Superblock-Based Source Code Optimizations for WCET Reduction

Paul Lokuciejewski, Timon Kelter, Peter Marwedel
Computer Science 12
D-44221 Dortmund, Germany
{*Paul.Lokuciejewski,Timon.Kelter,Peter.Marwedel*}*@tu-dortmund.de*

***Abstract*—Superblocks represent regions in a program code that consist of multiple basic blocks. Compilers benefit from this structure since it enables optimization across block boundaries. This increased optimization potential was thoroughly studied in the past for average-case execution time (ACET) reduction at assembly level.**

In this paper, the concept of superblocks is exploited for the optimization of embedded real-time systems that have to meet stringent timing constraints specified by the worst-case execution time (WCET). To achieve this goal, our superblock formation is based on a novel trace selection algorithm which is driven by WCET data. Moreover, we translate superblocks for the first time from assembly to source code level. This approach enables an early code restructuring in the optimizer, providing more optimization opportunities for both subsequent source code and assembly level transformations. An adaption of the traditional optimizations common subexpression and dead code elimination to our WCET-aware superblocks allows an effective WCET reduction. Using our techniques, we significantly outperform standard optimizations and achieve an average WCET reduction of up to 10.2% for a total of 55 real-life benchmarks.

I. Introduction

Modern embedded real-time systems are characterized by both efficiency requirements and critical timing constraints. Average-case performance, power consumption and resource utilization are objectives describing the efficiency of a system. In physical environments, such as safety-critical automotive or avionic systems, where time is a crucial resource, the precise knowledge of the maximal program run time, defined by the WCET, is mandatory. This key parameter is exploited for schedulability analyses and verification as well as for the design of hard real-time systems in order to enable a safe interaction with the system's environment.

With the increasing complexity of today's embedded software, program code is typically generated by a compiler. State-of-the-art compilers offer a vast variety of optimizations with the objective to minimize the average-case execution time or energy dissipation. In contrast, a compiler-guided reduction of the WCET is still a novel research area with an increasing academic and industrial interest. WCET-aware compilation requires the integration of static timing analyses into the compiler framework to provide a worst-case timing model that can be exploited for an effective reduction of the program's WCET.

Funded by the European Community's 7^{th} Framework Programme FP7/2007-2013 under grant agreement n^{o} 216008.

Numerous compiler optimizations are not able to deploy their full optimization potential since they are considerably limited by basic block boundaries found in the application code. To overcome this problem, a program structure called *superblock* was introduced. It comprises several basic blocks and allows optimizations across block boundaries. This technique was thoroughly studied in the past for ACET minimization and substantial program speedups were reported [1], [2]. To find promising block candidates for superblock formation, block execution counts are required. For ACET minimization, profiling typically identifies assembly blocks on the most frequently executed path within the program's *control flow graph* (CFG).

In this paper, we aim at an automatic WCET reduction for embedded real-time systems. Unlike the profiling-based ACET optimizations, our techniques rely on WCET data provided by a static timing analyzer to construct superblocks for an effective WCET reduction. In addition, our optimizations must face the challenge of a switching *worst-case execution path* (WCEP). The WCEP is the longest path through the CFG and its length corresponds to the program's WCET. Real applications typically consist of more than one path, however only the WCEP is relevant for the program's WCET and compiler optimizations aim at its reduction. The modification of WCEP π may lead to a *WCEP switch*, i. e., after reducing the length of π, a new path π' may become the longest path in the CFG. To enable a continuous WCET reduction, WCET-aware optimizations must ensure that they do not proceed on the outdated path π but perform further transformations on path π'. This path switch makes WCET-aware optimizations more demanding compared to traditional compiler optimizations.

The main contributions of this paper are as follows:

1) The superblock formation is driven by WCET data.
2) For the first time, the concept of superblocks has been translated from assembly to source code level in order to exploit the compiler's full optimization potential.
3) We propose a novel trace selection algorithm which is more suitable for WCET reduction than standard selection approaches.
4) The compiler optimizations *common subexpression (CSE)* and *dead code elimination (DCE)* were re-

designed to exploit superblocks for an effective WCET reduction.

5) Our techniques are evaluated on a large number of real-life benchmarks to show their practical use.

The rest of this paper is organized as follows: Section II gives a survey of related work. In Section III, concepts for the formation of WCET-aware superblocks are presented. These concepts are exploited for the WCET-aware optimizations common subexpression elimination and dead code elimination that are introduced in Section IV. A description of our experimental environment and results achieved on real-life benchmarks are given in Sections V and VI, resp. Finally, Section VII summarizes this paper and gives directions for future work.

II. Related Work

Superblocks are based on the model of *traces* which represent the most frequently executed paths in the program. The initial idea was presented by Fisher [3] who considered traces as extended regions in the code to perform instruction scheduling across basic blocks. Different trace selection algorithms were evaluated in [4].

