WCET-Driven, Code-Size Critical Procedure Cloning [*†]

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

Henrik Theiling
AbsInt Angewandte Informatik GmbH
Science Park 1
D-66123 Saarbrücken
theiling@absint.com

Abstract

In the domain of the worst-case execution time (WCET) analysis, loops are an inherent source of unpredictability and loss of precision since the determination of tight and safe information on the number of loop iterations is a difficult task. In particular, data-dependent loops whose iteration counts depend on function parameters can not be precisely handled by a timing analysis. Procedure Cloning can be exploited to make these loops explicit within the source code allowing a highly precise WCET analysis.

In this paper we extend the standard Procedure Cloning optimization by WCET-aware concepts with the objective to improve the tightness of the WCET estimation. Our novel approach is driven by WCET information which successively eliminates code structures leading to overestimated timing results, thus making the code more suitable for the analysis. In addition, the code size increase during the optimization is monitored and large increases are avoided.

The effectiveness of our optimization is shown by tests on real-world benchmarks. After performing our optimization, the estimated WCET is reduced by up to 64.2% while the employed code transformations yield an additional code size increase of 22.6% on average. In contrast, the average-case performance being the original objective of Procedure Cloning showed a slight decrease.

1. Introduction

Many embedded systems have to meet real-time constraints. One of their key parameters is the WCET and its knowledge is required for scheduling or the development of hardware platforms tailored towards given software resource requirements.

A sophisticated approach to calculate the worst-case execution time is the static WCET analysis [19] that determines a WCET estimate whose overestimation should be minimal w.r.t. the program's real (and often unknown) WCET. Besides the timing characteristics specifying the execution time of single instructions, the analysis relies on *flow facts* which define the iteration counts of loops and the recursion depth. One of their main goals is to ensure that the program will terminate. Since embedded systems software spends most of its execution time in loops, their precise specification is of eminent importance for the WCET analysis. State-of-the art timing analyzers like aiT [2] provide a loop analysis trying to find the number of loop iterations automatically. This, however, succeeds only for simply structured loops. For real-world benchmarks, the analysis frequently fails and the user must provide the loop iteration counts manually. The common form of this user annotation is a $min\,/\,max$ interval for each program's loop, defining the lower and upper bounds for the possible number of loop iterations.

The source code of typical embedded systems applications written predominantly in the high-level language C contains a large number of loops whose number of loop iterations depends on a parameter p of function f containing this loop. These functions are typically invoked multiple times from different program points with varying constant arguments used as the function parameter p in the function's loop. The result are different numbers of loop executions depending on a specific function call. This uncertainty has a negative influence on the WCET analysis.

The reason is the insufficient specification of flow facts of these parameter-dependent loops. Due to the different calling contexts of the loop's function f, i. e. calls of function f from different points of the program with possibly different arguments, the effective number of loop iterations may considerably vary. The $min\,/\,max$ interval, however, does not allow to explicitly express all context-dependent

[*] © 2008 by EDAA.

[†] The research leading to these results has received funding from the European Community's ARTIST2 Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008.

loop iterations but must cover all these different calling contexts to guarantee a safe WCET estimate. Hence, the lower bound of such flow facts must represent the globally minimal number of iterations for a particular loop found over the entire program execution, while the upper bound equivalently represents the global maximum. Since the flow fact specification of parameter-dependent loops does not explicitly consider different calling contexts of function f, the resulting WCET estimates are safe but lack precision and are thus highly overestimated.

This leads to the conclusion that the precision of timing information provided by the WCET analyzer significantly depends on the analyzability of the code. Thus, the basic idea of our optimization is not to minimize the real WCET but to improve the WCET estimation by transforming code structures that result in overestimations. A possible solution for a precise worst-case execution time analysis of parameter-dependent loops is the compiler optimization Procedure Cloning [14].

The optimization creates specialized copies of functions invoked with constant arguments by propagating the constant parameter values into the function body, thus achieving constant loop bounds. Hence, it makes different calling contexts explicit and enables a precise specification of loop bounds. The main drawback of the transformation is the resulting code size increase due to the additional function clones which is undesirable for resource-critical systems.

To cope with this problem, we present a novel WCET-driven compiler optimization framework that performs a modified Procedure Cloning focused on the improvement of the WCET estimation. Our algorithm evaluates the increasing precision of the WCET estimation as well as the resulting code size changes during cloning of functions on the worst-case path (called WC path) and successively performs the optimization on that function which promises the highest reduction of overestimation. In the further course, this function will be called *fittest* function. Additionally, a factor specified by the user which defines the maximally permitted code size increase is considered to avoid extreme code size increases. To the best of our knowledge, this is the first approach performing high-level compiler optimizations controlled by WCET information.

Due to both the complexity of many real-world benchmarks and sophisticated processor features (like caches, branch predictions and pipelines), the real WCET is often not known. For this reason, whenever we talk about the WCET in the following, we do not mean the (speculative) program's real WCET but refer to the estimate of the real WCET calculated by the analyzer.

