WCET-driven Cache-aware Memory Content Selection[1]

Sascha Plazar, Paul Lokuciejewski, Peter Marwedel
Computer Science 12
TU Dortmund University
D-44221 Dortmund, Germany
FirstName.LastName@tu-dortmund.de

***Abstract*—Caches are widely used to bridge the increasingly growing gap between processor and memory performance. They store copies of frequently used parts of the slow main memory for faster access.**

Static analysis techniques allow the estimation of the worst-case cache behavior and enable the computation of an upper bound of the execution time of a program. This bound is called worst-case execution time (*WCET*). Its knowledge is crucial to verify if hard real-time systems satisfy their timing constraints and the WCET is a key parameter for the design of embedded systems.

In this paper, we propose a new WCET-driven cache-aware memory content selection algorithm, which allocates functions whose WCET highly benefits from a cached execution to cached memory areas. Vice versa, rarely used functions which do not benefit from a cached execution are allocated to non-cached memory areas. As a result of this, unfavorable functions w. r. t. a program's WCET can not evict beneficial functions from the cache. This can lead to a reduced cache miss ratio and a decreased WCET. The effectiveness of our approach is demonstrated by results achieved on real-life benchmarks. In a case study, our greedy algorithm is able to reduce the benchmarks' WCET by up to 20%.

I. Introduction

Many embedded systems have to meet hard real-time constraints. The worst-case execution time (*WCET*) of a program is the upper bound of its execution time for all possible input data and all possible initial system states. The WCET is a key parameter for real-time scheduling and the development of hardware platforms which have to satisfy critical timing constraints.

Caches have become popular to bridge the growing gap between processor and memory performance. They are developed to work transparently to the software running on a system by integrating a fully autonomous hardware controller. Caches are essential in today's computer systems since they dramatically improve the average-case performance. In the following, we will consider the optimization of instruction caches (*I-caches*).

The latency of an access to a certain main memory address highly depends on the contents of the cache. If an instruction to be fetched already resides in the cache, then a so called *cache hit* occurs and the fetch can be usually performed within one processor clock cycle. Otherwise, the access results in a *cache miss*. The required data then has to be fetched from the slow main memory (e.g. Flash) leading to penalty cycles depending on the processor and memory architecture.

The disadvantage of systems with caches is the limited predictability. It is hard to determine if an access results in a cache hit or miss and thus, it is hard to predict the execution time of a program executed from a cached memory. This is caused by the fact that caches only speed up the execution if a program tends to reuse instructions in the near future. If, however, the code of a program is not suitably arranged in the address space and the memory accesses are random or widely spread over the address space, the performance can be also decreased by the usage of a cache. Hence, static analysis techniques have been developed to allow safe predictions of a cache's impact on the worst-case performance of a system [1] in order to estimate tight bounds for the WCET of a program. Since the real WCET of a system can not be determined, we use the term WCET as a synonym for estimated WCET which is determined by a static WCET analyzer.

Compared to state-of-the-art main memories, caches are rather small. They can only keep a fraction of the slower memories' content as a copy for faster access. If a cache miss occurs, a complete cache line containing the addressed item is fetched from the main memory and possibly evicts valid content from the cache. Depending on the ratio of cache size to memory size, the cache replacement policy and the structure of the executed program, many cache misses can occur leading to costly reloading of content from the slow main memory.

Intelligent allocation of beneficial functions to cached memory areas and unfavorable functions to non-cached memory areas can ensure that functions whose WCET highly profits from a cache are not evicted from the cache by functions with a low benefit. This can lead to a faster execution and a decreased WCET due to a dramatically decreased number of cache misses. Furthermore, it is possible to reduce the overestimation of the WCET which is introduced by pessimistic assumptions concerning the I-

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

cache performance: If the memory access pattern can not be determined at a certain point of the program execution, a static WCET analyzer has to assume a cache miss and invalidated cache content. By allocating only a promising subset of functions to cached memory areas, the amount of unpredictable cache accesses can be decreased. As result, the cache analysis assumes less cache misses caused by such unpredictable accesses and a tighter bound for the WCET can be computed.

