Accelerating WCET-driven Optimizations by the Invariant Path Paradigm – a Case Study of Loop Unswitching *

Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel
Computer Science 12
TU Dortmund University
D-44221 Dortmund, Germany
FirstName.LastName@udo.edu

Abstract

The worst-case execution time (WCET) being the upper bound of the maximum execution time corresponds to the longest path through the program's control flow graph. Its reduction is the objective of a WCET optimization. Unlike average-case execution time compiler optimizations which consider a static (most frequently executed) path, the longest path is variable since its optimization might result in another path becoming the effective longest path.

To keep path information valid, WCET optimizations typically perform a time-consuming static WCET analysis after each code modification to ensure that subsequent optimization steps operate on the critical path. However, a code modification does not always lead to a path switch, making many WCET analyses superfluous. To cope with this problem, we propose a new paradigm called Invariant Path which eliminates the pessimism by indicating whether a path update is mandatory. To demonstrate the paradigm's practical use, we developed a novel optimization called WCET-driven Loop Unswitching which exploits the Invariant Path information. In a case study, our optimization reduced the WCET of real-world benchmarks by up to 18.3%, while exploiting the Invariant Path paradigm led to a reduction of the optimization time by 57.5% on average.

1. Introduction

Embedded systems must often meet real-time constraints. Especially for safety-critical systems like in the avionic and automotive domain, the adherence of the worst-case execution time must be ensured to avoid system failure potentially leading to a disaster. The precise knowledge of this key parameter is also required for scheduling or the development of hardware platforms which have to satisfy critical timing constraints.

Due to the complexity of today's embedded systems, the software development relies on both a high-level language, predominantly C, and a compiler. State-of-the-art compilers offer a vast variety of optimizations with the objective to minimize the average-case execution time *(ACET)* [10] or energy dissipation [15]. On the contrary, a compiler-guided reduction of the WCET is still a novel research area. WCET-driven compiler optimizations require the integration of a static WCET analyzer into a compiler framework providing timing information taken into account to effectively minimize the program's WCET.

A well-known problem of a sophisticated static WCET analysis for today's embedded system applications is its complexity. Due to this reason, the major portion of the time required for a compiler-based WCET-driven optimization is spent for timing analysis while the contribution of the remaining parts of the optimization process is negligible. Moreover, an effective WCET-aware optimization typically relies on multiple invocations of the static WCET analyzer which substantially increase the optimization run time.

The static WCET analysis computes the worst-case behavior for a given program that is valid for all inputs without running the program but by performing a static program analysis. The WCET which is the upper bound of the maximum execution time corresponds to the longest path through the program's control flow graph *(CFG)*, called the worst-case execution path *(WCEP)*. Real applications typically consist of more than one path, however only the WCEP is relevant for the program's WCET and compiler optimizations aim at its reduction. The modification of WCEP WP may lead to a path switch, i. e. after reducing the length of WP, a new path WP' may become the longest path in the CFG. To enable a continuous WCET reduction,

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

the WCET-aware optimization must ensure that it does not proceed on the outdated WP but performs further transformations on the path WP'. In the approaches presented in the past, this is usually achieved by updating the internal information via another WCET analysis of the modified code. This necessity for repeated analysis is a well known problem and was frequently addressed in the literature.

In this paper we present a novel paradigm which allows a substantial decrease of the number of WCET analyzer invocations during WCET minimization. The main contributions of this paper are as follows:

1. We introduce the *Invariant Path* paradigm which detects subsets of a CFG that are *always* part of the WCEP, thus making an update of the WCEP superfluous after the modification of these sets.

2. We show how the Invariant Path information can be exploited in compiler-based WCET optimizations to drastically reduce the compilation time.

3. To show the practical use of the Invariant Path information, a novel optimization, the *WCET-driven Loop Unswitching*, is presented which exploits the new data to accelerate its optimization process.

The rest of this paper is organized as follows: Section 2 gives a survey of the related work. The concepts of the Invariant Path paradigm are presented in Section 3. Section 4 introduces the WCET-aware optimization Loop Unswitching, followed by a description of our experimental environment and results achieved on real-world benchmarks in Section 5 and Section 6, respectively. Finally, Section 7 summarizes this paper and gives directions for future work.

2. Related Work

Recently, the minimization of energy dissipation as an optimization goal of compilers has moved into the focus of research. However, WCET minimization by compiler optimizations is only sparsely dealt within today's literature. All these approaches rely on a static WCET analysis. A sophisticated analyzer, used also in this work, is the tool aiT [1] developed by the company AbsInt. [2] presents an approach to compute valid upper bounds for the WCET in the context of a system with preemptive scheduling. In [7], the authors present predictable code and data paging in order to enable a WCET analysis of systems using virtual memories.

