Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization *

Paul Lokuciejewski, Peter Marwedel
Computer Science 12
TU Dortmund University
D-44221 Dortmund, Germany
FirstName.LastName@tu-dortmund.de

Abstract

Program loops are notorious for their optimization potential on modern high-performance architectures. Compilers aim at their aggressive transformation to achieve large improvements of the program performance. In particular, the optimization loop unrolling has shown in the past decades to be highly effective achieving significant increases of the average-case performance.

In this paper, we present loop unrolling that is tailored towards real-time systems. Our novel optimization is driven by worst-case execution time (WCET) information to effectively minimize the program's worst-case behavior. To exploit maximal optimization potential, the determination of a suitable unrolling factor is based on precise loop iteration counts provided by a static loop analysis. In addition, our heuristics avoid adverse effects of unrolling which result from instruction cache overflows and the generation of additional spill code. Results on 45 real-life benchmarks demonstrate that aggressive loop unrolling can yield WCET reductions of up to 13.7% over simple, naive approaches employed by many production compilers.

1. Introduction

The optimization *loop unrolling* [22] has been thoroughly studied in the context of the average-case performance. Most important advantages of this optimization are the reduced loop increment-and-test overhead, increased instruction level parallelism, and, often most advantageous, the unrolled code establishes additional potential for other optimizations. However, loop unrolling is also entailed with negative side-effects. Examples are an adverse impact on the instruction cache (I-cache) and additional *spill code*. Thus, the optimization must be used with caution. Several studies, however, indicated that sophisticated heuristics for loop unrolling can significantly improve the average-case execution time (ACET) [24, 8, 13, 5]. The success of loop unrolling also highly depends on the knowledge of loop iteration counts and their lack diminishes optimization potential. Thus, an effective loop unroller must have access to this parameter [8].

In the last years, different objectives than the average-case performance moved into the focus of compiler optimizations. Timing constraints deserve special attention for embedded systems that often operate as real-time systems. Worst-case timing behavior is expressed by the worst-case execution time. Especially for safety-critical application domains the WCET bounds must be safe to guarantee that real-time constraints are met. Moreover, the precise knowledge of this key parameter is mandatory for various scheduling algorithms. With the growing complexity of embedded applications, an effective WCET reduction can be only achieved with aggressive compiler optimizations.

In this paper, we present a WCET-driven loop unrolling for real-time systems. Standard heuristics which have shown to be highly effective are integrated into our optimizer. Among others, the most important are related to an avoidance of I-cache overflows by means of a controlled code expansion and the estimation of spill code generation. In contrast to previous works, our novel optimization is completely driven by worst-case timing information to achieve a maximal WCET reduction. This makes our approach more reliable than previous ACET optimizations relying on profiling. Unlike WCET information, profiling data might change for different inputs, thus the optimization of a program based on this data might be suboptimal or even have an adverse influence on the performance when inputs change.

The main contributions of this paper are as follows:

*The research leading to these results has received funding from the European Community's Artist Design Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008.

1. To our best knowledge, this is the first study evaluating the impact of an aggressive loop unrolling on the program's worst-case performance.
2. Our WCET-driven loop unrolling exploits the trade-off between performance and code size, being typical for embedded systems, by optimizing loops that promise the maximal WCET reduction with a simultaneous small code expansion.
3. We exploit a static loop analysis providing the crucial knowledge of loop iteration counts.
4. Our developed prediction mechanism computes for each loop its unrolling profit that allows the determination of the most promising unrolling factor.
5. The transformation is applied at source code level to provide a large optimization potential for subsequent high- and low-level optimizations.

The rest of this paper is organized as follows: Section 2 gives a survey of related work. In Section 3, the standard optimization loop unrolling is discussed, followed by the description of our WCET-driven loop unrolling in Section 4. Section 5 presents our experimental environment, while results achieved on real-life benchmarks are discussed in Section 6. Finally, Section 7 summarizes this paper and gives directions for future work.

2. Related Work

Loop unrolling belongs to the class of compiler optimizations that has been extensively studied for ACET reduction. In [25], generalized loop-unrolling methods were presented that allow the transformation of even badly-structured loops. A combination of loop unrolling and automatic scheduling by the compiler was discussed by [15]. The positive effects of loop unrolling concerning an increased instruction-level parallelism were exploited for instruction scheduling in [20].

One of the central questions for loop unrolling is how to find an appropriate unrolling factor. It has been shown that the consideration of I-cache constraints and register pressure is significant for that computation. The problems of automatically selecting an appropriate unrolling factor was addressed in [24]. In addition to considering I-caches and register availability, the authors use a technique called *unroll-and-jam* to produce compact code for nested loops. [14] describes an unrolling heuristic taking information about data dependencies, reuse, and machine resources into account. Heydemann [13] presents an integer linear programming (ILP) based approach to compute the unrolling factor.

In addition, the selection of an appropriate unrolling factor essentially depends on the knowledge of loop iteration counts. In [8], a survey of different aspects of unrolling is given. One of the main conclusions is that missing loop bounds considerably decrease optimization potential. They conduct profiling to gather this crucial parameter. However, as described in Section 1, profiling information might not be reliable. Moreover, profiling is expensive and thus often not applicable. To overcome this dilemma, we apply a static loop analysis computing safe and input-invariant loop bounds that are valid for all input data.

