Memory Positioning of Real-Time Code for Smaller Worst-Case Execution Times

Amine Marref
Department of Computer Science
Umm Al-Qura University
Makkah, Saudi Arabia
ajmarref@uqu.edu.sa

Adam Betts
Mälardalen University
School of Innovation, Design, and Engineering
Västerås, Sweden
adam.betts@mdh.se

***Abstract*—The process of determining the worst-case execution time (WCET) is challenged in the presence of caches due to their unpredictable effect on the speed of memory references. In particular, when cache conflicts between program lines are common, thrashing occurs and this inadvertently increases the WCET, sometimes significantly so. One way to minimise the cache impact on the WCET, therefore, is to judiciously lay out code and data in memory to avoid cache conflicts.**

In this paper, we show how to formulate the WCET-aware code placement problem as a Constraint-Optimization Problem (COP), which can then be efficiently solved by off-the-shelf COP solvers. Experimental evaluation of our proposed analysis shows that the proposed analysis successfully identified code positioning that yields the minimum WCET for over half of the problems within manageable time.

I. Introduction

Real-time systems (RTSs) must operate in a timely manner to ensure their correct functioning. An RTS is a set of tasks that cooperate in order to deliver a specific functionality. To ensure that the RTS works correctly, a schedulability analysis is performed which checks whether or not all tasks can meet their deadlines at runtime — this requires knowledge about the worst-case execution time (WCET) of the individual tasks. In order to estimate the WCET of a task, either static analysis (SA) or dynamic analysis (DA) are performed [1].

The SA of a real-time task to obtain an upper bound on the WCET involves creating a mathematical model of the hardware where the task runs. Contemporary hardware inevitably contains caches since they speed-up computation time considerably. Consequently, the SA of modern hardware to obtain the WCET naturally includes the analysis of the cache behaviour that classifies cache accesses as *definite hits*, *definite misses*, and *unclassified accesses*. An unclassified cache access results when SA fails to determine whether a portion of the task will or not be in the cache when it is accessed because of some complex *cache-conflict* pattern that cannot be accurately captured by the SA. Since safety in SA is mandatory, the unclassified cache accesses are considered as cache misses — as long as this does not cause timing anomalies [2] i.e., situations where assuming a local worst-case behaviour leads a global non worst-case behaviour.

Large WCET estimations are not practical as they might eventually lead to unschedulable systems, and so decreasing the WCET is always desired. The magnitude of the WCET returned by SA is a function of the number and magnitude of cache hits and cache misses — the latter influences the estimation considerably since the cost of a cache miss is several orders of magnitude larger than the cost of a cache hit. Reducing the number of cache conflicts eventually reduces the number of definite cache misses and unclassified cache accesses, allowing SA to derive a tighter WCET bound.

The number of cache conflicts in the task also affects DA — which involves executing the task on its target hardware. When the WCET estimation is obtained by DA, a certain level of execution-time coverage [3] is usually needed to acquire more confidence in the WCET estimation. When the analysed task exhibits a complex cache behaviour, reasoning about timing coverage becomes very hard as the testing process of DA must be able to stress task execution in situations that cause worst-case cache behaviour which might only be observed in very pathological executions of the task. Reducing the number of cache conflicts reduces the number of cache misses during execution which contributes to a smaller WCET, and also minimizes the timing variations which allows a better reasoning about timing coverage.

One way to make the WCET smaller is to organise its code in memory in a way that minimizes cache conflicts at run-time — this is the problem for which we offer a solution in this paper. The proposed technique only applies to instruction caches which we refer to as simply caches throughout this paper.

In this work, we consider the problem of code positioning of real-time tasks that contain a number of functions. The problem for which we propose a solution is how to organise the functions that form the real-time task in main memory so that their interaction in cache at run-time yields a minimal number of cache conflicts, and hence decreases the WCET. The analysis could in theory additionally be applied at the basic-block level of the different functions i.e., minimizing cache conflicts resulting from both intra-function and inter-function cache interaction. We focus on this level of granularity because the computed layout can only be enforced with linker support, which typically only allows control over the start addresses of functions.

In this paper, we show how to model the problem of code positioning in memory for lower WCET as a constraint-optimization problem (COP). Using a COP to model the

problem has two main advantages. First, it is possible to obtain the *optimal* code positioning in memory that yields the minimal number of cache conflicts using a complete search method. Second, the problem can seamlessly be augmented by extra constraints that describe restrictions on code placement or reflect functionality constraints between portions of the code of the task being analysed. This is a particularly important consideration in industry where it is often necessary, for example, to force functions to particular address so that new layouts have minimal changes.

The contributions of the paper are the following.

- We show how to model the problem of code positioning to minimize the WCET as a COP.
- We show that the COP solution to the code-positioning problem can transparently be augmented by real-life constraints that normally govern code positioning in main memory.
- We prove that positioning task code in order to minimize the WCET never causes an increase in the WCET with respect to the most-commonly used hardware accelerators.
- We evaluate the quality and cost of the solutions obtained for the COP formulation using two off-the-shelf COP solvers.
- We implement our own COP search method in constraint-logic programming, and evaluate the quality and cost of its solution.

The rest of the paper is organised as follows. In Section II, we review related work. In Section III, we describe the COP formulation of the code-positioning problem in a WCET setting. In Section IV, we describe the search methods that we use to solve the COP formulation of the problem. In Section V, we present the enhancements we perform in the COP solver that decreases the effort of solving the COP. In Section VI, we conduct an empirical evaluation of the goodness and complexity of our proposed technique. In Section VII, we draw conclusions and set directions for future work.

II. Related Work

Code positioning is a compiler technique [4] that aims at reducing the average-case execution time (ACET). Using execution-profile data, the compiler decides on a particular code (and data) positioning that enhances spacial and temporal locality which at the end yields a better use of the cache.

Code positioning aimed at decreasing the ACET is not guaranteed to also decrease the WCET — it can potentially increase it. Therefore, code-positioning techniques to reduce the ACET are relevant to our work, but non-comparable.

In [5], [6], the problem of code positioning to reduce the WCET is considered. The work in [5] addresses the issue of positioning basic blocks in main memory in a way that reduces the branch penalties along the longest path. This work is different from ours in two ways. First, the positioning of code is performed at a basic-block level, while in our case, the positioning is performed at the function level. Second, the cost of positioning is derived from branch-prediction penalties, while in our case it is derived from cache-conflict penalties.

