WCET-driven Cache-based Procedure Positioning Optimizations [*]

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

Abstract

Procedure Positioning is a well known compiler optimization aiming at the improvement of the instruction cache behavior. A contiguous mapping of procedures calling each other frequently in the memory avoids overlapping of cache lines and thus decreases the number of cache conflict misses. In standard literature, these positioning techniques are guided by execution profile data and focus on an improved average-case performance.

We present two novel positioning optimizations driven by worst-case execution time (WCET) information to effectively minimize the program's worst-case behavior. WCET reductions by 10% on average are achieved. Moreover, a combination of positioning and the WCET-driven Procedure Cloning optimization proposed in [14] is presented improving the WCET analysis by 36% on average.

1. Introduction

Embedded systems often must 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 which have to satisfy critical timing constraints.

Due to the complexity of today's embedded systems, the software development relies on both a high-level language, predominantly C, and a compiler. State-of-the-art compilers offer a vast variety of optimizations with the objective to minimize the average-case execution time *(ACET)* [13] or energy dissipation [18]. On the contrary, a compiler-guided reduction of the WCET is still a novel research area with only a small number of published approaches. WCET-driven compiler optimizations require the integration of a static WCET analyzer into a compiler framework. The analyzer provides timing information taken into account to effectively minimize the program's WCET. Our developed optimizations base on the exploitation of memory hierarchies.

In contrast to the speed of memories, processor speed has increased dramatically in the past years. To bridge the increasingly large gap between the processor and the memory speed, memory hierarchies based on caches are today's state-of-the-art. Caches have the advantage of being transparent to the software running on a system since their management is controlled by the hardware. They are effective in reducing the ACET of a system and have become indispensable in today's desktop processors.

The possible unpredictibility of caches due to the lack of sophisticated static cache analyses made them inapplicable for real-time systems which have to meet hard timing constraints. Without caches these systems suffer a low average-case performance since code and data must be delivered from the slow memory. Today's high-performance demands, however, make the use of caches indispensable and are the motivation for research aiming at a thorough analysis of cache-based systems and their static prediction. This calls for the development of static WCET analyzers, like aiT [1], including a sophisticated cache analysis [8] which is able to precisely determine whether a cache access is a hit or a miss.

In this paper, we consider WCET-driven compiler optimizations which aim at the minimization of the WCET by achieving an improved instruction cache (I-cache) behavior. The goal is to place procedures which contribute to the WCET, i.e. procedure lying on the worst-case path (*WC path*), such that they are mapped contiguously in memory. This placement avoids overlapping of cache lines belonging to a caller and callee function and thus decreases the number of cache conflict misses. Since the procedures chosen for reordering might vary from those that would be chosen for an ACET optimization, a WCET-centric optimization guidance is mandatory.

In this paper we present three different approaches: a greedy and a fast heuristic approach and a combination of WCET-driven Procedure Cloning [14] with Procedure Positioning. The contributions of this paper are as follows:

1. In contrast to standard Positioning optimizations guided by profile data to improve average-case performance, our novel approaches are driven by WCET information to effectively reduce the program's WCET.

2. Our greedy Procedure Positioning approach takes changes to the worst-case path into account allowing an effective WCET minimization.

3. We show that our extended WCET-driven Procedure Cloning is effective in cache-based systems to improve WCET estimations.

The rest of this paper is organized as follows: Section 2 describes related work. The general ideas of Procedure Positioning and our three WCET-driven approaches 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

In past decades, development of compiler optimizations has concentrated on the ACET. One class of optimizations exploits memory hierarchies and aims at the improvement of both data and instruction cache behavior. The main idea behind all these techniques is to enhance spatial and temporal locality. Known data access optimizations encompass *Loop Interchange*, *Loop Tiling*, *Loop Fusion* or *Data Prefetching* [16].

I-caches being the cache we consider in this paper mainly profit from a reorganization of the code at procedure and basic block level. [19] propose two code placement methods for basic blocks to reduce the cache miss rate based on an integer linear programming problem. [11] propose a compiler with an integrated instruction placement algorithm reducing page faults.

The static cache analysis is an essential part of a WCET estimation for cache-based processors. Its goal is to classify each memory access into a cache hit and a cache miss. Ferdinand et al. use a *must* and *may analysis* based on abstract interpretation [6]. This approach is also employed in aiT, the WCET analyzer we apply for our experiments. Other approaches categorize the cache accesses into guaranteed cache hits and misses for all references encountered for a particular program line on the one hand and into guaranteed cache hits and misses encountered from the second loop iteration on the other hand [7].