The automatic compiler-based reduction of the WCET, which is a crucial objective in real-time systems, is still a novel research area. The only work that considers loop unrolling in the context of WCET minimization is presented in [30]. However, this work differs in several aspects from our approach. Most important, the authors do not develop a WCET-driven loop unrolling but apply standard ACET unrolling and study its impact on the WCET. Moreover, they do not exploit worst-case iteration counts of loops for an aggressive WCET optimization, but unroll each loop by a constant factor of two. Also, their unrolling is applied at the assembly level, and their target architecture has no caches making an evaluation of cache effects impossible.

Most published works in the domain of WCET minimization operate on assembly level and exploit memory hierarchies but not standard compiler optimizations as is done in this work. [4] presents an algorithm for static locking of I-caches based on a genetic algorithm. [29] combines compile-time cache analysis with static data cache locking. Iterative approaches for I-cache locking that repeatedly update the worst-case execution path (WCEP) information are studied in [11]. In [18], WCET-driven procedure positioning optimizations are presented.

Besides cache optimizations, fast scratchpad memories (SPM) are exploited for WCET minimization. A hybrid approach combining ILP and an iterative heuristic for WCET-centric dynamic SPM allocation is presented in [9]. In [26], a static allocation of data to SPM is based exclusively on an ILP model. A comparison between scratchpads and locked caches for real-time systems is presented in [23].

Contrary to these assembly optimizations, WCET-driven source code optimizations require an additional intermediate step making timing information, which are typically computed at assembly level, accessible at source code level. In literature, this step is called *Back-Annotation* and was utilized in [17] to conduct the optimization *procedure cloning* based on WCET data. The latter work, and loop unrolling which is presented in this paper, are integrated into a compiler using a detailed WCET timing model [10] provided by the static timing analyzer aiT [1].

3. Loop Unrolling

Loop unrolling is a code transformation that replicates the body of a loop a number of times and adjusts the loop-control accordingly. The number of replications is called the *unrolling factor* u and the original loop is often termed *rolled loop*. If the loop iteration count of a rolled loop does not correspond to an integral multiple of the unrolling factor

u, then either additional exit conditions must be added into the unrolled loop body or some *left-over* iterations must be separately handled by an additional loop (*remainder loop*).

Unrolling a loop positively affects the program performance for many reasons. The loop overhead comprising the increment-and-test instructions is reduced and the reduction of jumps might in particular have a positive influence on architectures with large control hazard penalties. Unrolling is also a fundamental code transformation for superscalar processors that can only exhibit a high performance when instruction level parallelism is exploited. In addition, other optimizations following unrolling might profit from the unrolled code, thus standard literature proposes to execute unrolling not *too* late in an optimization sequence [22]. Studies also showed that unrolled code might have an improved memory hierarchy locality [2].

Despite the large number of positive effects, it has been observed that loop unrolling can also have an adverse impact on the program's performance when the optimization is not applied elaborately. Since unrolling is a code-expanding transformation, an aggressive loop unrolling can overflow the I-cache leading to additional capacity cache misses that did not arise for the rolled loop [8]. An excessive loop unrolling can also lead to additional spill code when the register working set in the unrolled loop exceeds the number of available registers [6]. Remainder loops should be introduced with caution. They increase the code size but only a small fraction of the program execution is spent in this code [24]. It should be also noted that unrolling can increase the compilation time resulting from more code that has to be processed by subsequent optimizations.

Thus, sophisticated optimizing compilers should take I-cache constraints and the register pressure into account as well as avoid jumps in order not to cancel the optimization benefits when loops are unrolled.

4. WCET-driven Loop Unrolling

The central question for loop unrolling is which unrolling factor should be used for each loop. Its computation depends on several parameters:

1. **Loop iteration counts of each loop**
2. **I-cache and free program memory constraints**
3. **Approximation of spill code generation**

In the following sections, these parameters will be discussed and we show how we exploit them for our WCET-driven loop unrollig.

4.1. Worst-Case Loop Iteration Counts

The determination of the unrolling factor requires the knowledge about the loop iteration counts which should be known at compile time. This information can be provided either by profiling or by a static analysis. Profiling is the most common approach. However, the use of profiling data limits the scope of application since it is difficult to obtain reasonable profiles and the generation of this data might have high space and time requirements. A sophisticated static loop analysis is often not available in most compilers. Due to the lack of techniques to compute the iteration counts, most compilers use a small and constant unrolling factor (typically 2 or 4) [13, 30] which does not sufficiently exploit the optimization potential.

Some compilers provide a loop analysis that is only able to analyze simple, well-structured loops which are rarely found in real-life applications. Thus, many loops remain unoptimized. We combine our loop unrolling with a powerful static loop analyzer that is able to detect and analyze most loops found in today's embedded systems applications. To our best knowledge, this is the first study that evaluates the effectiveness of a sophisticated loop analysis in the context of loop unrolling.

Static Loop Analysis

Our static loop analysis [19] is based on Abstract Interpretation [7], a theory of a sound approximation of program semantics. The analysis is provided with the high-level intermediate representation (IR) of the program as input. Based on Abstract Interpretation, all potential values a program variable might have at any program point are considered. Since all possible input data is assumed, the analysis is not sensitive to a particular input set. This makes the analysis results suitable for a WCET minimization. In contrast, profiling information is not applicable since it reflects the program behavior of a particular input data and not the program's worst-case behavior.

Based on the analysis results, context-sensitive worst-case iteration counts for each loop are provided. Consider this code snippet:

```
void food( int n ) {
  for( int i = 0; i < n; ++i ) {
    // Loop body } }
int main( void ) {
  food( 4 );
  food( 10 );
  food( 16 ); }
```

As our review on embedded systems applications revealed, *variable* loop bounds, i. e. loops that have different iteration counts depending on their execution context, as can be seen in the code snippet, are common. However, many simple loop analyzers support only trivial counting loops and are not able to handle the class of loops from our example. Thus, optimization potential is not fully exploited.