A compiler guided trade-off between WCET and code size for an ARM7 processor was studied by [9]. They use a simplified timing analyzer to obtain WCET information employed in their code generator to produce code that exploits this trade-off and uses the two instruction sets (16-and 32-bit instructions) for different program sections.

In [3], an algorithm for static locking of I-caches based on a genetic algorithm is presented. [16] combines compile-time cache analysis with static D-cache locking. Both works aim at the minimization of the WCET and have in common that changes of the WCEP during the optimization are not considered. Thus, these algorithms are non-optimal. In contrast, the works of [4, 6, 14] propose to move parts of a program's code and data into the scratchpad memory or into a software-controlled cache while taking care that their optimizations always operate on the WCEP.

In [18], a code-positioning optimization driven by worst-case path information was presented. By rearranging the memory layout of basic blocks, branch penalties along the WCEP are avoided. To cope with the altering worst-case execution path, the authors perform a new WCET analysis after each block rearrangement.

A WCET-driven procedure positioning optimization was presented in [11]. We achieve an improved instruction cache behavior by reordering procedures in memory to reduce the number of cache conflict misses. In order to ensure that subsequent optimization steps after a previous memory layout modification operate on a valid WCEP, the static WCET analyzer updates the path information.

Most of these approaches have in common that they successively optimize the WCEP by modifying the code while keeping the WCEP up-to-date. This update is achieved by performing time-consuming static WCET analyses which mainly account for the long optimization time. To shorten the optimization time, we propose the Invariant Path paradigm which indicates whether an update of the WCEP is mandatory after a code modification. Since the found path is *invariant* to WCEP switches, redundant WCET analyses can be omitted without suffering a loss of significant information. The paradigm will be described in the next section.

The high-level compiler optimization *Loop Unswitching* is a well-known control-flow transformation. It moves a loop-invariant condition branch outside the loop. In case of an if-$else$ statement the loop body is duplicated and the modified version is placed inside the condition's $then$- and $else$-block [13]. The benefits of the optimization are the reduced number of executed branches and more opportunities for parallelization of the loop. The drawback is the increased code size. In Section 4 this standard ACET optimization will be extended by WCET concepts.

3. Invariant Path Paradigm

In general, the number of different mutually exclusive paths in a CFG results from branches in the program. They represent potential candidates for a WCEP switch and must be accounted by a WCET optimization. To cope with this problem, we introduce the Invariant Path:

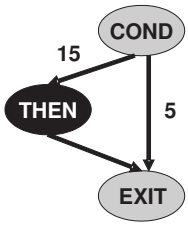

Figure 1. if statement on WCEP

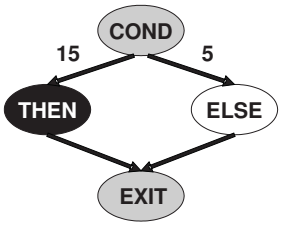

Figure 2. Both parts of if-$else$ statement on WCEP

Definition 1 *The Invariant Path is a sub-path of the WCEP which will always remain part of the WCEP independent of the applied code modifications.*

Any code that lies on the WCEP and is not part of any mutually exclusive paths obviously lies on the Invariant Path. The challenging issues concern different types of control-flow branches that will be discussed next.

The methods for a split of the control flow path depend on the abstraction level of the program. High-level programming languages use so-called *selection statements* like if, if-$else$ or $switch$ (general form of if-$else$) statements to model mutually exclusive paths. They perform a conditioned execution dependent on a condition expression. Low-level programming languages model mutually exclusive paths using conditional jump instructions. Other types of statements/instructions which alter the program's control flow like $call$ statements are not considered by the Invariant Path paradigm since they are irrelevant for a path switch. In this paper, the focus lies on the high-level language constructs but the ideas of the Invariant Path can be translated to low-level languages in a straightforward manner. For the sake of clarity we use the terminology of the programming language C, but the presented concepts are independent of the language.

To understand the concepts behind an Invariant Path computation, some details about a static WCET analysis must be introduced. Generally, a WCET analysis handles conditional statements that split the control flow as follows: if the condition can be statically evaluated, it is known which path will be taken during execution and this path contributes to the WCET. Otherwise, the static analysis assumes the longer path to be the WCEP. This assumption is safe but might lead to a WCET overestimation.

Moreover, we assume that the static WCET analysis is context-sensitive. Contexts represent either calling contexts, i. e. the analysis distinguishes between different calls to a particular function, or they can be used to consider each loop iteration separately. Sophisticated WCET analyzers support this technique to increase the precision of the computation since it introduces a dynamic view of the program during a static program analysis. By considering context-dependent program variable values for repetitively executed code fragments like functions or loops, context-dependent CFG paths can analyzed separately.

Based on this knowledge, the Invariant Path paradigm distinguishes between three different classes of selection statements leading to a generation of mutually exclusive paths which are potential candidates for a path switch. They are described in detail in the following.