The WCET of a program always corresponds to the length of the worst-case execution path (*WCEP*) which is that path of the control flow graph (*CFG*) with the highest execution time. Optimizations of elements like functions on the WCEP can shorten this longest path in a way that another path will become the new WCEP. Optimizations of elements not lying on the WCEP will not result in a reduction of the WCET. Hence, possible switches of the WCEP have to be taken into account during optimizations.

Due to the complexity of today's embedded systems, programs are predominantly developed in a high-level language like C or designed with graphical tools which in turn emit C code. Optimizing compilers with a vast variety of optimizations focussing on average-case execution time (*ACET*) reductions are employed to generate highly optimized code. In contrast to ACET-optimizing compilers, automatic compiler-driven reduction of the WCET is still a novel research area. The big challenge is to develop optimizations which are able to keep track of switching WCEPs. Therefore, a tight coupling of a static WCET analyzer and a fine grained timing model has to be integrated into any WCET-optimizing compiler.

We developed a novel WCET-driven cache-aware memory content selection algorithm to decide which functions should be placed in a cached memory area in order to improve the worst-case I-cache performance. The proposed algorithm uses a greedy approach and evaluates the impact of executing a function from a cached memory area on the WCET. To enable WCET-centric optimizations, we employ the sophisticated static WCET analyzer *aiT* developed by AbsInt [2]. The algorithm always selects the function with the largest profit and keeps track of changing WCEPs by subsequently performing a WCET analysis. The main contributions of this paper are as follows:

- We explicitly select the memory contents of cached and non-cached regions based on its impact on the WCET of a program.
- Our greedy memory content selection takes changes on the worst-case execution path into account and optimizes along this path in order to allow an effective minimization of the WCET.
- We show that our algorithm is capable of improving the performance of caches with inherently low cache miss rates.

```
void foo1() {
 for(i=0; i<100; i++) {
  foo2();
  foo3();
 }
}
```

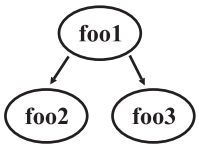

Figure 1. Exemplary program and resulting call graph

The paper is organized as follows: In the next section, we provide an overview of related work. Section III presents our new WCET-driven cache-aware memory content selection algorithm. Section IV introduces the WCET-aware C compiler *WCC* employed to develop our novel algorithm. An evaluation of the performance which is achieved by our WCET-aware memory allocation algorithm, is presented in Section V. Finally, we conclude our work and give a brief overview of future work.

II. Related Work

Theiling et al. present static WCET analyses for systems with caches based on their research in [1]. They separate microarchitecture analysis from path analysis in order to manage the overall analysis complexity. This way, a fast and still precise WCET analysis is feasible.

Chen et al. evaluate different types of optimizations and their influence on different cache sizes to show the implications of code expanding optimizations on instruction cache design [3]. The authors in [4] give an overview of cache optimization techniques and cache-aware numerical algorithms. They focus on the memory bottleneck which often limits the performance of numerical algorithms. Both [3] and [4] do not take the impact on the WCET of a system into account.

Work presented in [5] examines a combination of locking and partitioning of shared caches on multi-core architectures in order to guarantee a predictable system behavior. The proposed algorithms are not WCET-aware since their decisions are not based on any WCET values; nevertheless, the authors evaluate the impact of their caching schemes on the worst-case application performance.

In [6], Falk et al. present a cache locking algorithm which explicitly takes the worst-case execution path into account during each step of the optimization procedure. This way, they can make sure that always those parts of the code are locked in the I-cache that lead to the highest WCET reduction. Puaut et al. [7] propose locked instruction caches in multi-task real-time systems. They propose two low complexity algorithms for cache content selection. A drawback of statically locking the cache content is that the dynamic behavior of the cache is lost. In contrast to the techniques proposed in this paper, code is no more automatically loaded into the cache. Thus, code which is not locked into the cache can not profit from it anymore.