Exploiting the results of our static loop analysis, we have complete information about the loop behavior in different contexts. Our loop unroller would choose 2 as unrolling factor to maximally unroll the loop while avoiding adverse

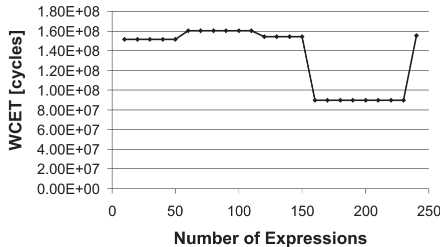

Figure 1. Impact of Unrolling

test conditions. A detailed description of the unrolling factor determination follows in Section 4.4.

For the sake of completeness, it should be noted that it is also possible to unroll loops when the loop counts are not known at compile time. This approach requires the insertion of additional code calculating the iteration counts during the program execution. However, as shown in [8], the cost of calculating the loop iteration counts does not always amortize leading to a performance degradation. Moreover, this approach prevents the determination of an unrolling factor that takes I-cache and spill code constraints into account.

4.2. I-Cache And Memory Constraints

In a second step, an appropriate unrolling factor *u* must be calculated based on the loop iteration count. A complete unrolling of the loop is often a bad choice due to several reasons. First, the size of the unrolled loop might exceed the available program memory. This is in particular crucial for embedded systems with restricted memory resources. Second, a too large unrolling factor might lead to I-cache overflows. In order to control such adverse effects, the optimization must be able to estimate the code size of the finally unrolled loop. This knowledge allows the calculation of an unrolling factor that restricts the loop size increase to a given I-cache capacity.

The determination of a precise loop size requires knowledge about the involved assembler instructions. On the one hand, when loop unrolling is applied at assembly level, the loop size can be easily extracted and used for the I-cache constraints. However, due to the late application of the optimization, other optimizations performed previously cannot profit from the unrolled code. On the other hand, unrolling loops at the source code level offers optimization opportunities for a larger set of following optimizations, but a precise estimation of code size at source code level is usually not possible. Thus, both solutions are not fully satisfactory.

A typical unrolling heuristic, found in many compilers to restrict the increase of the code size, is to unroll loops as long as the loop size does not exceed a constant boundary. Compilers performing unrolling define the size of the unrolled loop by the number of source code expressions or assembler instructions depending on the code abstraction level. This value is typically chosen relatively small to avoid negative side-effects due to extensive unrolling. However, with this approach also optimization potential is missed.

The influence of the unrolling factor on the WCET is illustrated in Figure 1 for the *fft* benchmark from DSPstone [28] containing two loops that can be unrolled. Increasing the unrolling limitation, which defines the permitted number of expressions in the final loop body, to 60 expressions allows unrolling one of the loops by a factor of 2, leading to an increase of the WCET since additional spill code is introduced. With 150 expressions, both loops are further unrolled providing optimization potential for other optimizations reducing register pressure and eliminating some spill code. This positive WCET effect is cancelled when the loops are further unrolled at the 240-expression boundary. At this point, additional spill code is added and further I-cache misses due to a cache overflow arise.

Our unrolling benefits from advantages of common approaches while avoiding their drawbacks. First, we apply loop unrolling at the source code level providing optimization potential for subsequent high- and low-level optimizations. To estimate the assembly size of the loop, we transform information from the compiler's back-end into the front-end where unrolling takes place. This transformation is called *Back-Annotation*, described in the following section. Our unrolling is placed in the applied sequence of high-level optimizations after all code-expanding transformations, e. g. *function inlining* or *loop unswitching*. Optimizations following loop unrolling, e. g. *common subexpression elimination* and *redundant load elimination*, naturally reduce the code size, thus we are able to provide a reliable approximation of the loop size based on the source code. This information is involved in the determination of the unrolling factor to avoid I-cache misses.

Second, our loop unrolling has access to various information about the processor's physical memory that is provided by our compiler. On the one hand, the size of the involved I-cache can be extracted. Using this parameters, an appropriate unrolling factor can be chosen reducing the size of the unrolled loop to the given cache size. On the other hand, our optimizer is provided with detailed information about the memory usage of the program under analysis. The compiler constructs a memory model of the available physical memories available in the underlying target architecture. During code generation and code modification, this model is kept up-to-date such that valid information about memory usage can be extracted at any time.

The amount of free space in the program memory is read during the calculation of the unrolling factor in order to find a value that does not exceed the available program memory when the loop is unrolled. In addition, the user can parameterize the memory usage considered during loop unrolling. For example, the user might want to use only half of the I-cache, or just allow loop unrolling to consume 60%

of the free program memory. With this flexible handling of the memory model, our loop unrolling can be effectively tailored towards particular memory requirements which are particularly imposed on embedded systems.

Back-Annotation

The Back-Annotation serves as a translation of information between the compiler back-end and the front-end. The static WCET analysis takes place at assembly level since timing information about the executed instructions is required. Within a WCET-aware compiler, the results of the WCET analysis are imported into the compiler back-end and exploited for WCET optimizations.