The main drawback of trace scheduling is the arising overhead for the insertion of *compensation code* after scheduling a trace to preserve program semantics. To overcome this complex *bookkeeping*, Chang introduced superblocks [1] which allow an easier instruction scheduling. Moreover, this work discusses ideas for the exploitation of superblocks by standard optimizations. These ideas serve as motivation for our novel WCET-aware optimizations. All these presented works have in common that they target at the ACET and are applied at assembly level.

To conduct WCET-aware compilation, a static WCET analyzers is required that provides the compiler with WCET data. In our work, we use the sophisticated analyzer *aiT* [6].

Most approaches to WCET reduction operate on assembly level or exploit memory hierarchies (e. g., scratchpad allocation or WCET-aware register allocation [14]). The exploitation of the optimization potential for WCET reduction at source code level requires the translation of worst-case timing information from the compiler back-end into the front-end. This step is called *back-annotation* and was introduced in [16]. Since our superblocks are constructed at source code level, we also rely on a back-annotation step.

The only work considering superblocks for WCET minimization was published by Zhao [7]. This paper is most related to our work but also significantly differs in several ways. Most importantly, Zhao performs the superblock formation at assembly level while we construct superblocks early in the compiler's optimization process at source code level. Thus, our approach enables further potential for both source code and assembly level optimizations as shown in [2]. Moreover, in [7] no novel superblock-based WCET optimizations were developed while we propose two techniques, the CSE and DCE. Finally, Zhao used small programs for the evaluation of his approach, thus it is not clear if his approach also scales well for realistic applications. In contrast, we apply our optimizations to large real-life benchmarks, which represent applications used in industry.

III. WCET-aware Source Code Superblock Formation

In this section, the required steps for the WCET-aware formation of superblocks are discussed.

A. Concepts of Superblocks

In general, a superblock is a program structure that comprises several basic blocks. The main advantage of this approach is that these blocks can be considered as a larger unit in the CFG, and thus provide higher flexibility to optimizations that operate on them.

The selection of basic blocks used for a superblock is based on the idea of program traces which were developed by Fisher [3] to enable a more efficient instruction scheduling across basic block boundaries.

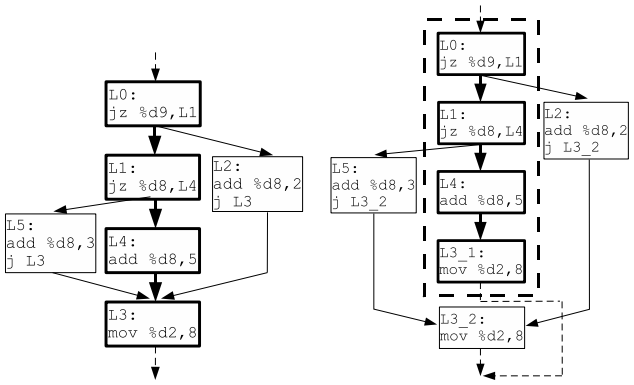
Definition 1: *Given a control flow graph, which is a directed (cyclic or acyclic) graph $G = (V, E)$ with nodes V corresponding to basic blocks and edges E connecting two nodes $v_i, v_j \in V$. A **trace** T is a sequence of basic blocks $T = (b_a, ..., b_k)$, such that for $a \leq i < k, (b_i, b_{i+1}) \in E$. If there is a loop L, with $\exists v_k \in L : v_k \in T$, then T is restricted by the respective loop boundaries, i. e., T does not span across basic blocks that lie outside L.*

Assembly Superblocks

Chang [1] developed superblocks at assembly level to circumvent some of the problems that exist in the trace-scheduling approach.

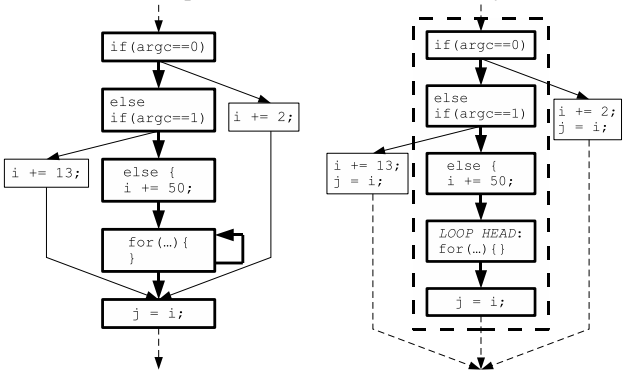
Definition 2: *A **superblock** S is a trace which can be entered only at the first basic block b_{start}, i. e., S is a trace $T = (b_{start}, ..., b_{end})$ such that $\forall b \in T \setminus \{b_{start}\} : \forall (b_i, b) \in E : b_i \in T$.*

Since the size of *natural* superblocks found in the program code is typically small, superblock enlarging optimizations [5] are used. The main technique is *tail duplication* which eliminates side entrances of arbitrary traces. For a side entrance b_{in} in a trace $T = (b_a, ..., b_{trc}, b_{in}, ..., b_e)$, a copy of the tail portion of the trace from the side entrance to the end $b_{in}, ..., b_e$ is created and all side entrance edges $e \in \{(b_{pred}, b_{in}) \in E \mid b_{pred} \neq b_{trc}\}$ are redirected to the corresponding duplicated basic block b'_{in}. This is illustrated in Figure 1. The trace is marked in bold, the superblock is represented by the dotted box. The superblock formation at this abstraction level of the code is easy since merely block labels have to be adjusted, while the code in the duplicated tail portions remains the same as in the original code.



