Tighter WCET Estimates by Procedure Cloning*

Paul Lokuciejewski, Heiko Falk, Martin Schwarzer, Peter Marwedel
Embedded Systems Group, Dept. of Computer Science 12
University of Dortmund
D-44221 Dortmund, Germany
{Paul.Lokuciejewski | Heiko.Falk | Martin.Schwarzer | Peter.Marwedel}@udo.edu

ABSTRACT

Embedded software spends most of its execution time in loops. To allow a precise static WCET analysis, each loop iteration should, in theory, be represented by an individual calling context. However, due to the enormous analysis times of real-world applications, this approach is not feasible and requires a reduction of the analysis complexity by limiting the number of considered contexts. This restricted timing analysis results in imprecise WCET estimates. In particular, data-dependent loops with iteration counts depending on function parameters cannot be precisely analyzed. In order to reduce the number of contexts that must be implicitly considered, causing an increase in analysis time, we apply the standard compiler optimization *procedure cloning* which improves the program's predictability by making loops explicit and thus allowing a precise annotation of loop bounds. The result is a tight WCET estimation within a reduced analysis time. Our results indicate that reductions of the WCET between 12% and 95% were achieved for real-world benchmarks. In contrast, the reduction of the simulated program execution time remained marginal with only 3%. As will be also shown, this optimization only produces a small overhead for the WCET analysis.

1. INTRODUCTION

Real-time systems acting in a safety-critical environment must meet timing constraints imposed by the system specifications which are based on the knowledge of the worst-case execution time (WCET). This key parameter can be calculated in several ways.

One technique is the static WCET analysis determining upper timing bounds of a program. Besides the hardware timing characteristics specifying the execution time of single instructions, the analysis relies on *flow facts*. They can be divided into two classes, namely the mandatory ones and those that improve the timing analysis. The mandatory flow facts serve as restrictions to overcome the halting problem by defining the iteration counts of loops and the recursion depth and thus to ensure that the program will terminate [11]. The second class contains information used to describe the program structure more accurately, in particular flow facts identifying *infeasible paths* that are potentially executable according to the control flow graph but are not feasible due to the program semantics and the given input data. This information is not mandatory to obtain a safe WCET estimate but might improve its tightness [7].

Embedded software spends a large amount of its execution time in loops. Thus, the WCET analysis of loops is

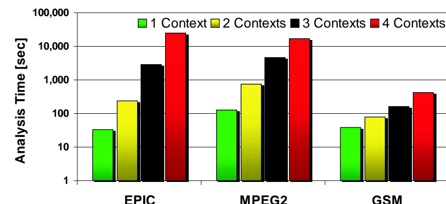

Figure 1: Context-Sensitive WCET Analysis Times

crucial and strongly relies on precise flow facts describing the number of loop iterations. To improve the analyzability of loops, each single iteration is represented by a context to explicitly model a particular state of the loop execution. When the number of contexts during the WCET analysis is not restricted and each loop iteration is assigned a separate context, the resulting WCET estimates are tight.

Real-world applications, however, are too complex to allow an efficient WCET analysis where each emerging context is taken into account. Figure 1 addresses this issue. For three complex benchmarks from the commonly used *MiBench* suite [8], a WCET analysis was performed using aiT [1], a static WCET analyzer developed by AbsInt. The number of distinguished contexts was varied from one to four for all benchmarks. As can be seen, the analysis time increases rapidly. For example, the analysis time for the MPEG2 benchmark took 125.79 seconds for one and 16723.22 seconds for four distinguished contexts. Despite these long analysis times, the WCET estimates are still highly overestimated since the small number of considered contexts was not sufficient for a precise timing analysis.

To cope with the exploding analysis time, the number of contexts must be restricted. A restriction to n contexts means that the first $n-1$ loop iterations are considered separately while the remaining loop iterations are summarized into a single context. This has a negative impact on the WCET estimation since loops cannot be analyzed precisely and worst-case assumptions must be done for the summarized context to guarantee safeness.

In particular, these assumptions might lead to large WCET overestimations for typical embedded systems applications which contain functions with loops whose number of iterations depend on function parameters. When the function is called multiple times with different arguments controlling the number of loop iterations, each loop execution requires individual loop bounds. State-of-the-art timing analyzers handle this issue by performing a loop analysis to determine explicit numbers of loop iterations for each loop execution. The approach enables a tighter WCET analysis since each loop contributes with its realistic runtime to the global WCET.