The remainder of this paper is organized as follows: Section 2 presents related work. In Section 3, the standard Procedure Cloning optimization will be briefly presented, followed by the description of our WCET-driven Procedure Cloning in Section 4. Section 5 describes the experimental environment, while results will be presented in Section 6. Section 7 summarizes the contributions of this paper and gives directions for future work.

2. Related Work

In past decades, development of compiler optimizations has concentrated on the average-case execution time (ACET). Recently, the minimization of energy dissipation as 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. Loop Nest Splitting [6, 7] is one of the few examples where the influence of an optimization originally developed for ACET and energy dissipation minimization on WCET was examined.

In [21], a genetic algorithm performing different low-level standard compiler optimizations to the program under test is applied. The objective is to find an effective optimization sequence that yields the largest reduction in the program's WCET. This approach does not exploit information about the WC path, but optimizes the program globally. In [22], a code-positioning optimization driven by worst-case path information was presented. By rearranging the memory layout of basic blocks, branch penalties along the WC path are avoided. The modified code has an improved performance and results in a reduced WCET. Both approaches operate on a low-level IR and their drawback is the lack of a high-level IR within the compiler leading to a costly re-generation of valuable high-level flow facts that are only available at the source code level. Moreover, the considered processor is quite simple as it has a simple pipeline and no caches.

A compiler guided trade-off between WCET and code size for an ARM7 processor was studied by [12]. The authors observed that applications implemented with 16-bit THUMB instructions are smaller but also slower than the same code using the full 32-bit instruction set. 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 for different program sections.

[5] presents a design study for an homogeneous WCET-aware compiler. However, that paper focuses on the overall design of the proposed compiler and concentrates on the integration of a WCET analyzer into the compiler. Since it does not focus on the WCET-awareness of built-in compiler optimizations, it is complementary to this work.

Procedure Cloning has been introduced by Cooper [4] and is nowadays part of many optimizing compilers [16]. This approach was mainly considered in the context of ACET and the main objective was the increase of the

```
int f(int *x,int n,int p) {
 for(i=0;i<n;++i) {
  x[i]=p*x[i];
  if(i==10) {...}
 }
 return x[n];
}

int main(void) {
 //calls of f(x,5,2);
 return f(a,5,2);
}
```

```
int f1(int *x) {
 for(i=0;i<5;++i) {
  x[i]=2*x[i];
  if(i==10) {...}
 }
 return x[5];
}

int main(void) {
 //calls of f1(x);
 return f1(a);
}
```

Figure 1. Example for Procedure Cloning

average-case performance while keeping the resulting code size increase small.

In [14], we studied the benefits of Procedure Cloning on the WCET analysis and showed that the optimization can be exploited for an efficient and precise timing estimation. We applied the standard Procedure Cloning developed to minimize the average-case execution time to real-world benchmarks and pointed out that the WCET estimates could be highly reduced while increasing the code size by up to 300%. In this paper, we extend Procedure Cloning by WCET-aware concepts and modify the optimization's cost function: in contrast to classical cloning aiming at the reduction of the ACET, our optimization explicitly focuses on the minimization of the WCET while controlling the resulting code size increase and possibly omitting cloning of large functions. In addition, we raise the effectiveness by exclusively considering functions on the WC path.

3. Standard Procedure Cloning

In this section, we describe the standard optimization Procedure Cloning. A detailed discussion is given in [14].

Procedure Cloning belongs to the class of inter-procedural compiler transformations where the optimizing compiler generates a specialized copy of the original procedure. Afterwards, the original function calls are replaced by calls to the newly created clones. The optimized code provides a more beneficial basis for aggressive inter-procedural data-flow analyses [4]. Furthermore, cloning often offers the opportunity for improved optimizations, like constant propagation or the elimination of paths through the control flow graph that will be never taken. Figure 1 demonstrates cloning of function *f* for the function parameters *n* and *p* resulting in cloned function *f1* [3, 16].

3.1. Improved WCET

The primary objective of the standard Procedure Cloning is the minimization of the ACET. In [14] we indicate why the optimization can be exploited to make the program code more predictable and thus enhance the WCET analysis. It tackles two issues: the explicit specification of loop bounds and the elimination of infeasible paths.

A code modified by Procedure Cloning makes the calling contexts explicit since function calls with particular constant arguments are represented by specialized clones where variables are replaced by constant values. The results are loops whose number of execution counts do not rely on function parameters but depend on constants and can thus be precisely specified by flow facts.

On the other hand, the optimized code allows the removal of infeasible paths. These are paths within the program control flow that are never executed within a particular scope but may unnecessarily contribute to the timing results leading to overestimated WCETs. They differ from dead code since there may be other scopes (called contexts), where, for the same code, these infeasible paths are executed [8]. An example for such a path is the if-block on the left-hand side of Figure 1. After cloning, compiler data- and control-flow analyses may be applied to remove these never-taken paths and enhance the timing results.

