# A Compile-Time Optimization Method for WCET Reduction in Real-Time Embedded Systems through Block Formation

MORTEZA MOHAJJEL KAFSHDOOZ, Sharif University of Technology
MOHAMMADKAZEM TARAM, Sharif University of Technology
SEPEHR ASAD, Sharif University of Technology
ALIREZA EJLALI, Sharif University of Technology

Compile-time optimizations play an important role in the efficient design of real-time embedded systems. Usually, compile-time optimizations are designed to reduce average-case execution time (ACET). While ACET is a main concern in high-performance computing systems, in real-time embedded systems concerns are different and worst-case execution time (WCET) is much more important than ACET. Therefore, WCET reduction is more desirable than ACET reduction in many real-time embedded systems. In this paper, we propose a compile-time optimization method aimed at reducing WCET in real-time embedded systems. In the proposed method, based on the predicated execution capability of embedded processors, program code blocks that are in the worst-case paths of the program are merged to increase instruction level parallelism and opportunity for WCET reduction. The use of predicated execution enables merging code blocks from different worst-case paths that can be very effective in WCET reduction. The experimental results show that the proposed method can reduce WCET by up to 45% as compared to previous compile-time block formation methods. It is noteworthy that compared to previous works, while the proposed method usually achieves more WCET reduction, it has considerably less negative impact on ACET and code size.

Categories and Subject Descriptors: C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Compile-Time Optimization, Hyperblock, WCET

## 1. INTRODUCTION

Basic blocks on average consist of only 5 to 7 instructions [Fisher et al. 2005]. This small number of instructions in each basic block restricts many optimization opportunities that could be provided in larger code blocks. To overcome this limitation, some block formation techniques were proposed to merge several code blocks into larger blocks such as trace [Fisher 1981], superblock [Chang et al. 1991] and hyperblock [Mahlke et al. 1992]. These block formation techniques, like many other compile-time optimizations, aim at reducing average-case execution time (ACET). In these techniques, using profiling data, most frequently executed code blocks are selected to be merged into larger code blocks. From the viewpoint of general purpose computing, this block selection approach is reasonable because ACET is the most important factor of performance in general purpose computing. However, in many real-time embedded systems, the most important factor, instead of ACET, is worst-case execution time (WCET).

There is little work on exploiting block formation for WCET reduction. Zhao et al [2005a] used block formation aimed at reducing WCET, where worst-case paths are identified by a timing analyzer and then superblocks are formed over these paths

New Paper, Not an Extension of a Conference Paper.
This work is supported by the deputation of research and technology of Sharif University of Technology, under grant G930827.

DOI: 0000001.0000001

at assembly-code level. Their method stops when code size limitation is reached or there is no possibility for more superblock formation. In a later work, Lokuciejewski et al [2010] proposed another block formation method for WCET reduction. Similar to [Zhao et al. 2005a], they also form superblocks on WCET paths; but in their method, superblock formation is conducted early at source code level and hence subsequent optimizations can benefit from larger code blocks. However, in superblock formation, only code blocks that are in one execution path can be merged together. This limitation results in missing some optimization opportunities that could be provided by merging code blocks from multiple execution paths.
In contrast to superblock formation, hyperblock formation by exploiting predicated execution allows code blocks from multiple execution paths to be merged. Therefore, hyperblock formation has more flexibility than superblock formation and as we discuss in this paper it can achieve more WCET reduction. To the best of our knowledge, the only previous work that exploits hyperblock formation for WCET reduction was published by Yan and Zhang [2008]. In their method, without considering worst-case paths, all execution paths (including worst-case and non-worst-case paths) are merged together. Although in some cases, this method (i.e. merging all execution paths) reduces WCET, in some other cases, especially when the program has several execution paths, it can even increase WCET.

In this paper, we propose a novel hyperblock formation method for WCET reduction. In the proposed method, based on information obtained from a timing analyzer, worst-case path(s) are identified and hyperblocks are formed on these paths. Since the proposed method exploits hyperblock, it has more flexibility than methods that exploit superblock and hence it can achieve more WCET reduction. Moreover, since the proposed method considers worst-case paths, it can also achieve more WCET reduction than other methods which use hyperblock formation but do not consider worst-case paths.