In [6], code positioning is performed at the function level, and its cost is based on instruction-cache conflicts. The work in [6] addresses the same problem that we attempt to solve — but we make the following observations. First, the formulation of the problem is different; [6] views the code positioning problem as a heuristic problem while we formulate it as a COP. Second, the algorithm presented in [6] is greedy whose solution is not guaranteed to be optimal. However, our formulation enables obtaining optimal solutions if desired. Third, [6] uses a third-party WCET analysis tool that performs cache modelling, which is not required in our case because we integrate the cache analysis as a set of constraints in our COP formulation. Fourth, [6] does not consider real-life constraints that may restrict code positioning, while our COP formulation can transparently be augmented with such constraints.

In [7], WCET reduction is implemented by careful inlining of task functions using machine-learning techniques. This is different from our work in the sense that the reduction of the WCET is obtained by inlining functions as opposed to positioning them.

III. Problem Formulation as a COP

The problem that we address in this paper is the following. An executable (real-time) task contains a set of functions whose interaction with each other is dictated by the functional semantics of the task. The functions are in object-code format i.e., compiled; and are awaiting to be linked in the task's memory-address space. The way by which the functions are linked yields a specific cache behaviour of the task characterised by a number of cache conflicts. We want to find the arrangement of functions in memory that yields a minimum number of cache conflicts and thus decreases the WCET of the task.

In order to formulate the code-positioning problem to minimize the WCET as a COP, we define what a COP is. First, a constraint-satisfaction problem (CSP) is defined as a set of variables and a set of constraints on the variables; and a solution to the CSP is a total assignment to the variables that satisfies all the constraints. A COP is a CSP augmented with an objective function and a cost variable; in this case a solution to the COP is a total assignment to the variables that satisfies all the constraints and optimizes (minimizes or maximizes) the cost variable according to the objective function.

In the problem of code positioning to minimize the WCET, the variables are the functions and their attributes e.g., position in main memory. The set of constraints dictates mandatory properties (e.g., no two functions can map to the same memory address) and optional properties of functions (e.g., two functions are mutually exclusive). The cost that we want to minimize is the WCET of the task.

Formally, we consider a task t of n functions F_i. Each function F_i has a start address s_i, an end address e_i, and a size z_i. The start and end addresses s_i and e_i respectively will indicate the placement of the functions in main memory

and hence finding an assignment to them — that minimizes the WCET — is the objective of the solution. The sizes z_i are given and they are a function of the number of instructions in the object code (known from the object code) and their respective sizes (known from the instruction-set description).

Table I shows the memory constraints of the COP that we attempt to solve. Constraint 1 defines the domain of the variables s_i and e_i between memory boundaries mem_l and mem_h (for memory low and memory high respectively); informally constraint 1 states that the functions can be positioned anywhere in the memory-address space. Here, the memory-address space is a finite range of integers from mem_l to mem_h, inclusive. Constraint 2 adds information about the sizes of the functions. Constraint 3 states the no-overlapping property i.e., the functions are arranged in disjoint portions of the address space available for the task. The constraints 1 to 3 are mandatory.

Optionally, the COP can be augmented with additional constraints specified by the user. For example, specific areas of memory cannot be used for procedure allocation because e.g., they are reserved for interrupt-service routines. This is encoded in the COP using constraint 4 in Table I where RS (for reserved space) is the set of address-space ranges where no function can be positioned. Another example is that a particular function must always be positioned at a specific address in memory. This is implemented by constraint 5 where k is a user-specified address where the function F_i is to be mapped. A third example is that, an extra memory space following the end address of some function is needed to be free in case the function's code increases in size in future. This is achieved using constraint 6 where k address-space units after the end of function F_i cannot be used to position other functions.

The problem as formulated so far is a CSP (or a COP with a non-variable cost) whose solution returns a feasible positioning of functions in memory. The next step is to devise a cost variable that guides the optimization of the CSP which will be a function of the memory layout of the functions, their execution behaviour in the task, and the cache architecture. In this paper, we restrict our analysis to direct-mapped caches, which are commonplace in embedded systems because they conserve power by preventing the simultaneous tag comparison needed by set-associative caches.

In order to define the cost variable, we need to determine which functions might conflict with each other in the cache. A naïve solution is to consider that every function can conflict with every other function. This does not yield an accurate positioning of functions in main memory. A more accurate solution is to consider the call and loop relationships between functions. For example, if some F_1 calls some F_2 and both conflict in the cache, a large number of misses is incurred compared to when they do not conflict. Similarly, if F_1 and F_2 are part of the same loop nest, a large number of cache misses is incurred if they conflict in the cache, compared to when they do not conflict.

For this purpose, we define the call-and-loop graph to represent the interaction of the functions in the cache. Formally, a call-and-loop graph $G = (V, E)$, $|V| = n$, $E \subseteq V \times V$ is a directed graph where each node $v_i \in V$ is a function F_i, and there is an edge $(v_i, v_j) \in E$ if and only if F_i and F_j call each other in task t or belong to the same loop nest. Notice that the construction of G is not sensitive to loop-nest levels. Building G might potentially encounter problems e.g., the resolution of function pointers — solving this problem which is relevant to many static code-analysis techniques is considered outside the scope of this work.

	F_1	F_2	F_3	$\cdots$	F_n
F_1	0	w_{12}	w_{13}	$\cdots$	w_{1n}
F_2	0	0	w_{23}	$\cdots$	w_{2n}
F_3	0	0	0	$\cdots$	w_{3n}
$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
F_n	0	0	0	$\cdots$	0

Fig. 1. The weight matrix W of graph G containing n functions F_i.

A pair of functions might conflict with each other more than another pair e.g., a pair of large functions executing in a loop that iterates many times will conflict more than a pair of functions that call each other once only. For this reason, the graph G should be weighted with maximum number of executions of functions to accurately model the sources of the cache conflicts. Since the positioning of functions is performed in a WCET setting, the weights on the graph must be safe. The information about these weights or execution counts more precisely (i.e., the number of times they are executed in any single run of the task) either comes as annotations from the user in the form of loop and recursion bounds or using a third-party flow-analysis tool for WCET e.g., SWEET [8].

The graph G is represented in the COP as a (square) weight matrix W of the form shown in Figure 1 where the entry $W(F_i, F_j)$ contains the weight w_{ij}. Notice that the entries of W below the main diagonal are zeros since the weights are already accounted for in their symmetric entries with respect to the main diagonal. This is because G is directed.