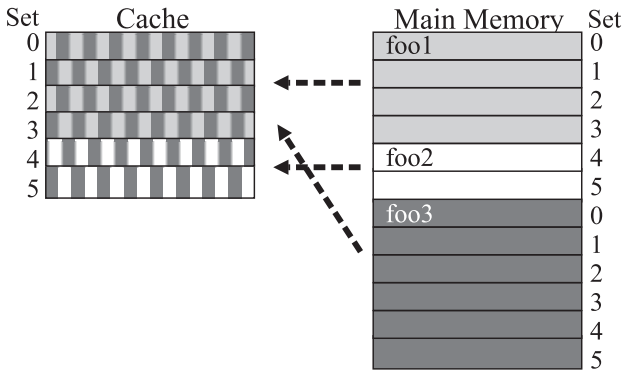

Figure 2. Cache thrashed by mutual evictions of functions

Vera et al. [8] combine cache partitioning, dynamic cache locking and static cache analysis of data caches to achieve predictability in preemptive systems. This eliminates overestimation and allows to approximate the worst-case memory performance, however, unlike our new approach, they are not able to explicitly optimize the WCET of a system.

In [9], a technique rearranging the positions of tasks to improve the cache performance is presented. The interdependency relation of tasks is evaluated in order to determine a memory layout which maximizes the number of persistent cache sets for each task.

A technique for procedure placement to reduce the cache miss ratio of programs is presented in [10]. Guillon et al. provide an optimal algorithm for memory placement which is improved regarding the unavoidable code size increase caused by gaps in the address space. In contrast to our optimizations, the presented approach does not take the WCET as metric into account.

Another technique for procedure positioning is presented in [11]. The authors propose an iterative algorithm which evaluates the worst-case calling frequencies and takes the WCET as metric into account. Unlike the work presented in this paper, their algorithm exploits preloading if parts of contiguous functions occupy the same cache line but suffers from small caches leading to increased cache miss rates.

Falk et al. counteract the possible predictability problems of caches with a static allocation of program code to so called scratchpad memories (*SPM*) [12]. They employ integer linear programming to select the optimal content of the SPM w. r. t. the program's WCET. Static SPM allocation has the disadvantage that the content is fixed during the program's execution. Thus, only the code located in the SPM profits from an accelerated execution.

III. WCET-driven Cache-aware Memory Content Selection

Nowadays, embedded systems are equipped with caches in kilobyte ranges, typically from 1 kB up to 16 or 32 kB. Compared to growing main memories in megabyte ranges, caches are rather small. The I-cache controller tries to keep copies of frequently executed memory lines containing sequences of instructions as cache content for a faster access.

Table I
WCETs for functions depending on the memory region

Function	WCET_{Flash}	WCET_{Cached}
`foo1`	350	195
`foo2`	690	470
`foo3`	500	369

The amount of cache misses highly depends on the ratio of cache to memory size, the cache replacement policy and the structure of the executed program. A high amount of cache misses implies costly reloading of content from the slow main memory and leads to a high number of penalty cycles due to pipeline stalls.

Figure 1 depicts the C code of a function `foo1` containing a loop which calls functions `foo2` and `foo3`. The resulting call graph of this simple program is shown on the right side. If the functions are consecutively arranged in the memory, `foo1` and `foo2` can be simultaneously stored in the cache (cf. Figure 2). Since the cache capacity is not large enough to store the whole program, `foo3` evicts the complete cache content during its execution. Thus, a lot of conflict misses occur and the complete cache has to be refilled twice during each iteration of the loop in `foo1`. This might lead to a dramatically increased WCET caused by pipeline stalls due to a high number of cache misses.

Many embedded systems have parameterizable caches and memory layouts which allow that parts of the address space can be included or excluded from caching. A WCET-aware compiler can exploit such a memory system and allocate functions to cached or non-cached memory regions. This strategy can ensure that functions which highly benefit from a cached execution can not be evicted by functions with a lower benefit.