Recently, the minimization of energy dissipation as an optimization goal of compilers has moved into the focus of research. However, WCET minimization by compiler optimizations is only sparsely dealt within today's literature. Loop Nest Splitting [4, 5] 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 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 on average by 7%. This work differs in two main points from our approaches. On the one hand, the underlying processor is quite simple since it has no caches, thus challenging side-effects are not considered. On the other hand, their optimization's objective is not the reduction of cache conflict misses but a decreased number of incurred pipeline delays caused by control transfer instructions like branches.

[3] 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.

Our third positioning algorithm is based on the compiler optimization Procedure Cloning. This optimization has been introduced by Cooper [2] 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 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 presented a modified version of the optimization which is tailored towards an effective WCET estimation. Presented results on real-world benchmarks indicate that the overestimation could be highly reduced. In this work, we briefly describe our extensions to the previously developed optimization and show how the memory layout can be exploited by an appropriate code positioning.

3   Procedure Positioning

The basic idea behind Procedure Positioning is to improve I-cache behavior by reducing the number of conflict cache misses. Caches reduce the average memory access time by exploiting spatial and temporal locality. The former refers to the reference of contiguous memory locations. Temporal locality means that particular memory locations will be accessed within a short period of time. Due to an inappropriate memory layout, the temporal locality may, however, degrade cache performance. This situation arises
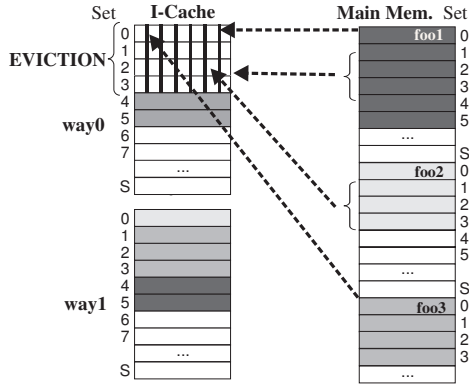
Figure 1. Cache content eviction before Positioning

when memory locations being accessed temporally close to each other are mapped to the same cache location. This overlapping results in an eviction of cache contents and a resultant repetitive cache refill.

Example

Assume that the considered cache is *set-associative*, i. e. the cache is divided into sets which hold multiple lines (located in ways) whose number is defined by the associativity. In our example, we consider the associativity of 2 (other associativities are equivalent). A memory location that was determined to be mapped to a particular set can be located in any of the set lines depending on the replacement strategy. The mapping of a memory block into a set, called *bit selection*, is performed by a modulo operation [9]:

$$(Block\ address)\ MOD\ (Number\ of\ sets\ in\ cache)$$

Furthermore, assume the memory layout given in Figure 1. This layout is used for the execution of the C code example in Figure 2 which is typical for embedded system's applications where multiple functions are invoked in the same loop.

```
void foo1( void ) {
   for( int i = 0; i < N; ++i ) {
      foo2();
      foo3();
      // remaining code of the loop body }
}
```

Figure 2. Code example for potential conflict misses

The function $foo1$ is located in main memory at an address which maps to cache sets 0-5 while functions $foo2$ and $foo3$ correspond to a cache mapping into sets 0-3. The set-associative I-cache consists of S sets distributed over the ways $way0$ and $way1$.

Starting with the execution of function $foo1$, the *for*-loop header is copied into the cache beginning at cache
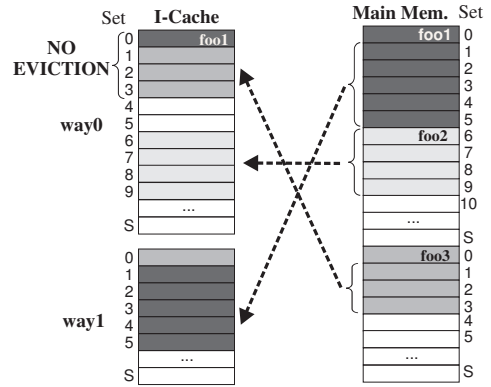

Figure 3. No eviction after Positioning

set 1 of way $way0$ (an empty cache is assumed). Subsequently, function $foo2$ is invoked. After moving the first block of $foo2$ into the free line of $way1$, the remaining code is copied into $way0$. The execution of $foo3$ leads to the first eviction for set 0 since both lines are already occupied. Depending on the replacement strategy (here *Least Recently Used*), $way0$ is chosen. The remaining code is moved into the free lines of $way1$. Finally, the remaining code of $foo1$ is executed and due to missing free lines, it evicts the lines in sets 1-3 of $way0$, copying the remaining two blocks into the free lines of $way1$. For the remaining loop iterations of $foo1$, the eviction of cache lines is continued resulting in multiple conflict misses which entail multiple accesses to the slow external memory.