A function F_i can have its size z_i superior to the cache i.e., $z_i > cache\_size$. In this case, the address range of function F_i is split into $\lceil \frac{z_i}{cache\_size} \rceil$ address ranges of hypothetical functions F_{i_k} with the constraints $(s_{i_{k+1}} = e_{i_k} + 1)$ to preserve the contiguousness of F_i — where $s_{i_{k+1}}$ is the start address of the $(k+1)^{th}$ (split) portion of F_i and e_{i_k} is the end address of the k^{th} one. For this reason, no function F_i in our formulation has a size larger than the cache i.e., no function can conflict with itself — but its split contiguous parts may conflict with each other, and hence the zeros along the main diagonal in matrix W.

Now that we know which functions can potentially conflict with each other in the form of a weighted graph G, we can define the cost variable. The amount of conflicts will depend on the positioning in memory e.g., consider Figure 2 where the cache is direct-mapped of size 600 units. In the left-hand side of the figure, functions F_1 and F_2 both fit disjointly in cache at address spaces $[0..199]$ and $[200..499]$ respectively resulting in

TABLE I
THE MEMORY CONSTRAINTS OF THE COP REPRESENTATION OF THE CODE-POSITIONING PROBLEM TO MINIMIZE THE WCET.

1	$\forall i \in [1..n] \bullet \{s_i, e_i\} \subset [mem_l..mem_h]$
2	$\forall i \in [1..n] \bullet e_i - s_i + 1 = z_i$
3	$\forall i, j \in [1..n], i \neq j \bullet [s_i..e_i] \cap [s_j..e_j] = \emptyset$
4	$\forall i \in [1..n], \forall rs \in RS \bullet [s_i..e_i] \cap rs = \emptyset$
5	$\exists i \in [1..n] \bullet s_i = k, k \in [mem_l..mem_h]$
6	$\exists i \in [1..n], \forall j \in [1..n], i \neq j \bullet [s_j..e_j] \cap [e_i + 1..e_i + k] = \emptyset, k \geq 1$

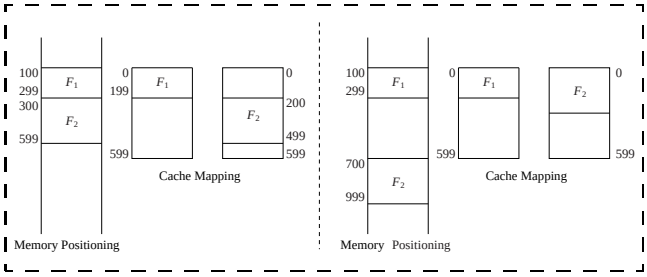

Fig. 2. An example of how positioning of functions in memory results in different cache conflicts: to the left, no conflicts occur; to the right, an amount of z_1 (size of F_1) conflicts occur.

zero cache conflicts — a cache line at address a in memory maps to ($a \ mod \ cache\_size$) in a direct-mapped cache. In the right-hand side of the figure, the memory mapping is different resulting in non-zero cache conflicts. Therefore, the cost variable will depend both on the number of edges in G and also their weights and magnitudes, and is computed according to Formula (1) where $cost_t$ is the cost of the positioning of functions in memory for the task t, $(v_i, v_j) \in E$ is an edge of the call-and-loop graph G, w_{ij} is the weight on the edge (v_i, v_j) computed by a safe WCET flow analysis, and c_{ij} is the number of the cache conflicts that occur between functions F_i and F_j given the chosen memory layout — c_{ij} is yet to be defined.

$$cost_t = \sum_{(v_i,v_j)\in E} w_{ij} * c_{ij}, G = (V, E) \quad (1)$$

What happens now (informally) is that when the COP is passed to a COP solver, the latter makes an initial assignment to the variables s_i and e_i that satisfies the constraints of the underlying CSP. Then, using a search method e.g., branch-and-bound, better solutions (i.e., other assignments to variables s_i and e_i) to the CSP are found that optimize (in our case minimize) the cost variable $cost_t$. At the end of the search, an assignment to the variables s_i and e_i that makes the value of $cost_t$ minimum (globally or locally depending on the search method) is found — which will be the best function positioning in our case.

Next, we define c_{ij} which represents the number of cache conflicts between functions F_i and F_j. In this case, we conjecture that the WCET $wcet_t$ of task t is reduced when the functions F_i are positioned in memory in an arrangement that minimizes $cost_t$; which is equivalent to saying that reducing inter-function cache conflicts reduces $wcet_t$. However, we must show that reducing inter-function cache conflicts does not cause an (unintentional) increase in $wcet_t$ by e.g., worsening intra-function cache behaviour.

In our formulation of the problem, we view a function as a black-box "chunk" of memory with a start address, an end address, and a size. The way by which the instructions or basic blocks are arranged inside the functions is not modelled in our formulation. We want to show that the amount of cache conflicts between the basic blocks of some function F_i does not increase if we "slide" its start address s_i over a range of addresses in memory. If we show this, then we can safely state that the way by which the functions are positioned in main memory does not lead to an increase in the number of intra-function cache conflicts.

Since a function is mapped to the cache according to a ($mod \ cache\_size$) relation, the cache behaviour of F_i when $s_i = a$ is the same as its cache behaviour when $s_i = b, a \neq b$ if ($a \ mod \ cache\_size = b \ mod \ cache\_size$). This means that when studying how positioning of F_i in main memory affects its internal cache behaviour, we can limit ourselves to a range of memory addresses of size $cache\_size$ since outside this range, the (internal) cache behaviour of F_i repeats itself because of the *modulus* mapping.

Now, all is needed is to show that the number of internal cache conflicts of function F_i does not change when its start address s_i changes over the memory-address range $r_i = [s_i^1..s_i^{cache\_size}]$. This reduces to showing that for any two basic blocks $B_{i.1}$, $B_{i.2}$ of function F_i, the magnitude of their conflict in the cache remains the same when s_i moves along the range r_i. For this, we need to determine the magnitude of the cache conflicts between two basic blocks — this is the number of cache lines occupied by one of the two blocks which are displaced by the execution of the other block. The number of displaced cache lines depends on how the blocks $B_{i.1}$, $B_{i.2}$ are mapped in the cache.