In this paper, we assumed that the target processor is a VLIW processor, which supports predicated execution. The main reasons why we chose a VLIW processor are as follows:

— VLIW processors have more predictable behavior than other modern processors such as superscalar processors [Yan and Zhang 2008]. This is because in VLIW processors instruction scheduling is conducted offline by the compiler and hence more details of execution are known at design time.
— In addition to high predictability, VLIW processors provide high computational power and hence are suitable for high performance real-time embedded systems [Fisher et al. 2005].
— In VLIW processors, the management of hardware resources is usually conducted at compile-time rather than at run-time. This makes VLIW processors more suitable for compile-time optimization techniques [Fisher et al. 2005]. Moreover, compile-time management of hardware resources simplifies the control unit of VLIW processors and hence increases energy efficiency of these processors. Energy efficiency is an essential requirement for many energy constraint real-time embedded systems.

It should be noted that the proposed method is not limited to VLIW processors and it can be exploited for the other processors that support predicated execution (e.g. 32bit ARM processors [Seal 2000]).

To evaluate the proposed method we implemented it in an optimizing compiler. We also implemented some other similar block formation methods for fair comparison. We tested our method and similar methods for several benchmarks from MiBench [Guthaus et al. 2001] and Mälardalen [Gustafsson et al. 2010] benchmark suites. The experimental results show that the proposed method significantly reduces WCET. In the best case, the proposed method reduces WCET by 33% compared to the recent previous method that merges all execution paths and by 45% compared to the superblock formation method. It should be noted that these results are estimated WCETs. This is because calculating or measuring the real WCET by using the state-of-the-art approaches is impracticable [Wilhelm et al. 2008] and just an estimation of the real WCET can be obtained. Therefore, the effectiveness of the proposed algorithm just can be measured and seen as a reduction in WCET estimates. However, we believe that the proposed algorithm just like the previous works (e.g., [Lokuciejewski et al. 2010; Zhao et al. 2005a]) is also effective in reducing the real WCET.

In addition to more WCET reduction compared to previous works, the proposed method also has considerably less negative impact on ACET and code size. ACET is important in real-time embedded systems that have energy constraints or reliability concerns, because with reduced ACET we achieve more dynamic slack time that can be exploited for reducing energy consumption [Aydin et al. 2004] or improving system reliability [Pradhan 1996]. Also code size is of great importance in many embedded systems, as it has been shown that code size efficiency has a considerable influence on the performance, and energy efficiency of embedded systems [Fisher et al. 2005; Henkel and Parameswaran 2007].

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 presents the proposed method. Section 4 describes the experimental setup. Section 5 shows the experimental results and finally Section 6 concludes the paper.

## 2. RELATED WORK

Several compile-time WCET optimization approaches have been proposed in the literature. WCET aware register allocation has been considered by [Falk 2009; Falk et al. 2011; Huang et al. 2014]. Falk [2009] and Falk et al [2011] proposed two methods based on graph coloring and integer linear programming respectively. The main idea behind these methods is to minimize the amount of spill codes on worst-case paths. Huang et al [2014] focused on the phase-ordering problem in clustered VLIW processors. They proposed an iterative heuristic method to jointly conduct register allocation, clustering and instruction scheduling for WCET reduction. Lokuciejewski et al [2008b; 2008a; 2009; 2010] considered various compile-time optimizations such as "*procedure cloning*", "*procedure positioning*", "*function inlining*", "*loop invariant code motion*" and "*trace scheduling*" and used them for WCET reduction. Falk and Schwarzer [2006] used "*loop nest splitting*" for optimizing WCET and improving time predictability.

Zhao et al [2005b] considered the overhead of jump instructions and proposed an approach aimed at decreasing the number of jump instructions in worst-case paths, thereby reducing WCET. "*Loop unrolling*" is exploited in [Lokuciejewski and Marwedel 2009] to reduce WCET. In this work, an unrolling factor is determined for each loop based on its maximum number of iterations and adverse effects of unrolling.

For processors that use scratchpad memory (SPM), several compile-time SPM allocation methods have been proposed (e.g. [Falk and Kleinsorge 2009; Suhendra et al. 2005; Wu et al. 2010]). Falk et al [2006] proposed a WCET aware C Compiler (WCC). Schoeberl et al [2011] proposed a time predictable VLIW processor. In their work, compiler optimizations are used to reduce WCET; however, there is no reference to the exploited optimizations.

Leupers [1999] exploited predicated execution to reduce WCET. In his approach, using dynamic programming, conditional jumps are selected to be eliminated by predicated execution. Puschner [2002] and Schoeberl et al [2009; 2011; 2015] used predicated execution for improving time predictability. In their approaches, all execution paths of the program code are merged into a single path. It is noteworthy that our work differs from the mentioned works as none of them exploits block formation for reducing WCET.