3.1. IF with feasible WCEPs

An if statement represents a conditional execution. Depending on the conditional expression either the path through the $then$-part is executed or the mutually exclusive path omitting the $then$-part is followed as can be seen in Figure 1. The edge labels denote the frequency of executing one of the two paths in the worst case, called *execution counts*, which means that both parts contribute to the WCEP.

If an optimization aiming at the reduction of the WCEP encounters an if statement, two possible cases must be distinguished during a context-sensitive WCET analysis. Either the WCEP goes along the $then$-part or the $then$-part does not contribute to the WCET.

In terms of the Invariant Path paradigm, a WCEP that traverses the $then$-part is also part of the Invariant Path since a modification of the code in the $then$-part is not crucial for path switching. This is due to the reason that the other feasible path of the if statement does not contain any code that might become the new WCEP.

3.2. IF-$ELSE$ with feasible WCEPs

This case occurs when the context-sensitive WCET analysis determined that the WCEP traverses the $then$- and $else$-part of a particular if-$else$ statement in different contexts as can be seen in Figure 2.

Compiler optimizations typically do not take different contexts into account since transformations are applied to the static code where dynamic aspects of the program execution are not available. Thus, most WCET optimizations evaluate selection statements statically by accumulating the context-sensitive WCET information over all contexts and

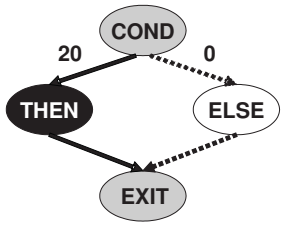

Figure 3. $then$-part of if-$else$ statement on WCEP

annotate the possible paths with these overall WCET values. In case of the if-$else$ statement, the $then$- and $else$-part would be annotated with the accumulated WCETs.

For a WCET optimization that must be aware of a valid WCEP, an if-$else$ statement, for which the WCEP goes either through the $then$- or $else$-part in different contexts, is not crucial since a path switch can not emerge. By treating this selection statement in a static manner, it is known from the WCET analysis that both parts always contribute to the program's WCET. Thus, this selection statement can be declared as part of the Invariant Path.

We call this type of branches as well as branches resulting from if statements *good-natured* since they are not prone to a path switch. This is the typical situation found in most applications, thus a large portion of the code can be declared as Invariant Path.

3.3. IF-$ELSE$ with an infeasible WCEP

The last class of selection statements differs from the previous one in terms of the WCEP flow. For all if-$else$ statements belonging to that class, the WCET analysis assumes that for all contexts the WCEP traverses exactly one of the mutually exclusive paths, i. e. either through the $then$- or the $else$-part. This situation might arise for two different cases. Either the value analysis, which is part of a sophisticated WCET analysis, evaluated the condition of the if-$else$ statement to be always $true$ or $false$, thus excluding one of the two paths. Or the condition could not be statically evaluated and one of the two mutually exclusive parts was determined to be always the longer path.

For the first case, the fact that one of the two branches is never executed *(dead code)* can be exploited. Since a WCET optimization would never consider a transformation of a non-executed code fragment, this code can be excluded and the other mutual path is declared as Invariant Path. Thus, this type of branches can be reduced to a *good-natured* if statement.

In the second case, the WCEP exclusively flows for all execution contexts either through the $then$- or the $else$-part. In addition, the branch not lying on the WCEP is not dead code. This combination is the only situation where a path switch can occur. These types of conditional statements are not declared as Invariant Path. An example is shown in Figure 3 where the $then$- but not the $else$-part (depicted by dotted arrows) lies on the WCEP. An optimization might optimize the $then$-part such that the path through the $else$-part becomes the new WCEP. To make sure that the next step of the optimization does not operate on an outdated path, a validation of the path information by a WCET analysis is required.

We call this type of branches described in this section *ill-natured* since they are prone to a WCEP switch.

3.4. Construction of the Invariant Path

The construction of the Invariant Path is performed recursively based on the assumptions from the previous sections. The computation begins with the entry point of the program's CFG, typically the first statement or instruction of the *main* function, and traverses the graph in a top-down manner.

All blocks lying on the WCEP that are not part of a branch (i. e. straight line code not control dependent of some branch statement) are declared by our algorithm as Invariant Path. Their transformation does not entail the risk of a path switch, thus they can be classified as Invariant Path. For good-natured selection statements, the Invariant Path is recursively computed for all feasible branches. Thus, all blocks that are part of the considered branch are taken into account. When function calls are encountered on the Invariant Path, their bodies are analogously analyzed. In that way, the Invariant Path is constructed also for nested selection statements.