However, the detection of loop bounds succeeds only for simply structured loops. In order to provide the essential in-

*This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/)

formation about the unrecognized loop iteration counts, the user must provide these annotations manually. This is done by defining an interval for each loop, modeling lower and upper bounds [6]. Any iteration counts lying in between the boundaries cannot be explicitly specified. Hence, exact loop bounds cannot be derived for a particular loop execution from these user annotations. This ignorance is an inherent source of imprecision since WCET analyzer must assume the worst case and consider each loop execution exclusively with the specified upper loop bounds regardless of their real behavior. To solve this problem, we propose procedure cloning to make loops more explicit.

Procedure Cloning (also known as *Function Specialization*) is a standard compiler transformation [10] optimizing functions called with different constant arguments. These functions are cloned, the constant parameters are removed from the parameter list and their occurrence is directly replaced by the constant values. The well-known benefits concerning the average-case execution time (ACET) are a simplified code that offers opportunities for further optimizations like constant folding as well as a reduced calling overhead.

Compared to the marginal improvements of the ACET, procedure cloning can be highly efficient for the WCET analysis since it makes different function call contexts explicit and thus allows a separate loop annotation for each context. This paper is the first to exploit this compiler optimization to improve a programs predictability. Loops that are data-dependent of constant function arguments now become better analyzable since their original variable loop bounds are replaced by constants allowing an explicit annotation within each cloned function. Hence, the optimized code offers the opportunity to calculate tighter upper timing bounds in an acceptable amount of time while the number of contexts remains restricted.

In addition, the tightness of the WCET estimates can be further improved by the elimination of infeasible paths. In the non-optmized code, loops might contain paths that are only traversed for a small number of loop executions. A safe WCET analysis must assume the worst case for the summarized context and also considers these paths even though they are never taken in reality for this particular calling context.

The rest of this paper is organized as follows: Section 2 describes related work. In Section 3, the concepts behind calling contexts and the resulting loss of analysis precision due to their restriction are discussed. Procedure cloning and its benefits for WCET analysis are presented in Section 4. Section 5 describes the experimental environment, followed by benchmarking results in Section 6. Section 7 summarizes the contributions of this paper and gives directions for future work.

2. RELATED WORK

Procedure cloning has been introduced by Cooper [3] and is nowadays part of many optimizing compilers [10]. Up to now, this approach was considered in the context of ACET and the main objective was the increase of the average-case performance while keeping the resulting code size increase small. This paper, in contrast, studies the benefits of procedure cloning on the WCET analysis and exploits them for an efficient and precise timing estimation.

In the previous decades, research on compiler optimizations mainly focused on the reduction of the average-case execution time. With the growing importance of embedded systems, other criteria like energy dissipation or code size become significant and various approaches have been presented describing how to use compiler optimizations to reduce them. However, there was little research on the minimization of the WCET by a compiler. The approach described in [5] is one of the few examples where a compiler optimization developed for ACET reduction was employed to study its influence on the WCET.

In [12], a code-positioning optimization driven by worst-case path information was presented. The algorithm rearranges the memory layout of basic blocks to avoid branch penalties along the WC path. The modified code has an improved pipeline performance and results in a reduced WCET.

A compiler guided trade-off between WCET and code size for an ARM7 processor was studied by [9]. 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.

A design study for a homogeneous WCET-aware compiler was presented in [4]. The compiler generates input data for a timing analyzer, starts the WCET analysis and finally imports WCET-relevant information back into its data structures. With the compiler knowledge about the program and the gained timing data, WCET-aware compiler optimizations can be realized. The focus of that paper was the design of a WCET compiler framework and can be considered complementary to this present work.

3. RESTRICTED CONTEXTS

This section discusses the concepts of contexts with a special focus on the analysis of loops. It shows why restricted contexts can be fatal for a WCET analysis and the resulting loss of tightness will be illustrated by an example.