Block formation has been exploited by several previous works aimed at reducing ACET (e.g. [Fisher 1981; Chang et al. 1991; Mahlke et al. 1992]). In these works, usually based on profiling data most frequently executed paths are identified and then code blocks that are in these paths are merged into larger code blocks. Although similar to these works we exploit block formation, our work differs from these works since as mentioned previously our goal is to reduce WCET instead of reducing ACET and hence we form code blocks on worst-case paths instead of most frequently executed paths. Moreover, since in many cases, profiling data cannot safely identify WCET and worst-case paths we instead use timing analysis to identify WCET and worst-case paths.
As we mentioned previously, there is little work on exploiting block formation for WCET reduction. Zhao et al [2005a] and Lokuciejewski et al. [2010] exploited superblock formation to reduce WCET. Our method differs from these methods as we form hyperblocks instead of superblocks to reduce WCET. In another work, Yan and Zhang [2008] exploited hyperblock formation for WCET reduction. However, in contrast to our method, their method is unaware of WCET and aggressively merges all execution paths.

Whitham and Audsley [2010] used the concept of virtual trace to increase the time predictability of out of order processors. A virtual trace is a sequence of basic blocks that instead of being scheduled at compile-time are scheduled at run time by processor. They proposed a trace formation algorithm at assembly level to select virtual traces. Moreover, they dealt with the unbiased conditional jumps issue and exploited predicated execution to solve it. This work also differs from our work as it considers superscalar processors and runtime trace scheduling.

## 3. PROPOSED METHOD

In this section, we first provide a brief review of some concepts that we use to explain the proposed method. Then, we provide the motivation behind using hyperblock formation for WCET reduction. Finally, we describe the proposed method and the technique that we use for timing analysis.

### 3.1. Background

In this subsection, we review some concepts that we used through the paper to explain the proposed method. It should be noted that we assume the reader is familiar with the basic concepts such as *predicated execution*, *control flow graph (CFG)*, and *basic block*,[Fisher et al. 2005].

*Trace* is a cycle free code block that is made by merging some basic blocks existing on a path of CFG. The definition of trace makes no restriction on the number of entry points to a trace, thus a trace can have more than one entry point. However, allowing more than one entry point to a trace complicates the generation of compensation code [Fisher et al. 2005].

*Superblock* is a trace with the additional constraint that all the entrances are allowed only to the first instruction of the superblock. In other words, a superblock is a single-entry, multiple-exit trace.

*Hyperblock* is also a single-entry, multiple-exit block. However, in contrast to superblock, hyperblock can contain several mutually exclusive paths of the program code by using predicated execution.

*Dominator and post-dominator*. We say that block B is a dominator of block A, if all paths from the entry block to block A, pass block B. Similarly, block B is called a post-dominator of block A, if all paths from block A to the exit block, pass block B. It should be noted that in this paper we assume (as is common for most of real CFGs) there is a single distinguished entry and exit block in the CFG. In Fig. 1(a), block $B_1$ is a dominator of block $B_3$ and block $B_6$ is a post-dominator of block $B_3$.

*Tail duplication.* Superblock and hyperblock definitions do not have side entrances by definition. This restricts the size of superblocks and hyperblocks that can be formed and thereby decreases the achievable optimization opportunity. To avoid this restriction, the tail duplication technique [Fisher et al. 2005] is used. In this technique, for each side entry point of a superblock or hyperblock, the part of the code between the side entry point and the end of the block is duplicated and the side entrances are retargeted to the duplicated counterpart. Fig. 1 shows an example for tail duplication.

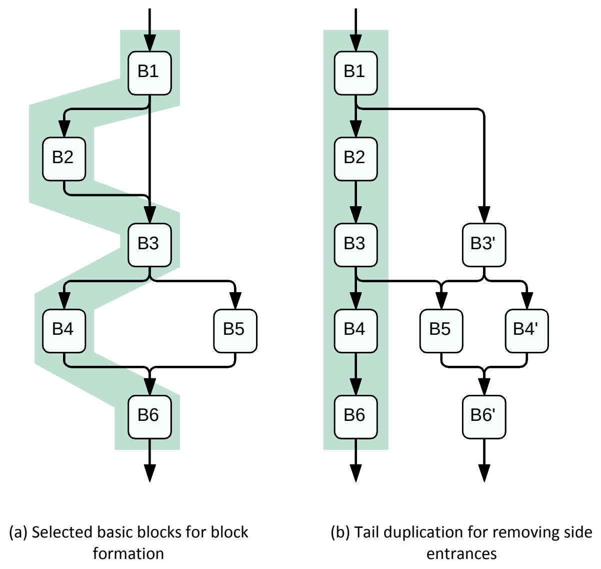

B1
B2
B3
B4
B5
B6
(a) Selected basic blocks for block formation
B1
B2
B3
B3'
B4
B5
B4'
B6
B6'
(b) Tail duplication for removing side entrances

Fig. 1: Tail duplication.