Figure 3 shows the possible ways by which two blocks $B_{i.1}$, $B_{i.2}$ can be mapped to the cache in the form of positioning scenarios labelled 1-16; and from which we should derive the number of cache lines between the two blocks that will potentially be displaced. Positioning scenarios 4 and 10 are annotated to explain the notation used in Figure 3. Table II lists the magnitude of the cache conflicts incurred by the execution of the two basic blocks $B_{i.1}$, $B_{i.2}$. Rows 1-3 in Table II correspond to the three rows (from top to bottom) of Figure 3 respectively. Each row of Table II captures the cache mapping of the blocks $B_{i.1}$, $B_{i.2}$ as a boolean expression in the column *Positioning-Scenario Condition*, and the respective

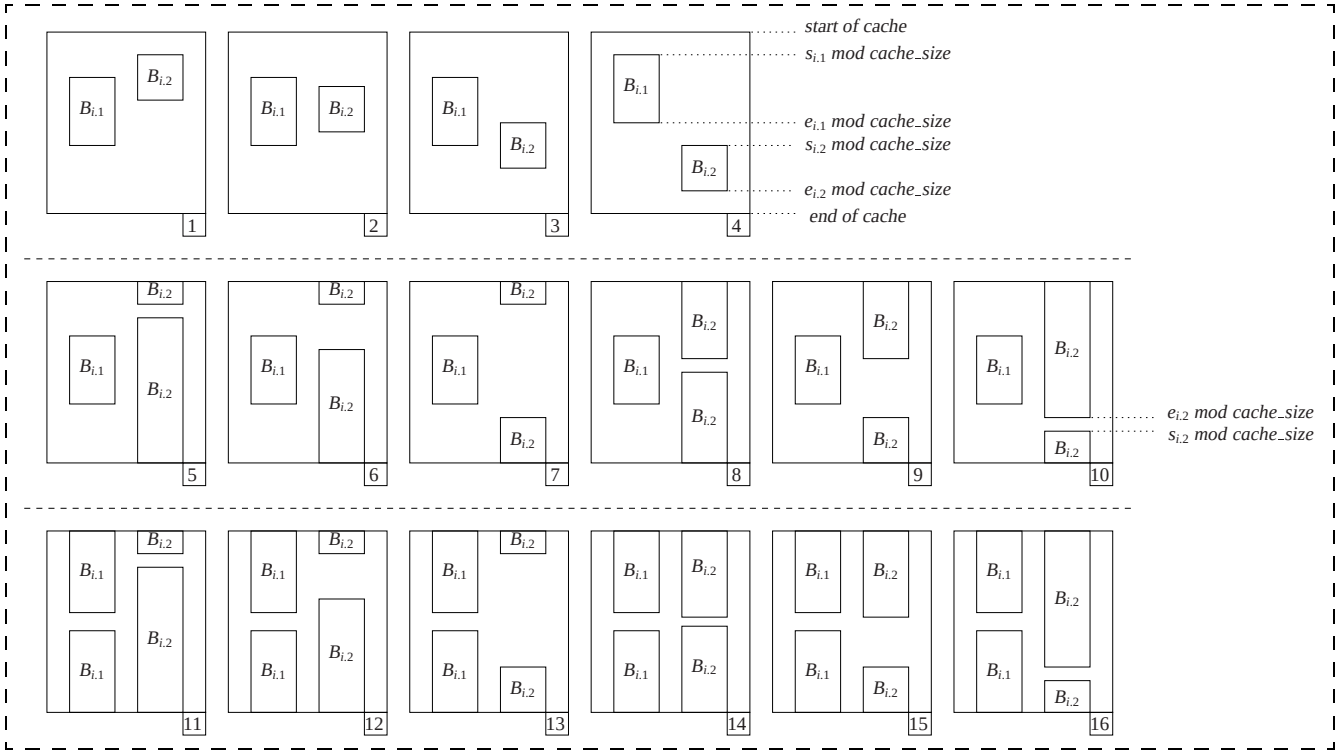

Fig. 3. Half of all the possible ways two blocks $B_{i.1}$ and $B_{i.2}$ can be arranged in the cache with respect to their start addresses $s_{i.1}$, $s_{i.2}$, and end addresses $e_{i.1}$, $e_{i.2}$ respectively. The other half is symmetric by making $B_{i.1} \leftarrow B_{i.2}$ and $B_{i.2} \leftarrow B_{i.1}$ simultaneously.

TABLE II
THE MAGNITUDE OF THE CACHE CONFLICTS INCURRED FROM THE EXECUTION OF $B_{i.1}$, $B_{i.2}$ AS SHOWN IN FIGURE 3. HERE, $x'_{i.k} = x_{i.k}\ mod\ cache\_size$, AND '$\oplus$' IS THE EXCLUSIVE-OR LOGICAL OPERATOR.

#	Positioning-Scenario Condition	Conflict-Magnitude Expression	
1	$cond_1 = (s'_{i.1} < e'_{i.1}) \wedge s'_{i.2} < e'_{i.2}$	$mag_1 =$	$max((min(e'_{i.1}, e'_{i.2}) - max(s'_{i.1}, s'_{i.2})), 0)$
2	$cond_2 = (s'_{i.1} > e'_{i.1}) \oplus s'_{i.2} > e'_{i.2}$	mag_2	$max((max(e'_{i.1}, e'_{i.2}) - max(s'_{i.1}, s'_{i.2})), 0)$ $+max((min(e'_{i.1}, e'_{i.2}) - min(s'_{i.1}, s'_{i.2})), 0)$
3	$cond_3 = (s'_{i.1} > e'_{i.1}) \wedge s'_{i.2} > e'_{i.2}$	$mag_3 =$	$min(e'_{i.1}, e'_{i.2}) + max((max(e'_{i.1}, e'_{i.2})$ $-min(s'_{i.1}, s'_{i.2})), 0) + cache\_size - max(s'_{i.1}, s'_{i.2})$

cache-conflict magnitude in the column *Conflict-Magnitude Expression* — which is the exact number of cache lines that conflict between the two blocks. Notice that the union of the three conditions in the table i.e., their disjunction is a tautology which means that they capture every possible positioning of blocks $B_{i.1}$, $B_{i.2}$: the positioning scenarios shown in Figure 3 *and* their symmetric counterparts.

