WCET-driven Cache-aware Code Positioning*

Heiko Falk
Institute of Embedded Systems / Real-Time Systems
Ulm University
D - 89081 Ulm, Germany
Heiko.Falk@uni-ulm.de

Helena Kotthaus
Computer Science 12
Technische Universität Dortmund
D - 44221 Dortmund, Germany
Helena.Kotthaus@tu-dortmund.de

ABSTRACT

Code positioning is a well-known compiler optimization aiming at the improvement of the instruction cache behavior. A contiguous mapping of code fragments in memory avoids overlapping of cache sets and thus decreases the number of cache conflict misses.

We present a novel cache-aware code positioning optimization driven by worst-case execution time (WCET) information. For this purpose, we introduce a formal cache model based on a conflict graph which is able to capture a broad class of cache architectures. This cache model is combined with a formal WCET timing model, resulting in a cache conflict graph weighted with WCET data. This conflict graph is then exploited by heuristics for code positioning of both basic blocks and entire functions.

Code positioning is able to decrease the accumulated cache misses for a total of 18 real-life benchmarks by 15.5% on average for an automotive processor featuring a 2-way set-associative cache. These cache miss reductions translate to average WCET reductions by 6.1%. For direct-mapped caches, even larger savings of 18.8% (cache misses) and 9.0% (WCET) were achieved.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers; Optimization; C.3 [**Real-time and embedded systems**]; B.3.2 [**Memory Structures**]: Cache memories; B.3.3 [**Memory Structures**]: Worst-case analysis

General Terms

Algorithms, Performance

*Funded by the European Community's ArtistDesign Network of Excellence and by the European Community's 7th Framework Programme FP7/2007-2013 under grant agreement n° 216008.

CASES'11, October 9–14, 2011, Taipei, Taiwan.

Keywords

WCET, Cache, Code Positioning, Locality, Cache Miss

1. INTRODUCTION

In contrast to the speed of memories, processor speed has increased dramatically in the past. To bridge this large gap between processor and 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 – no code modification has to be performed since caches are hardware controlled.

In current software design flows, compilers play an important role since they are able to apply automated optimizations improving the quality of the generated executable code. Unfortunately, even modern optimizing compilers are often unable to quantify the effect of an optimization since they lack precise timing models [11]. Hence, simple ad-hoc heuristics are applied during optimization in the hope that they finally improve code quality. But it is well-known that this is not always true: due to the absence of precise models, optimizations may have a negative impact on code quality.

This particularly holds for cache-based processors. Since they are fully hardware-controlled, the latency of a memory access can vary considerably. If an item to be fetched already resides in the cache, a cache hit occurs and the memory access usually takes one clock cycle. Otherwise, a cache miss happens taking several clock cycles. Compilers that are unaware of a cache's behavior and that do not include a formal cache model eventually generate code with poor performance due to an increased number of cache misses.

Thus, caches are usually effective in reducing the *average-case execution time (ACET)*, provided that the compiler somehow generates code which does not lead to massive amounts of cache misses.

However, caches are problematic for real-time systems with hard timing constraints. The correctness of a real-time system depends on both the logical results and on the time at which the results are produced. A program's *worst-case execution time (WCET)* is used to guarantee that real-time constraints are safely met. Since caches are hardware controlled, it is virtually impossible to determine the latency of a memory access for many popular cache architectures. However, this is what tools for static WCET analysis of executable code, like e. g. *aiT* [1], have to do – to classify each memory access in a program as definite cache hit or cache miss. Whenever a static WCET analyzer is uncertain about the classification of a memory access, it must assume the worst case and usually consider the access to

be a cache miss. Thus, WCET estimates computed during static WCET analysis may be heavily overestimated in the presence of caches.

It is well-known that caches effectively exploit *temporal locality* and *spatial locality*. The former means that particular memory locations will be accessed within a short period of time. The latter refers to the reference of contiguous memory locations. Particularly with regard to instruction caches,[1] it is possible to improve the cache's performance and to decrease the WCET overestimation by applying code positioning during compilation. Code fragments which are mapped to the same cache lines and which exhibit a high temporal locality, e. g. since they are executed within a loop, evict each other from the cache very frequently. The cache misses resulting from this scenario are called *conflict misses* since the code fragments lie in conflict if their cache lines overlap. This overlapping within the cache can be resolved by positioning such code fragments contiguously in the main memory, thus resulting in a decreased number of conflict misses.

This paper presents formal models and techniques for code positioning in a WCET-aware fashion. The overall goal of the proposed approach is to systematically reduce I-cache conflict misses and thus to reduce the WCET of a program. The key contributions of this paper are

- its innovative formal cache model based on a conflict graph which is able to capture all currently used cache architectures, including direct-mapped and set-associative caches; this cache model is combined with a formal WCET timing model, resulting in an entirely novel cache conflict graph weighted with WCET data,
- a couple of cache-related analyses which are used to refine the above cache conflict graph,
- the exploitation of these formal cache and WCET models by novel heuristics for positioning of both basic blocks and entire functions, and
- the achieved reductions in terms of accumulated cache misses by 15.5% and of WCET estimates by 6.1% on average for 18 benchmarks, while requiring only moderate runtimes for optimization.

Section 2 gives a survey of related work on cache-based and WCET-aware compiler optimizations. Our novel conflict graph based on formal cache and WCET models and our cache analyses are the subject of Section 3. Section 4 presents the heuristics used for code positioning at basic block and function level. Section 5 describes the benchmarking results, and Section 6 summarizes this paper and gives an outlook on future work.

2. RELATED WORK

Code positioning as such was studied in many different contexts in the past. In [28], positioning of basic blocks was proposed in order to reduce WCETs. The authors reorder the positions of basic blocks such that unconditional jumps between blocks are avoided. This leads to a reduction of jump-related pipeline delays and thus reduces the WCET. Unfortunately, [28] only considers positioning in the context of jumps, and not in the context of caches. The authors only consider simple processors without any caches.

In contrast, [8] describes an optimization which considers caches and positions functions based on their call graph. Functions which call each other frequently are placed contiguously in memory in order to avoid repeated cache eviction. However, this approach is limited to direct-mapped caches and only considers ACET reduction. For this purpose, the required data to construct the call graph is collected using profiling, which is not feasible in the domain of hard real-time systems.

A code positioning of functions that also supports set-associative caches is described in [12]. Here again, the focus lies on ACET reduction, and the data required for positioning is gained using profiling. A WCET-aware procedure positioning was presented in [14]. Using a greedy heuristic based on a program's call graph enriched with WCET data, procedures are placed contiguously in memory in order to reduce WCET. However, this work does not rely on any model of a cache's behavior. Due to this, the impact of a modified procedure placement is evaluated in a trial and error fashion.