The costly eviction of code lines can be eliminated by altering the order in which functions are mapped into memory. In general, this is accomplished by allocating functions which are accessed within a local time window (temporal locality) contiguously in memory as depicted in Figure 3. Function $foo2$ is mapped at an memory address corresponding to set 6. Obviously, allocating $foo3$ contiguously to $foo1$ would have the same result. This eases the pressure of mapping multiple memory locations to the same sets. Executing the code from Figure 2, the entire code for the three functions can be brought into the cache. This memory layout eliminates all set evictions, thus allowing a fast execution due to cache content reuse.

For *direct-mapped* caches, the positioning technique might be even more beneficial. In this cache architecture, each set can be considered as holding exactly one line. [10] reported that direct-mapped I-caches show a larger number of conflict misses compared to set-associative caches. Hence, altering the order of the code in the described manner would eliminate potentially more conflict misses. In addition to the reduced number of cache misses due to a minimized number of cache set overlappings, the altered memory layout after positioning might eliminate *translation lookaside buffer (TLB)* misses [17]. Reorganizing functions contiguously in memory, increases the probability that both functions will be mapped into the same page, reduc-

ing the page working set and potentially eliminating TLB misses.

The optimization offers another advantage especially relevant for embedded systems which are usually equipped with a battery and thus energy dissipation is a crucial constraint. Code positioning increases the number of cache hits by removing cache set eviction and thus eliminating cache misses.

A decreased number of cache misses results in less accesses to the main memory. [20] reported that the energy consumption for an access to main memory compared to accessing a cache might be larger by up to a factor of 40. Hence, code positioning may produce code that substantially saves energy consumption.

Please note that we used the term $function$ in this example to stay consistent with the high-level C terminology. Since our algorithms are based on the low-level (assembly-like) representation of the code, we will use the term $procedure$ as an equivalent to $function$ in the following.

3.1 WCET-Centric Call Graph-based Positioning

In this section, we present our novel positioning algorithms which are based on a call graph annotated with call frequencies obtained from a WCET analysis.

Procedure Positioning approaches are based on a call graph. This undirected graph (direction of the call is irrelevant for our approach) consists of nodes which represent program procedures. Edges between the nodes denote calling relationships between procedures and are weighted with call frequencies which, for ACET optimization, are gained during profiling.

In contrast to the standard optimizations, our approaches do not rely on execution profile data but extract their input data for the call graph from a WCET analyzer. This fundamental difference makes our approach more reliable than the standard optimizations. Execution profile data is critical since it reflects the program execution for a particular set of input data, i. e. profiling the program under test with varying inputs yields different results. For more complex programs consisting of numerous input-dependent execution paths, it is almost infeasible to find representative input values. This may result in a call graph which is annotated with profiling data that does not represent some particular program executions. The optimized code will possibly not improve cache behavior and may even suffer performance degradation.

Our approach does not rely on representative input data. The edge weights are computed by a WCET analyzer and are invariant for all program executions. The reason is the inherent nature of a static WCET analysis. It computes the worst-case behavior for a given program that is valid for
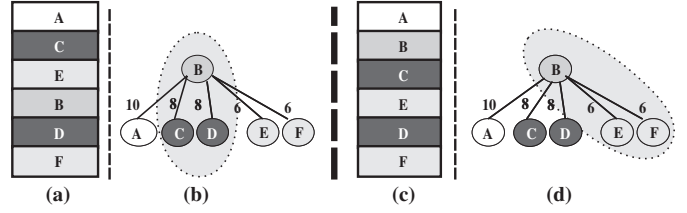

Figure 4. Inappropriate WCET Optimization

all inputs without running the program but by performing a static program analysis. This computation implies the determination of call frequencies that yield the longest program execution. Our WCET-centric call graph is based on this information and an edge with the heaviest weight potentially combines the most promising functions for optimization.

3.1.1 Greedy Approach

It is well known that the influence of a memory layout modification on caches is hardly predictable. A promising optimization is based on a greedy approach which evaluates the influence of a particular reallocation of procedures on the WCET. In case a WCET minimization was achieved, this altered memory layout is considered as a new starting point for the next optimization cycle and the next most promising function for positioning is considered. Hence, the approach successively reduces the WCET and guarantees that the modified code's WCET is never worse compared to the original code.

Our greedy approach is tailored to the WCET reduction and the procedure positioning order might substantially differ from the order chosen by the existing standard positioning optimizations. Figure 4 gives an example why Procedure Positioning guided by executing profile data is inappropriate for WCET minimization and might even yield a decreased worst-case performance.