In the following we assume the WCETs for functions `foo1`, `foo2` and `foo3` as depicted in Table I. WCET_{Flash} is the overall WCET of a function `f` if the entire function is executed from a non-cached memory area. WCET_{Cached} is `f`'s WCET if it is executed from a cached memory area. Obviously, it is more favorable if functions `foo1` and `foo2` can be kept in the cache since the accumulated WCET of `foo1` and `foo2` is decreased from 1040 cycles to 665 cycles. Overall, the WCET is reduced by 375 cycles compared to 131 if `foo3` is kept in the cache. An optimized memory layout for the example in Figure 1 is shown in Figure 3. Function `foo3` is moved to a non-cached memory area in order to prevent a constantly repetitive eviction of function `foo1` and `foo2`.

In the following, we describe our novel algorithm for a WCET-driven cache-aware memory content selection which moves functions to different memory areas depending on their impact on the overall WCET.

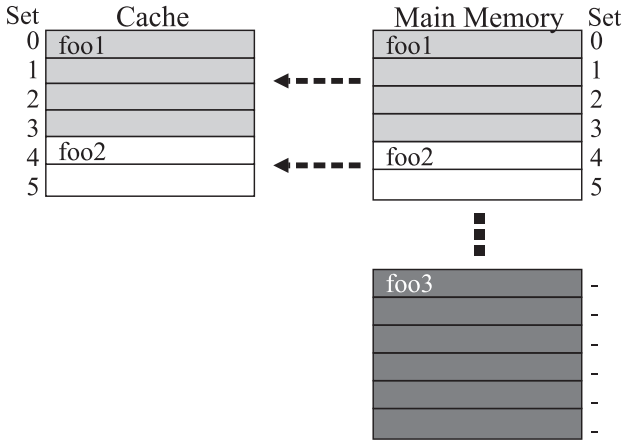

Figure 3. Exploiting non-cached memory areas to avoid cache thrashing

Greedy approach

It is well known [11] that even local code modifications can have a strong impact on the WCET of other parts of a program. This impact is hardly predictable without performing a complete static WCET-analysis of the program. Moreover, the situation becomes even more complicated if systems are equipped with caches. To deal with this handicap, a greedy optimization algorithm has been developed which moves single functions between memory areas and evaluates the influence on the WCET of the program to optimize by performing a WCET-analysis using aiT. The new assignment of functions to cached and non-cached memory areas acts as starting point for the next iteration which moves another function in case of a preceding WCET decrease. But if the WCET is degraded, the last modification is rolled back in order to guarantee that the optimized program is never worse than the original code w. r. t. its WCET.

Figure 4 shows the pseudo code of the greedy algorithm requiring a set *F* of functions to be optimized as input. The first two lines define sets of functions which are decided to be placed in a cached memory region (set *CF*) or a non-cached memory region (set *NCF*). The variable S is initialized with the cache size (line 3) and acts as counter for the free cache in bytes.

The profit for a function f if it will be moved from a non-cached memory region to a cached memory region is calculated in line 4. The profit of a function f is defined as:

$$profit(f) = \frac{WCET(f) - cachedWCET(f)}{size(f)}$$

$WCET(f)$ is the WCET of function f if it is allocated to a non-cached memory region, while $cachedWCET(f)$ is f's WCET if it is executed from a cached memory. To avoid side effects caused by cache conflict misses when two or more functions are mapped to the same cache lines,

Input: set<functions> F

```
1:  set<functions> CF = ∅
2:  set<functions> NCF = ∅
3:  S = cache size
4:  calculateProfit( F )
5:  while (S > 0 ∧ F ≠ ∅) do
6:    for f ∈ F : max( getProfit(f) ) do
7:      if ( getSize(f) > S ) then
8:        break;
9:      end if
10:     CF = CF ∪ f
11:     F = F \ f
12:     S = S − getSize( f )
13:   end for
14: end while
    /* From now on evaluate the WCET
    trend since the cache is full */
15: WCET lastWCET = evaluateWCET()
16: for all f ∈ F : max( getProfit(f) ) do
17:   CF = CF ∪ f
18:   F = F \ f
19:   WCET newWCET = evaluateWCET()
20:   if (newWCET > lastWCET) then
21:     CF = CF \ f
22:     NCF = NCF ∪ f
23:   else
24:     lastWCET = newWCET
25:     calculateProfit( F )
26:   end if
27: end for
28: return CF
```

Figure 4. Greedy WCET-driven memory content selection algorithm

either the cache has to be large enough to store copies of all functions or only a subset of functions has to be placed into the cached memory region and analyzed at a time.