To develop WCET-driven source code optimizations, WCET timing data must be made available at source code level. Our Back-Annotation establishes a connection between the high- and low-level IR of the code during the code generation of our compiler. This link enables to keep track of which objects in the high-level IR (i. e. source code constructs) correspond to which objects in the low-level IR (i. e. assembly constructs). The granularity of this mapping is basic block level. Using the Back-Annotation, any information that is available at the assembly code level can be easily accessed by the source code constructs.

We exploit three different types of information for our WCET-driven unrolling. Basic data required for any WCET optimization is the WCET of source code constructs. We exploit this information to find loops to be optimized. Moreover, we use WCET data to compute an unrolling profit that indicates how successful unrolling was. Details about the profit computation are provided in Section 4.5.

Moreover, we import data about the assembly size of the loop header and loop body. Using this information, we are able to estimate the size of the unrolled loop for a specific unrolling factor. Knowing the I-cache capacity size, an unrolling factor is chosen that avoids an I-cache overflow resulting in an adverse code transformation with a worst-case behavior degradation. Furthermore, the Back-Annotation allows the import of information about the amount of spill code within a basic block. This information can be used as an indicator for regions with a high register pressure that already suffer from spill code.

With the exploitation of the Back-Annotation, we overcome the common problem with the code size estimation at source code level. In [8], the authors criticized ad-hoc approaches that use a conversion factor to estimate the number of machine-language instructions produced for a source code line. Our approach provides exact size information that can be exploited as useful cost functions.

4.3. Prediction of Unrolling Effects

Besides I-cache overflows, additional spill code can have an adverse effect on the worst-case performance of the program. Spill code is introduced when more physical registers are required than available.

To predict the impact of loop unrolling at source code level on the register pressure, [24] proposed an approach that tries to approximate the maximal number of fixed- and floating-point values in the unrolled loop that may be simultaneously live. However, this approach has two main limitations. First, it is inflexible since its applicability highly depends on the involved register allocator. If the register allocator is modified or even exchanged by a register allocator pursuing another allocation strategy, then the register pressure approximation might fail. Second, the authors state in their paper that the approximation is conservative and may unnecessarily limit the amount of permitted unrolling.

Such ad-hoc approaches are required when information from the compiler back-end is not present. We propose a more realistic prediction of unrolling effects on the spill code exploiting the Back-Annotation. The prediction is based on a comparison between original loops and their unrolled version. During the evaluation, we create a copy P_{eval} of the original program P under analysis and unroll loops in P_{eval}. A comparison between the original and unrolled loops reveals if additional spill code was generated.

In detail, for each loop in P_{eval}, we consider all possible loop iteration counts derived from the static loop analysis (cf. Section 4.1). Based on the set of iteration counts, we determine the *smallest common prime factor (SCPF)* for a set S of possible iteration counts i for a loop L:

Definition 1 *With GCD_L being the greatest common divisor $\forall i \epsilon S$, the $SCPF_L$ is the smallest prime factor of GCD_L, if $GCD_L > 1$. Otherwise, SCPF is 1.*

For example, for a loop with the three (context-sensitive) iteration counts $\langle 4, 8, 20 \rangle$, the *SCPF* would yield 2. If in the rare case $SCPF_L$ is 1, L will be not unrolled.

The utilization of the *SCPF* for the prediction of unrolling effects is motivated by our unrolling strategy that tries to avoid additional conditional jump statements in the finally unrolled loops. Jumps are required when loops are unrolled with unrolling factors that do not evenly divide the number of iterations. Conditional jumps have several negative effects on the program performance, especially for a static program (and WCET) analysis. They introduce additional control pipeline hazards, they might produce high penalty cycles when mispredicted by the processor's branch prediction, and their use often results in a loss of precision for static analyses that merge data-flow results at CFG merge points. Thus, our optimization unrolls loop such that additional jumps are not required.

For a reliable prediction of the spill code, the unrolled loops in P_{eval} must also avoid jumps. This could be achieved by employing GCD_L as unrolling factor. However, this factor is often not applicable due to memory resource con-

straints. Some loops have a large *GCD* of their execution counts. If the loops were unrolled by this factor, they would possibly exceed the available program memory, whose free space is often restricted when multiple tasks reside in the same memory. Using *SCPF* for the evaluation, we avoid jumps in the unrolled loops on the one hand and memory overflows on the other hand since the *SCPF* is usually significantly smaller than the *GCD*. Moreover, using the *SCPF* has a positive side-effect on the compilation time since the optimization and analysis of minimally unrolled loops is faster than for excessively unrolled loops.

The prediction of the spill code impact on the program performance is implicitly integrated into the prediction of unrolling effects for each loop. Each loop is unrolled with the unrolling factor *u=SCPF*. The minimally unrolled code is translated into assembler code and analyzed in terms of its WCET. Please note that this evaluation is done once for the code where all loops are unrolled with *SCPF*.

Using the Back-Annotation, the comparison of different cost functions of the original and minimally unrolled loops allow an evaluation of the effects of *u=SCPF* on the worst-case performance. The results show how the WCET and the code size changed. Moreover, comparing the amount of spill code for the original and unrolled loop indicates the impact of unrolling on the register allocation. Our experiments on a large set of benchmarks showed that this approach produces sufficient results for the common allocation strategies: graph coloring based [3] and optimal [12] register allocation.