Assume the function memory layout given in Figure 4(a) and the corresponding call graph in Figure 4(b) representing a typical *if-then-else* statement where functions A, {C,D}, and {E,F} are invoked from mutually-exclusive alternative blocks in function B. Each edge is assigned the call frequency by execution profiling. Before positioning, assume the WC path, being the longest path through the program's control flow graph, is B → {C,D}. ACET code positioning would allocate functions B and A contiguously in memory to avoid conflict misses (Figure 4(c)). This might result in new conflict misses on the path B → {E,F} outweighing the costs for executing the current WC path B → {C,D}. Therefore, a modification in the positioning results in a different path becoming the WC path (*WC path switching*) whose execution might even take longer than that of the old WC path (Figure 4(d)). Thus, a decreased ACET was achieved at the cost of an increased WCET.

To avoid these unintentional effects, our greedy algo-

```
1    Input:    Program P
2    Output:   optimized Program
3
4    begin
5     boolean terminate := false
6     WCETAnalysis(P)
7     Graph G_ref := BuildCallGraph(P)
8     Graph G_wcet := BuildCallGraph(P)
9     repeat
10     WCETAnalysis(P)
11     UpdateOriginalGraph(G_wcet)
12     repeat
13      Edge e_max := FindMaxEdge(G_wcet)
14      if(e_max=∅)
15       terminate := true
16       break
17      fi
18     until(!Position(e_max,G_wcet,G_ref,P))
19    until(terminate≠true)
20    return (P)
21   end
```

Figure 5. Greedy WCET-driven Procedure Positioning algorithm

rithm reorders the procedures which are most promising for WCET minimization and additionally evaluates each potential memory layout modification on the WCET before finally applying it to the program under test. The basic idea for positioning was derived from the profile guided approach having the objective to reduce ACET as described in [17].

The formal definition of the greedy WCET-driven positioning algorithm is specified in Figure 5. The input of the algorithm is the program to be optimized. In line 5, the termination variable finishing the optimization is initialized. In the next threes lines, the WCET-centric call graphs are constructed based on the WCET analysis for program P in line 6. Reference graph G_ref remains unmodified by the algorithm and, as will be described later, it will be consulted during the optimization to find an appropriate positioning order. Graph G_wcet represents the WCET-centric call graph for the current program and will be updated whenever P changes.

Lines 9 - 19 are the core of the algorithm which evaluates potential procedure reorderings and possibly applies them to the final code P. In line 10, the WCET analysis for the current program P is performed and its code structures are annotated with timing information. Please not that this step is superfluous in the very first iteration due to line 6. The update step in Line 11 is another fundamental part of the WCET minimization since it makes the optimization aware of WC path switching. Any modifications to P after
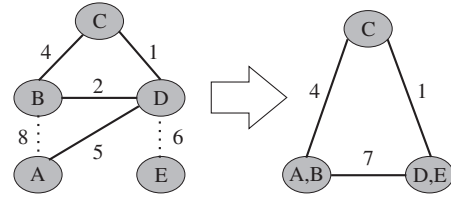

Figure 6. Coalescing Graph Nodes and Edges

an applied Procedure Positioning might result in a different worst-case path. In that case, the WCET-centric call graph must be updated by removing all call frequencies on the obsolete WC path and add them to the new path. Otherwise, the performed modifications might have no effect on the WCET.

For the call graph in Figure 4(b) this would mean that the call frequencies at the edge B → C and B → D would be deleted and the edge weight for B → E and B → F would be updated (in the WCET-centric graph), as depicted in 4(d). This update ensures an effective WCET minimization since exclusively code fragments relevant for the worst-case performance are optimized. Thus, WCET positioning optimizations are more challenging than similar ACET optimizations where frequencies of edges based on profile data do not change.

Lines 12 - 18 evaluate potential memory reorganizations resulting from Procedure Positioning. After finding the edge from the WCET-centric call graph G_wcet with the greatest call frequency, function $Position$ is invoked together with the WCET-annotated program P, the reference graph G_ref and the current graph G_wcet. In this function, procedures defined by e_max are reordered such that they are allocated contiguously in memory. The altered program is evaluated by the WCET analyzer. If the optimized program results in a larger WCET, Procedure Positioning was not successful for this particular e_max, P is restored and $Position$ returns with $false$ repeating the evaluation for the edge with the next largest call frequency (line 13).

Otherwise, if the reordering of procedures based on e_max was successful in $Position$, i. e. a WCET reduction was achieved, the new procedure order will be permanently retained by applying it to the passed input program P, and $Position$ returns with $true$ leading to a new iteration of the algorithm (line 9) with the optimized program P.

The reorganization of procedures in $Position$ is based on the edge e_max and the WCET-centric reference graph G_ref. In the beginning, all procedures in the call graph G_wcet reflecting the current state of P are represented by individual nodes. After choosing two functions to be placed contiguously in memory connected by e_max, both nodes of G_wcet are merged and their edges are coalesed. Figure 6 depicts an example where first nodes A and B, afterwards nodes D and E were coalesced. (The considered edges are marked by dotted lines.) The merged nodes guarantee

that the contiguous allocation of the corresponding procedures will be preserved for further invocations of $Position$. Nodes that are not connected in the graph are omitted.