The equations in Table II have been derived by hand; to prove their correctness, they are applied to their respective cache-mapping scenarios in Figure 3, in which case they yield the exact number of conflicting cache lines. For example, the amount of conflict between blocks $B_{i.1}$, $B_{i.2}$ as arranged in positioning scenario 14 in Figure 3 is derived according to the third row of Table II since the arrangement satisfies the condition $(s'_{i.1} > e'_{i.1} \wedge s'_{i.2} > e'_{i.2})$. By applying the expression mag_3 of row 3, we obtain the value $(e'_{i.1} + cache\_size - s'_{i.1})$ which is the exact number of cache lines that conflict between the two basic blocks. The correctness of the expressions is proved by applying them to all 16 scenarios which is straightforward and will not be shown here.

When the start address s_i of function F_i changes in the range r_i, the start addresses $s_{i.k}$ of the blocks $B_{i.k}$ of F_i also change. Since the function "slides" as a *whole chunk* in the memory space, the distance between the starting addresses $s_{i.k}$ of the blocks $B_{i.k}$ is preserved. This means that whenever the starting address s_i of function F_i changes by an amount d, the starting addresses of all blocks $B_{i.k}$ change by the same amount d since they all move together i.e., the relative placement of blocks $B_{i.k}$ to each other remains unchanged.

Consequently, we want to show that the number of cache conflicts between blocks $B_{i.1}$, $B_{i.2}$ is the same when their starting addresses $s_{i.1}$, $s_{i.2}$ change by the same amount $d \in r_i$. When the starting addresses $s_{i.1}$, $s_{i.2}$ change by the same amount, what happens is that the mapping of blocks $B_{i.1}$, $B_{i.2}$ changes through the scenarios enumerated in Figure 3. For example, if $B_{i.1}$, $B_{i.2}$ have initial positioning as shown in scenario 3 and their starting addresses $s_{i.1}$, $s_{i.2}$ increase by an amount $d > 0$ in memory, they could end up having the

TABLE IV
THE CACHE-CONFLICT COSTS c_{ij} OF THE COP REPRESENTATION OF THE CODE-POSITIONING PROBLEM TO MINIMIZE THE WCET.

1	$\forall i,j \in [1..n], i \neq j \bullet cond_1 \Rightarrow c_{ij} = mag_1$
2	$\forall i,j \in [1..n], i \neq j \bullet cond_2 \Rightarrow c_{ij} = mag_2$
3	$\forall i,j \in [1..n], i \neq j \bullet cond_3 \Rightarrow c_{ij} = mag_3$

positioning scenario 6 or remain in scenario 3 depending on the magnitude of the change d. Since the distance between $s'_{i.1}$, $e'_{i.1}$, $s'_{i.2}$, and $e'_{i.2}$ is always preserved, the number of cache lines that conflict is always preserved when F_i slides by some amount in main memory.

We have just shown that when a function F_i changes its position in main memory, the internal cache conflicts between its constituent basic blocks remain unchanged. This means that minimizing the inter-function cache conflicts of task t with respect to its constituent functions F_i does not cause an increase in $wcet_t$ due to intra-function cache effects. The positioning of functions F_i will not affect the internal pipeline timing behaviour of the individual functions since the timing behaviour of the pipeline depends on the intra and inter basic-block overlapping of instructions both of which are not affected by altering the position of the function in main memory. In addition to this, the branch prediction is not affected since it is not a function of the memory location of basic-block instructions. Finally, the data-cache timing behaviour depends on the positioning of data in main memory — not the positioning of code and so remains unaffected by the positioning of functions in main memory.

Pipelines, branch predictors, and data caches are the hardware accelerators normally accounted for in WCET analysis. We have shown that the positioning of functions in main memory with the aim of minimizing the WCET will not unintentionally cause the WCET to increase with respect to intra-function instruction-cache behaviour, pipeline behaviour, branch-predictor behaviour, or data-cache behaviour.

The values c_{ij} of the inter-function cache conflicts are computed in the same way by which the basic-block cache conflicts are computed in Table II. The reason for this is that the functions F_i just like blocks $B_{i.k}$ are regarded as whole chunks of code. Therefore, the positioning scenarios of basic blocks $B_{i.1}$, $B_{i.2}$ in Figure 3 also apply to functions F_i, F_j with attributes s_i, e_i, s_j, and e_j. Function-positioning can be illustrated by a figure equivalent to Figure 3 where $B_{i.1} \leftarrow F_i$, $B_{i.2} \leftarrow F_j$, $s_{i.1} \leftarrow s_i$, $e_{i.1} \leftarrow e_i$, $s_{i.2} \leftarrow s_j$, and $e_{i.2} \leftarrow e_j$. The values c_{ij} together with their associated positioning-scenario categories are shown in Table III. With reference to Table III, the costs c_{ij} are added to the COP as shown in Table IV.

In summary, we have shown that the CSP underlying our COP representation is modelled as shown by Table I, the cost function is shown in Formula (1), the costs of the pairwise-conflicting functions are captured by Tables III and IV. A solution to this COP — which minimizes the value $cost_t$ in Formula (1) yields the positioning of functions F_i of the task t that reduces the number of cache conflicts which minimizes the value $wcet_t$.

IV. SOLVING THE COP

A COP can be solved using a variety of techniques e.g., linear programming, constraint programming, evolutionary search, etc. — depending on the characteristics of the COP e.g., linearity of the constraints, integrality of the solution, guaranteed optimality, etc. It is not possible — within this paper — to exhaustively compare the applicability of all different optimization techniques to solve our COP, but we shall choose the following two criteria: integrality and optimality; the former is compulsory while the latter is desired.

By integrality, we mean that the assignment to the variables s_i, e_i must be (positive) integer. This suggests the use of an optimization method that solves COPs over discrete domains of which there is integer programming and finite-domain constraint programming. Both integer and constraint programming guarantee optimality through the use of (the NP-complete) branch-and-bound. Constraint programming, however, is superior in the sense that the search for the solution can be controlled by appropriate heuristics that aim at speeding-up the search by exploiting relevant COP attributes.

It is desired to obtain the optimal solution of the COP, however, it is not a pressing need. This by no means undermines the usefulness of obtaining an optimal placement of code that minimizes the WCET — which has both its scientific value and allows reasoning (via comparison) about the goodness of sub-optimal approaches to the problem. Nevertheless, in some COP instances, obtaining the optimal solution can be very costly since the process is NP-complete in the general case. A suboptimal solution to the code-positioning problem is a good-enough placement of functions in memory that reduces the WCET by some amount depending on the quality of the returned solution

We suggest to use a COP solution method that guarantees optimality (potentially expensive) and another solution method that does not guarantee optimality (potentially fast); and compare (empirically) the quality of the returned solution by both methods.