In a few cases, a negative profit is detected. Then function f incurs a WCET increase due to a cached execution and therefore is excluded from optimization.

$cachedWCET(f)$ could be determined by moving each function separately into the cached memory and determining its WCET using a timing analyzer. However, this approach has the drawback that several static WCET analyses are required to compute $cachedWCET$ for all functions. The employed static WCET analyzer aiT is able to compute a program's WCET for systems with different cache sizes. To save time consumed by repetitive WCET analyses, a virtual large I-cache larger than the code size of the analyzed program is chosen for the evaluation. Moreover, we align each function at the start address of a cache line in order to achieve the same values as if all functions were analyzed separately.

The loop in lines 5-14 allocates functions to the cached memory as long as the overall code size still fits into the cache or not all functions are allocated. Inside the loop, the function with the highest profit (line 6) is removed from the set of unhandled functions and allocated to the cached memory region (lines 10-11). Line 12 keeps track of the allocated overall code size and lines 7-9 breaks the loop if the function with the highest profist does not fit into the cache anymore.

If the cache can store no more functions simultaneously, it is yet possible that allocating another function to a cached memory can decrease the WCET. Hence, lines 15-27 test if moving another function to the cached memory can achieve further reductions of the WCET.

Line 15 stores the current WCET of the program with the allocation decisions made in the preceding loop. The loop starting in line 16 iteratively moves one of the remaining functions with the highest profit to the cached memory region (line 17-18) and evaluates its effect on the WCET (line 19). If a WCET increase was detected in line 20, the following two lines roll the last change in the memory layout back and function f is stored in the set *NCF* of non-cached functions. Otherwise, the new decreased WCET serves as reference for the next iteration (line 24).

A new profit calculation is performed in line 25. Here, again, the WCET of the remaining functions $f \in F$ is taken into account if stored in non-cached memory as well as stored in cached memory. Therefore, we can recycle the WCET analysis results gathered in line 4 where $cachedWCET(f)$ was already determined for all functions in a cached memory. $WCET(f)$ can be reused from the results computed in line 19 where the remaining functions were stored in the non-cached memory.

$WCET(f)$ for a function f stored in a non-cached memory will be zero iff f is not lying on the WCEP. Thus, $profit(f) \leq 0$ is true and f is not consideres as a possible candidate to move to a cached memory area during the next loop iteration. Hence, we inherently keep track of a possible change of the WCEP.

The algorithm terminates if all functions are either located in set *CF* for functions to be cached or set *NCF* of functions to exclude from caching.

Finally, all functions in set *CF* will be attached to a section which is allocated to a cached memory area by the linker. All remaining functions of set *NCF* belong to the `.text` section which is allocated to a non cached area by default.

IV. Workflow

WCET-driven optimizations and especially cache-aware memory content selection need support of an underlying compiler to collect WCET data and perform the required memory layout modifications. We employ our WCET-aware C compiler framework, called WCC [13], which is intended to assist the development of various high- and low-level