3.2. Increased Code Size

The major drawback of the standard Procedure Cloning is the code size increase due to the additional (cloned) functions. Whenever the optimization finds a candidate, the function will be cloned without taking the code size increase into account. For embedded systems applications with functions having a large number of function parameter and being invoked from different places with different constant arguments this yields significantly enlarged programs with numerous additional function clones. In [14], we have shown that the code size of some benchmarks increased by more than 300%. Thus, the classical Procedure Cloning should be used with caution, and a trade-off between the resulting WCET and the code size increase is required.

4. WCET-Driven Procedure Cloning

The standard Procedure Cloning described in Section 3 is not designed to explicitly focus on the improvement of the WCET estimation for two reasons. First, the optimization is not aware of the worst-case path (WC path) and thus optimizes all functions meeting specified constraints aiming at the reduction of the ACET. That means that functions not lying on the WC path will be also cloned although they do not influence the WCET but increase the code size.

```
 1 Input:   Program $P$, float $maxFactor$
 2 Output:  optimized Program
 3
 4 float $MCS \leftarrow$
    codesize($P$) * $maxFactor$
 5 repeat
 6  list<function> $WCPath \leftarrow$WCETAnalysis($P$)
 7  for all function $f \in WCPath$ do
 8    if ( meetConditions($f$) ) then
 9      codesize($P$)
10      program $P_{copy} \leftarrow P$
11      performCloning($P_{copy}$, $f$)
12      updateLoopBounds($P_{copy}$, $f$)
13      removeInfeasiblePaths($P_{copy}$, $f$)
14      WCETAnalysis($P_{copy}$)
15      codesize($P_{copy}$)
16    end if
17  end for
18  calculateBenefits()
19  function $f_{fittest} \leftarrow$findFittestFunc($MCS$)
20  if ( $f_{fittest} \neq \emptyset$ )
21    performCloning($P$, $f_{fittest}$)
22  end if
23 until ( $f_{fittest} == \emptyset$ )
24 return ( $P$ )
```

Figure 2. WCET-driven Procedure Cloning Algorithm

This is due to the inherent nature of the worst-case execution time estimation. WCET represents the execution time for the longest path within a program's control-flow graph, the worst-case path. After slightly modifying the code, however, the current WC path might switch to another path in the program (this behavior is also called *path switching*). If an algorithm continues to optimize code lying on the obsolete WC path, the transformation will not influence the WCET and is thus ineffective.

Second, the two main properties leading to a reduced WCET estimation as described in Section 3, namely the modification of parameter-dependent loops and the elimination of infeasible paths, are not explicitly exploited but all potential functions are specialized. Again, this results in additional cloned functions that might not be beneficial for a more precise WCET estimation.

Our WCET-driven Procedure Cloning modifies the standard version by focusing on the two issues mentioned above, hence systematically improving the estimation results during the WCET analysis. The underlying algorithm is depicted in Figure 2. It is provided with the original program P to be optimized and a float value $maxFactor$. This factor defines the maximal permitted code size increase for the optimized code compared to the original code size. For example, given the value 2.0 means that the code size must not increase by more than 200%. Hence, the WCET-driven optimization is best suited for WCET minimization in memory-restricted embedded systems. The factor is used to calculate the maximal code size MCS of the optimized program (line 4).

The algorithm consists of three main phases which are described in more detail in the following sections.

4.1. WC Path Determination

As mentioned previously, an optimization can successfully decrease WCET only when it improves the parts of the code that lie on the WC path. To meet this condition, the first step of our optimization is a WCET analysis of the original code with context-independent loop bound specifications, i. e. the bounds of the $min\,/\,max$ intervals depict the global minimal and maximal number of iterations for each loop. The calculated WCET is safe but overestimated.

Before finishing this phase, all functions on the WC path are collected (line 6) for subsequent steps. Also, the total WCET estimation for the original code is collected and will be used in the final phase to determine the fittest function that will be optimized in the returned program.

4.2. Data Collection

In this phase, all functions lying on the WC path, as determined in the previous step, are successively evalutated. In detail, each of these functions is only cloned when it meets one of the three constraints (line 8):
the functions parameter is used

- inside a loop statement to possibly allow a definition of more precise flow facts for the number of loop iterations
- inside a conditional expression so that possibly infeasible paths can be removed
- as an argument in the function's callee. Cloning for this case does not directly influence the WCET estimation but provides constant arguments for the callees that in turn might profit from later cloning.

Any other uses of the parameters are insignificant for an improved WCET and are thus omitted since cloning of these functions would merely result in additional function clones that increase the code size but do not provide a positive effect on the timing estimation.