3.5. Validation

To check that our assumptions about the Invariant Path are correct, i. e. a path switch never happens on the Invariant Path, we performed a validation of this paradigm on 42 real-world benchmarks from the MRTC WCET Benchmark Suite [12] and MediaBench suite [8]. The validation phase consists of two steps and was performed on the source-code level. First, is was verified whether the Invariant Path of a program lies on the WCEP. For this purpose, real-world benchmarks were compiled with the WCET-aware C compiler WCC [5], which will be described in detail in Section 5, with optimization levels O0 - O3. Next, using the WCET analyzer aiT and the compiler's high-level intermediate representation (IR) of the code annotated with WCET information, it was verified that code declared as Invariant Path is also lying on the WCEP.

The annotation of the high-level code with WCET data, which is originally computed at the low-level of the program, is called *Back-Annotation*. This process establishes a

connection between the high- and low-level IR of the employed WCET-aware compiler and translates worst-case execution data from the low-level to corresponding high-level constructs. Thus, WCET information are made accessible to high-level analyses and optimizations.

In the second step of the validation, it was tested whether the Invariant Path is prone to a path switch. For this purpose, branches in the program were explored since branches are the source for path switches that arise when the code is transformed such that the original longest past becomes outdated and another path in the CFG becomes the new longest path. For this step, each of the benchmarks under test was compiled within the WCET-aware compiler with optimization levels Oi ($i \in \{0, 1, 2\}$). Afterwards, the same benchmark was compiled with a higher optimization level Oj ($j \in \{1, 2, 3\}, j > i$). Using different optimization levels can be considered as a simulation of potential code transformations that might yield a path switch.

Finally, we verified if none of the *good-natured* branches compiled with Oi converted into an *ill-natured* branch after compilation with Oj. A violation of this condition would mean that (a part of) the WCEP was incorrectly classified as Invariant Path since a WCEP switch between mutually exclusive paths was encountered, i. e. the Invariant Path became outdated. In contrast, an *ill-natured* branch might convert into a *good-natured* branch. This might happen when an optimization removes one of the two paths that turned out to be never taken, thus practically converting an $if\text{-}else$ statements into an if statement. For all tests performed on the 42 different benchmarks, the validation was successful.

3.6. Invariant Path Ratio

The concepts of the Invariant Path are generic (not restricted to any programming languages) and can be combined with most WCET optimizations that are aware of the WCEP. Sub-graphs of the WCEP which are declared as Invariant Path can be modified without invalidating the WCEP information. To get an impression of how sensitive benchmarks are to WCEP switches, we computed the fraction of the code that is part of the Invariant Path. The computation was performed on the same benchmarks that were used for the validation.

We computed the *static* and *dynamic* ratio of code on the Invariant Path. The static value represents the number of basic blocks that are on the Invariant Path w.r.t. the total number of basic blocks in the program. The dynamic ratio indicates how many WCET cycles were consumed on the Invariant Path w.r.t. the total number of cycles on the WCEP, and were computed as follows:

$$\frac{\sum_{block} WCET_{est}(block) \cdot IP(block)}{\sum_{block} WCET_{est}(block)} \quad (1)$$

```
for(i=0; i<100; i++) {
 x[i] = x[i] + y[i];
 if (w)
  y[i] = y[i] * 2;
 else
  y[i] = 1; }
```

```
if (w)
 for(i=0; i<100; i++) {
  x[i] = x[i] + y[i];
  y[i] = y[i] * 2; }
else
 for(i=0; i<100; i++) {
  x[i] = x[i] + y[i];
  y[i] = 1; }
```

Figure 4. Example for Loop Unswitching

where $WCET_{est}$ represents the WCET estimation for each basic block, while $IP(block)$ indicates if $block$ is on the Invariant Path or not, i. e. it might have the value 0 or 1.

Our results show that the static ratio of blocks on the Invariant Path ranges between 74.1% and 77.9% for code optimized with optimization levels O0 - O3. For the dynamic ratio of cycles on the Invariant Path we observed that between 85.4% and 88.8% of the WCET cycles are spent for the execution of the code declared as Invariant Path. It is obvious that the ratio for the Invariant Path cycles is generally larger than the ratio for the blocks on the Invariant Path since iteratively executed code (e. g. in loops or functions) lying on the Invariant Path enlarges the dynamic ratio.

These results underline the optimization potential of the Invariant Path paradigm. Since approximately $\frac{3}{4}$ of the code is always lying on the critical WCEP without being sensitive to any path switches, the exploitation of this fact can significantly decrease the analysis time of WCET minimizations w.r.t. conservative approaches that, unaware of the Invariant Path, must run a costly WCET analysis after each code transformation to update their WCET information.

To demonstrate the practical use of our Invariant Path paradigm, we have implemented a WCET-driven Loop Unswitching that significantly benefits from that auxiliary information as shown in the next section.