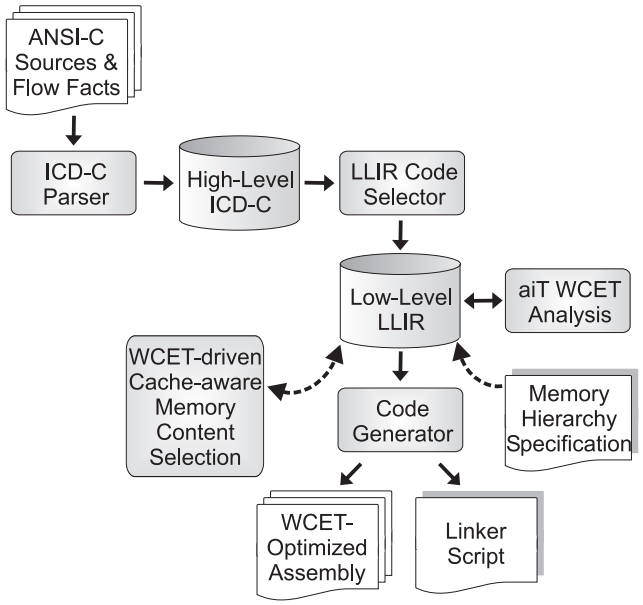

Figure 5. Workflow of the WCET-aware C compiler WCC

WCET-driven optimizations. WCC is a compiler for the Infineon TriCore TC1796 processor coupling AbsInt's static WCET analyzer aiT [2] which provides WCET data that is imported into the compiler backend and made accessible for optimizations.

Figure 5 depicts WCC's internal structure. One or more files of a program are read in the form of ANSI-C source files with user annotations for loop bounds and recursion depths, called *flow facts*. These source files are parsed and transformed into WCC's high-level intermediate representation (*IR*) *ICD-C* [14].

In the next step, the *LLIR Code Selector* translates the high-level IR into a low-level IR called *ICD-LLIR* [15]. On this TC1796-specific low-level IR, the cache-aware memory content selection is performed. To enable such a WCET-aware optimization, *aiT* is employed to perform static WCET analyses on the low-level IR. Therefore, mandatory information about loop bounds and recursion depths is supplied by flow fact annotations. These flow facts are automatically translated from the high-level IR to the low-level IR and are always kept valid and consistent during each optimization and transformation step of the compiler.

Optimizations exploiting memory hierarchies as well as the static WCET analyzer require detailed information about available memories, their sizes and access times. For this purpose, WCC integrates a detailed memory hierarchy specification available at *ICD-LLIR* level.

Finally, WCC emits WCET-optimized assembly files and generates suitable binaries using a linker script reflecting the utilized internal memory layout.

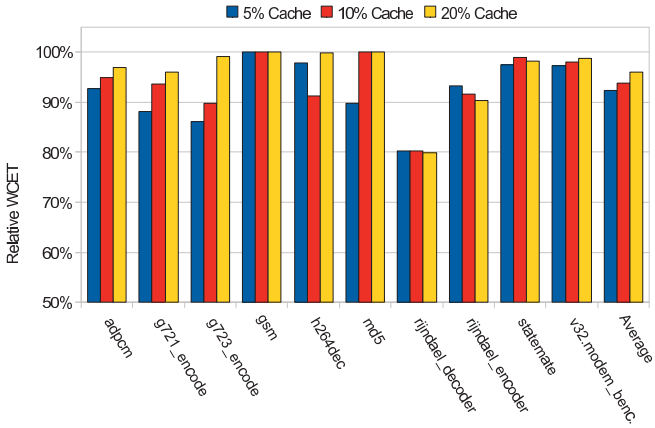

Figure 6. Optimization results for different cache sizes

V. Evaluation

This section evaluates the capability of our WCET-driven cache-aware memory content selection algorithm applied to real-life benchmarks. For benchmarking, we used the optimization level *O3* for which the WCC compiler activates 35 different optimizations in order to evaluate the performance of our new algorithm on highly optimized code. Our compiler supports the TC1796 processor with a 16 kB 2-way set associative I-cache and *last recently used* (LRU) replacement policy. The TC1796 integrates a 2 MB program Flash as main memory which is mapped to two different addresses in the memory layout. The first memory region allows a cached access to the Flash whereas code executed from the other region in the address space is excluded from caching.

For our measurements, we used the 10 largest benchmarks from the benchark suites *DSP Stone* [16], *Mediabench* [17], *MiBench* [18], *MRTC* [19], *NetBench* [20] and *UTDSP* [21]. The code size of the benchmarks range from 5 kB (*v32.modem_bencode*) up to 15 kB for the two *rijndael* benchmarks.

Today's embedded systems are equipped with main memories in megabyte ranges and caches typically ranging from 1 kB up to 16 kB or at most 32 kB. Due to this ratio of small cache compared to a large main memory, we artificially limited the cache sizes to 5, 10 and 20% of the program's overall code size. This guarantees, that we use a similar ratio of cache size to program size for all optimizations and static WCET analyses, found in current embedded systems in order to generate comparable results.