The use of contexts is a common approach for static program analyses. It enhances the analysis of functions by representing each function call as an individual context. Thus, all program details like the passed arguments can be explicitly specified. To depict a function call invoked within another function, its calling history must be considered. This is achieved by specifying this particular context with a call string describing the sequence of functions called previously. Loops resemble recursive functions calls since they invoke themselves. To exploit the concepts of contexts for loops, and thus to enhance the precision of their timing and stack analysis, loops are transformed into dedicated functions calling themselves. After this loop transformation, each loop iteration is assigned an unique context.

As indicated in Section 1, real-world applications possesse a structure too complex to analyze each emerging context. In particular, nested loops can cause a rapid increase of contexts (*state explosions*). For example, the loop nest of three nested loops with ten iterations each is represented by 1000 contexts. Such large numbers of contexts make a static analysis infeasible since both a vast amount of memory resources and computation time are required. The only way out is to reduce the complexity of the static analysis by restricting the number of distinguished contexts.

In general, restricting the number of contexts simplifies the calling history of function calls by restricting the length of the call string. A definition of maximally n distinguished contexts means that the first $n-1$ functions calls will be analyzed separately with all their details concerning the system properties at this particular point of the program. All

following calls, in contrast, are summarized into the last context. Thus, all precise calling information like the encountered value ranges during a particular call cannot be explicitly expressed. The result is an inappropriate input data for the value analysis required for a successful cache and pipeline analysis.

Besides the analysis of function calls, an enormous loss of tightness may also occur when loops are annotated manually and iterations are summarized into single contexts. The loss of tightness arises from the mandatory conservatism to guarantee the safeness of the timing results. If a loop is part of the WC path, its contribution to the global WCET is calculated as follows: the maximal iteration count specified by the user annotations is multiplied by the local WCET of the loop body and added to the WCET for the evaluation of the loop's termination condition.

This entails two potential problems. First, program functions containing loops whose loop bounds depend on the function parameters might be invoked with varying arguments. The result are function loops with different iteration counts and thus have individual WCETs. Due to the limited user annotations that simply define the lower and upper loop bounds, the real number of iterations cannot be derived. A conservative static timing analysis must expect the worst case and assumes for the summarized context that the loop is executed as often as defined by the user maximum. Second, the loop body might consist of different paths each with an individual execution time. The WCET analysis must again proceed pessimistically and assume that all loop iterations summarized into the last context traverse the WC path.

3.1 Exemplary loss of tightness

To emphasize the loss of precision, an exemplary scenario is presented in the following. Assume that the function f from the left-hand side of Figure 2 is invoked twice with the values 100 and 5 for the parameter n. Furthermore, it is assumed that the WCET of the path containing the if-block is twice as large as when this block is omitted and that the number of contexts is restricted to $k < 5$. With the safe user annotation defining the upper loop bound to be 100, the first call to the function f results in an overestimation since for the summarized context it is assumed that the remaining $100 - k$ loop iterations contribute with the local WC path which includes the if-block. The overestimation is even worse for the second call to f. Here, for the summarized context the static analysis must assume that the loop is not iterated k, but 100 times and that the remaining $100 - k$ loop iterations traverse the if-block being the WC path. The result is a safe but also unacceptably large. Due to its immense overestimation, the analysis result represents an upper timing bound that does not expresses the actual worst-case behavior of the given program.

To cope with the dilemma of shortening the analysis time by restricted contexts but still obtaining tight WCET estimations, the standard compiler optimization procedure cloning can be expoited. We will show in the next section how it improves the predictability by generating a program structure that allows a precise annotation of loop bounds.

4. PROCEDURE CLONING

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

```
int f(float *x, int n, int p) {
 for (i=1; i<=n; i++) {
  x[i] = pow(x[i], p);
  if (i==10) {...} }
 return x[n]; }

int main(void) {
 return f(a, 5, 2); }
```

```
int f1(float *x) {
 for (i=1; i<=5; i++)
  x[i] = x[i]*x[i];
 return x[5]; }

int main(void) {
 return f1(a); }
```

Figure 2: Example for Procedure Cloning

more beneficial basis for aggressive inter-procedural dataflow analyses [3]. On the other hand, cloning often offers the opportunity for improved optimizations, particularly for constant propagation and folding, copy propagation and strength reduction. Also, entire paths might be eliminated when cloning yields conditions that can be evaluated by the compiler as always false and thus be never executed.