Assuming the definition of the amount of spill code (within a loop *L*) per unrolling factor *u*:

$$\psi_{\text{u,L}} = \frac{\sum spill\ code\ instructions\ \epsilon L}{unrolling\ factor\ u} \tag{1}$$

we observed that if $\psi_{\text{u,L}}$ did not increase for unrolling the loop with the factor *SCPF*, i. e. no extra spill code was inserted, then naturally also no extra spill code was introduced for $u > SCPF$. As an example, assume that the original loop has 10 spill code instructions. When the loop is unrolled by a factor of 2 and the unrolled loop has 20 spill code instructions, then spill code was just replicated and no extra spilling was introduced. However, if the number of spilling instruction is greater than 20, then in total more instructions must be executed for the unrolled loop.

Using this approach, we are more flexible than [24] since our approximation of the spill code is not restricted to a particular register allocator for which a deep knowledge about the allocation strategy is required. Using the evaluation run with the *SCPF* factor, a realistic prediction of the spill code can be even made when unrolling is followed by further optimizations. The results of the Back-Annotation gathered during the evaluation run are exploited for the computation of the unrolling profit as described in Section 4.5.

```
 1  Input:Loop L
 2  Output:int Factor_Final
 3  begin
 4   int sizeHeader := BackAnnotation(L)
 5   int sizeBody := BackAnnotation(L)
 6   set<int> iterations := LoopAnalysis(L)
 7   int GCD := FindGCD(iterations)
 8   for int i := GCD to 1
 9    int sizeUnrolled := sizeHeader+i×sizeBody
10    if((i−1)*sizeBody ≤ freePMem &&
11        sizeUnrolled      ≤ cacheSize  &&
12        GCD  mod i        == 0) )
13        return i
14   return 1
15  end
```

Figure 2. Algorithm for Unrolling Factor Determination

4.4. Determination of the Worst-Case Unrolling Factor

For the application of our WCET-driven loop unrolling, a final unrolling factor for each loop must be computed. The determination considers constraints imposed by the memory system and cost functions that were collected during the evaluation run using the unrolling factor *SCPF* on the program copy P_{eval}. The final unrolling factor is based on the *GCD* of the loop iteration counts, thus it is usually larger than *SCPF* to allow aggressive unrolling. The algorithm is shown in Figure 2.

In lines 4 and 5, the header and body size of the original loop L is computed. The greatest common divisor GCD of the iteration counts based on the loop analysis is computed. Using this factor, we avoid conditional jump instructions, as described in Section 4.3, that might decrease the WCET of the loop. In line 9, we compute the estimated size of the unrolled loop and check if free program memory and I-cache constraints are not violated in lines 10 and 11, respectively. With this strategy, we avoid an overflow in the program memory and possibly cache conflict misses. If these constraints are not met, the unrolling factor is decreased and the size constraints are tested again. If no appropriate unrolling factor was found, 1 is returned in line 14 meaning that this loop will be not unrolled.

4.5. WCET-driven Unrolling Heuristics

Knowing the unrolling factor that promises most WCET reduction for a loop, the complete algorithm for our WCET-driven loop unrolling is depicted in Figure 3.

In line 4, a copy of the original program is created. Since our compiler has full control over the code generation, we can establish a connection between loops in P and the copy P_{eval} which is later required to evaluate the unrolling effects.

```
1   Input:    Program P
2   Output:   optimized Program P
3   begin
4    Program P_eval := Copy(P)
5    list<Loop> loops_eval := Loops(P_eval)
6    for all Loop L_eval ϵ loops_eval
7     UnrollSCPF(L_eval) /*use SCPF*/
8    EvaluateSCPFUnrolling(P_eval)
9
10   list<Loop> loops := Loops(P)
11   RemoveSpillingLoops(loops)
12   SortByProfit(loops)
13   for all Loop L ϵ loops
14    WCETUnrolling(L) /*use Factor_Final*/
15   return P
16  end
```

Figure 3. Algorithm for WCET-driven Loop Unrolling

In line 7, for all innermost loops, which are typically considered by loop unrolling since they consume most execution time, unrolling is performed using *SCPF* as unrolling factor (cf. Section 4.3). Finally, the copied program with the minimally unrolled loops is evaluated by performing a WCET analysis and collecting the loops' cost functions in line 8.

In line 11, function $RemoveSpillingLoops$ removes all original loops for further consideration that turned out to produce extra spill code when unrolled. This evaluation is based on the amount of spill code per unrolling factor $\psi_{\text{u,L}}$ as defined in Equation 1. When ψ is larger for the loop unrolled with factor *SCPF* than ψ for the corresponding (original) loop, this loop is excluded from unrolling.

Before finally unrolling all loops collected in $loops$, the loops are sorted by their profit. The profit represents the expected WCET reduction and code size increase when the loop is unrolled using final unrolling factor computed by the algorithm in Figure 2. The goal of the profit calculation is to unroll those loops first that promise most benefits. Doing so, strict memory resource constraints typically found in embedded systems are taken into account. The remaining free program space is consumed with those loops for which a maximal trade-off between WCET reduction and code expansion is expected. We define the profit ρ_{L} for a loop L for which a final unrolling factor $Factor_{\text{Final}} > 1$ was computed:

$$\rho_{\text{L}} = \frac{\triangle\, WCET_{\text{SCPFunrolling}}}{\triangle\, CodeSize_{\text{SCPFunrolling}}} \times \frac{Factor_{\text{Final}}}{Factor_{\text{SCPF}}} \qquad (2)$$

The first fraction expresses the ratio between the WCET reduction and the code size expansion after unrolling the loop with the *SCPF* unrolling factor. Dividing this fraction by the *SCPF* unrolling factor represents the estimated profit for a single loop body replication. Multiplying this value by the final unrolling factor, a profit for unrolling loop L can be estimated. Loops with larger profits promise a higher benefit and are thus optimized first. All loops with a negative profit will likely have a negative impact on the WCET and are excluded from unrolling.