[7] positions entire tasks of a multi-task application in order to minimize the number of cache misses. The authors concentrate on the preemptive scheduling of tasks. Tasks preempting each other are placed in memory such that they are assigned to different regions of the cache. The goal here is to maximize the number of persistent cache sets to allow more precise WCET estimates for preemptively scheduled tasks. A similar approach was proposed in [16] where the I-cache is partitioned into disjoint regions which are then assigned to the tasks of a multi-task application. This way, it is ensured that the execution of one task never influences the cache partition of other tasks. Thus, task preemption does not have side-effects on other tasks which in turn leads to highly precise WCET estimates for multi-task applications.

Several processors support software-control of caches in the sense that the cache content can be locked and will not be evicted from the cache afterwards. This feature basically disables a cache's replacement mechanism and turns the cache into a rather conventional memory which is highly timing predictable. However, cache locking requires sophisticated compiler support in order to allocate frequently accessed code or data fragments into the cache before locking. Locking of data and instruction caches has been considered by e. g. [4, 13, 18, 24].

As an alternative to caches, so-called *scratchpad memories (SPMs)* were studied extensively in the past. SPMs have the advantage that they are fully timing predictable since an access to the SPM never has varying latencies. An SPM can be compared to a locked cache and thus requires dedicated compiler support, too. Approaches for SPM allocation of code and / or data in order to minimize energy dissipation or WCET have been proposed e. g. in [2, 20, 21].

The techniques mentioned in the last two paragraphs concerning cache locking and SPM allocation are complementary to code positioning since they all aim at avoiding a cache's inherent unpredictability using additional hardware – be it special instructions or registers for locking a cache or be it the presence of scratchpad memories. In contrast,

[1] We exclusively consider I-caches in this paper due to their practical relevance for safety-critical real-time systems. D-caches are usually completely deactivated in such systems since it is extremely difficult to get their worst-case timing behavior under control.

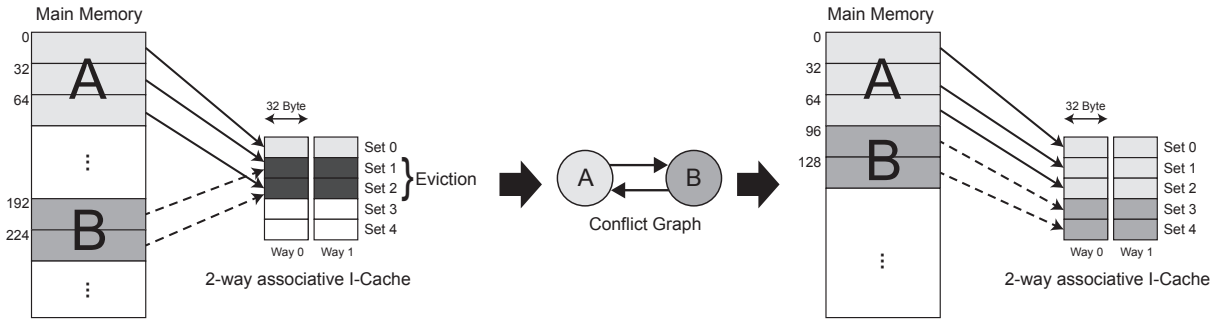

Figure 1: Resolution of Cache Conflict Misses using Code Positioning based on a Conflict Graph

this paper focuses on cache-based processors without such hardware extensions.

Finally, [25] proposes a cache-aware SPM allocation. The goal of this work is to minimize energy dissipation by allocating code fragments which lead to high numbers of cache conflict misses to the SPM. The interaction of code fragments in the I-cache is modeled using a conflict graph. This graph contains information about basic blocks, whether pairs of basic blocks evict each other, and whether this happens frequently and thus leads to a large energy overhead. [25] serves as a basis for our work in that sense that we also model cache conflicts by a conflict graph. However, the conflict graph described in Section 3 relies on WCET data instead of energy data. Furthermore, our conflict graph is used to model both functions and basic blocks. Finally, we use the conflict graph for cache-aware code positioning, without relying on the presence of additional hardware like e. g. SPMs.

As can be seen from this survey of related work, there currently exist no code positioning techniques which focus on WCET reduction under simultaneous consideration of actual cache models. The remainder of this paper thus presents our formal cache and WCET model in the next section, followed by code positioning heuristics which are able to deal with both functions and basic blocks.

3. WCET-AWARE CACHE CONFLICT GRAPH AND ANALYSES

The code positioning techniques proposed in this paper rely on a cache conflict graph as the underlying model of a cache's behavior. Section 3.1 presents the formal definition of this conflict graph. However, it is likely that the conflict graph contains many superfluous conflict edges that do not model actual cache conflicts. Thus, analyses are applied which help to refine the conflict graph so that such superfluous conflict edges are removed. A refinement of the conflict graph based on control flow analyses is presented in Section 3.2. The exploitation of cache may analyses in the context of the conflict graph is the subject of Section 3.3. Finally, information on a program's memory layout is used in a last refinement step described in Section 3.4.

3.1 The WCET-aware Cache Conflict Graph

The conflict graph used throughout this paper is defined as follows:

Definition 1 (Conflict Graph, CG)
A conflict graph $G = (V, E, w)$ is a directed graph whose edges are weighted.

A node v_i denotes a code fragment of a program. The set of nodes $V = \{v_1, \ldots, v_n\}$ includes a node v_i if the code fragment v_i lies in conflict with some other code fragment v_j. Either, all nodes of V represent basic blocks, or they all represent entire functions.

The set of edges E contains an edge $e_{i,j}$ starting at v_i and ending at v_j if a cache line that contains code of v_i can be replaced by code of v_j.

The weight $w_{i,j}$ of edge $e_{i,j}$ approximates the number of possible cache misses that are caused during the execution of v_i due to v_j, since v_j evicts cache lines of v_i.

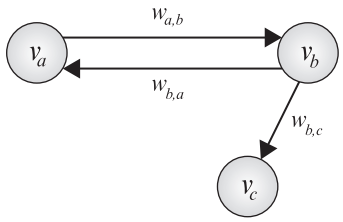

Figure 2: Conflict Graph according to Definition 1

Figure 2 depicts an example conflict graph. An edge like e. g. $e_{a,b}$ is directed to node v_b if v_b is responsible for cache misses of the edge's source node v_a. Depending on whether the node set V represents basic blocks or entire functions, the conflict graph is called local or global. As depicted in Figure 1, the conflict graph models in how far code fragments overlap in the cache and thus potentially evict each other from the cache. The conflict graph can then be exploited in order to place such conflicting code fragments contiguously in memory. This helps to resolve such conflict misses, since nodes of the conflict graph are no longer mapped to the same cache set unless the combined size of the nodes exceeds the size of the cache.