(a) The original assembly code (b) After tail duplication

Figure 1. Superblock Formation at Assembly Level



(a) The original source code (b) After tail duplication

Figure 2. Superblock Formation at Source Code Level

Source Code Superblocks

The definition of basic blocks at source code level is equivalent to its counterpart at assembly level, making a translation of the superblock concepts possible. Following Definition 2, superblocks at source code level are defined as traces in a high-level control flow graph with no side entrances. However, our source code superblocks differ in one point from Chang's definition: we allow that a superblock $S = (b_0, b_1, ..., b_n)$ contains *inner loops*. These loops are represented in the trace by their loop headers. Hence, a loop L is referred to as an inner loop if its loop header is a basic block b_{head} such that $b_{head} = b_k$ with $k \in 1, ..., n$, and block b_{k+1} is the next basic block after loop L. This idea of a special handling of inner loops is similar to Fischer's trace definition [3] which treats loops as a single operation.

An example for a superblock formation at source code level for the programming language C is depicted in Figure 2. Basic blocks at this level are not distinguished by unique block labels, making an explicit control flow modification from one block to another arbitrary block difficult. As can be seen in Figure 2(b), the superblock formation requires a duplication and insertion of the statement `j = i` into the CFG paths other than the trace. A detailed discussion of the WCET-aware superblock formation at source code level follows in Section III-C.

The triplication of statement `j = i` could be avoided if, similar to assembly level, a new basic block holding this statement would be generated and control flow would be redirected to this block by using `goto`s. We intentionally avoided this since many compiler optimizations, which could possibly be enabled by the superblocks, rely on well-structured, `goto`-free code. Also, `goto`s generate additional jump instructions that decrease performance.

B. Trace Selection

The previous section introduced the concept of superblocks and motivated the need for traces. Here, we show why common trace selection approaches are not suitable for a WCET-aware superblock formation and present a novel trace selection algorithm as a possible solution.

Existing Approaches

The popular trace selection algorithms [4] rely on execution counts of basic blocks ($w(b_i)$) or control flow edges between blocks ($w(e_i)$). For a trace T_i, both approaches begin at a block b_{start} having the highest execution count $w(b_{start})$ and being not part of any other trace T_j. In the following steps, the trace $T_i = (b_a, ..., b_{start}, ..., b_k)$ is alternately extended at both ends. With *Traces* denoting the set of already selected traces and $\delta^+(b)$ denoting the set of outgoing edges from block b, the two trace selection algorithms for the expansion of a trace at its end (extension at the trace beginning works equivalently) operate as follows:

- **Selection via node weights**: Select b_{new} such that edge $(b_k, b_{new}) \in E$, $\forall T_j \in \mathit{Traces} : b_{new} \notin T_j$ and $w(b_{new}) = \max\{w(b_i) \mid (b_k, b_i) \in E\}$.
- **Selection via edge weights**: Select b_{new} such that edge $e_{new} = (b_k, b_{new}) \in E$, $\forall T_j \in \mathit{Traces} : b_{new} \notin T_j$ and $w(e_{new}) = \max\{w(e) \mid e \in \delta^+(b_k)\}$, with $\delta^+(b_k) = \{(b_i, b_j) \in E \mid b_i = b_k\}$.

The trace selection based on node weights may find less suitable traces than the edge weight-based selection since adjacent CFG blocks with high node weights may be infrequently executed in sequence.

Longest Path Approach

The previously presented greedy trace selection algorithms may produce traces that do not enable full optimization opportunities, as a result of their limited, local view on the program's CFG.

Consider the weighted CFG in Figure 3, where blocks are annotated with their worst-case execution times (in parentheses) and edges are annotated with their worst-case execution counts. Starting at the `if`-condition, both the node and the edge weight-based approach would select `blockC` for trace expansion. Such a trace following the *false* edge comprises code that *consumes* $6 * 100$ $cycles + 6 * 220$ $cycles = 1920$ $cycles$. However, if the *true* edge were taken, the

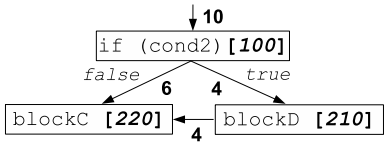


Figure 3. Failure of Existing Trace Selection Approaches

trace would comprise all three blocks with a length of $4{*}100\,cycles{+}4{*}210\ cycles{+}4{*}220\,cycles = 2120\,cycles$. Obviously, focusing on the longer trace promises more optimization potential.