If one of the conditions is met, the function's original code size is determined (line 9) and a copy P_{copy} of the original program P is created (line 10). At this point, a copy is required since the subsequent Procedure Cloning (line 11) modifies the program. This data collection step is exclusively meant to collect optimization results and it must be ensured that the given program is not permanently modified for following collection steps. The function *performCloning* is handed the copied program and the current function f which is used to identify the corresponding function in P_{copy}.

After cloning, two significant steps for the cloned function are performed (lines 12 and 13): if possible, the flow facts for the function's loops are automatically tightened, i. e. the $min \,/\, max$ intervals are adjusted to the new loop bounds resulting from Procedure Cloning and the constant parameter propagation. This step does not require user interaction but relies on a static loop analyzer which tries to find the number of loop iterations automatically. If the analyzer succeeds, the flow facts will be adjusted to represent tight loop bounds for the transformed code. Otherwise, the flow facts remain unmodified. This leads to the same WCET overestimation as was found in the original function but still represents a safe timing approximation.

A cloned function with non-adjusted loop bounds might even result in a slightly increased WCET. The reason is the execution of the modified code for which the WCET analysis can not profit from tighter flow facts. The changed sequence of executed instructions might have a negative impact on a superscalar pipeline executing multiple instructions simultaniously. Its parallelism can be only exploited when particular combinations of adjacent instructions are fetched by the processor sequentially from memory. After cloning, the original instruction sequence leading to a good pipeline performance might, however, change and results in less opportunities for parallel instruction execution which degrade the program's execution.

The second step is the removal of emerging infeasible paths in the function clones (line 13). This step and the adjustment of flow facts modify the copied program and take the currently considered function f as reference.

For the modified program, the WCET is estimated (line 14) which might, as described in Section 3.1, be smaller than the WCET estimation of the original code from the first phase. Moreover, the total code size of the modified code is calculated allowing the determination of the code size changes w.r.t. to the original code (line 9). At the end of this step and before continuing with the next function $f \in WCPath$, the algorithm knows for each function how Procedure Cloning would influence the WCET estimation and code size if performed on the current function.

It should be noted that it is not possible to perform the WCET analysis exclusively for the currently processed function, but the entire program must be analyzed after cloning to determine the new total WCET (cf. line 6). This is due to processor features like caches and pipelining which influence the timing properties of a particular code fragment based on previously executed code. Thus, a realistic WCET of a function depends on the program history and can not be considered as an independent code part.

4.3. Benefit Calculation

In the final phase, the collected information for the original and the optimized code from the first two phases are evaluated (line 18). For each function to be optimized, the benefit is calculated as follows:

$$benefit_{\text{function}} = \frac{WCET_{original} - WCET_{optimized}}{code\ size_{optimized} - code\ size_{original}}$$

The values $WCET_{original}$, $WCET_{optimized}$, $code\ size_{optimized}$ and $code\ size_{original}$ were calculated in lines 6, 14, 15 and 9, respectively. Functions which yield a declined WCET estimation, i. e. the difference between the WCET before and after cloning for this particular function is negative (e. g. due to a worse pipeline performance as described above), are counterproductive and are thus omitted. To accelerate the optimization, the algorithm keeps track of those functions and they are completely excluded for further iterations. When there is a single function clone that can replace the original function, the size of the optimized code might get smaller resulting in a negative denominator. In that special case we consider the dominator neutral with the value of one.

The function with the greatest benefit is the fittest function whose cloning yields the best result w.r.t. the minimized WCET and the code size. The quotient expresses a trade-off between the influence on the WCET and the code size. For functions with a comparable WCET minimization, the one with the smaller code size increase will be chosen.

Finally, the fittest function among those functions whose benefit was calculated in the previous step, is chosen (line 19). The algorithm guarantees to preserve a maximal code size increase for the optimized code. It searches the function with the greatest benefit and checks if the additional code size increase resulting from cloning of this function will not exceed the permitted maximal code size defined by the parameter MCS. If the code size increase is too large, this function will be omitted and the function with the next largest benefit is considered. If either none of the evaluated functions had a positive influence on the WCET estimation or all exceeded the maximal code size, no fittest function was found and $\emptyset$ is returned.

In contrast, if a fittest function has been found in this phase, this function will be permanently cloned (line 21). This is achieved by running Procedure Cloning on the original program P with function $f_{fittest}$. Any further search for the next function to be cloned will take the so far optimized program P as new code basis which covers all optimized (fittest) functions from the previous iterations.

As long as a fittest function was found (line 23), the algorithm continues its search for the next function to be cloned which further improves the WCET estimation. Otherwise, the algorithm terminates and the so far optimized code is returned (line 24).

A repeated WCET analysis for the complete code after applying Procedure Cloning to the fittest function is mandatory for an effective optimization to take WC path switchings into account (cf. line 6). Therefore, our algorithm can not proceed its transformations on the previously determined WC path functions, but must update this information. Due to the typically relatively small number of possible candidates for cloning, the number of WCET analyses remains small and yields an acceptable analysis run-time as will be indicated in the result section.