4. WCET-driven Loop Unswitching

Loop Unswitching is a typical ACET compiler optimization where a trade-off between the execution time improvement and the resulting code size increase must be taken into account. It shifts loop-invariant conditions out of the loop at the cost of loop body duplications. Due to this reason, this optimization can not be applied to all possible loop candidates if strict code size constraints, as often found in the embedded system domain, must be met. Rather, potential candidates should be evaluated before optimized, thus enabling a successive improvement in the execution time through unswitching of most promising candidates while keeping the code size increase minimal. However, most compilers lack execution frequencies and timing information about the code to be optimized which makes a sophisticated preced-

```
1  Input:  Program P, size increase MAX
2  Output:  optimized Program P
3
4  begin
5   performWCETAnalysis(P)
6   set<Loop> S := FindUnswitchCand(P)
7   while(!S.empty()) do
8    boolean allOnIP := CheckInvariance(S)
9    repeat
10    Loop bestCand := FindBest(S)
11    LoopUnswitching(bestCand)
12    DeleteCandidate(S,bestCand)
13    if(CodeSizeIncrease(P) ≥ MAX)
14     return P
15    fi
16   until(!S.empty() && allOnIP == true)
17   performWCETAnalysis(P)
18   S := FindUnswitchCand(P)
19  od
20  return P
21 end
```

Figure 5. WCET-driven Loop Unswitching algorithm

ing evaluation impossible. In this work, we focus on an effective WCET reduction and solve this dilemma by exploiting information from a static WCET analyzer for a prior evaluation before optimizing the loop which promises the highest decrease in the WCET.

An example of Loop Unswitching is given in Figure 4. Standard Loop Unswitching, which was up to now employed to reduce the ACET, works in two steps. In the first step, it iterates over all statements of a function and searches for loop-invariant selection statements. If a proper candidate was found, its loop is unswitched by moving the $if/if\text{-}else$ statement outside the loop and copying the loops inside the $then$- and $else$-parts. Since most compilers are unaware of the execution frequency of the selection statements, a common but also ineffective heuristic used for Loop Unswitching is to transform loops in the order as given in the source code until a restriction to the code size increase is reached.

In contrast to previous works, we have extended the standard Loop Unswitching as follows:

- We exploit Unswitching to automatically reduce the WCET and not the ACET.
- Our novel optimization is based on more sophisticated heuristics based on worst-case execution frequencies and the WCET.
- To accelerate the WCET optimization, we combine it with the Invariant Path paradigm.
- To keep the increase of the code size small, most promising loops for the WCET minimization are transformed first.

Before describing the optimization steps in detail, the use of the Invariant Path information should be motivated in the context of unswitching. The danger for a WCEP switch comes from the code restructuring and the sensitivity of today's processors to any memory layout modifications. The reasons are twofold. On the one hand, instruction caches might show a different behavior w.r.t. incurred cache misses. By shifting the selection statement, the $then$- and $else$-parts are mapped to different addresses in the memory and the cache. This might lead to new cache misses on the non-WCEP which becomes the new longest path. On the other hand, many processors incur a penalty when performing a fetch to a misaligned target instruction (also called *line crossing*). By shifting code during unswitching, additional misaligned instructions on the non-WCEP might be introduced making this path the longest path [17]. Exploiting the Invariant Path information, we can distinguish between loops which might entail a WCEP switch after Loop Unswitching and those where a path switch can be definitely excluded making the updating WCET analysis redundant and thus accelerating the optimization.

The algorithm for the extended unswitching is depicted in Figure 5. It expects the program to be optimized and the maximal code size increase as input. After performing a WCET analysis and Back-Annotation (line 5), all possible candidates in the source code are collected in set S (line 6) together with their execution frequencies and WCETs. Note that all loops which are not on the WCEP, i. e. their WCET is equal zero, are excluded and not added to S. The outer loop iterates as long as candidates for Loop Unswitching are found (line 7). In line 8, it is determined if all of the collected loops are on the Invariant Path and the result is store in $allOnIP$.

Next, loop unswitching is performed iteratively in a loop (lines 9-16) without updating the program's timing model by an expensive WCET analysis. This step is done as long as unprocessed unswitching candidates are found in S and all of them are lying on the Invariant Path.

The candidates are evaluated in function $FindBest$ (line 10). As a heuristic for finding the most promising candidate we consider the worst-case execution frequencies of the selection statements in the collected loops. The loop holding the selection statement which is executed most frequently on the WCEP will be unswitched (line 11) and deleted from the list of candidates (line 12). If two selection statements have the same worst-case execution frequency, the one with the larger WCET is chosen. In the rare case of equal WCETs, the selection statement with the smaller

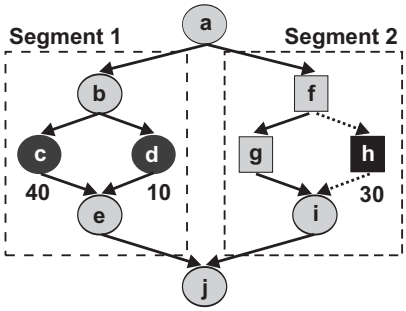

Figure 6. WCEP switch in different segments

(assembly) code size provided by the Back-Annotation is chosen for the optimization. If the maximal code size increase has been reached after unswitching, the optimization is terminated (line 14).