The selection of a trace based on the *longest path* outperforms the greedy algorithms. It requires precise information about the WCET of each basic block, which is obtained from the static WCET analyzer aiT. This information is computed at assembly level and propagated to the source code level using *back-annotation*.

If the starting block (see below) is located within a loop, the algorithm finds a trace T within this loop. Otherwise, an entire function is used for trace selection. We describe the former case, the function-wide selection works analogously. The following steps have to be performed:

1) Find block b_{start} with the maximal WCET serving as a heuristically promising starting point for the trace.
2) For loop L, with $b_{start} \in L$, a directed, acyclic graph $G_L = (V_L, E_L)$ is constructed, such that all blocks in L that have the same loop depth as L are added to V_L. Inner loops I of L are represented by a special node b^I_{loop}. The set of edges E_L contains all edges between blocks $b \in V_L$. Moreover, each edge (v_i, v_j) with $\exists I : v_i \notin I \cap v_j \in I$ is replaced by an edge (v_i, b^I_{loop}) and is added to E_L (the symmetric case is handled analogously).
3) L is entered through the source block denoted as b_{source}. Each block having successors outside L is called b^i_{sink}. Furthermore, a distinguished node $b_{supersink}$ is created and connected with all b^i_{sink}.
4) To find the longest path in G_L, we use a cost function c_L that models the WCET of an edge $e \in E_L$ so that for any path P through G_L $\sum_{e \in P} c_L(e) = WCET(P)$ holds. Note that G_L is acyclic by construction, so that we can compute the longest path in G_L by supplying G_L together with the negated cost function $-c_L$ to a Bellman-Ford shortest path algorithm.
5) The final trace is selected by starting at b_{start} and adding alternately one block from the longest path to the beginning and end of the trace. Our trace selection at each step takes user-defined code expansion restrictions into account to avoid extensive code expansion during the superblock formation.

C. Superblock Formation

The superblock formation consists of a preprocessing and the actual formation phase.

Algorithm 1 Algorithm for Source Code Superblock Formation

Input: *WCET-aware Trace* T

```
1: block currBB ← endNode(T)
2: block lastBB ← endNode(T)
3:
4: /* Iterate trace, starting at the end. */
5: while currBB ≠ startNode(T) do
6:   /* Perform tail duplication. */
7:   if |δ⁻(currBB)| > 1 then
8:     for all predBB ∈ δ⁻(currBB) do
9:       if predBB == tracePred(currBB) then
10:        moveStmts(firstStmt(currBB),
             lastStmt(lastBB), lastStmt(predBB))
11:      else
12:        copyStmts(firstStmt(currBB),
             lastStmt(lastBB), lastStmt(predBB))
13:      end if
14:    end for
15:  end if
16:  currBB ← tracePred(currBB)
17:  if isPrecededByInnerLoop(currBB) then
18:    currBB ← predBeforeLoop(currBB)
19:  end if
20:  if isConditional(lastStmt(currBB)) then
21:    lastBB ← currBB
22:  end if
23: end while
```

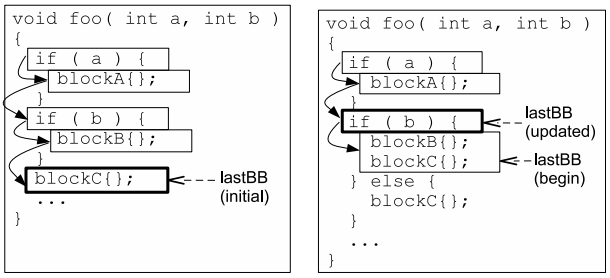
Preprocessing

The preprocessing phase begins with the application of the well-known optimizations *function inlining* and *loop unrolling* to enlarge code fragments suitable for superblocks [5]. In the next step, programming language constructs that lead to *unstructured* code are tried to be eliminated since they prevent a proper superblock formation. Examples are C `goto`-statements. A detailed approach for the elimination of unstructured code can be found in [9]. If the undesired construct can not be removed, then functions containing these constructs are omitted for superblock formation.

Formation Algorithm

After preprocessing, a WCET analysis of the program is performed at assembly level and the worst-case timing model is made available at source code level using back-annotation. This information allows to perform superblock optimizations in decreasing order of the functions' WCETs. Such a strategy allows to exploit maximal optimization potential before code expansion restrictions are exceeded.

Next, for each function a trace based on the longest path approach is iteratively selected and used to build a superblock which is subsequently exploited by our superblock-based optimizations. Afterwards, the WCEP is updated (to



(a) Original code with trace (b) Side entrance elimination

Figure 4. Successive Source Code Superblock Formation

cope with path switches) and the next trace is processed if there still are unprocessed blocks which are not part of any trace and have a WCET > 0.