To obtain the conflict graph from Definition 1, it is necessary to perform a static WCET analysis beforehand. The WCET of a program P is equal to the length of the longest possible execution path from the start node to an end node in P's *control flow graph (CFG)*. For such a path, its length is the sum of the products of WCET and worst-case execution frequency for all basic blocks of the path. This longest path is also known as *Worst-Case Execution Path (WCEP)*.

Static WCET analyzers a priori compute how often pieces of code are executed in the worst case. For basic blocks, a WCET analyzer thus computes the so-called *worst-case execution count (WCEC)*, and for functions the *worst-case*

calling frequency (WCCF). It is possible that WCET analysis reveals that certain code fragments are never executed in the worst case since they never lie on a program's WCEP. Such code fragments have a WCEC or WCCF of 0. Since such pieces of code do not contribute to a program's WCET at all, it is not necessary to include them into our conflict graph. Thus, the CG's node set V consists of all basic blocks or functions with non-zero WCEC or WCCF, respectively.

For the generation of the graph's edge set E, it must be known which code fragments overlap in the cache and thus are in conflict with each other. For this purpose, Equation (1) defines a function which maps an address of a code fragment to its belonging cache set:

$$Map(Addr) = (Addr \gg Off) \bmod \frac{C}{A * S} \tag{1}$$

Addr denotes a main memory address of a code fragment. The least significant bits of an address contain the offset which is used for addressing of a word within a cache line. Since the offset does not influence to which cache set an address is mapped, this offset is stripped from *Addr* by shifting *Addr* right by the required number *Off* of bits.

The remaining relevant part of the address is then mapped to a cache set by simply computing the modulus of the right-shifted address and the total number of sets available in a cache. This amount of available cache sets is computed by dividing the cache's capacity C given in bytes by its associativity A multiplied by the cache line size S in bytes.

Assuming that it is known at compile-time at which addresses the code fragments of a program start and end, it is easy to compute which cache sets are occupied by the code fragments using Equation (1). For each pair of code fragments v_i and v_j that have at least one cache set in common, two directed edges $e_{i,j}$ and $e_{j,i}$ are inserted into the CG. Furthermore, Equation (1) obviously shows that our conflict graph is inherently able to model a broad class of cache architectures with varying associativities, including e. g. direct-mapped caches with an associativity of 1. In the following we will focus on cache architectures using the *least-recently-used (LRU)* replacement policy. The LRU policy is known to allow static analyses to predict the cache content best [19].

By default, the edge weights of the CG are initialized with the value 0. In addition to the abovementioned worst-case execution frequencies, a static timing analyzer also computes worst-case numbers of cache misses. All this information is computed for edges of the control flow graph during WCET analysis. A cache miss value x at CFG edge (v_i, v_j) denotes that a number of x cache misses will occur in the worst case if control is transferred from node v_i to v_j.

Unfortunately, this CFG edge-related data computed during static WCET analysis can not directly be used as edge weights $w_{i,j}$ for the conflict graph of Definition 1. This is due to the fact that the CG contains edges which might not be present in a program's actual CFG: whenever any two nodes of the conflict graph share at least one common cache line, edges are introduced into the CG, irrespective of the control flow relationship of the two nodes.

In order to bridge this gap between the CFG model used during static WCET analysis and the conflict graph model proposed in the present paper, we accumulate the CFG edge-related data computed by the timing analyzer, attach this accumulated data to CFG nodes and finally translate this CFG node-related data to edge weights in our CG.

Modern WCET analyzers are context-sensitive – they consider via which control flow paths each CFG node can possibly be reached, and they compute individual WCEC and cache miss values for each CFG edge and context. For example, basic blocks and their edges inside loops typically have different contexts which represent the individual iterations of the surrounding loop. For the very first loop iteration and its corresponding context, WCET analysis typically computes a relatively large number of cache misses in the worst case since the loop's code needs to be loaded into the cache. For the remaining iterations and contexts, usually lower cache miss numbers are determined.

To model the CG's edge weights $w_{i,j}$, this context-related data produced during WCET analysis is used. For each CFG node v_i, its outgoing CFG edges and contexts are considered and accumulated cache misses are computed as follows:

$$AccMisses_i = \sum_{\substack{\forall v_j \in succ(v_i): \\ (v_i, v_j) \in CFG\ edges}} EdgeMisses_{i,j} \tag{2}$$

$$EdgeMisses_{i,j} = \sum_{contexts\ c} Misses_{(i,j),c} * WCEC_{(i,j),c}$$

According to Equation (2), the $Misses_{(i,j),c}$ of a CFG edge leaving node v_i are multiplied by the edge's worst-case execution count $WCEC_{(i,j),c}$ for each context c. CFG edges which are provably never executed since they are e. g. infeasible are properly treated this way, because such edges have a WCEC of 0 so that they do not influence a block's accumulated cache misses. Thus, only true cache misses which can actually occur in the worst case are considered by Equation (2).

Accumulating all context-related data for $EdgeMisses_{i,j}$ implies a certain loss of information since fine-grained data for individual contexts is no longer available. However, this is not a limitation, since fine-grained context-sensitive information is not required for code positioning. This is because code fragments v_i and v_j occur exactly once in the machine code, and not several times as virtually assumed when distinguishing between different contexts. In order to decide on the code positioning of v_i and v_j, the overall number of cache conflicts between these code fragments is of importance, irrespective of the circumstance from which context individual cache misses stem.