We shall use constraint programming as a solution method to the COP which guarantees both integrality and optimality.

As for the quick (and potentially suboptimal) solution method, there is an issue to be faced. Most optimization techniques in the literature are suitable for continuous domains i.e., the returned solutions are not guaranteed to be integral. A quick fix to this problem involves rounding up/down the assignments to the variables s_i, e_i to their closest integer values. Such rounding of values might result in a lower-quality solution, but most importantly, might result in a solution that does not satisfy the constraints. Therefore, the resulting integral assignment is checked for consistency against the constraints of the COP. Notice that this process of checking the feasibility of the solution is not a CSP: we check the feasibility of a *given assignment* to the variables against a set of constraints as opposed to finding *an assignment* to the variables that satisfies the set of constraints which is a way

TABLE III
THE MAGNITUDE OF THE CACHE CONFLICTS INCURRED FROM THE EXECUTION OF F_i, F_j.
HERE, $x'_i = x_i \ mod \ cache\_size$, AND '$\oplus$' IS THE EXCLUSIVE-OR LOGICAL OPERATOR.

#	Positioning-Scenario Condition	Conflict-Magnitude Expression
1	$cond_1 = s'_i < e'_i \wedge s'_j < e'_j$	$mag_1 = max((min(e'_i, e'_j) - max(s'_i, s'_j)), 0)$
2	$cond_2 = s'_i > e'_i \oplus s'_j > e'_j$	$mag_2 = max((max(e'_i, e'_j) - max(s'_i, s'_j)), 0)$ $+max((min(e'_i, e'_j) - min(s'_i, s'_j)), 0)$
3	$cond_3 = s'_i > e'_i \wedge s'_j > e'_j$	$mag_3 = min(e'_i, e'_j) + max((max(e'_i, e'_j)$ $-min(s'_i, s'_j)), 0) + cache\_size - max(s'_i, s'_j)$

cheaper process. The loss in the quality of the solution after a successful rounding might or not be significant; it will be considered as one of the shortcomings of the quick approach. When the rounding of the non-integral solution yields a non-feasible solution, the quick solution method is declared as not finding a solution to the COP.

Next, we have to decide on which continuous-domain solution method to use. Given that our cost variable $cost_t$ is implemented using the implication and modulus operators, a linear programming method — which otherwise is very attractive — is not suitable since the two operators cannot be expressed as linear equalities or inequalities to our best knowledge.

A COP can also be solved by an iterative algorithm or search method (IASM) which in general attempts to find a solution to a problem where it is possible to evaluate the goodness of a particular solution with respect to another solution. Using this kind of comparison, the space of solutions is visited always looking for a better solution. The way by which the space of solutions is visited depends on the implementation of the IASM which can be e.g., a genetic algorithm (GA) [9]. It is not possible to state whether or not the final solution of the IASM is optimal, nor is it guaranteed that the IASM finds better sub-optimal solutions than the COP in a shorter time. However, in general, IASMs have successfully been applied to optimization problems that otherwise do not have efficient algorithms to solve them.

Formally, the IASM optimizes a vector v_t or a set of vectors v_t of the form shown in Formula (2) where s_i, e_i are the start and end addresses of function F_i.

$$v_t = \langle s_1, e_1, s_2, e_2, \cdots, s_n, e_n \rangle \tag{2}$$

Assignments to v_t are explored in the search space until no further improvement in the quality of the solution is found, or until a predefined amount of time has elapsed.

When using an IASM, we are normally concerned about how fast it converges to a good-enough solution. The speed of the IASM depends on the goodness of its starting vector(s) v_t, the cost of evaluating the goodness of a solution, and the cost of finding new solutions. With respect to this work, the initial input to the IASM is random since any other initial code positioning is not guaranteed to be better than random unless it comes from a code-positioning technique in a WCET setting — in which case applying the IASM is only useful if further improvement is sought. Here, we want to evaluate the goodness of using an IASM for finding a good code positioning as opposed to improving an existing one.

A major speed hindrance in the IASM could potentially be the generation of new solutions: it is compulsory to produce only new vectors v_t that satisfy the constraints of Table I. For this reason, the generation of new solutions is not completely free in choosing new vectors v_t — but rather goes through the process of checking the constraint consistency of every newly created vector v_t. For IASMs that base their solution generation on an element of randomness e.g., GAs, this can be very costly.

In summary, a solution to the COP formulation of code positioning to minimize the WCET must be integral and preferably optimal. In order to guarantee integrality and optimality, constraint programming over finite domains is used with a complete search method e.g., branch-and-bound. When the constraint-programming solution to the COP turns out to be costly, a quicker solution method consisting of an IASM combined with integer rounding of solution and constraint-consistency checking is used.

V. ENHANCING THE CONSTRAINT SEARCH

The reason for using constraint programming (CP) — over integer programming — to solve the COP is that it allows the user to define own search heuristics that reduce the effort of finding a solution. A search method in CP explores systematically the variables in the problem, and attempts to make new value assignments to them from their respective domains that satisfy the constraints and optimize the cost variable. The efficiency of such search method depends to a great extent on the order in which the variables are visited, and also on the order by which the values are assigned to the variables. In CP, we normally have heuristics for *variable ordering* and heuristics for *value ordering*.

For instance, it makes sense to start with the variables that are mostly constrained as they have smaller domains and will enhance constraint propagation if they are *labelled* (i.e., assigned a value) first. The order of choosing values to label variables is equally important, for example, in a maximization problem, if the cost variable is proportional to some problem variable, then it makes sense to label that problem variable by choosing its largest value first.

With respect to code positioning to minimize the WCET, variable ordering is important. The cost variable $cost_t$ increases with respect to the terms $w_{ij} * c_{ij}$ according to Formula (1); and each term $w_{ij} * c_{ij}$ increases with respect to

w_{ij}, and c_{ij} since both are always non-negative. The ordering of variables must be determined at compile time where the costs c_{ij} — which depend on the values of s_i, e_i that are determined at runtime — are not available. Therefore, the only available information is the weights w_{ij} from the call-and-loop graph G. A variable ordering in this case is to first label the s_i, e_i, s_j, and e_j of functions F_i, F_j whose w_{ij} is highest. This obviously can be misleading as some $w_{12} * c_{12}$ can be smaller than $w_{34} * c_{34}$ even though $w_{12} > w_{34}$ — this is why it is a *heuristic*.