5. Experimental Environment

This section describes the choice of benchmarks used to evaluate the influence of our novel Procedure Cloning on the WCET. Furthermore, the benchmarking workflow is described.

To evaluate our WCET-driven optimization, we used three benchmarks from the widely used MiBench suite [9] representing different applications typically found in the embedded systems domain. The first benchmark is MPEG, a motion estimation for frame pictures. EPIC is an experimental lossy image compression utility. GSM is a standard for voice encoding and decoding based on a combination of Time- and Frequency-Division Multiple Access, while G.721 represents a reference implementation of the CCITT for voice encoding and decoding. The fifth benchmark comes from NetBench [15], a benchmarking suite for network processors. It represents the message digest algorithm (MD5) used in the Diffie-Hellman key encryption-decryption mechanism. These benchmarks were chosen since they represent typical applications found in the embedded systems domain.

The workflow is depicted in Figure 3. For the integration of our WCET-driven Procedure Cloning, we use our WCET-aware C compiler for the Infineon TriCore 1796 processor [5]. The general structure of our compiler consists of a high-level intermediate representation, the ICD-C [10], and a low-level representation, called the LLIR [11], which is coupled to AbsInt's WCET analyzer aiT. The components of the compiler which are significant for the realization of the algorithm given in Figure 2 are described in more detail in the following.

The input is a C source code representing the application under test which is manually annotated in the source code with pragmas representing the flow facts for the loop iteration counts in the form of min / max intervals. After parsing the code, the application is transformed into the ICD-C

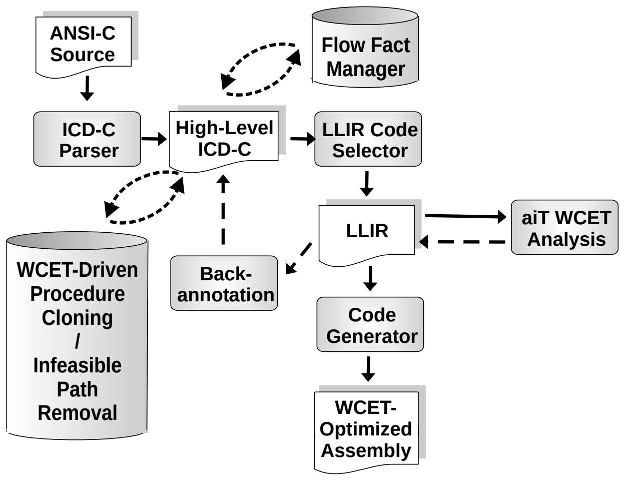

Figure 3. Workflow for WCET-driven Procedure Cloning

IR. Next, the code is passed to the code selector transforming it into the assembly-level LLIR.

The next step is the WCET analysis. For this purpose, a common exchange format between the compiler's backend and the timing analyzer is required. The analyzer aiT works internally with its own intermediate representation called CRL2 [1], storing the application under analysis as well as analysis results. Thus, our compiler translates the LLIR into an equivalent CRL2 representation which is used as input for the subsequent run of aiT. After finishing the WCET analysis, our compiler automatically reads the timing results stored in a temporary CRL2 and imports them back into the LLIR. Hence, the compiler backend is aware of WCET information which can be exploited for optimizations. Readers who are interested in the interface between the compiler and the WCET analyzer are referred to [13, 5]. The workflow up to the WCET analysis is depicted in Figure 3 by solid arrows.

To provide WCET information for Procedure Cloning, the gained data must be transformed from the low- into the high-level IR. This process is called *Back-annotation*. As far as we know, this is the first compiler framework providing highly accurate WCET information from a sophisticated timing analyzer into the compiler front-end. The transformation of WCET information from aiT to our high-level ICD-C IR is marked by dashed lines.

The WCET-annotated ICD-C IR is the starting point for all WCET-aware high-level compiler optimizations. The optimization framework is designed in a modular and generic way and can be easily extended by any high-level optimization. It performs a successive WCET minimization as described in Section 4. For a given optimization, the WCET influence is evaluated for each function separately and the fittest one, determined by an configurable benefit cost function, is chosen for permanent optimization. The framework can also be extended to perform the evaluation on a more fine-grained level like basic blocks. Being aware of the actual worst-case path after a code transformation tackles the path switching problem and thus enables effective optimizations in order to minimize the worst-case execution time. It should be noted that this optimization framework can be used for both improving the precision of the WCET estimation (as is done for Procedure Cloning) and reducing the real program's WCET.

For the presented work, we integrated Procedure Cloning into the WCET-aware optimization framework. After obtaining the initial WCET information as well as the program's code size, the information collection for functions on the WC path is started. This second step of the optimization creates a copy of the original ICD-C IR and successively clones each function meeting the constraints described in 4.2.