The decision for an appropriate procedure order becomes more complicated when two already coalesced nodes have to be merged since there are two possibilities: the second node can be placed before or after the first node. To find the most promising memory allocation, the unmodified WCET-centric call graph G_ref of the original input program is consulted. In this graph, the original call frequencies are saved before the nodes were coalesced. They enable to determine whether the right-most procedure (in the order of the call chain) of the first node and the left-most procedure of the second node have a greater call frequency and vice versa. In the right-hand side of Figure 6, coalescing of nodes A, B and D, E represents such an issue. From the original graph in the left-hand side of the figure, it can be seen that procedures B and E have no calling relation, while the call frequency for procedures A and D is 5. Hence, placing the procedures such that A and D are contiguously allocated in memory is more likely to reduce cache set evictions.

The algorithm terminates when all possible edges in the WCET-centric call graph G_wcet were considered without finding a new ordering that minimizes the WCET (line 15). Obviously, this greedy algorithm may in the worst-case take multiple iterations since each evaluation of a new procedure positioning is accompanied by a WCET analysis. However, our experiences, as will be indicated in the result section, show that for all considered benchmarks the optimization run time was acceptable. Obviously, this is not a proof but an investigation of our heuristics revealed that in many cases the initial choice of the procedures to be positioned was successful w.r.t. to WCET minimization. Thus, we think that also for other benchmarks a comparable behavior of our algorithm can be expected.

3.1.2 Heuristic Approach

For comparison, we also implemented a fast heuristic approach which works exclusively on the information from the WCET-centric call graph constructed from the original input program. The basic idea is the same as for the greedy algorithm described in Section 3.1.1. Procedures related by high call frequencies are allocated contiguously in memory to avoid cache conflict misses.

In contrast to our greedy algorithm, the heuristic algorithm performs exactly one WCET analysis to construct the call graph. Based on this data, the algorithm tries to ensure that at least the initial positionings will have a positive effect on the WCET since edges with the heaviest call frequencies between procedures on the worst-case path are exploited. The speed advantages come at the cost of efficacy. First, the reordering of procedures is based exclusively on the initial call graph and is performed without re-evaluating its influence on the WCET. Hence, also undesired WCET increases are accepted. Second, worst-case path switching is not taken into account. Since the call graph is not updated, the heuristic approach will operate on an outdated WCET-centric call graph when the WC path changes. The performed positionings would then possibly not affect the WCET.

The formal algorithm can be considered as a simplified version of the one given in Figure 5. After performing a WCET analysis on the original program P (line 6), a WCET-centric call graph is constructed (line 8). Next, an edge with the heaviest weight is searched (line 13) and directly taken for procedure positioning in P without any evaluation. After coalescing the nodes and edges in the call graph, the next edge with the heaviest weight is analyzed. The optimization terminates when all edges were processed and returns the modified program P.

3.2 Cache-aware Procedure Positioning for WCET-driven Procedure Cloning

In the domain of the worst-case execution time 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 static timing analysis.

This is due to the inherent nature of the static WCET analysis which does not execute the program but extracts most of its information from the program code by static program analyses. However, some data can not be automatically derived and must be provided by the user (so-called *user annotations*) such as the iteration counts of loops. 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.

For loops whose number of iterations depends on function parameters, the specification of flow facts is insufficient since no individual calling contexts are considered. This results in a lack of precision with safe but also highly overestimated WCET bounds. In [14], we exploited a modified version of the standard compiler optimization *Procedure Cloning* to improve the WCET estimation. Our approach generates specialized versions of functions, making their calling context explicit and thus enabling a precise specification of loop annotations. The optimizations presented in the previous work were performed in a system with disabled caches. Any newly created function clone was placed behind the last function in the code with no regard to cache effects. Such a simple placement strategy is sufficient for cache-free systems since undesired cache conflict misses

can not emerge.

In this paper, we, for the first time, perform the WCET-driven Procedure Cloning on a cache-based system to show that the influence of the resulting code size increase during Cloning does not revoke the positive effect of the optimization on the WCET estimation. The idea of memory layout modifications to avoid cache conflict misses was also exploited for this optimization. After cloning a function, the question arises where to place the cloned function in memory. A promising idea is again to place the new functions close to the function invoking them most frequently. In this manner, a new optimization is becoming feasible by combining Procedure Cloning with Procedure Positioning.