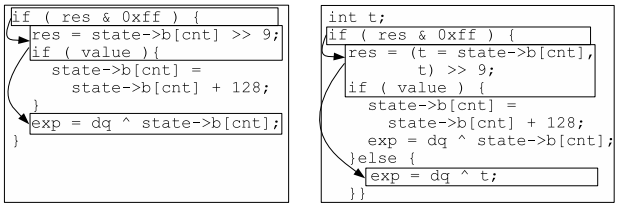
Source code superblock formation is depicted in Algorithm 1. Figure 4 shows an example code with a trace marked by the arrows and serves for illustration of the algorithm. The basic idea of the algorithm is to traverse trace T (line 5) backwards beginning at the last block and to eliminate found side entrances by duplicating the so far traversed tail portions of the trace in all CFG paths other than the trace. The original tail portion is moved behind the predecessor block on the trace to enlarge the superblock. Following this strategy, the superblock is iteratively increased by merging it with the predecessor blocks on the trace.

Variable $curBB$ (line 1) represents a pointer to the basic block that is currently considered for superblock formation and used to traverse the trace backwards. In Figure 4 it is marked by the bold box. The current tail portion that must be duplicated, is enclosed by its first (beginning of $curBB$) and last statement (end of $lastBB$).

If $curBB$ has more than one incoming edge ($\delta^-(curBB)$), tail duplication is performed (lines 7-14). If the predecessor of $curBB$ is on the trace (line 9), then all statements (including inner loops) of the current tail are moved behind the last statement of the trace predecessor $predBB$ (line 10). Otherwise, this set of statements is copied into the other CFG paths not part of the trace (line 12). After the elimination of all side entrances, $curBB$ is set to the next trace predecessor block (line 16). Inner loops are omitted (line 17-19). Finally, if the last statement of $currBB$ is a conditional statement, $lastBB$ is set to $currBB$ (line 20-22). Doing so ensures a duplication of entire conditional statements. Figure 4(b) shows code after the first side entrance elimination. `blockC` was moved onto the trace in the $then$-part of the second `if`-condition and copied into the non-trace predecessor ($else$-part). Moreover, $lastBB$ was updated to the beginning of its embedded conditional statement.

IPET-based WCEP Update

During superblock formation and its optimization, a WCEP path switch may occur. To make sure that subsequent optimizations do not operate on an outdated WCEP, the timing information must be updated. A frequent use of a costly WCET analysis is not feasible. Thus, we update the WCEP data after superblock optimization by solving an ILP model based on IPET [8]. The ILP model requires WCETs for basic blocks which we extract from the preceding run of the timing analyzer. Since the ILP is less complex than a full WCET analysis, the WCEP re-computation is faster but also less precise. However, precision is not our main objective here since the IPET-based computation is not meant to replace a safe static WCET analysis but should be considered as a fast heuristic which helps to indicate potential WCEP switches. For blocks that were modified by the optimizations, estimations of their WCET are made by the optimizer. The estimation uses the *tree-based approach* [10] which computes WCETs for statements by a bottom-up run over the syntax tree. After some iterations (defined by the user), a full WCET analysis is performed to get precise timing information again.



(a) Original code (b) After Superblock-CSE

Figure 5. Example for Superblock-CSE

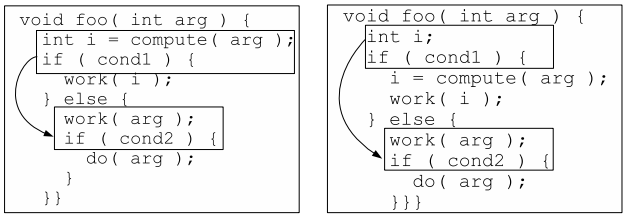
IV. WCET-aware Superblock Optimizations

In this section, we exploit the WCET-aware superblocks for compiler optimizations. Section IV-A briefly introduces basic techniques from static program analysis required for the development of the superblock-based optimizations CSE and DCE. These two optimizations were chosen for our case study due to their popularity and applicability. The optimizations are discussed in Sections IV-B and IV-C, resp.

A. Static Program Analysis

Static program analysis tries to determine dynamic properties of the program without actually executing it. For our optimizations, we need to know which statements access which storage locations (variables).

An *alias analysis* determines which targets a memory reference (i. e., pointers in the C language) may point to. For many compiler optimizations it is not sufficient just to know to which variables a pointer may point to, but which variables are read or written by which expressions. These results can be expressed by *def/use sets*, that represent sets of symbols from/to which an expression *may* read/write ($\mathrm{USE}_{\mathrm{may}}$ and $\mathrm{DEF}_{\mathrm{may}}$). Our computation of def/use sets integrates results of an alias analysis and is based on *syntax directed definitions* [11]. The last analysis required for the superblock-based optimizations is the livetime analysis. It is a classical data flow analysis which determines for each



(a) Original code (b) After Superblock-DCE

Figure 6. Example for Superblock-DCE

program point if a variable is *live* or otherwise *dead*. A variable v is called live on a CFG edge if there is a directed path from that edge to a use of v that does not contain any redefinition of v. A variable is *live-in* at a statement s if it is live on any of the incoming edges of s. A variable v is called *live-out* at a statement s if it is live on any of the outgoing edges of s [12]. $\text{LIVE-IN}_{\text{may}}(s)$ and $\text{LIVE-OUT}_{\text{may}}(s)$ are the *may*-sets of variables v that are live-in or live-out at s.