WCET Estimations

Figure 6 depicts the results achieved by our new memory content selection algorithm for the considered 10 benchmarks. For each benchmark, each of the three bars represent the results for different cache sizes. The bars depict the estimated WCET of the optimized program computed by the static WCET analyzer relative to the estimated WCET if the benchmark is executed with all functions located in a cached memory region.

Our algorithm was able to reduce the estimated WCET of the benchmarks by up to 19.5% for 5% and 10% cache size for the *rijndael_decoder*. For the same benchmark, we were able to achieve a WCET reduction of 20.1% for 20% cache size. The reasons why the worst-case performance of benchmark *gsm* could not be improved are twofold. First, the loop of a filter function which consumes about 95% of the program's estimated WCET amounts to only 3% of the program size and fits into all considered cache sizes. Second, the programs control flow is highly serial with only few branches. Thus, only few cache conflict misses occur and the best performance is achieved if all functions reside in a cached memory area during execution.

Due to the fact that our optimization is based on a heuristic to select the most promising functions, wrong decisions could be made leading to suboptimal results. The benchmark *h264dec* clearly indicates suboptimal results for the cache sizes of 5% and 20% of the program size. For these cases, the memory content selection algorithm optimized on local minimums where a different set of functions allocated to the cached memory would achieve a better worst-case performance.

On average, we were able to reduce the estimated WCET for all benchmarks by 8% for the smallest considered cache size and by 6 and 4% for 10% and 20% cache size, respectively.

For most of the benchmarks, it can be observed that our algorithm performs better for smaller cache sizes which has two reasons. On the one hand, there is more optimization potential since the number of cache misses necessarily grows with caches diminishing in size. On the other hand, the *principle of locality* [22] which states that programs tend to reuse data or instructions they have used recently also applies to most of the programs running on embedded systems. Based on the principle of locality, a widely known rule of thumb is that most of the programs spend 90% of their execution time in only 10% of the code [23]. As a consequence, the considered 10% cache size can often already keep copies of a program's hot spots. Thus, it is very likely that the execution time of the program can be improved by at most 10% by storing the remaining 90% of the code in the cache. The middle bar of each benchmark shown in Figure 6 represents exactly these 10% cache size. The 100% line represents the WCET of a system with a normally operating cache which can store copies of just these program's hot spots. Seen from this angle, the achieved average WCET reductions of 4-8% seem to be hardly improvable.

Optimization Time

To consider the optimization time, we utilized an Intel Xeon X3220 (2.40 GHz). Most of the time necessary for our novel WCET-driven cache-aware memory content selection algorithm was consumed by the repetitive WCET analyses using aiT. The maximal number of WCET analyses amounts to $n + 2$ where n is the number of functions. Initially, two analyses have to be performed in order to calculate the profit, and another after each allocation of a function to evaluate its impact on the overall WCET.

Most of the time was consumed by the optimization of the $rijndael\_decoder$ where 6 WCET analyses require almost 2 hours of CPU time. The highest number of WCET analyses (exactly 17) was performed during the optimization of the two benchmarks *g721_encode* and *g723_encode* which amounts to 8 respective 10 minutes of analysis time.

VI. Conclusions and Future Work

In this paper, we showed how the I-cache performance can be improved by intelligent memory content selection in order to decrease the WCET of a program. Frequently used functions whose WCET highly benefits from a cached execution are allocated to cached memory regions. This ensures that they can not be evicted from the cache by functions with a lower WCET decrease being allocated to non-cached memories.

We introduced a novel algorithm for WCET-driven cache-aware memory content selection which selects the set of functions to be cached. The presented greedy algorithm evaluates the influence on the WCET of a program when a promising function is moved to a cached memory region in order to optimize along the WCEP. Applying this technique, we were able to achieve a WCET decrease of up to 20% and ensure that the performance of the optimized program is never worse than the original. On average, WCET reductions between 4% and 8% were achieved for cache sizes ranging from 5-20% of the overall code size.