As a final step, these CFG node-related accumulated cache misses $AccMisses_i$ need to be translated to edge weights $w_{i,j}$ in our conflict graph. $AccMisses_i$ represents the total number of worst-case cache misses of v_i. This total number of cache misses needs to be distributed among all other nodes v_j which v_i lies in conflict with. For this purpose, $AccMisses_i$ is multiplied by the number of cache sets $Sets_{i,j}$ that v_i and v_j have in common, divided by the total amount of cache sets $Sets_i$ occupied by v_i. $Sets_{i,j}$ and $Sets_i$ can be computed easily using Equation (1). Thus the CG's edge weights are defined as follows:

$$w_{i,j} = \begin{cases} AccMisses_i * \frac{Sets_{i,j}}{Sets_i} & \text{for } \mathrm{CG}_{\mathrm{local}} \\ AccMisses_i * \frac{Sets_{i,j}}{Sets_i} * WCCF_i & \text{for } \mathrm{CG}_{\mathrm{global}} \end{cases} \tag{3}$$

3.2 Refinement using Control Flow Analysis

Code fragments which are mapped to some common cache sets need not necessarily lie in conflict with each other. If e. g. one fragment belongs to the *then*-part and the other

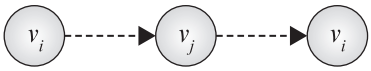

Figure 3: Control flow path of a conflict edge $e_{i,j}$

one belongs to the *else*-part of an *if-else* statement, the corresponding two nodes of the CG potentially need not be connected by edges. The fact that one piece of code evicts another piece of code in the cache thus heavily depends on the location of both code fragments in a program's CFG.

A CG edge $e_{i,j}$ denotes the situation that v_i encounters cache misses due to v_j. However, this can only happen if v_i is executed both before and after v_j within a program's control flow, as depicted in Figure 3.

v_i needs to be executed *before* v_j in order to be evicted from the cache by v_j. Due to the very first execution of v_i, its code is loaded into some cache sets. If v_i is not executed before v_j, the cache cannot contain any code of v_i which will later be evicted by v_j.

Additionally, v_i also has to be executed *after* v_j so that it actually comes to cache misses due to v_j. v_j has evicted some sets from the cache which were originally occupied by v_i. During v_i's second execution, these evicted cache sets have to be reloaded from main memory which leads to actual cache misses. If v_i is not executed after v_j, parts of v_i can be evicted from the cache by v_j, but this does not lead to actual cache misses since v_i is no more executed.

This argumentation holds for any two nodes v_i and v_j which are connected by a control flow path as shown in Figure 3. The two nodes do not necessarily have to be direct neighbors in a program's CFG. It is sufficient that one path exists within the CFG leading from v_i to v_j, and another CFG path from v_j back to v_i. This is indicated by the dashed edges in the above figure.

Thus, all edges in the conflict graph whose corresponding nodes do not fulfill the CFG-based criterion sketched above do not model actual cache conflicts. For this reason, edges are removed from the conflict graph as follows:

Definition 2 (CG Refinement: Control Flow-based) *An edge $e_{i,j}$ is removed from the conflict graph if either no directed path from v_i to v_j or from v_j to v_i exists in the program's control flow graph.*

The criterion of Definition 2 is easy to check by simply applying a standard reachability analysis to the CFG. Apparently, only such edges $e_{i,j}$ are kept in the CG where the corresponding nodes lie in a (possibly nested) loop structure.

3.3 Refinement using May Analysis

A code fragment v_j can only evict some other block v_i from the cache if v_i is actually contained in both nodes' common cache sets prior to v_j's execution.

For this purpose, the control flow-based analysis presented in Section 3.2 checks the execution order of both code fragments. However, the criterion of Definition 2 cannot determine whether v_i is still contained in the cache immediately before v_j is executed. For example, v_i could be completely evicted from the cache by some other block v_k lying somewhere on the control flow path from v_i to v_j.

Thus, an edge $e_{i,j}$ in the conflict graph would not represent an actual cache conflict between v_i and v_j since the eviction of v_i is not caused by v_j but by some other block v_k. If it can be excluded that v_i is contained in the cache immediately before it comes to the execution of v_j, the edge $e_{i,j}$ can be removed from the conflict graph.

In order to detect such superfluous CG conflict edges, we apply a cache *may analysis* as already published in [5, 6]. For each point in a program's CFG, the abstract cache states and thus the states of the individual cache sets are estimated. For a cache may analysis, the cache states include information which code fragments *possibly* reside in a cache set. Thus, it can be deduced which code fragments definitely do not reside in the cache at certain points in the CFG.

For this purpose, incoming cache states are attached to the CFG nodes. They model the possible contents of cache sets immediately before the node's execution. May analysis iteratively traverses the CFG and refines these cache states by applying the principle of abstract interpretation. During the progress of may analysis, update and join functions are used to update a node's incoming cache states.

May analysis terminates as soon as all cache states are stable and are no longer changed. Since may analysis is a well-established technique, we refrain from describing its details here. Instead, the interested reader is referred to the original publications [5, 6]. However, it is worth to note that cache may analysis is used in a generic fashion in order to support configurable cache associativities. Thus, also this heavily cache-dependent part of our conflict graph-based framework supports broad classes of cache architectures.

Based on the results of cache may analysis, edges are removed from the conflict graph as follows:

Definition 3 (CG Refinement: May Analysis) *An edge $e_{i,j}$ is removed from the conflict graph if node v_i is not part of the incoming cache states of node v_j.*

3.4 Refinement using the Memory Layout

The code positioning approach described in this paper aims at reducing cache conflict misses by positioning code fragments contiguously in memory. This has the advantage that it is very likely that contiguously placed pieces of code are not mapped to the same cache sets. However, the initial conflict graph as specified in Definition 1 can contain edges between nodes v_i and v_j which are already placed contiguously in memory. This occurs if the combined size of two code fragments exceeds the size of one way. Thus, such CG edges denote conflicts which do not exhibit any actual optimization potential.

For this reason, CG edges are removed depending on the memory layout of their corresponding nodes:

Definition 4 (CG Refinement: Memory Layout) *An edge $e_{i,j}$ is removed from the conflict graph if nodes v_i and v_j are already placed contiguously in memory.*

These refinement criteria of Definitions 2 to 4 are sequently applied to the initial conflict graph of Definition 1. The resulting refined local or global conflict graph is then used as input for the code positioning heuristics described in the following.

4. CODE POSITIONING HEURISTICS

To reduce WCETs, cache-aware code positioning resolves conflict misses between basic blocks and functions by placing

```
1  LLIR WCET_CodePositioning( LLIR P ) {
2    wcet_cur = WCETAnalysis( P );
3    foreach ( function f ∈ P )
4      do {
5        Graph CG_local = BuildCG( f );
6      } while ( Position( CG_local, P ) );
7    do {
8      Graph CG_global = BuildCG( P );
9    } while ( Position( CG_global, P ) );
10   return P;
11 }

12 bool Position( Graph G, LLIR P ) {
13   Edge e_max_i,j = G.removeMaxEdge();
14   if ( e_max_i,j ≠ ∅ ) {
15     positionNextTo( v_i, v_j );
16     if ( G.isLocal() )
17       adjustJumps();
18     wcet_new = WCETAnalysis( P );
19     if ( wcet_new < wcet_cur ) {
20       wcet_cur = wcet_new;
21       return true;
22     } else
23       revertPositioning();
24   }
25   return false;
26 }
```

Figure 4: Algorithm for WCET-driven Cache-aware Code Positioning

those conflicting code fragments contiguously in main memory. For this purpose, local conflict graphs are generated to position basic blocks. Afterwards, positioning is done at function level with the aid of a global conflict graph.