## 3.2. Motivation

Conventional global optimization techniques must be simultaneously conscious of all execution paths in the program code, which makes compilation very complex and time consuming. In contrast, superblock formation techniques only consider one path in the program code at a time. Although superblock formation reduces the complexity of compilation, some optimizations related to the multiple paths of the program code can be missed. Hyperblock formation provides flexibility in the number of paths each block contains. Therefore, it potentially provides more chances for compiler optimizations than superblock formation and also does not have the complexity of global optimization techniques [Fisher et al. 2005]. Hyperblock formation also eases the combination of speculative and predicated execution, which previously was difficult [Mahlke et al. 1992]. Exploiting predicated execution in hyperblock formation removes time consuming conditional jumps and hence can reduce execution time. Moreover, hyperblock formation can improve the efficiency of software pipelining [Fisher et al. 2005].

In addition to the mentioned benefits, hyperblock formation has another important advantage in reducing WCET. The general idea behind WCET reduction is to merge basic blocks that are on the worst-case path. However, other paths may exist which have a WCET equal (or nearly equal) to the WCET of the worst-case path. In such cases, optimizing one worst-case path not only does not reduce the whole WCET of the program, but also probably increases it because of increasing the execution time of the other paths. This happens when there are conditional jumps in the worst-case paths in which their taken and not-taken branch paths have equal or nearly equal execution time. In these cases, hyperblock formation by merging those paths can reduce the WCET. To illustrate this, we provide an example.

Fig. 2(a) shows an example program in which the alternative paths have equal execution time. The CFG of the program is shown in Fig. 2(c). Fig. 2(b) denotes how variables are mapped to the registers and Fig. 2(d) shows the machine instructions after compiling the code of Fig. 2(a). The basic blocks' instructions are separated by the horizontal lines. The basic block scheduling of these instructions is shown in Fig. 2(e). It should be noted that in this paper (just like the previous works [Lokuciejewski et al. 2010; Yan and Zhang 2008]) we do not address the problem of instruction scheduling (e.g., the internal instructions of a basic block) and we intend to just focus on block formation. Therefore, in this example we have assumed that to overcome the complexity of global scheduling (which is almost infeasible due to the huge compilation time), all block formation methods (e.g., superblock and hyperblock formation) are using an optimal local scheduler for scheduling the instructions within blocks. We have also assumed that such scheduler exists and can be implemented using the techniques like the one in [Wilken et al. 2000] (Indeed, we have implemented such an optimal scheduler for our studies; nevertheless, as it is beyond the scope of this article we do not address this issue).

We assumed that in this example the processor has two ALUs, one memory unit and one branch unit. In the table of Fig. 2(e), the numbers in the leftmost column show the time in cycles and the number in each cell shows the instruction that is executed at the corresponding time slot in the corresponding functional unit. The empty cells show that the corresponding functional units are idle. The WCET of the example basic block scheduling as is shown in the following expression is calculated by adding the WCET of BB1, MAX(WCET(BB2),WCET(BB3)) (as either BB2 or BB3 is executed and not both of them) and the WCET of BB4.

$$\text{WCET(Program)=WCET(BB1)+MAX(WCET(BB2),WCET(BB3))+WCET(BB4)}$$
$$\text{=4+MAX(5,5)+2=11 cycles}$$

Fig. 3 depicts superblock formation and scheduling of the example program. The figure shows that forming superblock on each branch path of the jump increases the execution time of the other branch path and hence increases the overall WCET. In Fig. 3(a) two superblocks are formed which are specified by dashed line. In this example, forming superblocks causes the WCET of the program to increase by one cycle.