Finally, for all remaining loops in L, loop unrolling is performed (line 14) utilizing the final unrolling factor computed by the algorithm in Figure 2. Please note that we do not explicitly consider a switch of the worst-case execution path *(WCEP)*. WCEP switches might happen when a path in the CFG lying on the WCEP is optimized such that another path becomes the new WCEP. Such effects typically arise for mutually exclusive paths resulting from `if-then-else` statements which have a similar WCET. Optimizing one path slightly reduces its WCET such that the other path becomes the longest path. However, a WCEP switch is very uncommon for paths containing loops. Usually, a context-sensitive WCET analysis assumes that all loops in the program are executed at some point of the execution and these loops are also considered by our loop unroller. This was also confirmed by our review of the benchmark codes for which no switches of WCEP containing loops occurred. Thus, the high overhead of considering potential path switches which usually entails repetitive runs of the WCET analysis is not mandatory. The only exception is dead code, i. e. code fragments that are never executed. Unrolling *dead* loop would not improve the WCET but would just waste valuable program memory. These loops are implicitly excluded from unrolling by Equation 2.

The profit calculation serves as an estimation of unrolling effects before the actual unrolling. This option is missing in most compilers, thus loops are often unrolled that decrease program performance. With our profit calculation, we detect in advance if a loop should be unrolled or if its unrolling will likely decrease the worst-case behavior. The correctness of our prediction is emphasized by our results. As will be shown in Section 6, for all considered benchmarks, WCET reductions were achieved.

5. Experimental Environment

Our WCET-driven loop unrolling is integrated into a WCET-aware C compiler for the Infineon TriCore TC1796 [10]. The processor is equipped with a 16 kByte I-cache and 2 MByte program Flash that we use for our experiments. The I-cache size can be modified for the WCET analysis to evaluate different cache sizes. The processor does not have a data cache. However, unrolling marginally changes memory reference patterns, thus D-cache misses can be ignored for unrolling factor selection [13].

The workflow is depicted in Figure 4. The compiler is provided with a C file as input data. The program is parsed into the high-level IR, called ICD-C, and further translated into the low-level IR LLIR. Using the tight integration of the

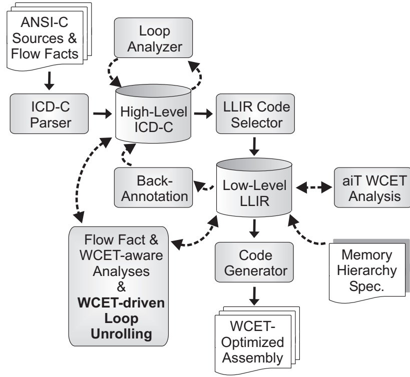

Figure 4. Workflow for WCET-driven Loop Unrolling

WCET analyzer aiT, WCET timing information is imported into the compiler back-end which allows the construction of a timing model. Using the Back-Annotation, information about the WCET, code size, and spill code is transformed back into the ICD-C and can be exploited for our WCET-driven loop unrolling. Additional data is provided by the loop analyzer operating on the ICD-C and the memory hierarchy specification attached to the LLIR. The register allocation used for benchmarking was performed by an optimal register allocator [12]. Please note that our analyses are *flow fact-aware*. Flow facts (e. g. information about loop iteration counts and recursion depth) are modelled within the compiler, and adjusted during optimizations before passed to the WCET analyzer.

Our WCET-driven loop unrolling was applied to 45 real-life benchmarks from the DSPstone [28], MRTC [21], MediaBench [16], and UTDSP [27] benchmark suites. The code sizes of the considered benchmarks range from 302 bytes up to 14 kByte with an average code size of 1.9 kByte per benchmark. The number of innermost loops considered for loop unrolling ranges between 1 and 15, depending on the benchmark complexity.

6. Results

For benchmarking, the code was compiled with optimization level *-O3* comprising 42 different high- and low-level optimizations. Thus, highly optimized code is considered.

Worst-Case Execution Time

Figure 5 shows the results of our optimization on the WCET. The bars represent the average WCET reduction for all 45 benchmarks when the code was compiled with the highest optimization level *-O3* including our WCET-aware unrolling w.r.t. to the code that was generated using the highest optimization level and disabled unrolling. For the results, we modified the I-cache size between 512 bytes and 16 kByte to simulate the impact of our optimization

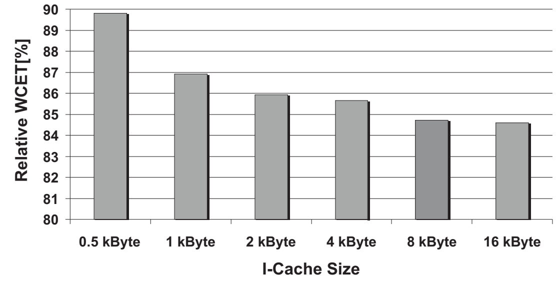

Figure 5. Cache-Size Dependent WCET

on different architectures. As expected, the WCET reduction can be increased from 10.2% for the smallest cache capacity (512 bytes) to a WCET decrease of 15.4% for the largest cache (16 kByte). The reason for this increase is obvious. Using larger caches, our approach is offered more optimization opportunities since extensive unrolling is less frequently limited by the I-cache constraints. It can be also seen that for larger caches the WCET reduction becomes smaller. The reason are smaller benchmarks that can be already fully optimized for modest cache sizes, thus relaxing the I-cache constraints due to larger cache capacities does not result in any further benefit. It should be noted that for a 16 kByte I-cache, standard ACET loop unrolling, as employed by many compilers, achieved an average WCET reduction of 1.7%, thus we outperform the standard optimizations by 13.7%.