The condition whether all collected unswitching candidates are on the Invariant Path (line 16) is mandatory for an effective WCET minimization since changes in one CFG segment might influence the WCEP in another segment. This situation is depicted in the CFG in Figure 6 as could be possibly found at the beginning of our WCET-aware Loop Unswitching. For simplicity, nodes represent either single basic blocks or loops and the edge labels represent relevant worst-case execution counts. Nodes that are represented by circles are lying on the Invariant Path. The WCEP traverses all nodes but node h (path marked by dotted line). Further, it should be assumed that nodes c, d and h represent loops that are candidates for unswitching (marked by dark nodes). Our algorithm would begin with node c which has the highest execution count (40). However, the transformation of this node in segment 1 can potentially have an influence on segment 2 leading to a WCEP switch from the left to the right branch (to h). Thus, in the next step, not the previously collected loop d but h should be considered for unswitching. Obviously, this information is provided after another WCET analysis following the first transformation.

In our algorithm, variable $allOnIP$ is responsible for the indication of potential WCEP switches among different CFG segments by forcing a new WCET analysis (line 17) to keep relevant unswitching candidates updated. At first glance, this condition might seem restrictive preventing an extensive exploitation of the Invariant Path. However, as shown in Section 3.6, a large portion of the code lies on the Invariant Path, thus $allOnIP$ can be expected to be often $true$. As will be seen later in Section 6, this assumption could be also validated for real-world benchmarks for which a large number of redundant WCET analysis could be avoided. If the Invariant Path information was not available, each Loop Unswitching would be followed by a WCET analysis to ensure further optimization steps to operate on a valid WCEP.

It should be also noted that $allOnIP$ is only relevant for code size critical optimizations that aim at a reduction of the WCET while keeping the increase of the code size minimal. In that case, as also for our WCET-driven Loop Unswitching, always the most promising candidate (e. g. a loop for unswitching) for a WCET minimization should be chosen to improve the program's worst-case performance with a minimal number of transformations that increase the code size. In contrast, when a maximal WCET reduction is the exclusive goal of an optimization, Invariant Path information can be fully exploited. Since the optimization of the Invariant Path always reduces the WCET, the compiler can iteratively optimize this part of the code without performing further WCET analyses. Afterwards, the WCET information must be once updated for the transformed code to check if possibly new optimization candidates, due to a WCEP switch in the remaining code, occurred that can be optimized next.

Our algorithm in Figure 5 returns an optimized program. However, due to the potentially omitted WCET analyses during the optimization, no reliable assumptions about the WCET of the transformed code can be made. Thus, to get a safe WCET estimation of the optimized program, a final WCET analysis should be performed. These estimations are also the results presented in Section 6.

5. Experimental Environment

Benchmark	#Cand.	Code Size	Description
transupp	13	7224 B	JPEG transformation
wrbmp	4	682 B	JPEG conversion
block	7	16050 B	H264 decoding
macroblock	5	18520 B	H264 decoding

Table 1. Benchmark Characteristics

To indicate the efficacy of Invariant Path information and the WCET reductions achieved with our Loop Unswitching, tests were performed on real-world benchmarks. The benchmarks come from the widely used MediaBench suite representing different applications typically found in the embedded systems domain. They contain typical routines that are frequently used in larger benchmarks, e. g. the JPEG-2000 image and the H.264 video compression. Due to the high complexity of the original benchmarks which can not be handled by today's state-of-the-art static WCET analyzers, some major kernel routines were used instead as described in Table 1. The second column of the table indicates the number of candidates for the WCET-aware Loop Unswitching.

The techniques presented in Sections 3 and 4 are fully implemented. The workflow is depicted in Figure 7. For the computation of the Invariant Path information and the