After each cloning of a function, the compiler automatically adjusts the flow facts representing the number of loop iterations within the function clones. This task is performed by the *Flow Fact Manager* [18]. This module is coupled to the ICD-C IR and keeps track of the consistency of any defined flow facts. Whenever a high-level optimization influencing the flow facts is applied, the manager updates them accordingly. For example in the case of loop unrolling, the corresponding $min \,/\, max$ interval for the modified loop is adapted. The new loop iterations are determined by a static loop analysis which operates on the ICD-C IR. Moreover, the Flow Fact Manager is coupled to the LLIR for two reasons. First, it takes care to perform a correct translation of flow facts from the high- to the low-level IR. Second, similar to the ICD-C, any flow facts assigned to LLIR components which are modified by an LLIR optimizations, are updated accordingly [1]. In this way, it is always ensured that flow facts reflect the real program behavior. The manager relies on results of a static loop analysis and does not adjust the flow facts when no loop bounds could be determined, thus guaranteeing a safe WCET estimation.

Moreover, the optimization removes any infeasible paths from the high-level IR. After cloning a function, a technique based on Loop Nest Splitting [7] analyses all conditional statements and removes those parts of the code whose condition are always false. This, in conjunction with the previous step yields a code that is more suitable for WCET analysis and allows a more precise estimation.

For the second input parameter of our algorithm $maxFactor$ (cf. Fig. 2) which is a restriction to the maximally permitted code size increase, the factor 2.0 was chosen allowing maximally a doubling of the code size w.r.t. to

[1]Please note that the loop analyzer, the fbw fact manager and the optimization modules for the ICD-C and LLIR are not included in Figure 3 for clarity reasons.

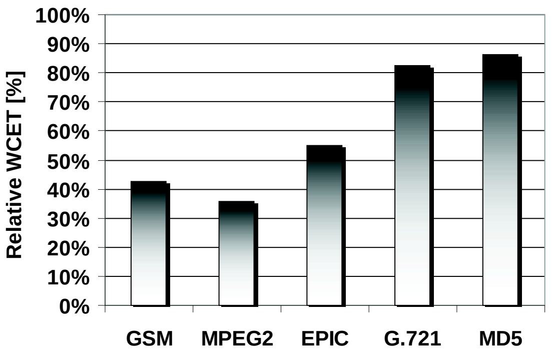

Figure 4. Relative WCET after Cloning

the original code size. This setting is meant to prevent too heavy code size increases. Varying this factor to the values 2.5 and 3.0 provided very similar results as those presented in the next section.

According to the algorithm given in Figure 2, the collection of data and the evaluation of the influence of Procedure Cloning on each function lying on the WC path continues as long as there are further candidates for cloning. This cycle is represented in the workflow in Figure 3 as a loop between the High Level ICD-C and the LLIR/aiT WCET analysis. Finally, the WCET-driven Procedure Cloning generates an optimized IR, which is transformed into an LLIR. The latter is dumped as an assembly code using the code generator. This code is more suitable for a precise WCET analysis and serves as input for the assembler and linker.

For the results provided in the next section, we measured the WCET, the simulated time running a TriCore Instruction Set Simulator on the generated binary, and the run-time of the analysis with and without our WCET-driven Procedure Cloning. For all tests, we disabled the cache by placing the code into non-cachable memory areas. This was done on purpose to avoid unexpected cache effects due to the positioning of the cloned functions. These effects are often difficult to interpret and might lead to falsified conclusions. In this paper we fully concentrated on how transformations performed by Procedure Cloning might be exploited for a more precise WCET analysis. It is well known that different code positioning techniques [17, 20] may significantly influence the program run-time in the presence of caches, so varying placements of the function clones might be the source of different timing results and hence hide the influence of cloning.

6. Results

Worst-Case Execution Time

Figure 4 presents the timing results, with 100% corresponding to the WCET estimation of the original code. As can be seen, our WCET-driven Procedure Cloning achieves WCET reductions of up to 64.2% on real-world benchmarks which will be discussed in detail in the following.

The benchmark GSM contains a function representing a filter for the short term residual signal invoked with varying constants (13, 14 and 120) defining the number of iterations for its loop. Our WCET-driven Procedure Cloning solves this problem by cloning this function for each of the three constant arguments. Due to the improved analyzability, the loops can be exactly specified by our automatic flow fact manager. This has a positive effect on the estimated WCETs. A WCET reduction of 57.3% was achieved.

The second benchmark, MPEG2, contains two functions that were optimized by our WCET-driven optimization. The first function implements the *Fullsearch* algorithm to detect macro-blocks called with different constant values. Some of these arguments are passed to a callee containing parameter-dependent loops. The other constant values instead do not meet the constraints defined in Section 4.2 and are not considered for cloning since it is not expected that they reduce the WCET. At this point, our optimization shows its strength. It performs Procedure Cloning for the parameters that are suitable for WCET reduction while omitting the others. We achieve a WCET reduction by 64.2%. The reason for the strong reduction is the large number of calls to the cloned function. For the unoptimized code with imprecise loop bound specifications, each analyzed loop contributes to the overestimation. In contrast, the classical Procedure Cloning would yield worse results since it would also perform cloning for parameters that do not lead to a WCET minimization but increase the code size.