B. Common Subexpression Elimination

The optimization common subexpression elimination replaces recomputations of expressions by temporary variables which hold the already computed result (called *common subexpression*). Local CSE operates on the limited scope of a single basic block. Global CSE works on entire functions but side entrances in the control flow graph often cancel opportunities for CSE since common subexpressions may be overwritten in the side entrance path. Superblock-CSE (*SB-CSE*) can outperform the local and global CSE since it operates on multiple basic blocks and removes conflicting side entrances.

Our SB-CSE is based on Ghiya's approach [13] and relies on def/use sets. It traverses a superblock in a top-down manner and updates the set of available expressions $availList$ at each statement s. An expression e is called **available** at a statement s of a superblock if e was computed in a preceding superblock statement s_{comp} and there is no redefinition of any operands $o \in e$ in the superblock code between s_{comp} and s (including s). At each expression $e \in s$ it is checked if e redefines any of the operands of the available expressions $availExp \in availList$, i. e., $\text{USE}_{\text{may}}(availExp) \cap \text{DEF}_{\text{may}}(e) \neq \emptyset$. If so, $availExp$ is removed from $availList$. Otherwise, if e equals $availExp$, a mapping $availExp \rightarrow e$ is registered. If inner loops are encountered during the superblock traversal, $availList$ must be updated, i. e., expressions that are redefined within the loop are removed from $availList$. Finally, CSE traverses all registered mappings, creates a temporary variable $t = availExp$, and replaces all redundant $e_1, ..., e_n$ by t. An example from the $G721$ benchmark for SB-CSE is shown in Figure 5 (trace is marked). As can be seen, the expression `state->b[cnt]` can be reused in the superblock.

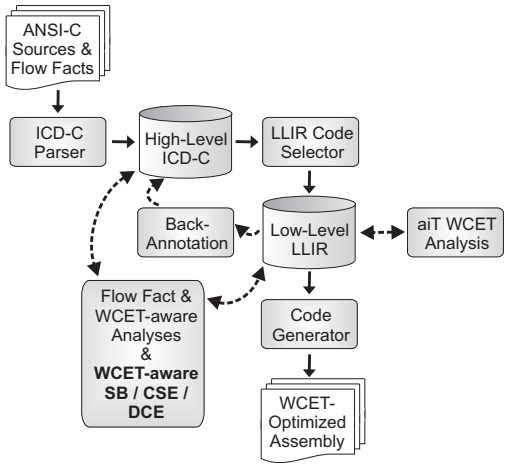


Figure 7. WCET-aware C Compiler WCC

C. Dead Code Elimination

Classical dead code elimination deletes dead statements. A statement s is called **dead** if $\text{LIVE-OUT}_{\text{may}}(s) \cap \text{DEF}_{\text{may}}(s) = \emptyset$. Results computed by s are then not required in further program flow. In case of the superblock-based DCE (*SB-DCE*) not only superblock-dead statements are removed but they are also moved outside the superblock (aka. *variable migration*).

Definition 3: *A statement s^{SB} is said to be **superblock-dead** in superblock SB iff: (a) it is not dead and (b) none of its defined variables $(\text{DEF}_{\text{may}}(s^{SB}))$ is read in SB before being re-defined.*

A superblock-dead statement s^{SB}_{dead} can be removed from a superblock *SB* by copying s^{SB}_{dead} into all control flow paths that exit the superblock and which contain statements that require the results of s^{SB}_{dead} [1]. Our SB-DCE algorithm moves each superblock-dead statement s^{SB}_{dead} downwards in the superblock, passing all succeeding statements. When a side exit is passed, our algorithm copies s^{SB}_{dead} into all successor blocks b_{succ} which are not part of *SB* and for which s^{SB}_{dead} is live at block b_{succ}, i. e., $\text{LIVE-IN}_{\text{may}}(b_{succ}) \cap \text{DEF}_{\text{may}}(s^{SB}_{dead}) \neq \emptyset$. The motion of s^{SB}_{dead} is stopped if the superblock end is reached. If s^{SB}_{dead} is live in the superblock end-block b_{end} $(\text{LIVE-OUT}_{\text{may}}(b_{end}) \cap \text{DEF}_{\text{may}}(s^{SB}_{dead}) \neq \emptyset)$, s^{SB}_{dead} is kept at the superblock end to preserve data dependencies, otherwise s^{SB}_{dead} can be completely removed. As shown in Figure 6, the computation of `i` is removed from the superblock that traverses the *else*-part of the outermost `if`-condition.