In the future, we intend to develop an integer linear programming based approach to select the optimal set of functions which should be allocated to a cache memory region w. r. t. the program's WCET.

Furthermore, we plan to combine our WCET-aware cache partitioning for multi-task hard real-time systems [24] with the optimizations presented in this paper. In systems with partitioned caches, each task is mapped into a separate part of the cache, thus, the cache capacity is split among the set of tasks. The performance of these tasks could be increased by a smart selection of functions which should be cached and those which would cause a performance decrease due to a high number of cache misses.

Acknowledgments

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

References

[1] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and Precise WCET Prediction by Separated Cache and Path Analyses," *Journal of Real-Time Systems*, vol. 18, no. 2-3, 2000.

[2] AbsInt Angewandte Informatik GmbH, "Worst-Case Execution Time Analyzer aiT for TriCore," 2009, http://www.absint.com/ait.

[3] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design," *IEEE Transactions on Computers*, vol. 42, no. 9, 1993.

[4] M. Kowarschik and C. Wei, "An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms," in *Algorithms for Memory Hierarchies*. Springer, 2003.

[5] V. Suhendra and T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores," in *Proc. of DAC*, Anaheim, USA, 2008.

[6] H. Falk, S. Plazar, and H. Theiling, "Compile Time Decided Instruction Cache Locking Using Worst-Case Execution Paths," in *Proc. of CODES+ISSS*, Salzburg, Austria, 2007.

[7] I. Puaut and D. Decotigny, "Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems," in *Proc. of RTSS*, Washington, DC, USA, 2002.

[8] X. Vera, B. Lisper, and J. Xue, "Data Caches in Multitasking Hard Real-Time Systems," in *Proc. of RTSS*, Cancun, Mexico, 2003.

[9] G. Gebhard and S. Altmeyer, "Optimal Task Placement to. Improve Cache Performance," in *Proc. of EMSOFT*, New York, USA, 2007.

[10] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez, "Procedure Placement using Temporal-Ordering Information: dealing with Code Size Expansion," *Journal of Embedded Computing*, vol. 1, no. 4, 2005.

[11] P. Lokuciejewski, H. Falk, and P. Marwedel, "WCET-driven Cache-based Procedure Positioning Optimizations," in *Proc. of ECRTS*, Prague, Czech Republic, 2008.

[12] H. Falk and J. C. Kleinsorge, "Optimal Static WCET-aware Scratchpad Allocation of Program Code," in *Proc. of DAC '09*, San Francisco, USA, 2009.

[13] H. Falk, P. Lokuciejewski, and H. Theiling, "Design of a WCET-Aware C Compiler," in *Proc. of ESTIMedia*, Seoul, Korea, 2006.

[14] R. Pyka and J. Eckart, "ICD-C Compiler Framework," http://www.icd.de/es/icd-c, 2009, Informatik Centrum Dortmund.

[15] J. Eckart and R. Pyka, "ICD-LLIR Low-Level Intermediate Representation," http://www.icd.de/es/icd-llir, 2009, Informatik Centrum Dortmund.

[16] C. S. Živojnović, J. Martinez and H. Meyr, "DSPstone: A DSP-Oriented Benchmarking Methodology," in *Proc. of ICSPAT*, Dallas, USA, 1994.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proc. of MICRO 30*, Washington, DC, USA, 1997.

[18] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proc. of IISWC*, 2001.

[19] Mälardalen WCET research group, "Mälardalen WCET Benchmark suite," http://www.mrtc.mdh.se/projects/wcet, 2008.

[20] G. Memik, W. H. Mangione-Smith, and W. Hu, "Netbench: A benchmarking suite for network processors," in *Proc. of ICCAD*, Piscataway, USA, 2001.

[21] "UTDSP Benchmark Suite," http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 2008.

[22] P. J. Denning, "The Locality Principle," *Communications of the ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 2003, p. 47.

[24] S. Plazar, P. Lokuciejewski, and P. Marwedel, "WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems," in *Proc. of WCET'09*, Dublin, Ireland, 2009.