Figure 2 demonstrates cloning of function f allowing improved optimizations across function call boundaries [2, 10]. Replacing the function parameters n and p by the constants 5 and 2, respectively, offers a significant amount of optimization potential for the cloned function *f1*. First, applying *strength reduction* allows the replacement of the expensive call to function *pow* by a multiplication. Second, the propagated constant value of n results in a simplified control flow graph. By exposing the value range of the loop induction variable i, it is known at compile time that the condition *(i==10)* will never become true. Thus, this infeasible path can be eliminated yielding a smaller number of instructions and a better pipeline behavior due to the reduced number of control hazards. Last but not least, the calling overhead is reduced. The decreased number of passed arguments minimizes the number of required instructions for both the caller and the callee.

Besides the improvements concerning the program runtime, the optimization has one drawback. Each specialized copy of the function body increases code size. In general, it is also not always permitted to remove the original function even if it is not called anymore in the optimized program. On general purpose systems there is no guarantee that this function might be called from another compilation unit not considered in the current optimization course and its removal would be illegal. In the domain of embedded systems, this restriction is usually not given and the removal of original functions can be performed more aggressively. The designer knows in advance what software will be running on the system and can thus definitely determine the functions never called from other modules than the one they are located in. These original functions can be removed after cloning without endangering the systems consistency.

Hence, this compiler optimization should be used with caution, and a trade-off between the resulting speed-up and the increased code size, especially in the domain of embedded system's with restricted memories, should be taken into account.

4.1 Selection of functions to be cloned

There are different strategies to define how extensively procedure cloning should be performed. Two factors are relevant for the optimization. First, the maximum size of the function permitted to be cloned must be specified. This parameter can, for example, be defined by the number of source code expressions found within the function. All functions that exceed this parameter are omitted and not considered for procedure cloning since they may possibly result in a too large code size increase.

The second factor guides the choice of functions to be cloned by setting constraints on the occurrence of the constant arguments. It defines how frequently a particular constant argument must occur within all calls of the function to be cloned. For example, the user might specify that constant argument values must be present in more than half of all function calls. When this frequency is not reached, it will not be considered for optimization and the function will not be cloned for this parameter. If the code size increase is crucial, the number of additionally generated functions must be kept minimal. The only candidates for cloning are functions that are called most of the time with the same constant argument. The extreme case is the choice of functions that are always invoked with the same constant value for a particular function parameter.

Procedure cloning is performed in three stages where each function is analyzed separately. In the first step, constant arguments and the number of their occurrences for each function parameter are collected. Hereafter, the collected arguments that do not meet the specified frequency are removed and omitted for procedure cloning. This is done by counting all function calls the considered argument is used in and comparing it to the number of parameter occurrences from the previous step. In the final stage, all constant arguments that were not removed are used for procedure cloning. The original function is cloned and assigned a unique function name. The specialized argument is removed from the parameter list and directly propagated into the code by replacing the parameter variables by the constant value. Finally, the original function calls within the source code are redirected to the cloned functions.

4.2 Improvements to the WCET

In addition to the previously mentioned pure code optimizations that yield a better code quality, procedure cloning yields a program structure that strongly improves the WCET analysis by making the code more predictable. It tackles the two major problems discussed in Section 3: the explicit specification of loop bounds and the elimination of infeasible paths that may otherwise contribute to the WCET for the sake of safeness. Both contributions of procedure cloning enhance the tightness of the estimates since they result in a more accurate description of the program behavior.

Typical embedded software is loop-dominated. As studies on MiBench benchmarks [8] pointed out, many loops are located in functions and their number of iterations is often specified by function parameters as shown on the left-hand side of Figure 2. These functions, in turn, are called multiple times with varying constant arguments resulting in strongly deviating execution times spent in the loops.

To statically analyze these loops, the timing analyzer must be provided manually with loop iteration counts. To preserve WCET safeness, the loops are annotated with the maximal number of iteration counts the loop is ever executed with, i. e. the annotations must represent the global maximum of iterations for this loop.

These loose loop annotations can be specified more precisely after procedure cloning. When a function is called multiple times with varying constant values that dictate the upper loop bound, this is exploited by the transformation (see right-hand side of Figure 2). The variables in the specialized functions are replaced by the individual constants and thus provide clones that are *dedicated* to individual loop executions. The new user loop annotations can focus on each specialized function explicitly and annotate their loops more realistically. During WCET analysis with restricted contexts, these loops contribute with their corrected maximal number of iterations. Thus, the transformation aims at making the code more predictable.