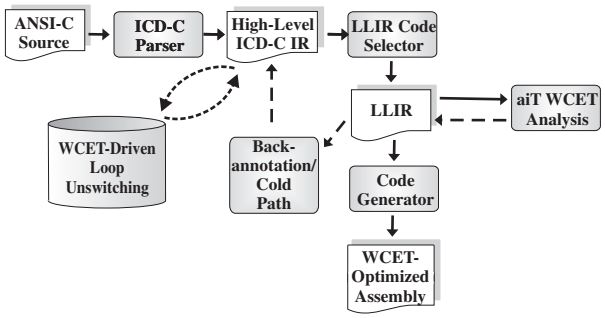

Figure 7. Workflow for WCET-driven Unswitching

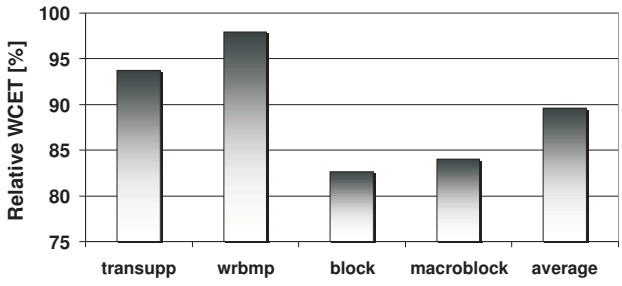

Figure 8. Relative WCET after Unswitching

integration of our WCET-driven Loop Unswitching, we use our WCET-aware C compiler WCC for the Infineon TriCore TC1796 processor [5]. The general structure of our compiler consists of a high-level IR, the ICD-C IR, and a low-level IR, called the LLIR, which is coupled to AbsInt's WCET analyzer aiT. Parsing of the C source code, the transformations into the corresponding intermediate representations, and the automatically performed static WCET analysis are depicted by solid arrows.

After the WCET analysis performed with an unlimited number of distinguished contexts, information about the execution frequencies and the WCETs provided by aiT are imported into the LLIR. Since Loop Unswitching is a high-level optimization, this data must be transformed from the LLIR into the ICD-C IR using the compiler's Back-Annotation. In parallel, the calculation of the Invariant Path exploiting Back-Annotation data is performed. The ICD-C annotated with timing information is finally used as input for our WCET-driven Loop Unswitching. The flow of these steps is marked by dashed arrows in Figure 7.

For our experiments, we disabled all compiler optimizations except dead code elimination. With this scenario, the impact of the WCET-driven Loop Unswitching on the WCET can be best studied. By applying dead code elimination, we make sure that our optimization does not optimize unswitching candidates that represent traditional dead code and as such could be easily eliminated before WCET analysis. In this paper, we also decided not to perform our unswitching within an optimization sequence since other optimizations performed before or after might hide its impact on the code preventing a clear interpretation.

6. Results

6.1. Worst-Case Execution Time

Figure 8 presents the timing results, with 100% corresponding to the WCET estimation of the original code after dead code elimination. The tests were performed with enabled 8 kB I-cache of the TriCore processor. Note that the default cache capacity of 16 kB was reduced to take cache effects for the benchmarks under test into account. It can be seen that for all benchmarks on average a WCET minimization of 10.4% was achieved. The maximal WCET reduction of 18.3% was achieved for the *block* benchmark. It contains 7 loop-invariant selection statements executed between 4 and 16 times in the worst-case. By unswitching, their execution frequencies could be significantly reduced.

6.2. Optimization Run Time

To indicate the effectiveness of the Invariant Path paradigm, we measured the optimization time of our WCET-driven Loop Unswitching with and without exploiting Invariant Path information. The optimization time encompasses the entire optimization process from parsing the source code to the generation of the optimized assembly code. However, since the WCET analyses consume the most optimization time, the results reflect the core of a typical WCET analysis.

The results given in Figure 9 were generated on an Intel Xeon 2.13GHz system with 4GB RAM. The 100% mark corresponds to the optimization time of standard Loop Unswitching without any WCET heuristics. The diagram shows that for all benchmarks the optimization time could be drastically reduced when the Invariant Path information is exploited. On average, the optimization time using the Invariant Path information could be reduced by 57.5%. The conventional WCET optimization without employing the new paradigm was 872.7% of the optimization time for the standard ACET Unswitching. Taking the Invariant Path into account reduces the optimization time to 379.0% of the standard Loop Unswitching.

These significant optimization time reductions result from the reduced number of performed WCET analyses. For example, the number of mandatory WCET analyses to update the WCEP which was performed for the benchmark *macroblock* could be reduced from seven analyses to two analysis (including the final mandatory aiT run), reducing the optimization time from more than 75 minutes to less

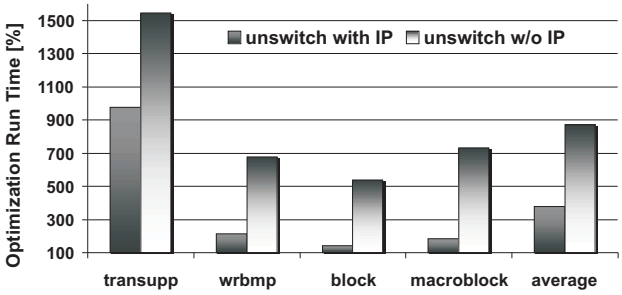

Figure 9. Relative Optimization Run Time with and w/o Invariant Path

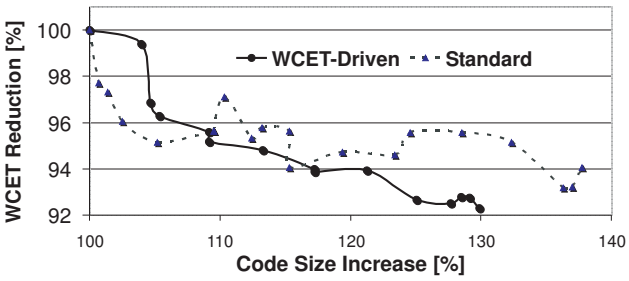

Figure 10. Comparison of standard and WCET-driven Unswitching for *transupp*

than 19 minutes.