Figure 6 answers the question which of our strategies provided the most benefit. Can the WCET reductions be addressed to the additional information generated by the static loop analysis, or are our unrolling heuristics the main source for the decreased WCET? The results show the WCET for three different strategies using a 2 kByte I-cache to capture cache effects, with 100% corresponding to the WCET of the benchmarks compiled with the highest optimization level and disabled unrolling. For the sake of readability, we show results on a subset of 19 representative benchmarks. As can be seen, using our standard ACET loop unrolling (LU) with a simple loop analysis and restricting the size of the finally unrolled loop to 50 expressions has minimal positive effects on the WCET. Integrating our sophisticated loop analysis (LA) into standard unrolling, slightly improves the average WCET by 2.9%. Notably improved results are achieved when loop unrolling is extended by our novel heuristics. WCET reductions of up to 39.5% are observed. These high WCET reductions achieved e. g. for the benchmarks *edge_detect* and *fir2dim* are the result of unrolling some loops completely. Such straight line code can be effectively improved by succeeding optimizations. However, our review on the optimized code reveals that it cannot be said in general which effects are most beneficial. For some benchmarks, like *countnegative*, the reduced loop overhead was the key factor, while other benchmarks profited from enabled optimization potential after unrolling.

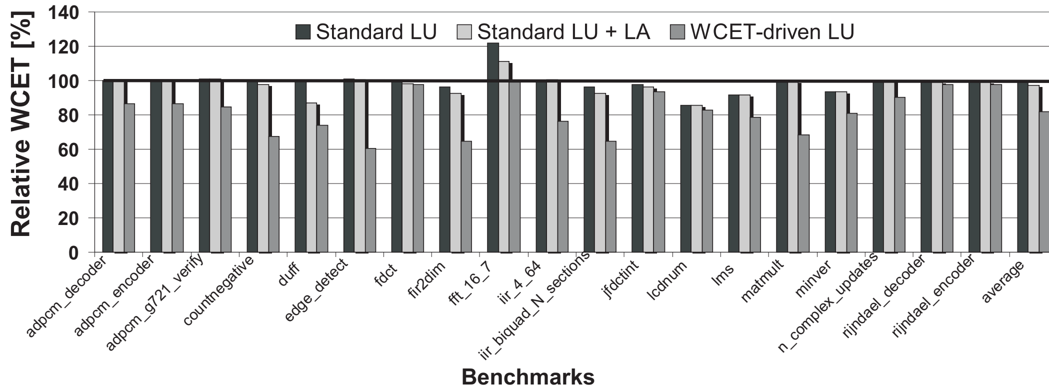

Figure 6. Comparisons of Unrolling Strategies

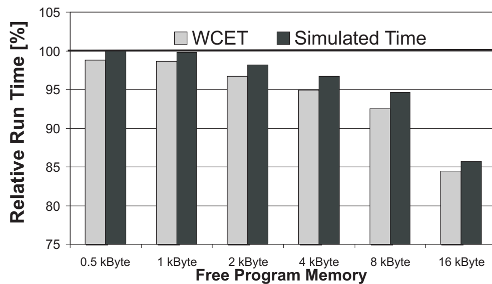

Figure 7. Impact of Program Size Heuristic

Simulated Time

To evaluate our program memory heuristic, we simulated a system for which program memory is restricted. This situation is common for embedded systems that execute multiple tasks residing in the same memory. The native 16 kByte I-cache of the TC1796 is assumed to enable a comparison between WCET and simulated time provided by a cycle-true simulator. Figure 7 shows the results for the WCET and simulated time when our WCET-driven unrolling is applied together with *-O3* for the example MRTC benchmark *ndes*. 100% correspond again to the WCET for *-O3* and disabled unrolling. First, it can be seen that with increasing program memory, the reduction of the benchmark's WCET is also increasing. This is expected as with more program memory, unrolling can be applied more aggressively. Second, from the results it can be inferred that our optimization is tailored towards WCET minimization. For all memory sizes, we achieve a higher reduction of the WCET than the simulated time. One reason for this behavior is that our approach begins to unroll those loops that promise the highest WCET reduction. Since the program memory is restricted, not all loops can be unrolled and the transformed loops most beneficial for WCET reduction need not to be implicitly the most beneficial loops for ACET reduction.

Code Size

Figure 8 shows the impact of our optimization on the average code size w.r.t. to the average code size of the benchmarks with disabled loop unrolling. Six different I-cache configurations ranging from 512 bytes up to 16 kByte are considered. As expected, the code size increase becomes larger with an increasing cache size. This is due to the growing potential of unrolling which can unroll loops more aggressively for larger caches without exceeding their capacity. It should be noted that the average code size increase is mainly reflecting small benchmarks, while the code size increase for larger benchmarks is modest. For example, the highest increase was found for the small DSPstone benchmark *matrix1* with 1261% for the 16 kByte cache, resulting in a final code size of 4114 Bytes. However, this code size is still fully acceptable. In contrast, the code size increase for the largest benchmark, *rijndael_encoder* from MediaBench, had only a maximal increase of 6.4%. This is also a typical increase for other larger benchmarks. As a conclusion, our approach produces enlarged code whose size is acceptable for modern processors and if required using the optimization parameters the code generation can be restricted to meet even strict code size constraints.

