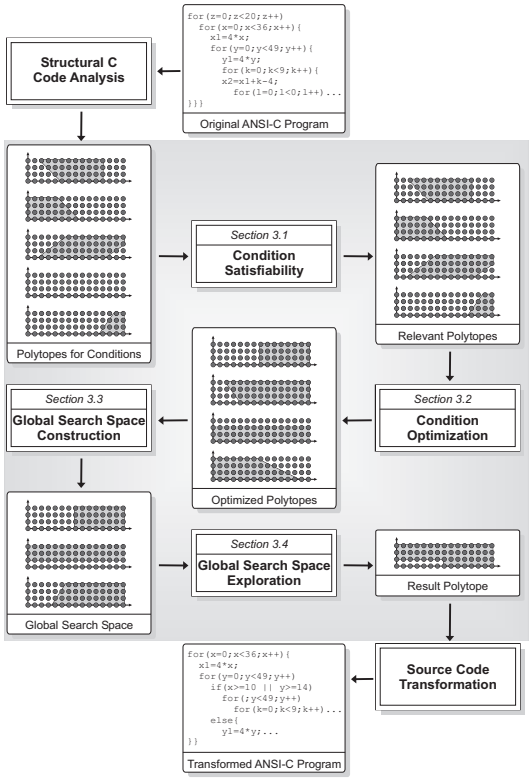

Figure 3. Design Flow of Loop Nest Splitting

- loop nest splitting by all values lb'_l and ub'_l leads to the minimization of *if*-statement execution.

lb'_l and ub'_l are used to construct the splitting *if*-statement. The substeps of loop nest splitting as shown in Figure 3 are briefly outlined in Section 3.1 (cf. [6] for a more in-depth discussion). Section 3.2 deals with the automatic generation of high-level flow facts during loop nest splitting.

3.1. Workflow of Loop Nest Splitting

During an initial phase ("Structural C Code Analysis" in Figure 3), only suitable loop nests and *if*-statements with affine bounds and conditions are extracted from the source code. The output is a set of polytopes, each of them representing a single condition within the source code. The core optimization algorithm (the shaded region in Figure 3) consists of four sequentially executed tasks. In the beginning, all conditions in a loop nest are analyzed separately without considering any inter-dependencies among them.

First, it is checked if conditions are ever true or not ("Condition Satisfiability"), e. g. conditions `x3<0` and `y3<0` are removed from Figure 1, since they are provably false in each iteration and are thus modeled by empty polytopes.

Second, an optimized search space is built for all satisfiable conditions ("Condition Optimization"). Here, a polytope P modeling an original condition C is replaced by a polytope P' modeling a condition C' such that $C' \Rightarrow C$ holds. P' is generated such that C' is significantly simpler than C. For example, it is detected that $C' =$ `x >= 10` implies $C =$ `4*x + k + i >= 40` for Figure 1.

Third, all polytopes P' obtained by condition optimization are combined to form a global search space G ("Global Search Space Construction"). The previous phases only considered single conditions of entire *if*-statements in isolation. 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, the global search space G is explored leading to the final result of loop nest splitting ("Global Search Space Exploration"). This phase selects a subset of constraints of G to build a final polytope R representing the splitting *if*-statement. For the code of Figure 1, the outcome of this phase is the polytope $R = \{\mathtt{x} \geq 10\} \cup \{\mathtt{y} \geq 14\}$.

The final polytope R is then used to rewrite a loop nest ("Source Code Transformation"). The splitting-if is generated and placed in the loop nest. Its *then*- and *else*-parts are created by copying parts of the original loop nest.

3.2. Flow Fact Generation

Since a program spends most of its time in loops, iteration counts are crucial for WCET estimation. The WCET analyzer requires precise loop iteration numbers to compute safe and tight WCET bounds. As already stated in Section 3.1, polytopes model conditions and loops. Since polytopes base on linear inequations, the bounds of a loop L_l must be affine expressions of surrounding loops for loop nest splitting. The bounds lb_1 and ub_1 of an outermost loop L_1 have to be constant. This way, all loop's iterations can be non-constant but still are fully analyzable at compile time.