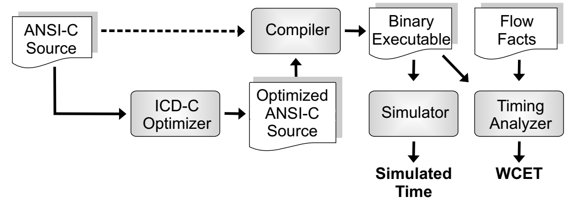

Figure 3: Workflow of Procedure Cloning

Yet another code simplification has a positive effect on the tightness of the WCET estimates. Loops often consist of multiple paths. Some paths may have the longest execution time (WC path in the loop) but are never executed due to unfulfilled conditions. A conservative timing analysis with restricted contexts must assume the worst-case scenario where each loop iteration represented by the summarized contexts goes through the WC path. After procedure cloning, these infeasible paths can be eliminated in the specialized functions.

This is illustrated by an example. As shown in function *f1* on the right-hand side of Figure 2, the path through the *if*-block is never traversed for parameter $n = 5$. Compiler data- and control-flow analyses are capable of detecting conditions that are evaluated as being always false and remove them from the control-flow graph. Thus, these infeasible paths are not taken into account during WCET analysis and don't unnecessarily contribute to the estimated upper timing bounds.

In the following sections, procedure cloning transformations described in this section are applied to real-world benchmarks and their improvements concerning the WCET estimates are presented.

5. EXPERIMENTAL ENVIRONMENT

This section describes the choice of benchmarks used to evaluate the influence of procedure cloning on the WCET. Furthermore, the benchmarking workflow is described.

The benchmarks come from the widely used MiBench suite representing different applications typically found in the embedded systems domain. The first benchmark is EPIC, an experimental lossy image compression utility. MPEG2 is a motion estimation for frame pictures, while GSM represents a speech compression.

All measurements were performed for two different 32bit processors. The first was an Infineon TriCore 1796 microcontroller. The other was an ARM7TDMI that supports two different instruction sets: a full set of 32bit instructions and the so called THUMB instruction set consisting of 16bit instructions. Both modes were exploited for the evaluation of the results.

The workflow is depicted in Figure 3. Two different binary executables are generated. One derived from the original code that is used as reference object (marked with the dotted line). The other binary is the resulting program after procedure cloning and the standard optimizations constant folding, constant propagation and dead-code elimination [5] to remove infeasible paths. The optimizations are automatically performed by a source-to-source optimizer (*ICD-C optimizer*). Our parameters for procedure cloning as described in Section 4.1 were a maximal function size of 2000 expressions and a frequency of 50% (constant argument to be cloned must occur in at least half of all function calls). The output of the optimizer, the optimized source code, is

provided as input for the compiler. Depending on the architecture under test, different commercial compilers were employed. For the TriCore processor the tricore-gcc was used. For the ARM7TDMI the compilers armcc and tcc for the ARM and THUMB mode, respectively, generated the binary executable.

In the last test phase, the binaries are passed to the cycle-true simulator to obtain the simulated processor cycles representing the program execution time for a typical input data set. In addition, the binary executables are passed to the timing analyzer aiT together with the configuration file containing the manually generated information about the loop bounds (flow facts). The result is the WCET for the evaluated TriCore and ARM instructions sets for both the original and the optimized code. Due to the complexity of the timing analysis as indicated in Figure 1, the number of contexts was restricted to 1 (no remarkable improvements were observed for two or three contexts).

6. RESULTS

WCET

Figure 4 depicts the relative WCET for the optimized code with respect to the WCET estimated before procedure cloning (corresponds to 100%). As can be seen, significant WCET reductions of up to 95% were achieved. In the following, the transformations performed by the ICD-C optimizer are briefly discussed for each benchmark and the resulting improvements are pointed out.