V. Experimental Environment

Our WCET-aware superblock formation and optimizations are integrated into the WCET-aware C compiler *WCC* [15] for the Infineon TriCore TC1796 processor that is equipped with an instruction cache. The workflow is depicted in

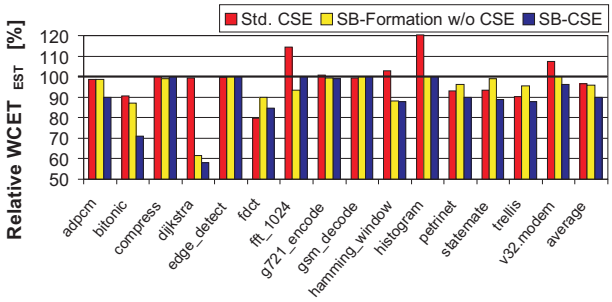


Figure 8. Relative WCET Estimates for CSE

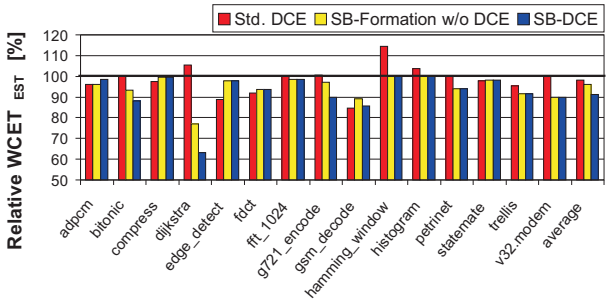


Figure 9. Relative WCET Estimates for DCE

Figure 7. The compiler is provided with C source files annotated with *flow facts* in the form of C pragmas. Flow facts are mandatory for a static WCET analysis and represent information about the loop iteration counts and the recursion depth. The parsed C code is translated into the high-level intermediate representation *ICD-C*, where standard compiler analyses and source code ACET optimizations can be applied. Next, the code selector translates the code into the low-level intermediate representation *LLIR*, also equipped with different analyses and assembly level optimizations. In total, the compiler features 43 different ACET optimizations which are activated at the highest optimization level *O3* including a local CSE and DCE.

As already mentioned, the sophisticated WCET analyzer aiT is tightly integrated into WCC's back-end. It allows the import of a worst-case timing model into the compiler to exploit it for analyses and optimizations. Via back-annotation, WCET information is propagated to the front-end and can be exploited for WCET-aware superblock formation and for superblock-based optimizations. Our analyses are *flow fact-aware*, i. e., flow facts are modeled within the compiler and are automatically updated during optimization,

To demonstrate the practical use of our approach, experiments on a large number of different benchmarks were conducted. The 55 benchmarks come from the test suites DSPstone, MediaBench, MiBench, MRTC WCET Benchmark Suite, and UTDSP.

For the experiments the following optimization parameters were used: the code size restrictions allow the maximal superblock size to be 5 times as large as the original trace, and a maximal increase of a function and the entire program by a factor of 3 and 2.5, resp. Moreover, the timing analyzer aiT was run after each 4th superblock formation including the application of the superblock optimizations to update WCET information. For the remaining steps, the IPET-based approach (cf. Section III-C) was used. These settings were empirically determined and showed good performance.

VI. Results

Worst-Case Execution Time

Figure 8 shows the impact of different common subexpression elimination strategies on the WCET estimates ($WCET_{EST}$) for a subset of 15 representative benchmarks. The *average*-bars on the right-hand side of the figure indicate the results averaged for all considered 55 benchmarks. The 100% base line represents the WCET of the benchmarks compiled with the highest optimization level (*O3*) and disabled CSE. The first bars per benchmark (labeled with *Std. CSE*) represent the results for the standard local ACET CSE. As can be seen, the WCET improvements are marginal with an average improvement of 3.4%. The second bars represent the WCET for the code optimized with *O3*, disabled CSE, and the WCET-aware superblock formation. An average WCET improvement of 4.0% was achieved.

It should be noted that the superblock formation as well as the SB-CSE and the SB-DCE exploit WCC's capability to quantify transformation effects. If the estimated WCET is larger after the code modification than before, then the modification is rolled back. This can happen because of phenomena which are not predictable from the high-level, such as register spill code or cache overflows. Thus, the diagram does not show any results larger than 100%.

Best results were achieved for the CSE based on the WCET-aware superblocks (bars labeled with *SB-CSE*). WCET improvements of up to 42.1% for the *dijkstra* benchmark were observed. The average improvement for the 55 benchmarks amounts to 10.2%. This result is remarkable since it increases the optimization potential of a traditional, intensively studied compiler optimization by a factor of 3. From the comparison between the second and third bars also the conclusion can be drawn that superblock formation often provides optimization potential which can be only fully exploited by a tailored superblock-based optimization (cf. e. g., benchmark *statemate*).

The results for the superblock DCE achieved for the 15 representative benchmarks are shown in Figure 9. Compared to the 100% bas line, which represents the WCET for the highly optimized code using *O3* without DCE, the standard local ACET DCE achieved on average for all 55 benchmarks a WCET reduction of 2.0%. The superblock formation amounts to an average WCET reduction of 4.0%. Again, the most effective WCET reduction was achieved with the DCE based on the WCET-aware superblocks, yielding an average improvement of 8.8%.