The WCET for EPIC was decreased by 44.9%. This is due to the code structure containing a large number of nested loops. The image coder benchmark contains a filter function with 32 loops nested up to four times and their number of iteration counts partially depend on function parameters. This function is invoked with different constant values. Again, the WCET-driven Procedure Cloning propagates these values into the loops of the cloned functions making them accessible for precise loop bound specifications. For this benchmark, the power of our optimization was exploited by specifying the maximally permitted code size increase to a factor of 2. This restriction did not allow cloning of all possible candidates but successively chose the fittest functions according to the previously mentioned benefit function. Without this restriction, the WCET reduction would be even larger due to more cloned functions but would also result in a heavy code size increase which might be undesired for embedded systems.

For the last two benchmarks, G.721 and MD5, each benchmark contains a frequently called function being a candidate for Procedure Cloning. After cloning these functions and allowing a more precise loop bound specification in the optimized functions, a WCET reduction for G.721 and MD5 of 17.4% and 13.7%, respectively, was achieved.

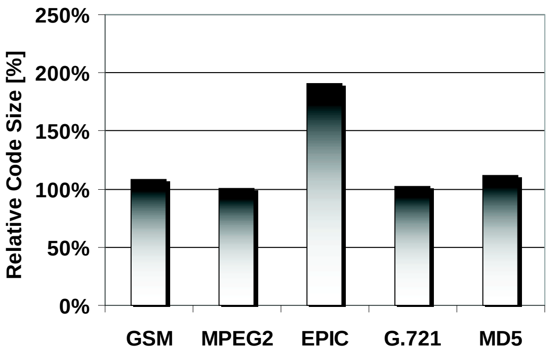

Figure 5. Relative Code Size after Cloning

Code Size

Figure 5 depicts the code size results, with 100% being the original code size of the benchmarks after the infeasible paths were removed. As can be seen, the increase was negligible in most cases, ranging from 0.3% for MPEG2 to 11.7% for MD5. This is due to the fact that the cloned functions were relatively small. One exception is EPIC with 90.9% whose further increase was restricted by the algorithm. The reason is a large function with numerous parameters to be specialized yielding large multiple function clones. Without our extensions to Procedure Cloning, the standard optimization would triple the benchmark's code size.

Simulated Program Run-Time

To examine the impact of our WCET-driven Procedure Cloning on the average-case execution time (ACET), the benchmarks were run in a TriCore cycle-true simulator. For each benchmark, the simulation was performed for the original version and for the fully optimized code that was provided as output after running our algorithm. To have comparable results, the optimization for infeasible paths removal was run for the original benchmark as a pre-pass step. This step is mandatory since it removes conditional expressions which are given at the beginning of infeasible paths and would be always executed in the original code while missing in the optimized code after cloning. Hence, the simulated times reflect the changes to the program run-time due to code modifications performed by Procedure Cloning.

The results are shown in Figure 6. The 100% mark corresponds to the simulated run-time of the original benchmark. As can be seen, our optimization had less influence on the simulated time than on the WCET estimation. On average, a run-time decrease of less than 3% was achieved. The strongest decline was achieved for the benchmark EPIC

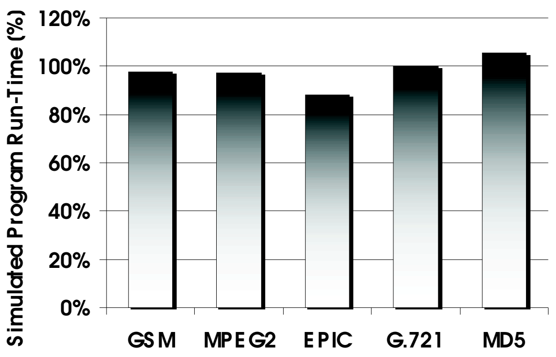

Figure 6. Relative Run-Time after Cloning

with 12.86% where 12 function clones were produced in total, allowing the removal of numerous infeasible paths. This leads to an improved pipeline behavior due to the elimination of conflict hazards.

However, Procedure Cloning might even be disadvantageous for the average run-time. The simulated time of the benchmark MD5 increased by 5.70%. This is a reason of an increased number of instructions. In the original version, the function parameters were stored once in a register at the beginning of the function and this register was re-used during the entire function execution. After cloning, the use of the constant parameters in the function body led to additional MOV instructions since many instructions (especially conditional jumps) do not support large constants as immediate operands. Thus, each of these instructions requires an additional load of the constant into a register before it can be used.

The results of the simulated run-time point out that Procedure Cloning does not improve the code quality remarkably. Compared to the results of the WCET analysis where gains of more than 64% are achieved, it becomes clear that Procedure Cloning is an effective approach in eliminating the WCET overestimation during the timing analysis. It may not improve the real WCET of the program but helps the timing analyzer to achieve tighter WCET estimates that are closer to the real program's WCET.