For this purpose, our algorithm first performs Procedure Cloning for a particular function and constructs the corresponding WCET-centric call graph. Next, a function f_max with the greatest call frequency (edge weight) to the cloned functions is determined. In case cloning generated only one function (meaning that there was only one function parameter in the original function which controlled a loop), the cloned function will be placed directly after the function f_max. For two cloned functions, the first clone will be placed after f_max, while the second clone will be positioned before f_max. This positioning tries to achieve best cache performance by minimizing the number of cache evictions. If there are more than two cloned functions, there is no chance to place them all contiguously in memory w.r.t. f_max, so they are placed behind the first clone to be located as close as possible to f_max to reduce cache overlappings.

4 Experimental Environment

This section describes the choice of benchmarks used to evaluate the influence of our three WCET-driven Positioning algorithms on the WCET. Furthermore, the benchmarking workflow is described.

Benchmark	Code Size [bytes]	Description
expint	972	Function series expansion
g721_encode	19880	G.721 encoder
g723_encode	19972	G.723 encoder
gsm_decode	34112	GSM voice decoder
gsm_encode	41900	GSM voice encoder
mpeg2	39763	MPEG2 decoder

Table 1. Benchmark Characteristics

To evaluate our WCET-driven optimizations, we used a set of applications from different benchmark suites representing applications typically found in the embedded systems domain. The benchmark *expint* is part of the Mälardalen WCET benchmark suite [15], the G.72X, GSM
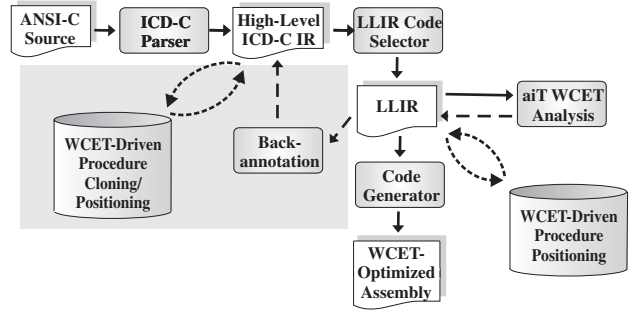

Figure 7. Workflow for WCET-driven Procedure Positioning

and MPEG2 coder come from the MediaBench [12]. The benchmarks with their size in bytes are described in Table 1 and were all in their original layout without any modifications to the function order. The initial code layout is crucial for the improvements achieved by the optimization since different layouts offer different potentials for positioning and might yield even better or worse results than the result presented in this work.

The workflow is depicted in Figure 7. For the integration of our WCET-driven compiler optimizations, we use our WCET-aware C compiler for the Infineon TriCore 1796 microcontroller [3]. The compiler consists of our compiler consists of a high-level intermediate representation, the ICD-C, and a low-level representation, called the LLIR, which is coupled to AbsInt's WCET analyzer aiT. The I-cache utilized for our tests is a 16 kB 2-way set associative cache with cache line size of 256 bits and a *Least-Recently Used* replacement strategy. Due to the predictable LRU strategy, the cache is fully supported by the static WCET analysis of aiT and highly reliable results are achieved. In the following, the workflow for the greedy Procedure Positioning aprroach is described in more detail. The additional workflow used for Procedure Cloning is marked by the gray box in Figure 7.

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

The next step is the WCET analysis of the program. After finishing the analysis, our compiler automatically imports the timing results 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 [3]. The workflow up to the WCET analysis is depicted in Figure 7 by solid arrows.

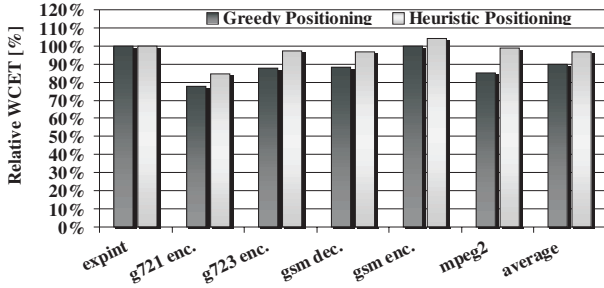Based on the WCET information, the WCET-centric call

Figure 8. Relative WCET after greedy and heuristic Procedure Positioning

graph is constructed and exploited by our greedy algorithm. After choosing the call graph edge with the heaviest weight, the LLIR procedures are reordered and the resulting effect on the WCET due to the modified memory layout is evaluated by another run of the WCET analyzer. The evaluation progress is represented in Figure 7 by dashed lines (not included in the gray box). After terminating the optimization, a code generator is used to produce an equivalent assembly code from the WCET-optimized LLIR representation. The heuristic Procedure Positioning algorithm follows the same workflow. However, after an initial run of the WCET analysis, the WCET-annotated LLIR serves as basis for the construction of the WCET-centric call graph. Hence, the workflow depicted by the dashed lines in Figure 7 is omitted.

