Influence of Procedure Cloning on WCET Prediction [*]

Paul Lokuciejewski, Heiko Falk,
Martin Schwarzer, Peter Marwedel
Computer Science 12
University of Dortmund
D-44221 Dortmund, Germany
FirstName.LastName@udo.edu

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

ABSTRACT

For the worst-case execution time *(WCET)* analysis, especially loops are an inherent source of unpredictability and loss of precision. This is caused by the difficulty to obtain safe and tight information on the number of iterations executed by a loop in the worst case. In particular, data-dependent loops whose iteration counts depend on function parameters are extremely difficult to analyze precisely. Procedure cloning helps by making such data-dependent loops explicit within the source code, thus making them accessible for high-precision WCET analyses.

This paper presents the effect of procedure cloning applied at the source-code level on worst-case execution time. The optimization generates specialized versions of functions being called with constant values as arguments. In standard literature, it is used to enable further optimizations like constant propagation within functions and to reduce calling overhead.

We show that procedure cloning for WCET minimization leads to significant improvements. Reductions of the WCET from 12% up to 95% were measured for real-life benchmarks. These results demonstrate that procedure cloning improves analyzability and predictability of real-time applications dramatically. In contrast, average-case performance as the criterion procedure cloning was developed for is reduced by only 3% at most. Our results also show that these WCET reductions only implied small overhead during WCET analysis.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Performance

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

1. INTRODUCTION

Embedded systems often must meet real-time constraints. The correctness of such a system depends on both the logical results and the time at which the results are produced. Besides safe real-time systems, the market demands high performance, energy efficient and low cost products. Without knowledge about the worst-case timing, designers tend to oversize hardware to guarantee the safeness of the real-time system. Knowing the worst-case execution time enables to use a hardware platform tailored towards the software resource requirements significantly reducing production costs while still guaranteeing the safeness of the system.

Static WCET analysis [9] relies on *flow facts* specifying iteration counts of loops and recursion depths. They serve as restrictions to overcome the halting problem to ensure that the program will terminate. Since typical embedded applications spend most of their execution time in loops, flow facts for loops have a significant impact on an application's WCET. Modern timing analyzers like e. g. aiT [1] include a loop analyzer automatically computing such flow facts for loops. However, this only succeeds for a limited class of loops so that most of the loops found in real-life applications are not analyzed. As a consequence, flow facts for loops not recognized by the timing analyzer have to be specified manually by the designer. Usually, this is done by specifying a $min\,/\,max$ interval for each loop of an application, modeling lower and upper bounds of the possible number of iterations of a loop.

Today, software development for embedded systems uses high-level languages like C. The C source codes of typical embedded real-time applications often contain a number of loops whose number of iterations depends on a parameter p of the function f containing this loop. In addition, such a function f can be called from various places within the code of the entire application with different values for p. These observations on real-life code have a very negative influence on the WCET computed by a timing analyzer.

This is due to the flow fact specification of such a data-dependent loop. Since the loop's function f is called from many places with possibly different arguments, the effective number of loop iterations within f can vary considerably, depending on the context how and with which parameters f is called. However, the flow facts for such a loop must cover all these different contexts in which the loop may be executed to result in safe WCET estimates. Hence, the lower bound of such flow facts must represent the global minimum of iterations executed by such a loop over all contexts in which f is called, and the same holds for the upper bound. Since such flow facts for data-dependent loops do not consider possible different execution contexts of a function f, the flow facts are safe but lead to a highly overestimated WCET.

Procedure Cloning (also known as *Function Specialization*) is a standard optimization [8] exploiting functions that are often called with constant values as arguments. If the caller invokes a callee with some constant parameters, the callee can be cloned and the constant parameter can be removed from the list of parameters and instead imported into the callee itself. In standard literature, the benefits of cloning are said to be twofold. First, procedure cloning potentially enables further optimizations like e. g. constant propagation and constant folding within the clone. Second, the calling overhead is reduced since the constant parameters are not passed any more between caller and callee.

This paper analyzes the impact of procedure cloning on the WCETs of embedded real-time applications. Procedure cloning is done if a caller invokes a callee f containing data-dependent loops, and if the iterations of such loops depend on a parameter p being a constant in the function call. Cloning results in a specialized version f' of f now having constant loop bounds with respect to p. The data-dependence of the number of loop iterations is resolved by cloning so that highly precise flow facts for such data-dependent loops in f' can be provided. Hence, procedure cloning can be seen as a way to express different calling contexts at the source-code level, thus enabling a high-precision WCET analysis of these specialized functions.