The WCET for the EPIC benchmark decreased by 94.61% for the TriCore processor. Similarly remarkable improvements were achieved for the ARM processor, namely 95.72% for the ARM mode and 95.65% for the THUMB mode. This is due to the code structure containing a large number of nested loops. The image coder benchmark contains a function that is highly appropriate for cloning. It is a filter containing 32 loops nested up to four times, and their number of iteration counts partially depends on the function parameters. Furthermore, the function is called six times with different constant values. After procedure cloning, each function call is specialized. The passed constants are propagated and in some cases explicitly define the upper loop bounds. The result is tighter *min / max* intervals for each loop execution e.g. $[1 \dots 15]$ (in non-optimized code) becomes $[1 \dots 1]$ after the transformation, meaning that the timing analyzer can assume one loop iteration in contrast to the pessimistic assumption of 15 iterations.

The benchmark MPEG2 contains two functions that were cloned. The first function implements the *Fullsearch* algorithm to detect the motion of macro-blocks. It is called with two different constant values defining the height of the image block. Within this function, another procedure is called computing the distance between these blocks. It is invoked with the same block height constants as passed to its caller. These values are used to control the number of iteration counts for multiple loops. The source-to-source optimizer performs cloning for each of these functions. The result is a transformed code that has a dedicated version of the Fullsearch implementation for each block size. The loop bounds in the nested function can again be defined more precisely. As for EPIC, the timing analysis of the loops becomes better analyzable and thus more predictable. This is confirmed by the benchmark results. For TriCore, the WCET after procedure cloning was reduced to 70.08% compared to the unoptimized code. Similar improvements were gained for the ARM processor: the worst-case execution time was reduced to 66.75% and 66.55% for the ARM and THUMB

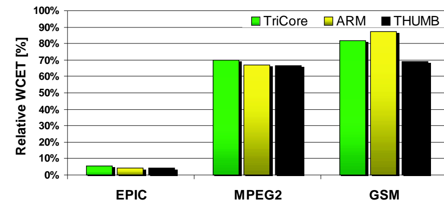

Figure 4: Relative WCETs after Procedure Cloning

modes, respectively. The reason for the strong reduction is the large number of function calls for the nested function. In total, it is called more than 1.4 million times. For the unoptimized code with imprecise loop bound specifications, each analyzed loop contributes to the overestimation.

GSM, the last evaluated benchmark, contains a function representing a filter for the short term residual signal. The function is called with strongly varying constants (13, 14 and 120) defining the number of iterations for its loop. Without procedure cloning, the designer must specify the loop bounds safely, and annotate the maximal number of loop iteration with 120. Obviously, for all calls with the constants 13 and 14, the timing results show an overestimation since the timing analyzer assumes 120 loop iterations. Procedure cloning solves this problem by cloning this function twice, one specialized version for the constant 13 and one for 120. Due to the small number of occurrences of the constant 14, this argument was not considered for specialization. The original version is kept and handles calls with the constant 14. Due to the improved analyzability, the loops can be exactly specified by the system designer. This has a positive effect on the estimated WCETs. Reductions from 12.73% (ARM mode) up to 31.03% (THUMB mode) with regard to the WCET of the non-optimized code were achieved.

The significant WCET reductions after procedure cloning comes basically from the possibility to tighten the *min / max* intervals specifying the loop bound iterations. Another reason for the success of this compiler transformation was the fact that the specialized functions were part of the WC path. Otherwise, their optimization would have had no effect on the WCET. As discussed in Section 4.2, the elimination of infeasible paths might also have positive effects on the timing results. However, for the benchmarks considered in this paper, they were marginal as comparisons between the original and optimized code indicated. Although some paths could be eliminated in the cloned function, they did not improve the WCET results since they had not lied on the WC path and were thus irrelevant for the timing analysis.

Simulated Program Execution Time

To examine the impact of procedure cloning on the ACET, the simulated execution times for typical program input data sets of the original and optimized code were compared. The improvements were negligible. For the EPIC benchmark, the simulated execution time even slightly increased between 0.02% and 3% for TriCore and THUMB mode, respectively. For MPEG2, the optimization gain was between 0.21% (for TriCore) and 2.6% (for the THUMB mode). No improvements were achieved for the GSM benchmark executed on the ARM7TDMI, and a minimal simulated time reduction of 0.01% was observed for the TriCore processor. The minimal degree of the execution time reduction came from the fact that the cloned functions did not provide additional opportunities to further improve the code by the performed source-to-source optimizations, i. e. the newly created functions did not allow to additionally simplify the code with the performed optimizations like constant folding.