During loop nest splitting, polytopes model the loop currently under analysis by defining affine constraints for the loops' lower and upper bounds and for all surrounding loops. In such a polytope, each integral point represents one execution of the loop body for one actual assignment of values to the loops' index variables. Counting the number of integral points of these polytopes leads to the exact number of executions of the loop body. For this purpose, so-called *Ehrhart polynomials* [3] are applied to the polytopes.

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

The number of points is used in flow facts for WCET analysis that exactly specify the executions of a loop compared to code lying outside the outermost loop L_1. For the aiT WCET analyzer [1] used here, annotations like `flow 0x40007c / 0x40002e is exactly 142884;` are created, where the given addresses represent basic blocks inside the current loop and outside the outermost loop, resp.

In addition, annotations for the splitting-if created by

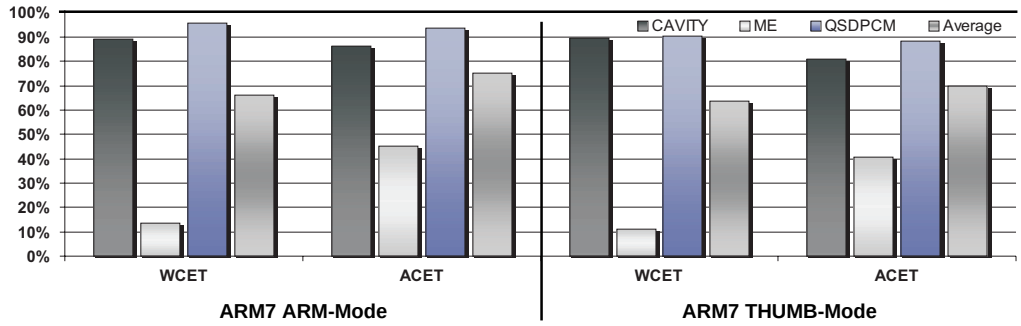

Figure 4. Relative WCETs and ACETs after Loop Nest Splitting

loop nest splitting are provided to aiT. The final solution of loop nest splitting (cf. Section 3.1) is a polytope R that is used to generate the splitting-if. Computing the size of R using Ehrhart polynomials leads to the number of iterations with a provably true splitting-if. Analogously, the number of iterations with a false splitting-if is also computed. These values precisely model the control flow after loop nest splitting and are passed to aiT as additional `flow` annotations.

4. Evaluation

This section evaluates the impact of loop nest splitting on WCET. First, 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 SUIF [13] and Polylib [12]. Our tool was applied to three multimedia programs: a medical tomography image processor (*CAVITY*), an MPEG 4 full search motion estimation (*ME*, cf. Section 1) and the QSDPCM algorithm for scene adaptive coding. Since all polyhedral operations have exponential worst case complexity [12], loop nest splitting also has exponential complexity. Nevertheless, the effective runtimes of our tool are very low (max. 1.58 CPU seconds for CAVITY on an 1.3 GHz AMD Athlon).

The effect of loop nest splitting on the benchmark's WCET was quantified for an ARM7 processor, having a 32-bit wide ARM instruction set and a 16-bit THUMB ISA. For both instruction sets, the native ARM compilers *armcc* and *tcc* were used to generate executables from the benchmark's source codes. Compilers are always invoked with all optimizations enabled to generate highly optimized code.

First, all source codes were compiled for both ISAs without loop nest splitting applied. The resulting binaries were analyzed with aiT [1] to obtain WCETs before our optimization. In addition to the binaries, specifications of the exact number of loop iterations are also provided to aiT. In parallel, the same executables were processed by the cycle-true native ARM simulator. These simulations used typical input data, and the resulting cycle counts are considered as the ACETs of the benchmarks in the following.

Second, loop nest splitting was applied to the benchmark's source codes. The resulting optimized source codes were processed the same way as described previously, leading to the corresponding WCETs and ACETs after loop nest splitting. For WCET analysis of the optimized codes, specifications containing the `flow` annotations for loop bodies and the splitting-if (cf. Section 3.2) are also provided.

4.2. Benchmarking Results

WCET and ACET

Figure 4 shows the effect of loop nest splitting on the benchmark's WCETs and ACETs for both the ARM and THUMB ISA. It shows the values for the optimized benchmarks as a percentage of the unoptimized versions denoted as 100%.