Our code positioning heuristic is based on a greedy approach. The edge weights of a conflict graph are used to identify code fragments with a high number of conflict misses and thus with a high optimization potential. Starting with the edge having the largest weight, the algorithm evaluates the influence on the WCET when this edge – and therefore the conflict misses it models – is removed by a reallocation of the corresponding basic blocks or functions. If a WCET reduction was achieved, the resulting altered memory layout of the program is kept so that it serves as starting point for the next optimization cycle. This ensures that the WCET of the altered memory layout is never worse compared to the original layout.

The formal definition of the WCET-driven code positioning algorithm is depicted in Figure 4. The input of the algorithm is the *low-level intermediate representation (LLIR)* of program P. The positioning at basic block level takes place in lines 3-6, while lines 7-9 illustrate the positioning at function level. In line 5, a local conflict graph CG_{local} is built for each function f in P, based on the information of the WCET analysis performed in line 2. $wcet_{cur}$ represents the current WCET of the program P and is used later for evaluation. The positioning at basic block level terminates if every local conflict graph is processed by function `Position()`. `Position()` is also used for positioning at function level when processing the global conflict graph CG_{global}, which is built in line 8. Positioning at function level terminates if a reordering of two functions was not successful, i. e. the resulting WCET starts to get worse, or all conflict graph edges were processed. In that case, function `Position()` returns `false`.

The function `Position()` itself is defined in lines 12-26. The input of this function can be a global or a local conflict graph. Based on the edges belonging to graph G, the algorithm positions the corresponding code fragments. Thereby, positioning always starts with the edge having the largest weight and thus the largest optimization potential. In line 13, the edge with the largest weight is removed from conflict graph G and is stored in $e_{max_{i,j}}$. In line 15, the two corresponding basic blocks or functions v_i and v_j are contiguously placed in memory by the function `positionNextTo()`. This leads to an altered memory layout.

In contrast to positioning at function level, code positioning at basic block level requires additional jump adjustments (line 17) after each reallocation. Therefore, `adjustJumps()` is called right after a reallocation of two basic blocks. Unlike positioning at function level, the reallocation of basic blocks has to consider the control flow of a program. If two basic blocks A and B initially located next to each other in control flow and in main memory are being separated, jump instructions need to be inserted, or existing jump instructions have to be modified in order to ensure the correctness of control flow. For example, if basic block A ends with a conditional jump instruction and the condition won't meet, the subsequent basic block B will be executed. If those basic blocks will be separated due to a reallocation, they have to be reconnected via an additional jump instruction by the function `adjustJumps()`. Also, superfluous jump instructions can be removed if two basic blocks previously separated are positioned contiguously due to a reallocation.

To evaluate the resulting memory layout, a WCET analysis of program P (line 18) is necessary. The new WCET of P is stored in $wcet_{new}$ and is compared with the previously computed value of $wcet_{cur}$ (line 19). If the positioning of the code fragments v_i and v_j based on $e_{max_{i,j}}$ was successful and achieves a WCET reduction, the new memory layout will be the starting point for the next optimization cycle. In that case, `Position()` returns `true`. Otherwise, the positioning is reverted by function `revertPositioning()` in line 23, and `Position()` returns `false`. In that case, positioning at basic block level processes the conflicts within the next function f of P (line 3), or the positioning at function level terminates (line 9).

As stated in Section 3.1, a program P's WCET is equal to the length of its longest execution path WCEP. Thus, a compiler aiming at WCET reduction has to shorten the length of the WCEP. Assume p_1 is P's current WCEP and some disjoint path p_2 is the second longest path in the CFG. If a compiler optimization is successful in shortening p_1 by more than $|p_1| - |p_2|$ time units (where $|p|$ stands for the length of p), p_2 becomes the new WCEP after this optimization. This phenomenon of switching WCEPs in the course of an optimization is known as the instability of the WCEP.

Due to the instability of the WCEP, any modification of the memory layout of P might potentially change the WCEP. Once two code fragments are reallocated success-

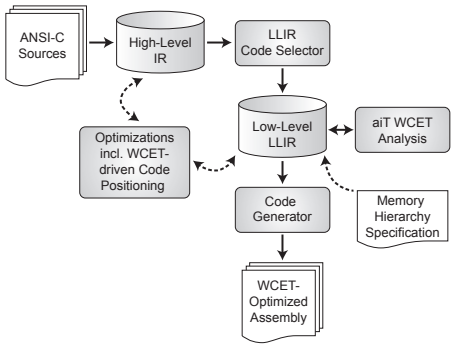

Figure 5: WCET-aware C Compiler WCC [27]

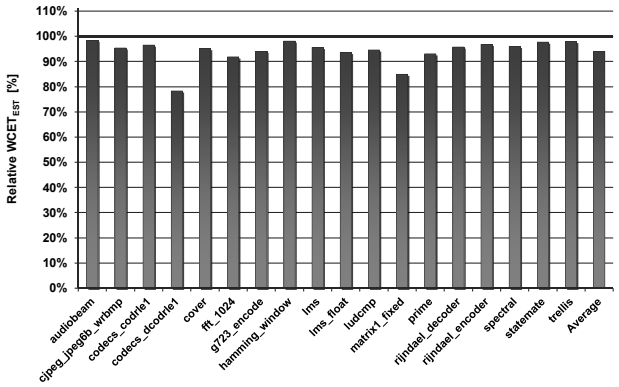

Figure 6: Relative WCET Estimates after Code Positioning (associativity 2)

fully, this can affect other code fragments and might cause new conflicts or even solve conflicts between other code fragments. Thus, one single invocation of `Position()` potentially changes the memory addresses and cache misses of lots of other code fragments. For this reason, the conflict graph is fully rebuilt after a successful reallocation of code fragments, i. e. if `Position()` returns `true` in line 6 (basic block level) or in line 9 (function level). This necessarily includes a complete WCET analysis in order to capture the novel worst-case cache misses and to obtain the actual CG edge weights after one single code positioning. This approach ensures an effective WCET reduction, since only code fragments relevant for the WCET are optimized. Without a rebuild of the CG, the next positioning might be superfluous and thus might not lead to a further WCET reduction. Only after rebuilding the CG in lines 5 and 8, function `Position()` is called again to continue code positioning using the edge with the next largest optimization potential.