But not only loops will benefit from procedure cloning. Propagation of constants as specialized function arguments may enable the elimination of *infeasible paths*. Infeasible paths are paths which are executable according to the control flow graph structure, but not feasible when considering the semantics of the program and the possible inputs. Removal of infeasible paths may thus lead to tighter WCET estimates.

The contributions of this paper are twofold: First, it is the very first study of the impact of procedure cloning on WCET. Up to now, only the effect of cloning on average-case execution time *(ACET)* and code size was studied. We carefully examine how cloning influences the specification and consideration of precise loop bounds and the detection of infeasible paths during WCET analysis. Experimental results underline the performance of this optimization in terms of WCET. Our measurements show WCET reductions from 12% up to 95% for real-life benchmarks. In contrast, cloning affects ACETs only marginally with maximum speed-ups of 3%. Second, we show that these considerable improvements do not translate into an increased overhead during WCET analysis. It can be concluded that procedure cloning is highly effective in WCET minimization with no additional overhead required during WCET analysis.

The rest of this paper is organized as follows: Section 2 describes related work. Procedure cloning and its benefits for WCET analysis are presented in Section 3. Section 4 describes the experimental environment, followed by benchmarking results in Section 5. Section 6 summarizes the contributions of this paper and gives directions for future work.

2. RELATED WORK

Procedure cloning was initially published in [3] and is now part of standard literature on compiler construction [8]. However, cloning was exclusively studied in the context of

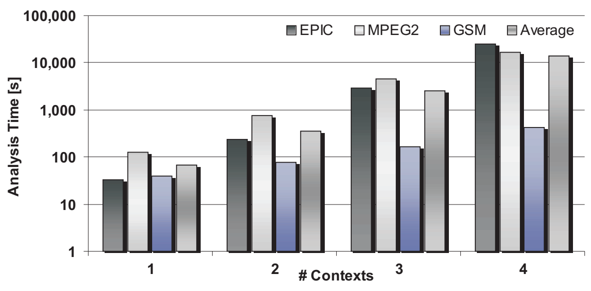

Figure 1: Context-Sensitive WCET Analysis Times

ACET and code size. All publications on cloning discussed how it reduces function call overhead, how it enables other optimizations focusing on ACET, and how growth of code size can be limited. Procedure cloning was not yet discussed in the context of compilation and optimization of real-time embedded software which is the key contribution of this paper.

In general, development of compiler optimizations concentrated on ACET in the past decades. Recently, the minimization of energy dissipation as optimization goal of compilers moved into the focus of research. However, WCET minimization by compiler optimizations is only sparsely dealt within today's literature. Loop Nest Splitting [5, 6] 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 [10], an algorithm for code positioning for WCET reduction is presented. It rearranges basic blocks in functions such that the number of branches taken on the worst-case execution path is minimized. Since WCET minimization by compiler optimizations is only barely researched, an in-depth survey of the influence of standard optimizations on WCET is missing in literature. This paper is a first step towards such a survey by studying one well-known optimization.

[4] presents a design study for an entire 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.

As shown in Section 1, functions containing data-dependent loops have a strong negative impact on WCET analysis since the effective number of loop iterations depends on the calling context of the surrounding function. A calling context represents a history of a call of function F and thus indicates a particular way of calling F [1, 9]. However, modern WCET analyzers [1] are able to distinguish between different calling contexts of functions and thus can theoretically deal with precise numbers of loop iterations depending on such contexts.