Optimization Run-Time

The run-time of our optimization strongly depends on the number of functions that are potential candidates for cloning since they are all evaluated w.r.t. their influence on the WCET, i. e. multiple runs of the WCET analyzer are required. For typical benchmarks with few functions, the optimization time ranges from 1 minute to 21 minutes for G.721 and GSM, respectively. Most time was spent on the optimization of EPIC with 294 minutes caused by the large number of evaluated functions. However, the structure of this application is exceptional and does not represent the general program structure found in embedded system applications. On the other hand, WCET optimizations are not performed as frequently as standard optimizations on general-purpose systems but are run once to generate the final production code. Thus, the optimization run-time is not a key issue and longer analysis times are acceptable.

7. Conclusions and Future Work

In this paper, a novel WCET-driven Procedure Cloning is presented. The timing analysis of loops is an inherent source of unpredictability since the number of executions of parameter-dependent loops can be rarely specified precisely leading to high WCET overestimations. The standard compiler optimization Procedure Cloning helps make these loop bounds explicit within the source code, thus allowing a more precise WCET analysis.

We extend the classical cloning by WCET-aware concepts which exclusively optimize functions on the worst-case path. Moreover, we increase the effectiveness of the transformation by successively evaluating the effects on both the WCET estimation and the code size before finally cloning a function. By resolving context-dependent function calls, we eliminate the overestimation during the WCET analysis. Hence, we improve the analyzability of the program and significantly improve the tightness of the WCET estimation. In addition, we tackle the main problem of the standard optimization, the code size increase, by allowing the user to define a maximal code size bound for the final code. For these reasons, the optimization is well suited for resource-critical embedded systems.

The effects of the WCET-driven Procedure Cloning were evaluated with different real-world benchmarks and the WCET estimations were reduced by up to 64.2%. Resulting from the performed transformations, an acceptable code size increase of 22.6% on average was achieved. The decrease of the simulated time, in contrast, was negligible with 3% on average. Our results also show that the optimizations were performed in an acceptable run-time.

In the future we intend to study the influence of Procedure Cloning on instruction caches. A promising idea is to place the cloned functions based on a *closest is best* strategy, i. e. place the function clones close to their callers in memory to avoid cache conflict misses. Moreover, we plan to improve our loop analysis employed to revise flow facts which specify loop bounds within the cloned function. This would result in improved WCET reductions since our current loop analysis is not capable of finding iteration counts for some classes of loops. Another idea is to perform Procedure Cloning exclusively for the WCET analysis to obtain more precise estimations but to not apply the transformations to the final code to avoid code size increases.

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. CRL Version 2. *http://www.absint.com/artist2/doc/crl2*, 2007.

[2] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore. 2007.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[4] K. D. Cooper, M. W. Hall, and K. Kennedy. A Methodology for Procedure Cloning. *Computer Languages*, 19(2):105–117, 1993.

[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 and P. Marwedel. Control Flow driven Splitting of Loop Nests at the Source Code Level. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10410, Munich, 2003.

[7] H. Falk and M. Schwarzer. Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, October 2006.

[8] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Washington, DC, USA, 2001.

[10] Informatik Centrum Dortmund. ICD-C Compiler framework. *http://www.icd.de/es/icd-c*, 2007.

[11] *ICD Low Level Intermediate Representation backend infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund, 2007.

[12] 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 *SCOPES '04: Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems*, pages 244–258, Amsterdam, 2004.

[13] P. Lokuciejewski. *A WCET-Aware Compiler. Design, Concepts and Realization.* Vdm Verlag Dr. Müller, 2007.

[14] P. Lokuciejewski, H. Falk, M. Schwarzer, P. Marwedel, and H. Theiling. Influence of procedure cloning on wcet prediction. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 137–142, New York, NY, USA, 2007.

[15] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: a benchmarking suite for network processors. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42, Piscataway, NJ, USA, 2001.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[17] A. D. Samples and P. N. Hilfi nger. Code reorganization for instruction caches. Technical Report UCB/CSD-88-447, EECS Department, University of California, Berkeley, Oct 1988.

[18] D. Schulte. Modeling and transformation of fbw facts within a wcet optimizing compiler (in german). Master's thesis, Technical University of Dortmund, Dortmund, Germany, May 2007.

[19] S. Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[20] H. Tomiyama and H. Yasuura. Code placement techniques for cache miss rate reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):410–429, 1997.

[21] W. Zhao, P. Kulkarni, D. Whalley, et al. Tuning the WCET of Embedded Applications. In *RTAS '04: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 472, Washington, DC, USA, 2004.

[22] W. Zhao, D. Whalley, C. Healy, et al. Improving WCET by Applying a WC Code-Positioning Optimization. *ACM Transactions on Architecture and Code Optimization*, 2(4):335–365, Dec 2005.