In the algorithm shown in Figure 4, positioning at function level takes place after positioning at basic block level. It is also possible to switch this order, because the order is not crucial for WCET reduction. Results achieved by applying the presented code positioning heuristic at both basic block and function level are presented in the following section.

5. EVALUATION

This section presents results obtained by applying the proposed WCET-driven cache-aware code positioning techniques to real-life benchmarks. Section 5.1 describes the experimental setup used to perform benchmarking. The following sections discuss the results in terms of worst- and average-case execution times, code sizes and optimization runtimes.

5.1 Experimental Setup

Our code positioning techniques are integrated into a compiler for Infineon TriCore TC1797 processors. The TC1797 features a program flash memory with a size of 2 MB and an instruction cache with a size of up to 16 kB which is 2-way set-associative and uses the LRU replacement. The cache's line size is 32 bytes. In case of a cache hit, an access to program code memory takes place within one cycle whereas a cache miss takes up to 12 cycles. In order to demonstrate the flexibility of our approach w. r. t. different cache architectures, we also assume varying associativities ranging from 1 up to 16 in the following.

Figure 5 depicts the structure of the compiler which we used for benchmarking [3]. One of its key features is the tight integration of the static WCET analyzer *aiT* [1] into the compiler's backend. This way, WCET timing data required to generate the conflict graph is available in the compiler. The second important aspect of the compiler is the fact that it is up to the compiler to decide about a program's memory layout. For this purpose, the compiler has detailed knowledge about the processor's memory hierarchy, and the compiler explicitly specifies at which addresses individual basic blocks or functions are placed in memory. This property is exploited by our code positioning approach during conflict graph generation (cf. Equation (1)) and memory layout-related refinement (cf. Section 3.4).

Both at C and assembly level, code optimizations are applied to the high-level intermediate representations, called ICD-C [9] and to the LLIR. One of these optimizations, applied as very last one at assembly level, is the code positioning discussed in this paper. The compiler features a total of 42 different optimizations. For benchmarking, all of them are activated such that code positioning is always applied to already highly optimized code.

Code positioning was applied to a total of 18 different real-life benchmarks from the MRTC [15], MediaBench [10], UTDSP [23], DSPstone [26] and other benchmark suites. The code sizes of the benchmarks range from 160 bytes up to 10.4 kB with an average code size of 3.3 kB per benchmark. Since the benchmarks' code sizes are considerably smaller than the totally available cache capacity, we artificially limit the available cache size for benchmarking.

For every single benchmark, a cache size of 10% of the benchmark's code size was used [17]. For the smallest benchmarks where 10% of their code size results in a cache size of less than 2 sets, a minimal cache size of 2 sets was assumed. Whenever static WCET analysis was done during benchmarking, the analyzer *aiT* was invoked such that highest precision is achieved, i. e. an infinite number of execution contexts was assumed wherever feasible. Only for very few and complex benchmarks, we manually reduced the number of execution contexts in order to keep analysis times of *aiT* acceptable. For example, we set the number of contexts for benchmark `audiobeam` to 10 for this reason.

5.2 Worst-Case Execution Time Estimates

Figure 6 shows the impact of our code positioning techniques on the WCET estimates *($WCET_{est}$)* of all considered

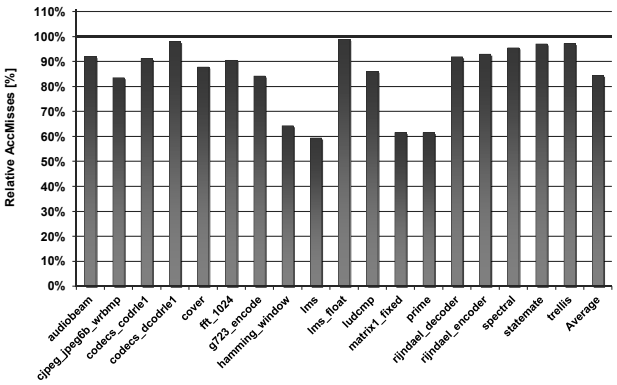

Figure 7: Relative Accumulated Cache Misses (associativity 2)

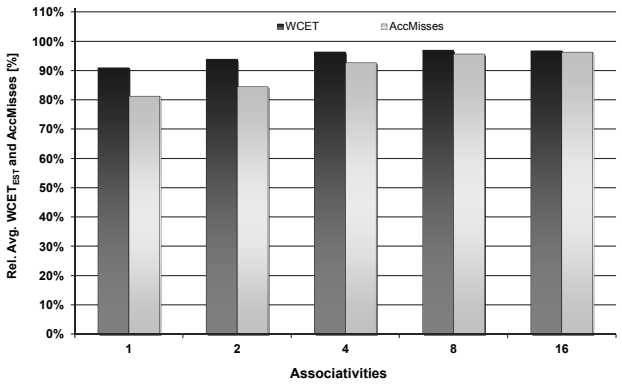

Figure 8: Relative average WCET Estimates and Accumulated Cache Misses for varying associativities

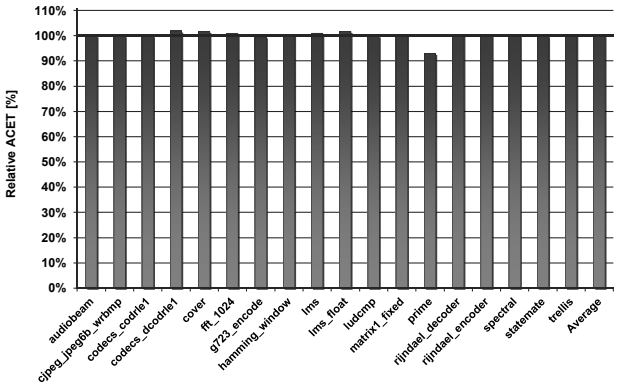

Figure 9: Relative ACETs after Code Positioning (associativity 2)

benchmarks. The 100% baseline of the diagram represents the benchmarks' $WCET_{est}$ if no code positioning is applied, i. e. the WCET estimates resulting from the compiler's highest optimization level *O3*. Results are presented as the percentage of the benchmarks' $WCET_{est}$ after code positioning.

As can be seen, our WCET-driven and cache-aware code positioning is able to reduce the $WCET_{est}$ for all benchmarks. For `audiobeam`, the $WCET_{est}$ after code positioning is 98.4% of the original $WCET_{est}$, i. e. a minor $WCET_{est}$ reduction of 1.6% was achieved. The largest gain in terms of $WCET_{est}$ was observed for `codecs_dcodrle1` where the $WCET_{est}$ after our code positioning amounts to only 78.3% of the original $WCET_{est}$, leading to savings of 21.7%.