In practice, such a context-sensitive WCET analysis often is not feasible since runtimes of WCET analyses grow enormously. This phenomenon is depicted in Figure 1. For three complex real-life applications, it shows the time taken for WCET analysis using aiT [1] for different (small) numbers of considered calling contexts. As can be seen, WCET analyses require exponential runtimes for increasing numbers of contexts. In practice, this makes context-sensitive WCET analysis infeasible and leads to the WCET overestimations described in Section 1. Another important advantage of the

```
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

techniques studied in this paper is that they achieve a significant WCET reduction without increasing analysis times noticeably.

3. 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 more beneficial basis for aggressive inter-procedural data-flow 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, 8]. 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 considered.

3.1 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.

This strict policy of trying to keep the code size small would however strongly restrict the use of procedure cloning for most embedded systems applications since the constant values for a particular function parameter rarely remain the same, but vary between a small set of constant values. The second parameter mentioned above, defining the frequency of the occurrence of identical constant values, must thus be chosen adequately to allow procedure cloning at all.

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.

3.2 Improvements to the WCET

Primarily, the objective of procedure cloning is to reduce the ACET. But obviously, the optimization also reduces the WCET when code considered by the timing analyzer is improved. These pure improvements resulting from a better code quality, however, yield a minimal reduction of both the ACET and WCET for typical embedded systems applications as will be indicated in the following section.

In addition to the pure code improvements mentioned above, 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 1: 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 [7] 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.

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 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 like *Loop Nest Splitting* [6] 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.

4. 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. Table 1 lists their number of code lines, followed by the number of functions and finally the number of loops.

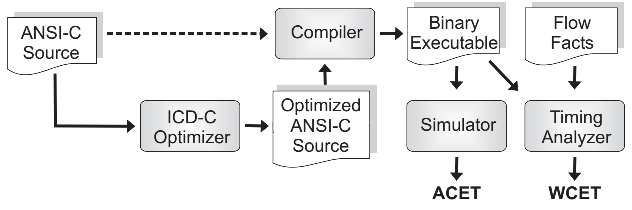

Figure 3: Workflow of Procedure Cloning

Benchmark	LoC	# Functions	# Loops
EPIC	334	6	41
MPEG2	1226	14	33
GSM	1066	36	48

Table 1: Evaluated MiBench benchmarks

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 [6] 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 3.1 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 program's ACET. 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. Since the ARM7TDMI does not use a cache, code and data for the TriCore processor were mapped to non-cached areas in order to get equivalent results. Due to the complexity of the timing analysis as indicated in Figure 1, the number of contexts was restricted to one distinguished context (no remarkable improvements were observed for the distinction of two or three contexts).

5. RESULTS

WCET

Compared to the ACET, the impact on the WCET was enormous. As presented in Figure 4, on average reductions of the WCET by 47.56% for TriCore, 47.23% for the ARM mode and 53.38% for the THUMB mode were achieved. In the following, the transformations performed by the ICD-C optimizer are briefly discussed for each benchmark.

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. $[1 \ldots 15]$ (in non-optimized code) becomes $[1 \ldots 1]$ after the transformation, meaning that the timing analyzer can assume one loop iteration in contrast to the pessimistic assumption of 15 iterations.

The second benchmark MPEG2 contains two functions that were optimized by Procedure Cloning. 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 is 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 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.

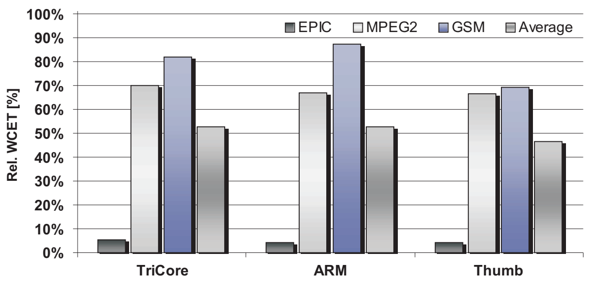

Figure 4: Relative WCETs after Procedure Cloning

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 3.2, 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.

ACET

To examine the impact of procedure cloning on the ACET, the simulated execution times of the original and optimized code were compared. The improvements were negligible. For the EPIC benchmark, the ACET 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 ACET reduction of 0.01% was observed for the TriCore processor. On average for all benchmarks, an ACET decrease between 0.06% and 1.8% for TriCore and THUMB mode, respectively, was achieved. The minimal degree of ACET 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.

Code size

Finally, the code size is examined. As mentioned in Section 3, 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% corre-

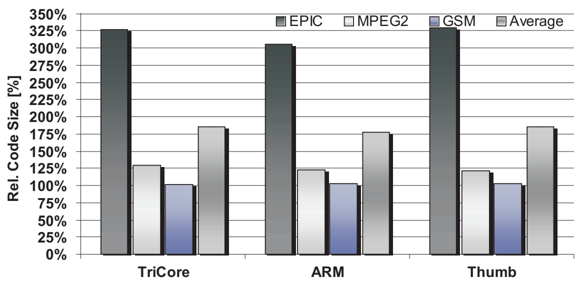

Figure 5: Relative Code Sizes after Cloning

spond 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 the 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.

6. CONCLUSIONS AND FUTURE WORK

This paper studies the positive impact of procedure cloning on the worst-case execution time. The timing analysis of loops is an inherent source of unpredictability since the required number of loop iterations can be rarely specified precisely. The resulting WCET is heavily overestimated. Procedure cloning improves the analyzability and thus makes the code more predictable since specialized functions allow an explicit definition of the loop bounds. Moreover, the optimization aids at detecting infeasible paths that must not contribute to the timing analysis for tight WCET estimates.

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 ACET 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.

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.

7. 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 *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, October 2006.

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

[6] 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.

[7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. pages 3–14, December 2001.

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

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

[10] 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.