Loop nest splitting reduces both ACET and WCET significantly. Concerning ACET, improvements of 6.4% (QSDPCM) – 54.8% (ME) were measured for the ARM ISA. Similarly, ACET is reduced by 11.5% (QSDPCM) – 59.4% (ME) for the THUMB ISA. On average for all benchmarks, ACET is reduced by 25.0% (ARM) – 30.1% (THUMB). This clearly shows that generating homogeneous control flow in loop nests increases average performance since a large amount of code found in innermost loops before our optimization is eliminated.

Figure 4 also shows that loop nest splitting reduces WCETs by a similar order of magnitude. Here, gains reach from 4.4% (QSDPCM) – 86.5% (ME) for the ARM ISA. For the THUMB ISA, WCET reductions by 9.6% (QSDPCM) – 89.0% (ME) were reported by aiT. On average over all benchmarks, the WCET reductions after loop nest splitting are significantly larger than the corresponding ACET reductions. In terms of WCET, average improvements by 34.0% (ARM) – 36.3% (THUMB) were measured.

Even though WCET is reduced more than ACET, Figure 4 does not imply invalid WCET underestimations. This is due to the presentation of WCETs and ACETs as a percentage of the unoptimized values in Figure 4. Thus, it is legal to reduce WCET by 89% and ACET by just 59%.

These differences in the average WCETs and ACETs are due to the huge improvements of WCET for the ME bench-

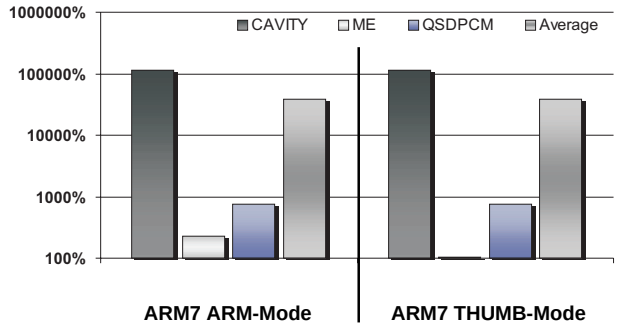

Figure 5. Relative WCETs after Loop Nest Splitting without Flow Annotations

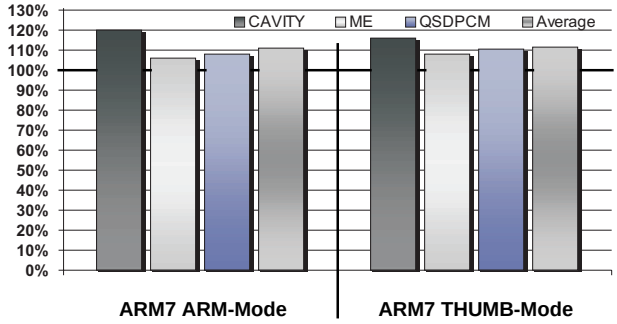

Figure 6. Relative Code Sizes after Loop Nest Splitting

mark. For CAVITY and QSDPCM, WCET reductions scale with the corresponding ACET improvements. This shows that the achievable gains in terms of WCET also depend on the overall structure of the unoptimized benchmark's code.

E. g., ME and QSDPCM have similar structures (cf. Figure 1). The varying WCET reductions of both benchmarks are caused by the structure of the *then*- and *else*-blocks of the *if*-statements. For ME, both *then*-blocks just contain assignments of constants to a variable, whereas both *else*-blocks contain complex array accesses and address computations. Since these address computations use division and modulo operators, calls to runtime libraries are generated. For QSDPCM, the situation is vice versa – the *then*-blocks are more complex than the *else*-blocks.

This difference makes the WCET path contain both *else*-parts for the unoptimized ME, whereas it contains the *then*-parts for QSDPCM. After loop nest splitting, the new WCET path contains the *then*-part of the splitting-if for both benchmarks. For ME, this *then*-part now just contains the assignments of constants. Thus, the optimized WCET path of ME no longer contains the costly address computations, leading to the very high gains reported here. For QSDPCM, the *then*-part of the splitting-if still contains the complex address computations after optimization. Hence, this complex code still lies on the WCET path resulting in not as high WCET reductions as compared to ME.