On average over all 18 considered benchmarks, and assuming a cache associativity of 2, we were able to obtain a $WCET_{est}$ of 93.9% of the original worst-case execution time estimate, corresponding to a total average $WCET_{est}$ reduction of 6.1%.

Figure 7 shows the benchmarks' accumulated cache misses as defined in Equation (2). Again, these results are presented in a relative fashion where the 100% baseline represents the cache misses of the benchmarks without code positioning.

In particular, benchmarks `lms` or `prime` show a clear reduction of accumulated cache misses. In contrast, only few accumulated cache miss reductions were observed for benchmark `lms_float`. This is caused by the extremely tiny code size so that a minimal cache size of 2 sets was assumed (cf. Section 5.1). This has the effect that the cache is larger than the usually assumed fraction of 10% of a benchmark's code size. Thus, only a rather limited conflict potential exists for this benchmark.

As can be seen from Figure 7, our code positioning approach reduces the accumulated cache misses from 1.2% (`lms_float`) up to 40.6% (`lms`). On average over all benchmarks, our approach is able to reduce accumulated cache misses by 15.5%.

Besides the TriCore TC1797's native cache associativity of 2, the analyzer *aiT* allows to perform static WCET analyses assuming a variety of other associativities. We exploited this fact and our conflict graph's flexibility in terms of associativity and provide results for caches with associativities of 1 (direct-mapped), 2, 4, 8 and 16 in the following. Figure 8 shows the relative WCET estimates and accumulated cache misses as average values over all 18 considered benchmarks.

As can be seen from the figure, the optimization potential exploited by our code positioning steadily decreases for increasing cache associativities. For a direct-mapped cache, the accumulated cache misses are reduced by 18.8%, which translates to $WCET_{est}$ reductions by 9.0%. However, for a 16-way set-associative cache, significantly smaller improvements by 3.7% in terms of accumulated cache misses and by 3.2% in terms of $WCET_{est}$ were achieved. This behavior is not surprising, as the number of cache conflict misses decreases with increased associativity since more cache lines are available per set and thus less evictions happen.

5.3 Average-Case Execution Times

Figure 9 shows the impact of code positioning on average-case execution times (ACET) of our benchmarks for a 2-way set-associative cache. ACETs were measured using the commercial simulator *CoMET* [22] for the TC1797. Simulations were done using one input set per benchmark stemming from the benchmark suites mentioned above. Once again, ACETs after WCET-driven cache-aware code positioning are depicted as a percentage of the ACETs resulting from optimization level *O3* without code positioning.

A comparison of Figures 6 and 9 shows that the measured ACETs behave completely different than the WCET estimates resulting from our code positioning. It can be seen that, in contrast to WCETs, ACETs are not significantly

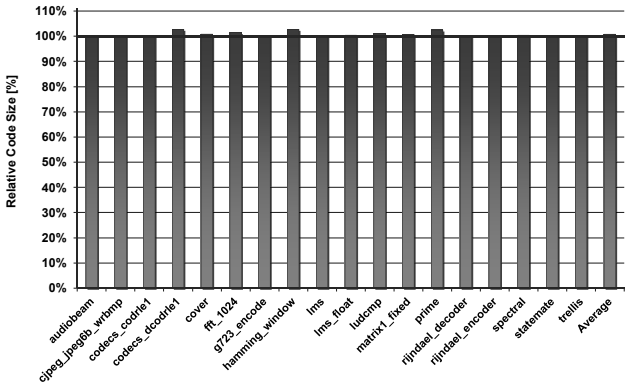

Figure 10: Relative Code Sizes after Code Positioning (associativity 2)

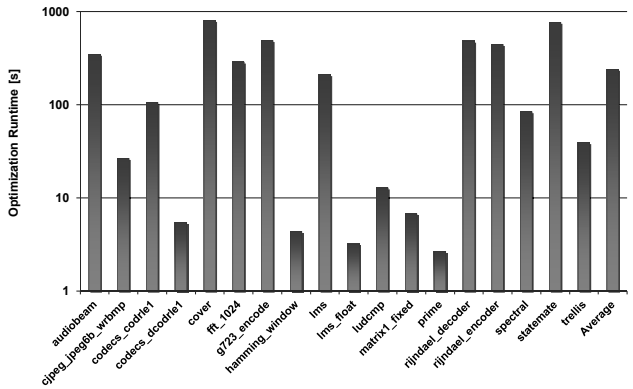

Figure 11: Absolute Runtimes required for Code Positioning (associativity 2)

reduced by our code positioning. Even worse, ACETs are increased by a few percents for some benchmarks, e. g. by 1.8% for `codecs_dcodrle1`.

This behavior can be explained by the fact that our code positioning is a fully WCET-oriented optimization which keeps on optimizing along a program's worst-case execution path (WCEP) which is usually not identical to the path that is executed in a typical average-case scenario. Hence, our approach reduces cache misses of those basic blocks and functions which contribute most to the global WCET estimate of a program. This strategy can have the effect that ACETs are increased since e. g. the layout of such basic blocks which lead to a low ACET is changed so that they generate more cache misses in the average case afterwards. For this reason, it is likely that the total amount of cache misses in the average case is probably displaced within our benchmarks, but not systematically reduced.

Only for four benchmarks, ACET reductions were observed: e. g. `prime` exhibits an ACET reduction by 7% while the ACET of `statemate` is reduced by 0.03%. On average for all considered benchmarks, our code positioning approach leads to a minor ACET increase by 0.03%. It can be concluded that optimizing for WCET is a completely different issue than optimizing for average-case performance.

When considering cache associativities ranging from 1 up to 16, it turns out that ACETs do not vary notably. For a direct-mapped cache, a negligible ACET reduction of 0.07% was achieved, while an ACET increase by 0.28% was observed for a 16-way set-associative cache. Due to the obviously irrelevant impact of associativity on ACETs, we refrain from presenting more detailed results here.

5.4 Code Sizes and Optimization Runtimes

Figure 10 depicts the influence of our code positioning on the benchmarks' code sizes. Again, results are presented as a percentage relative to the code sizes achieved when not applying code positioning. As can be seen from this diagram, code positioning has only a marginal effect on code size. For most benchmarks, code size is not affected at all. For others, code sizes are slightly increased, up to a maximal code size increase by 2.8% for benchmark `codecs_dcodrle1`.