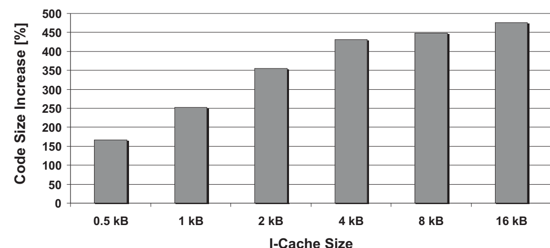

Figure 8. Impact on Code Size

Compilation Time

Finally, we measured the compilation time of our optimization on an Intel Xeon 2.4GHz system with 8GB RAM. Compilation of all 45 benchmarks including a single WCET analysis took 29 minutes when compiling the codes with *-O3* including the standard loop unrolling with a simple loop analysis. In contrast, the compilation time for our optimization took 390 minutes in total, using an 16 kByte I-cache. This increase mainly results from standard optimizations which have to analyze (in some cases significantly) enlarged

basic blocks. However, since the main objective of embedded systems compilers is performance maximization even at the cost of a longer compilation time, the observed compilation times are considered as fully acceptable.

7. Conclusions and Future Work

This paper is the first to present a WCET-driven loop unrolling. To be effective, our loop unrolling is applied at source code level. This early application in the optimization process establishes optimization potential for a large number of other compiler optimizations. By integrating a precise worst-case timing model into the compiler, unrolling decisions based on the WCET are taken in order to achieve a maximal WCET reduction. Moreover, we exploit results from a static loop analysis, take I-cache and memory constraints into account, and are able to approximate adverse spill code generation. Using our profit calculation, adverse effects of unrolling a loop can be predicted in advance. The effectiveness of our approach is demonstrated on 45 real-life benchmarks for which average WCET reductions of up to 13.7% are achieved.

In the future, we intend to study the impact of further source code optimizations on the WCET and possibly extend their heuristics to make decisions based on WCET data. Moreover, we plan to extend our framework by further cost functions, e. g. power dissipation, to enable a development of multi-objectives optimizations.

References

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore. 2009.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[3] P. Briggs. *Register Allocation Via Graph Coloring*. PhD thesis, Houston, TX, USA, 1992.

[4] A. M. Campoy, I. Puaut, and A. P. I. et al. Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In *Proc. of ECRTS*, 2005.

[5] S. Carr and Y. Guan. Unroll-and-Jam Using Uniformly Generated Sets. In *Proc. of MICRO*, 1997.

[6] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. Program. Lang. Syst.*, 16(6), 1994.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[8] J. W. Davidson and S. Jinturkar. An Aggressive Approach to Loop Unrolling. Technical report, 2001.

[9] J.-F. Deverge and I. Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *Proc. of ECRTS*, 2007.

[10] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *Proc. of ESTIMedia*, 2006.

[11] H. Falk, S. Plazar, and H. Theiling. Compile-Time Decided Instruction Cache Locking using Worst-Case Execution Paths. In *Proc. of CODES+ISSS*, 2007.

[12] D. W. Goodwin and K. D. Wilken. Optimal and Near-Optimal Global Register Allocations Using 0–1 Integer Programming. *Softw. Pract. Exper.*, 26(8), 1996.

[13] K. Heydemann, F. Bodin, and P. M. W. Knijnenburg. Global Trade-Off between Code Size and Performance for Loop Unrolling on Vliw Architectures. Technical report, 2001.

[14] A. Koseki, H. Komastu, and Y. Fukazawa. A Method for Estimating Optimal Unrolling Times for Nested Loops. In *Proc. of ISPAN*, 1997.

[15] D. M. Lavery and W.-M. W. Hwu. Unrolling-based Optimizations for Modulo Scheduling. In *Proc. of MICRO*, 1995.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of MICRO*, 1997.

[17] P. Lokuciejewsi, H. Falk, P. Marwedel, and T. Henrik. WCET-Driven, Code-Size Critical Procedure Cloning. In *Proc. of SCOPES*, 2008.

[18] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proc. of ECRTS*, 2008.

[19] Lokuciejewski, Paul and Cordes, Daniel and Falk, Heiko and Marwedel, Peter. A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models. In *Proc. of CGO*, 2009.

[20] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, and W.-M. W. Hwu. Compiler Code Transformations for Superscalar-based High Performance Systems. In *Proc. of Supercomputing*, 1992.

[21] Mälardalen WCET Research Group. Mälardalen WCET Benchmark Suite. http://www.mrtc.mdh.se/projects/wcet, 2009.

[22] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[23] I. Puaut and C. Pais. Scratchpad Memories vs Locked Caches in Hard Real-Time Systems: a Quantitative Comparison. In *Proc. of DATE*, 2007.

[24] V. Sarkar. Optimized Unrolling of Nested Loops. *Int. J. Parallel Program.*, 29(5):545–581, 2001.

[25] L. Song and K. Kavi. What Can We Gain by Unfolding Loops? *SIGPLAN Not.*, 39(2), 2004.

[26] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. of RTSS*, 2005.

[27] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, January 2009.

[28] C. S. V. Zivojnovic, J. Martinez and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proc. of ICSPAT*, 1994.

[29] X. Vera, B. Lisper, and J. Xue. Data Cache Locking for Higher Program Predictability. In *Proc. of SIGMETRICS*, 2003.

[30] W. Zhao, W. Kreahling, D. Whalley, et al. Improving WCET by Optimizing Worst-Case Paths. In *Proc. of RTAS*, 2005.