Formally, variable ordering is determined as follows. First, the weights w_{ij} available at compile time are ordered from largest to smallest to obtain the sequence $\langle w_{i1j1}, w_{i2j2}, \cdots, w_{imjm}\rangle$ where $|E| = m$. Then, the search method in the COP is instructed to visit the variables s_i, e_i in the order $\langle s_{i1}, e_{i1}, s_{j1}, e_{j1}, s_{i2}, e_{i2}, s_{j2}, e_{j2}, \cdots, s_{jm}, e_{jm}\rangle$. Let us call this *greedy variable ordering* to reflect the fact that it promotes the labelling of variables involved in the largest conflicts w_{ij}.

We can also have a (rather unimaginatively) *less-greedy variable ordering* which promotes the labelling of variables s_i, e_i whose associated functions F_i are involved in large global conflicts. For example, the variables s_1, e_1 of some F_1 involved in conflicts w_{1j} are labelled before the variables s_2, e_2 of some F_2 involved in conflicts w_{2k} if $(\sum w_{1j} \geq \sum w_{2k})$.

Formally, for each function F_i, we compute $(w_i = \sum w_{ij} + \sum w_{ki})$, and then the variables are labelled in the order $\langle s_{i1}, e_{i1}, s_{i2}, e_{i2}, \cdots, s_{in}, e_{in}\rangle$ if $(w_{i1} \geq w_{i2} \geq \cdots \geq w_{in})$.

As for value ordering, choosing (maximum) boundary values does not benefit the search since the cost variable does not increase/decrease in a proportional way to the increase/decrease of the variables s_i, e_i. Intuitively labelling the variables s_i, e_i starting by mem_l is better than labelling them starting from mem_h for obvious reasons.

In summary, the CP solver for the COP is augmented by heuristics specifying variable ordering and value ordering during search. The usefulness of this approach can only be evaluated through empirical experiments.

VI. Evaluation

In this section we describe our experimental environment, evaluate the goodness, and evaluate the scalability or practical applicability of our proposed COP approach.

A. Experimental Setup

All experiments are conducted on a personal computer with the following specification: 32-bit x86 CPU running at 2.8GHz and 4GB memory on Ubuntu 9.10.

The COP is implemented and solved by both the constraint-logic programming (CLP) solver *ECLiPSe* and the Matlab optimization Toolbox [10] (MOT). The reason for choosing ECLiPSe as the CP language is its ease of use for CP "outsiders", and also efficiency. The reason for choosing the MOT to implement the quick COP solution is that it contains mature, well-engineered algorithms to solve general COPs, and we argue in favour of using a good solver if one can have access to it.

The CLP solver is a CP solver with syntax based on logic programming. We will evaluate the goodness and complexity of finding a solution to the COP in three different ways. First, we use the default variable and value ordering provided by the solver as-is. Second, we use a conjunction of the greedy variable ordering and value ordering. Third, we use a conjunction of the less-greedy variable ordering and value ordering.

The MOT contains a comprehensive set of IASMs, of which we choose to use the GA for its common use in solving general optimization problems (function `ga` in the MOT). The GA implementation in Matlab allows the user to add constraints on the vectors v_t which are to be evolved. The MOT also contains an efficient implementation of the sequential quadratic-programming (SQP) method [11] (function `fminicon` in the MOT) which can be used to solve multivariate (non-linear) constrained optimization problems. An SQP method uses a quadratic model for the objective function and a linear model of the constraints and solves a series of sub-problems that optimize the objective function subject to the constraints. The objective function is expressed using conditionals to implement the different cost scenarios (which cannot be achieved using linear programming). The returned solution by the SQP method is not guaranteed to be integral and so the (non-integer) assignments to the variables s_i, e_i are rounded up or down to their closest integer values. This rounding of values may yield less optimal, potentially infeasible solutions.

B. Evaluation

An evaluation metric for the goodness of our approach is to compute by how much the WCET of the task is reduced when the linker uses the output of the COP solver. For this, we need a reference WCET to compare to, apply the optimization, obtain the new WCET, and compare the old WCET to the new WCET. One way to proceed is to use real-life programs e.g., from the WCET benchmarks [12] compiled and linked using some compiler — with maybe some inter-function code optimization techniques for the ACET — which are then re-linked using the suggested positioning by our approach, and finally the WCETs obtained from compiler positioning and our suggested positioning are compared.

The problem with the above way of evaluation is that it only reflects the goodness of the positioning for the specific benchmarks, and using a specific compiler. It could be the case that a compiler with no inter-function optimization for the ACET gives a better WCET positioning than that of a compiler that performs inter-function positioning to minimize the ACET — in which case evaluating the goodness of our approach by comparing to the compiler's positioning is biasedly in our favour. It could also be the case, that for the specific benchmarks used, the positioning of functions to minimize the ACET always leads to a large WCET — which might not be true for another set of benchmarks.

In order to conduct a more convincing evaluation, we suggest to estimate both the (inter-function) code positioning that yields the largest WCET (by maximizing $cost_t$), and the one that yields the smallest WCET (by minimizing $cost_t$). In this case, we obtain a range of WCETs $[wcet_{t_l}..wcet_{t_u}]$ ($wcet_{t_l}$ and $wcet_{t_u}$ not optimal in the general case). If the size of the range is large then it means that the method is good practically since it could be the case that the positioning generated by the compiler for the ACET case yields $wcet_{t_u}$. The method can obviously not generate positioning that yields a smaller $wcet_t$ if the compiler positioning to reduce the ACET already results in $wcet_{t_l}$.

The inputs to our suggested approach are the weighted call-and-loop graph G, the sizes z_i of the functions F_i, the constraints on the positioning of functions F_i as shown in Table I, the cache size, and the main-memory size. To perform our evaluation, we prefer to generate COP instances ourselves and solve them instead of using real-life programs since all that matter in our analysis are the number of functions F_i per task t, their sizes z_i, and their interaction that makes the graph G. As for the number of functions and sizes, they can be generated randomly while obeying real-life constraints e.g., they must all fit in main memory at the same time. As for the interaction between functions in the task, it can be anything — from a function conflicting with a few functions to a function conflicting with every other function in the task. In this evaluation, we generate hundreds of COPs — which again is superior to basing the evaluation on a small set of benchmark programs.