These minor code size increases can be explained by the jump correction stage which is necessary after code positioning. Placing basic blocks contiguously in memory might have the effect that the control flow through a program is illegally modified. In order to obtain valid code after our optimization, jump correction adjusts the programs' control flow by inserting additional jump instructions into basic blocks whenever necessary. Obviously, these additional jump instructions contribute negatively to the benchmarks' code size. However, an average code size increase of only 0.8% was obtained for all considered benchmarks which is tolerable in practice.

During each iteration of our WCET-driven code positioning, one static WCET analysis using *aiT* is required (cf. Section 4). This is required in order to cope with the instability of the WCEP. Thus, most of the runtime of our positioning technique is not used for the algorithm given in Figure 4, but for all the necessary WCET analyses.

Figure 11 finally shows the absolute runtimes of our code positioning approach for all benchmarks. Runtimes are given in CPU seconds for an Intel Xeon processor running at 2.5 GHz. They include the times needed for code positioning itself, all WCET analyses, and that for compiling and optimizing the benchmarks at optimization level *O3*.

The least runtime is consumed for the optimization of benchmark `prime`. Here, only a total of 5 WCET analyses is required for code positioning, yielding an absolute runtime of only 2.7 CPU seconds. The highest compilation time is required for `cover` with a runtime of 804 seconds. During the optimization of this benchmark, 15 WCET analyses are performed overall. For all considered 18 benchmarks, an average optimization runtime of 240 CPU seconds was

If, however, shorter runtimes are required, it is still possible to reduce the number of execution contexts during static WCET analysis (cf. Section 5.1) which allows to trade analysis precision with runtime.

It again turned out that different associativities have almost no direct impact on code sizes. For all considered associativities, average code size increases only vary between 0.2% (8-way set-associative cache) up to 0.8%. In terms of optimization runtimes, the highest average runtime of 240 CPU seconds was achieved for a 2-way set-associative cache, while a minimal average runtime of 171 CPU seconds was observed for the 8-way set-associative cache.

6. CONCLUSIONS

This paper is the first one to present a WCET-driven, cache-aware code positioning technique. It introduces a for-

mal model of cache conflict misses based on a conflict graph which is enriched by data stemming from static WCET analysis. Since it is likely that this conflict graph initially contains many superfluous edges that do not model actual cache conflicts, this paper also proposes techniques to refine the conflict graph. These refinement approaches rely on control flow analyses, cache may analyses and on the inspection of a program's memory layout.

On top of this conflict graph model, a greedy heuristic for code positioning is presented. In contrast to previously published positioning approaches, our technique is inherently able to apply code positioning both to basic blocks and to entire functions. Furthermore, we are able to optimize for broad classes of cache architectures with varying associativities. The effectiveness of our approach is shown by average reductions of accumulated cache conflict misses by 15.5% and by average reductions of WCET estimates by 6.1% for an industrially relevant processor with a 2-way set-associative cache. Considering a direct-mapped cache even yields significantly larger savings in terms of both accumulated cache misses and WCET estimates.

Despite the fact that the average runtimes of 240 CPU seconds required by our optimization are still moderate, we are aware of the inherent scalability issues of an optimization heuristic where a full static WCET analysis is performed during each individual iteration of the heuristic's optimization loop. For this reason, our future work will concentrate on approximations of a WCET timing model which are faster than a fully-featured WCET analysis using *aiT*, but which still are accurate enough to guide our code positioning heuristics towards a systematic WCET reduction. Furthermore, we will apply our code positioning techniques to large industrial benchmarks in the future.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using *aiT*. The authors would also like to thank Synopsys for the provision of the instruction set simulator *CoMET* enabling the determination of ACETs.

7. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers. `http://www.absint.com/ait`, July 2011.

[2] H. Falk and J. C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of DAC*, July 2009.

[3] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2), Oct. 2010.

[4] H. Falk, S. Plazar, and H. Theiling. Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths. In *Proceedings of CODES+ISSS*, Oct. 2007.

[5] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of LCT-RTS*, June 1997.

[6] C. Ferdinand and R. Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2), Nov. 1999.

[7] G. Gebhard and S. Altmeyer. Optimal Task Placement to Improve Cache Performance. In *Proceedings of EMSOFT*, Sept. 2007.

[8] C. Guillon, F. Rastello, T. Bidault, et al. Procedure Placement using Temporal-Ordering Information: dealing with Code Size Expansion. In *Proceedings of CASES*, Sept. 2004.

[9] Informatik Centrum Dortmund e. V. ICD-C Compiler framework. `http://www.icd.de/es/icd-c`, July 2011.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of MICRO*, Washington DC, Dec. 1997.

[11] E. A. Lee. Absolutely Positively On Time: What Would It Take? *Embedded Systems Column, IEEE Computer*, July 2005.

[12] Y. Liang and T. Mitra. Improved Procedure Placement for Set Associative Caches. In *Proceedings of CASES*, Oct. 2010.

[13] T. Liu, M. Li, and C. J. Xue. Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking. In *Proceedings of RTAS*, Apr. 2009.

[14] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proceedings of ECRTS*, July 2008.

[15] Mälardalen WCET Research Group. WCET Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, July 2011.

[16] S. Plazar, P. Lokuciejewski, and P. Marwedel. WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *Proceedings of WCET*, June 2009.

[17] S. Plazar, P. Lokuciejewski, and P. Marwedel. WCET-driven Cache-aware Memory Content Selection. In *Proceedings of ISORC*, May 2010.

[18] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of DATE*, Apr. 2007.

[19] J. Reineke, D. Grund, C. Berg, et al. Timing Predictability of Cache Replacement Policies. *Real-Time Systems*, 37(2), Nov. 2007.

[20] S. Steinke, L. Wehmeyer, B.-S. Lee, et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of DATE*, Mar. 2002.

[21] V. Suhendra, T. Mitra, A. Roychoudhury, et al. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of RTSS*, Dec. 2005.

[22] Synopsys, Inc. `http://www.synopsys.com`, July 2011.

[23] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, July 2011.

[24] X. Vera, B. Lisper, and J. Xue. Data Cache Locking for Higher Program Predictability. In *Proceedings of SIGMETRICS*, June 2003.

[25] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Proceedings of DATE*, Feb. 2004.

[26] V. Živojnović, J. M. Velarde, C. Schläger, et al. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of ICSPAT*, Dallas, USA, Oct. 1994.

[27] WCET-aware Compilation. `http://ls12-www.cs.tu-dortmund.de/research/activities/wcc`, July 2011.

[28] 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.