Impact of Flow Facts on WCET

Figure 5 depicts the benefits of the flow facts gathered during loop nest splitting. It shows the results of WCET analysis of the benchmarks after loop nest splitting, but without precisely annotating the splitting-if for aiT (cf. Section 3.2). Results are presented in a relative way such that the 100% baseline represents the WCETs before loop nest splitting.

Obviously, the flow facts computed during loop nest splitting are essential for effective WCET reduction. Without them, aiT can not compute tight WCETs for the optimized control flows. For all benchmarks, WCETs without flow facts are worse than WCETs before the optimization. For ME, WCET degrades between 4% (THUMB) – 231% (ARM). For QSDPCM, WCETs without flow facts are even worse: they increase between 743% – 767% compared to the unoptimized WCETs. For CAVITY, WCETs increase between 113,031% (ARM) – 113,953% (THUMB) if aiT does not have any knowledge on the splitting-if.

Code Size

Since code is replicated, loop nest splitting obviously increases code sizes. However, Figure 6 shows that these increases are limited. To measure code sizes, the size of the text sections in bytes was extracted from the benchmark's ELF binaries before and after loop nest splitting. For CAVITY, code sizes increase by 19.9% (ARM) – 15.9% (THUMB). Although ME is accelerated most, its code enlarges least by just 5.8% (ARM) – 8.1% (THUMB). Finally, QSDPCM enlarges between 7.9% (ARM) – 10.5% (THUMB). On average over all benchmarks, code size increase by just 11.2% (ARM) – 11.5% (THUMB).

For fine tuned embedded systems with hard constraints on both WCET and code size, such increases might 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.

Loop nest splitting places a splitting-if like `if (x >= 10 || y >= 14)` in the `y`-loop (cf. Figure 2), since this is the innermost loop the splitting-if directly depends on. Copies of the remaining `k`-, `l`-, `i`- and `j`-loops are placed within the splitting-if. Since the splitting-if does not depend on these remaining index variables by definition, it is always legal to place the splitting-if in any of these loops. This way, portions of code replicated by loop nest splitting are reduced on the one hand. On the other hand, smaller WCET reductions will be achieved since more *if*-statements are executed, leading to the mentioned trade-off.

Figure 7 shows the corresponding Pareto curves for ME and QSDPCM. The x-axis shows the benchmark's relative WCETs, the y-axis shows the corresponding relative code sizes (100% = unoptimized codes). Each point is labeled with the loop now holding the splitting-if. The code versions used to generate all previously presented results are

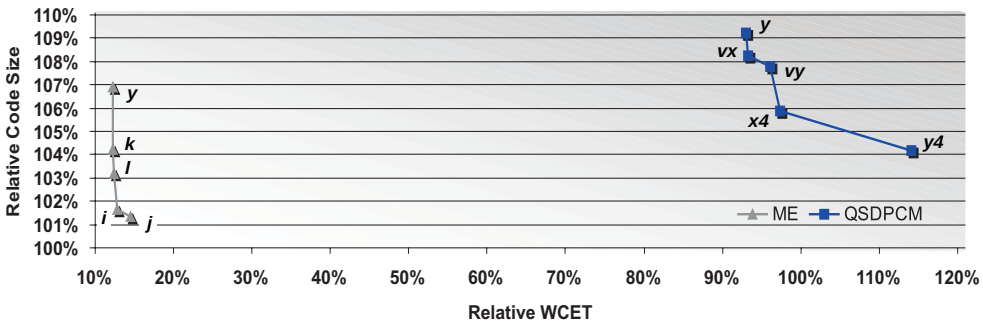

Figure 7. Possible WCET / Code Size Trade-Offs for Loop Nest Splitting

marked with y (ME and QSDPCM). As expected, they lead to the lowest WCETs and largest code sizes. 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 loop nest splitting optimization in the context of WCET. The optimization removes control flow redundancies in multimedia applications. Using polytopes, conditions without effect on the control flow are removed. Genetic algorithms identify iteration space ranges where all *if*-statements are provably satisfied. The source code of an application is rewritten such 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 [4]. 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] P. Clauss and V. Loechner. Parametric Analysis of polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, 19(2), July 1998.

[4] 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.

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

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

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

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

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

[10] 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.

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

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

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