The workflow of the WCET-driven Procedure Cloning resembles the one shown in Figure 7. The fundamental distinction lies in the code abstraction level the optimization is performed on. In contrast to the previously presented approaches, Procedure Cloning is not performed on the low-level but the high-level representation, in our example the ICD-C IR. To profit from WCET information at this level, the timing information must be transformed from the LLIR where it was obtained from the WCET analyzer back into the ICD-C. This process is called *Back-annotation*.

Hereafter, based on the ICD-C code, the WCET-centric call graph is constructed and the functions are successively optimized. Each potential function promising an improvement of the WCET is cloned, the resulting cloned functions are placed in an appropriate order consulting the call graph, as described in Section 3.2, and their influence on the WCET is evaluated. This greedy approach is repeated as long as there are potential functions in the code that might be beneficial for an improvement of the WCET estimation. After terminating, the WCET-optimized ICD-C code is translated into an LLIR and assembly code.

5 Results

Worst-Case Execution Time

In this section, the results on the WCET for all three presented approaches are discussed. Figure 8 shows the results for the greedy and heuristic Procedure Positioning, with 100% corresponding to the WCET estimation of the original code. First of all, it can be seen that for most benchmarks a WCET reduction was achieved. The greedy algorithm achieved on average a WCET minimization by 10%, while the heuristic approach reduced the WCET on average by 4%.

The results are strongly dependent on the initial order of the benchmarks' procedures. If the original memory layout already yields a good cache performance, the improvements due to positioning might be smaller than for benchmarks whose execution incurs more cache conflict misses. Moreover, small benchmarks whose text section is small enough to fit completely into the cache (like *expint*) do not profit from this optimization since no conflict misses can occur. However, applications which can be completely stored in the (usually) small I-cache of an resource-restricted embedded system are uncommon for today's software.

A different case is observed for the *gsm enc.* benchmark where the greedy algorithm could not achieve an improvement and the heuristic approach even worsens the WCET. This is due to the theoretical concepts discussed in Section 3 which do not guarantee an improved global cache performance with a reduced total number of conflict misses. The reason is the unpredictability of the global cache performance resulting from (even slight) local code modifications. A reordering of functions might improve the cache behavior locally but might simultaneously induce new cache misses for the execution of other code fragments leading to a degraded overall cache performance.

Finally, the difference in the achieved results between the greedy and the heuristic approach should be noted. For all benchmarks, the greedy positioning achieved better results since it does not allow a degradation of the WCET. This might result in a local optimum missing the global minimum as could be potentially achieved by the heuristic approach. However, for the considered benchmarks this case did not arise. For the heuristic approach, it might also happen that the WCET becomes worse after the optimization as experienced for the GSM encoder. Hence, it can be concluded that for best results it is worth to invest time for the evaluation as done by our greedy approach.

The results also show that the heuristic to always chose the edge with the heaviest weight is appropriate in general since the WCET could be reduced for most benchmarks and accomplish minimizations of up to 15% (for *g721 enc.*).

For WCET minimization, we do not achieve the same reductions as are reported for ACET minimization by Pettis [17]. This is due to the concepts a static WCET anal-
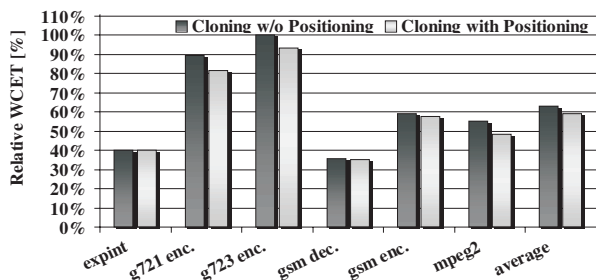
Figure 9. Relative WCET after Procedure Cloning without and with Positioning

ysis is based on. For any uncertainties encountered during the static program analysis, a safe assumption must be made to guarantee a sound approximation for the worst-case behavior of the program. These worst-case assumptions (penalties) often result in an overapproximation and hide the real program's WCET. The WCET estimation is said to lose tightness and is often encountered during the crucial cache analysis. This effect might also conceal the positive influence of our Procedure Positioning on the real program's WCET. Although the modified code yields an improved worst-case cache behavior with less conflict misses and thus an improved real WCET, the static cache analysis may not certainly classify all the cache accesses as cache hits and thus must assume a cache miss. This counterproductive effect does not occur for the measurement of the simulated time.

The results for WCET-driven Procedure Cloning are presented in Figure 9. Again, 100% corresponds to the WCET estimation of the original code. The optimization achieves WCET reductions of up to 64%. The partially heavy reductions in the WCET estimation result from the ability to annotate the optimized code with more precise loop bound specifications. Thus, the overestimation during the WCET analysis is removed. As an example consider the MPEG2 benchmark. It 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. We achieved a WCET reduction by 45% and 52% for Procedure Cloning without and combined with positioning, respectively. 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 results in Figure 9 allow two conclusions. First, it can be seen that the optimization is best suited in a cache-based system. Although Procedure Cloning increases the code size by additional functions (the cloned functions), the resulting WCET estimates are still more precise than for the original code. The benefits from the improved WCET analysis exceed the disadvantages that may emerge from more cache conflict misses due to the increased working set.