Optimization Level	Average ACET
O3 with Std. CSE	97.9%
O3 with SB-Formation w/o CSE	97.5%
O3 with SB-CSE	95.1%
O3 with Std. DCE	98.6%
O3 with SB-Formation w/o DCE	98.2%
O3 with SB-DCE	97.9%

Table I
ACET RESULTS FOR SB-OPTIMIZATIONS

Average-Case Execution Time

Our superblock formation is driven by WCET data. Thus, it is interesting to explore its impact on ACET. As rows one and four in Table I show, the reductions of the ACET for standard local CSE and DCE applied to the 55 benchmarks are comparable with those achieved for the WCET. Moreover, a comparison between the ACET and WCET results for the superblock formation and the superblock-based optimizations shows that higher improvements are achieved for the WCET reduction. One reason is that the WCET-aware optimizations focus on the WCEP which might be different from the most frequently executed path. This emphasizes the need for tailored WCET-aware optimizations.

Code Size

Since superblock formation is a code expanding transformation, the resulting code size increase is critical. We measured the code size for SB-CSE and SB-DCE for two different scenarios. In a first scenario, the same code size restrictions during trace selection were utilized as for the previous WCET and ACET experiments. Here, an average code size increase of 23% for the SB-CSE and 28% for the SB-DCE was observed. In a second scenario, code expansion was not limited. Code size increases of approximately 107% were measured on average for both optimizations. Simultaneously, the WCET results slightly degraded, most probably due to adverse instruction cache overflows or similar cache effects. Thus, it can be inferred that a code size restriction is mandatory for a balanced trade-off between WCET improvements and code size increases.

Compilation Run Time

The compilation run time of our optimizations was measured on an Intel Xeon system (2.4 GHz, 8 GB RAM). In a first scenario, we ran aiT after each 4th superblock formation and optimization as in the previous experiments. This leads to a compilation time increase by 540% compared to the case with standard optimizations only. For performance-oriented embedded systems this increase is still acceptable. To check if no WCEP switches were missed between two runs of the timing analyzer, aiT was run after each second superblock formation in a second scenario. This led to a compilation time increase by 757% and marginal WCET improvements of less than 1%. Thus, it can be concluded that too frequent WCEP updates by a costly WCET analysis do not pay off.

VII. CONCLUSIONS AND FUTURE WORK

It has been shown that superblocks are an effective technique for compiler optimizations since they restructure code such that additional optimizations are enabled. This paper is the first one to build superblocks at source code level for an effective WCET reduction. We propose a novel trace selection algorithm and re-design the traditional optimizations common subexpression and dead code elimination such that they operate on WCET-aware superblocks. Our experiments show that we significantly outperform standard CSE and DCE and achieve average WCET reductions on 55 real-life benchmarks for our superblock-based CSE and DCE of 10.2% and 8.8%, resp.

In the future, we intend to develop further WCET-aware source code compiler optimizations that exploit superblocks. Moreover, we plan the development of a WCET-aware instruction scheduling which operates on assembly code but reuses the source-code superblock structures.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software – Practice & Experience*, vol. 21, no. 12, 1991.

[2] R. Kidd and W. W. Hwu, "Abstract Improved Superblock Optimization in GCC," in *GCC Summit*, 2006.

[3] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, 1981.

[4] P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode," in *Proc. of MICRO*, 1988.

[5] W. W. Hwu, S. A. Mahlke, and W. Y. Chen et al., "The Superblock: an Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, vol. 7, 1993.

[6] AbsInt GmbH, "Worst-Case Execution Time Analyzer aiT for TriCore," http://www.absint.com/ait, 2010.

[7] W. Zhao, W. Kreahling, D. Whalley *et al.*, "Improving WCET by Optimizing Worst-Case Paths," in *Proc. of RTAS*, 2005.

[8] Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software," in *Proc. of RTSS*, 1995.

[9] A. Erosa and L. J. Hendren, "Taming Control Flow: A Structured Approach to Eliminating Goto Statements," in *Proc. of ICCL*, 1994.

[10] A. C. Shaw, "Reasoning About Time in Higher-Level Language Software," *IEEE Transactions on Software Engineering*, vol. 15, 1989.

[11] P. Tonella, "Effects of Different Flow Insensitive Points-to Analyses on DEF/USE Sets," in *Proc. of CSMR*, 1999.

[12] A. W. Appel, *Modern Compiler Implementation in C*. New York, NY, USA: Cambridge University Press, 1997.

[13] R. Ghiya and L. J. Hendren, "Putting Pointer Analysis to Work," in *Proc. of POPL*, 1998.

[14] H. Falk, "WCET-aware Register Allocation based on Graph Coloring," in *Proc. of DAC*, 2009.

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

[16] P. Lokuciejewski, H. Falk, P. Marwedel, H. Theiling, "WCET-Driven, Code-Size Critical Procedure Cloning," in *Proc. of SCOPES*, 2008.