6.3. Code Size Increase

The drawback of Loop Unswitching is the code size increase. For the four benchmarks we measured an average increase of 19.7%. To bound the code size increase, the WCET-driven Loop Unswitching might be terminated as soon as a desired WCET reduction or the maximally permitted code size increase is achieved. The diagram in Figure 10 shows the relationship between the WCET reduction and the code size increase for each optimization step of unswitching for the benchmark *transupp*. The solid curve represents the measurements for the WCET-driven Loop Unswitching, while the dotted curve depicts the measurements for standard unswitching. The points, which were used to construct the curves, represent the relative WCET and relative code size w.r.t. code (after dead code elimination) which was measured after each unswitching transformation. Based on the solid curve representing measurements for the WCET-driven Loop Unswitching, the parameters for the desired optimization objective can be extracted and used to terminate the optimization when the desired objective is achieved.

Moreover, a comparison between the solid curve and the dashed curve shows that our WCET-driven approach continuously reduces the WCET with each optimization step. In contrast, the standard approach does not guarantee to minimize the WCET after unswitching a selection statement. For example, a code size increase to 105.2% with standard unswitching results in a relative WCET of 95.1%. After unswitching two further selection statements, the code size increases to 110.1% having a negative effect on the WCET which is increased to 97.1%. This points out that our new optimization is tailored towards an effective WCET reduction and outperforms the standard Loop Unswitching for this objective.

7. Conclusions and Future Work

One of the major challenges of WCET-driven optimizations is to keep track of valid information about the worst-case execution path. After a code modification, the path information might, however, become outdated and typically a time-consuming WCET analysis must be performed for update purposes. These updates drastically increase the optimization time and might make a WCET optimization even impractical. To cope with this problem, we introduce the Invariant Path paradigm indicating which parts of the code will always be lying on the WCEP independently of the performed code modifications. To point out the practical use of the Invariant Path, it is combined with our newly developed optimization called WCET-driven Loop Unswitching which reduces the WCET of real-world benchmarks by up to 18.3% and the exploitation of Invariant Path information allows an optimization time reduction by 57.5% on average. In the future, we would like to assist further WCET-driven optimizations with Invariant Path information to reduce the optimization time. Moreover, we work on an application of the Invariant Path information at the low-level code representation.

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. Worst-Case Execution Time Analyzer aiT for TriCore. 2008.

[2] S. Altmeyer and G. Gebhard. WCET Analysis for Preemptive Systems. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 105–112, Prague, Czech Republic, July 2008. OCG.

[3] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[4] J.-F. Deverge and I. Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[5] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, October 2006.

[6] H. Falk, S. Plazar, and H. Theiling. Compile-Time Decided Instruction Cache Locking using Worst-Case Execution Paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis*, pages 143–148, New York, NY, USA, 2007. ACM.

[7] D. Hardy and I. Puaut. Predictable Code and Data Paging for Real-Time Systems. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, pages 266–275, Prague, Czech Republic, July 2008. IEEE Computer Society.

[8] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of MICRO*, December 1997.

[9] S. Lee, J. Lee, C. Y. Park, and S. L. Min. A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor. In *Proc. of "Intl. Workshop on Software and Compilers for Embedded Systems" (SCOPES)*, Amsterdam, Sept. 2004.

[10] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *DATE '00: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 4 – 8, Paris, Mar. 2000. IEEE.

[11] P. Lokuciejewski, H. Falk, and P. Marwedel. Wcet-driven cache-based procedure positioning optimizations. *ecrts*, 0:321–330, 2008.

[12] Mälardalen WCET Research Group. Mälardalen WCET Benchmark Suite. http://www.mrtc.mdh.se/projects/wcet, November 2008.

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

[14] I. Puaut and C. Pais. Scratchpad Memories vs Locked Caches in Hard Real-Time Systems: a Quantitative Comparison. In *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[15] S. Steinke, L. Wehmeyer, et al. The *encc* Compiler Homepage. *http://ls12-www.cs.uni-dortmund.de/research/encc*, 2002.

[16] X. Vera, B. Lisper, and J. Xue. Data Cache Locking for Higher Program Predictability. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–282, New York, NY, USA, July 2003. ACM.

[17] W. Zhao, W. Kreahling, D. Whalley, et al. Improving WCET by Optimizing Worst-Case Paths. In *Proc. of "11th RTAS Symposium"*, San Francisco, Mar. 2005.

[18] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET Code Positioning. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 81–91, Washington, DC, USA, 2004. IEEE Computer Society.