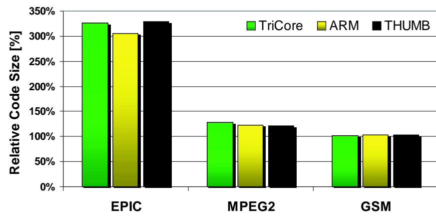

Figure 5: Relative Code Sizes after Cloning

Code Size

Finally, the code size is briefly examined. As mentioned in Section 4, the code size increase is a drawback. Each cloned function increases the code size in particular when the functions are large or multiple specialized copies of a function are created. In Figure 5, the relative code size is shown. 100% correspond to the code size of the benchmarks before procedure cloning. The code size of the optimized EPIC benchmark rose to more than 300% for all instruction sets since a large function with 32 nested loops was cloned six times. However, this increase is acceptable since the absolute code size of the optimized code remains small, namely 21 kilobytes. For MPEG2, the code size increased between 22.0% and 28.94% for TriCore and ARM7TDMI in the THUMB mode, respectively. Although procedure cloning created multiple specialized copies, they all were relatively small so that the code size increase was acceptable. For the GSM benchmark, the code size increase was negligible and achieved its highest value of 3.28% for the ARM mode. Again the reason is that just few and small functions were specialized.

Runtime of the WCET Analysis

To evaluate the impact on the complexity of the WCET analysis, all benchmarks were analyzed in their original and optimized version. The runtime of the timing analyzer aiT on an AMD Sempron 3000+ with 2 GB RAM on average increased by 14% for the analysis of the optimized code by procedure cloning compared to the analysis of the original code. The reason is a larger control-flow graph containing more functions than the original one. However, concerning the large WCET improvements, the increased analysis runtime is acceptable.

7. SUMMARY AND FUTURE WORK

This paper solves the problem of how to preserve the tightness of WCET results for programs with data-dependent loops that require a limited analysis with a restricted number of calling contexts. The analysis of these loops is an inherent source of unpredictability since their number of loop iterations can be rarely specified precisely. The resulting WCET is heavily overestimated. We propose procedure cloning as an approach to improve the analyzability and thus to make the code more predictable. The optimized code allows an explicit specification of loop bounds and furthermore advances the elimination of infeasible paths for tighter WCET results.

The effects of procedure cloning were evaluated with real-world benchmarks from the MiBench suite. The results emphasize the effectiveness of procedure cloning, a WCET reduction between 12% and 95% was achieved. In contrast, the simulated program execution time for the optimized code hardly changed after the optimization. The results also show that the optimization implies only a small overhead for the WCET analysis runtime. Thus, procedure cloning is best suited to improve the analyzability of real-world applications that require a reduced analysis complexity.

In the future, we plan to incorporate a WCET-aware C compiler into the workflow by replacing the commercial compilers. This would enable an improved exploitation of procedure cloning. Currently, the compiler is not aware of any WCET information and employs heuristics to improve the average-case execution time. Within the WCET-aware compiler, cloning could be guided by data provided by the timing analyzer and primarily functions on the worst-case path could be aggressively optimized. The integration into a WCET-aware compiler also offers the opportunity to perform a trade-off between the improvements concerning the WCET and the resulting code size. Thus, to meet the restrictions on the code size defined by the system, the compiler could evaluate the functions that produce the best gain after specializing and exclusively optimize them. Furthermore, we want to study if it might be possible to perform procedure cloning virtually, i. e. to optimize the program exclusively for the WCET analysis and keep the original code structure for the generation of the binary executable. Hence, we would be able to precisely annotate the loops and avoid an code size increase in the final code.

Acknowledgments

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

8. REFERENCES

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

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

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

[4] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *ESTIMedia'06*, pages 121–126, 2006.

[5] H. Falk and M. Schwarzer. Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In *ESTIMedia'06*, pages 115–120, 2006.

[6] C. Ferdinand, R. Heckmann, H. Theiling, and R. Wilhelm. Convenient User Annotations for a WCET Tool. In *WCET'03*, pages 17–20, 2003.

[7] 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*, pages 57–66, 2006.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC'01*, pages 3–14, 2001.

[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 *SCOPES'04*, pages 244–258, 2004.

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

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

[12] W. Zhao, D. Whalley, C. Healy, et al. WCET Code Positioning. In *RTSS'04*, pages 81–91, 2004.