A COP instance is generated like follows.

- The number of functions F_i per task is $n \in [10..100]$. This is way above the number of functions per benchmark programs in e.g., [12]. Choosing a large number of functions per task stresses more the COP and IASM since more variables and constraints are added to the problem — which is useful for the complexity evaluation later on.
- The constraint $\frac{mem_z}{2} \leq \sum_{i=1}^{i=n} z_i \leq mem_z$ is always satisfied where $(mem_z = mem_h - mem_l + 1)$. This constraint ensures that the functions cover (together) at least half of the memory space because there are always costs that need to be minimized when choosing hardware for real-time systems e.g., memory space, i.e., a memory size is chosen depending on the code and data size of the intended applications. A more relaxed constraint e.g., $\sum_{i=1}^{i=n} z_i \leq mem_z$ could lead to easy-to-solve COPs since there is more "freedom" in placing the functions in memory.
- A weight on an edge (v_i, v_j) in G is $w_{ij} \in [1..1000]$. The magnitudes of the weights w_{ij} — alone — are orthogonal to the complexity of solving the COP but a combination of the magnitude and their disparity is of relevance as each w_{ij} pushes the optimization search in a potentially different direction depending on its magnitude; thus increasing the search effort. Choosing a random weight from the range $w_{ij} \in [1..1000]$ ensures enough disparity in the generated COP.

We have generated 1000 instances of COPs which are then encoded both in *ECLiPSe* and the MOT. An exhaustive listing of all the results is not feasible here, but we report the following.

- We have computed the reduction in the number of cache conflicts as follows. First, we solve each of the generated COPs using both optimization methods while maximizing $cost_t$ which results in $cost_t^{max}$ (the largest value of $cost_t$ obtained by any of the two optimization methods). Then, we solve the COP using both methods while minimizing $cost_t$ which results in $cost_t^{min}$. Finally, we compute $(cost_t^{max} - cost_t^{min})/cost_t^{max}$ which represents the percentage of the cache conflicts that can potentially be avoided using the positioning that yields $cost_t = cost_t^{max}$. We took the average of this percentage across the 1000 COP solutions and this gave about 28%. The smallest recorded percentage is 19% while the largest is 42% with a standard deviation of about 11 for all the 1000 percentages. This result shows that the method has the potential to minimize the WCET considerably by reducing the total number of cache conflicts by a good percentage.
- The time taken to solve the COP using the GA function in the MOT is unmanageable because of the random generation of solutions and checking their constraint satisfaction. However, the time taken using the SQP function was less than 10 seconds in all generated COPs. The time taken to solve the COP using *ECLiPSe* ranged from less than one second to just below 15 minutes which was set as the maximum time allowed to find the solution after which the COP solver is forced to return its (so far) best solution. This result shows that solving (and modelling) the code-positioning problem using constraint-logic programming or sequential-quadratic programming is not expensive.
- In the generated COPs, 58% were solved optimally by *ECLiPSe* in less than 15 minutes each. A further 20% could be solved optimally by increasing the time limit to 1 hour each. The solver could not decide whether or not the current solution is optimal for the remaining 22% even when increasing the time limit to 2 hours. This result shows that it is possible to determine the optimal code positioning for a large number of problems.
- In 23% of the obtained optimal COP solutions by *ECLiPSe*, the SQP method generated an optimal solution too. The time it takes the SQP method to find the optimal solution is significantly less (5% in some cases) than the time it takes *ECLiPSe* to find the optimal solution. This result shows that using an IASM method has the advantage of quickly obtaining a good solution with a good probability of it being optimal.
- The less-greedy variable ordering combined with value

ordering converges to a solution quicker than the greedy variable ordering with value ordering in 79% of the cases. The difference in solution time in these cases is on average 10% of the slowest-heuristic's solution time. In few cases, the quicker heuristic reached the optimal solution before the time limit while the slower heuristic either reached the optimal solution after time-limit extension or never decided whether it has reached it or not. This result shows that heuristics can reduce the time taken by the COP solver to reach an (optimal) solution.

VII. Conclusion

In this paper we have shown how to formulate and solve the code-positioning problem to reduce the worst-case execution time (WCET) as a constraint-optimization problem (COP). We have shown via empirical evaluation that the solutions we obtain using our proposed approach are optimal in numerous cases; and when they are not optimal, they still result in a significant (potential) decrease of the WCET by reducing instruction-cache conflicts.

In a future paper, we explain how to apply the method to set-associative caches. We also aim to devise even better search heuristics to reduce the COP solution time even further.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research (SSF) through the Research Centre for Predictable Embedded Software Systems (PROGRESS).

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools," *ACM Transations on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[2] T. Lundqvist and P. Stenström, "Timing Anomalies in Dynamically Scheduled Microprocessors," in *Proceedings of the IEEE Real-Time Systems Symposium*, 1999, pp. 12–21.

[3] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel, "WCET Coverage for Pipelines," TU Vienna, Tech. Rep., 2006.

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

[5] W. Zhao, D. Whalley, C. Healy, and F. Mueller, "Improving WCET by applying a WC code-positioning optimization," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 4, pp. 335–365, 2005.

[6] P. Lokuciejewski, H. Falk, and P. Marwedel, "WCET-driven Cache-based Procedure Positioning Optimizations," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems (ECRTS'08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 321–330.

[7] P. Lokuciejewski, F. Gedikli, P. Marwedel, and K. Morik, "Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining," in *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, Paphos / Cyprus, 2009, pp. 1–15.

[8] WCET-IDT-MRTC, "SWEET," *http://www.mrtc.mdh.se/projects/wcet/sweet.html*, June 2010.

[9] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.

[10] MathWorks, "The MathWorks — Optimization ToolBox," *http://www.mathworks.com/products/optimization/*, June 2010.

[11] R. Brayton, S. Director, G. Hachtel, and L. Vidigal, "A new algorithm for statistical circuit design based on quasi-Newton methods and function splitting," *Circuits and Systems, IEEE Transactions on*, vol. 26, no. 9, pp. 784 – 794, September 1979.

[12] Mälardalen WCET Research Group, "WCET project/benchmarks," *http://www.mrtc.mdh.se/ projects/wcet/benchmarks.html*, June 2010.