Second, for most benchmarks, Procedure Cloning combined with Procedure Positioning achieved better results. The goal of this additional feature is to compensate the potential conflict misses caused by additional function clones. Obviously, the benefits achieved are more marginal than for the positioning approaches shown in Figure 8 because the function reordering was exclusively restricted to the function clones. However, the additional positioning of the clones is negligible for the complexity of the algorithm and should be exploited for better results.

Simulated Time

For all three approaches the simulated time was measured with the TriCore instruction set simulator before and after the optimization. The goal was to compare the WCET results with the ACET. For all benchmarks the results were very similar: the simulated time did not remarkably change for the optimized code.

For the greedy and heuristic Procedure Positioning the simulated time decreased by 2% on average. This indicates that the WCET-centric call graph used for an effective WCET minimization might vary from a call graph based on execution profiling data. Thus, any decisions leading to a memory layout modification by positioning might be different for WCET and ACET optimizations concluding that a WCET minimization guided by profile data is inappropriate. For Procedure Cloning, the simulated time was decreased by 4% on average, mainly from the reduced number of function parameters which were substituted by constants in the function bodies. Again, this indicates that cloning has a different influence on the ACET than on the WCET estimation.

Optimization Run Time

Finally we measured the run time of our approaches on an Intel Xeon 2.13GHz system with 4GB RAM. Obviously, the heuristic Procedure Positioning was the fastest optimization since its execution is mainly dominated by the single WCET analysis. The run time of the greedy Positioning strongly depends on the number of evaluations accompanied by a WCET analysis. The longest time was spent for the MPEG2 benchmark where the total analysis time was 183 minutes. Also, the run time of Procedure Cloning is highly dependent on the number of functions which promise a WCET improvement and are thus optimized. The longest time was again measured for the MPEG2 benchmark which took 103 minutes to optimize. The long execution times are not acceptable for repetitive optimizations in a short period

of time. However, 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.

6 Conclusions and Future Work

In this paper, three novel compiler optimizations were presented aiming at the minimization of the WCET in a cache-based system. The key idea is the modification of the memory layout to reduce the number of cache conflict misses. This is achieved by a contiguous allocation of procedures in memory which have a high call frequency relation. All three approaches are based on a WCET-centric call graph.

The first approach is based on a greedy algorithm which evaluates the influence on the WCET before finally modifying the order of functions. The second approach is based on a heuristic and requires exactly one WCET analysis. Thus, the first approach is suitable for the generation of production code where longer optimization run times are acceptable and best results are desired. The second approach has the smallest optimization run time and can be utilized for a quick attempt to reduce the WCET. The third approach combines the compiler optimization Procedure Cloning improving the WCET analysis with positioning to additionally improve the cache performance.

The results show that the positioning algorithms could reduce the WCET by up to 22% while cloning removed overestimations during the analysis resulting in a reduced WCET estimation by up to 65%. From the marginal improvements of the simulated time it can be concluded that the influence on the ACET and WCET substantially differ when compiler optimizations focused on a particular criterion are applied. Hence, an effective WCET minimization requires novel specialized approaches which vary from the standard ACET optimizations.

In the future we intend to extend our framework to support multi-objective optimizations like a simultaneous improvement of the worst-case behavior and reduced power dissipation.

Acknowledgments

References

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore. 2008.

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

[3] 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.

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

[5] 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.

[6] C. Ferdinand, R. Heckmann, M. Langenbach, et al. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, Lake Tahoe, USA, 2001.

[7] C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.

[8] R. Heckmann, M. Langenbach, et al. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), 2003.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3 edition, 2003.

[10] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[11] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of ISCA '89*. ACM, 1989.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of MICRO 30*, Washington, DC, USA, 1997.

[13] R. Leupers. Code selection for media processors with simd instructions. In I. B. P. Marwedel, editor, *Proceedings of DATE*, pages 4 – 8, Paris, Mar. 2000. IEEE.

[14] P. Lokuciejewsi, H. Falk, P. Marwedel, and T. Henrik. WCET-Driven, Code-Size Critical Procedure Cloning. In *11th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, March 2008.

[15] Mälardalen WCET research group. Mälardalen wcet benchmark suite. http://www.mrtc.mdh.se/projects/wcet, January 2008.

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

[17] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of PLDI '90*, New York, NY, USA, 1990.

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

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

[20] M. Verma and P. Marwedel. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer, 2007.

[21] 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), Dec 2005.