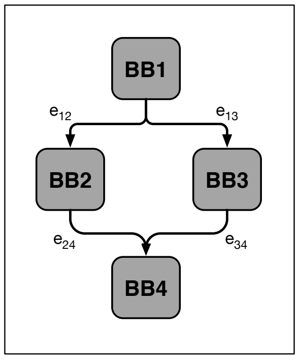
As mentioned earlier, because of the complexity of global scheduling techniques, superblock formation uses a local optimal scheduler for scheduling instructions whithin superblocks [Lokuciejewski et al. 2010]. Thus in Fig. 3(c) the scheduling of superblocks SB1 and SB2 are locally optimized. For example, assume that instead of the schedule of Fig. 3(c), we scheduled instructions 1-3 earlier than the other instructions. Then As it is depicted in Fig. 4 the execution time of the superblock SB1 would be 7 cycles. However, it can be seen from Fig. 3(c) that the execution time of its schedule is 6 cycles, which means that if we scheduled instructions 1-3 earlier than the other instructions, we would have a non-optimal schedule. Indeed, the CFG of the program has two possible execution paths, SB1 and SB1-SB2. Therefore, to calculate the WCET of the program we should consider the maximum of the execution time of these paths.

```
coff = 0.5 * n;
coff = coff*coff;
if (coff > 1)
    j=(c*k+3)*b;
else
    j=(d*k+3)*b;
out1=k*coff;
out2=j*out1;
```

(a) Program Code

| Var | Reg |
|---|---|
| n | R1 |
| coff | R2 |
| b | R3 |
| d,c | R4 |
| k | R5 |
| j | R6 |
| out1 | R7 |
| out2 | R8 |

(b) Regs



(c) Control Flow Graph

| | | |
|---|---|---|
| 1 | | ld R1,(n) |
| 2 | (BB1) | multf R2,0.5,R1 |
| 3 | | multf R2,R8,R2 |
| 4 | | blt R2,1,L1 |
| 5 | | ld R3,(b) |
| 6 | | ld R4,(c) |
| 7 | | ld R5,(k) |
| 8 | (BB2) | multf R6,R4,R5 |
| 9 | | addf R6,R6,3 |
| 10 | | multf R6,R6,R3 |
| 11 | | j L2 |
| 12 | L1: | ld R3,(b) |
| 13 | | ld R4,(d) |
| 14 | (BB3) | ld R5,(k) |
| 15 | | multf R6,R4,R5 |
| 16 | | addf R6,R6,3 |
| 17 | | multf R6,R6,R3 |
| 18 | L2: | multf R7,R5,R4 |
| 19 | (BB4) | multf R8,R7,R6 |

(d) Instructions

| | Mem | Br | ALU1 | ALU2 |
|---|---|---|---|---|
| BB1 | | | | |
| 1 | 1 | | | |
| 2 | | | 2 | |
| 3 | | | 3 | |
| 4 | | 4 | | |
| BB2 | | | | |
| 1 | 6 | | | |
| 2 | 7 | | | |
| 3 | 5 | | 8 | |
| 4 | | | 9 | |
| 5 | | 11 | 10 | |
| BB3 | | | | |
| 1 | 13 | | | |
| 2 | 14 | | | |
| 3 | 12 | | 15 | |
| 4 | | | 16 | |
| 5 | | | 17 | |
| BB4 | | | | |
| 1 | | | 18 | |
| 2 | | | 19 | |

(e) Schedule Table

Fig. 2: Basic block Scheduling.

The execution time of the path SB1 can be obtained from the scheduling table of Fig. 3(b). The execution time of the path SB1-SB2 is the sum of the execution time of SB1 when the program flow leaves SB1 through the edge $e_{13}$ and the execution time of SB2. Therefore, the WCET of the program after superblock formation is calculated by the following expression:

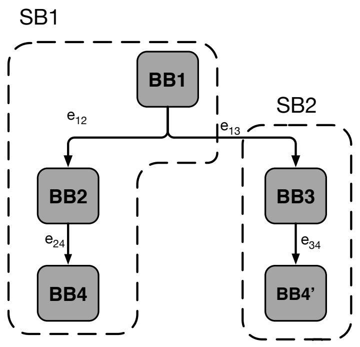
$$\begin{gathered}\text{WCET(Program)=MAX(WCET(SB1),WCET(SB1-SB2))}\\ \text{=MAX(6,6+6)=12 cycles}\end{gathered}$$

In contrast to superblock formation, hyperblock formation can increase compiler optimization opportunity and can decrease WCET by merging two branch paths of a jump. Fig. 5 shows in our example the hyperblock scheduling decreases WCET by one cycle compared to basic block scheduling. In this example, the hyperblock HB1 is formed by merging all the basic blocks. The predicated code of HB1 is shown in Fig. 5(b) and the scheduling is shown in Fig. 5(c).

Although the hyperblock formation has several benefits, it must be used consciously. This is because hyperblock formation increases the number of instructions that must be executed in parallel and due to the lack of hardware resources (esp. issue-width) it might increase (worst-case) execution time. In the next subsection, we present the proposed hyperblock formation algorithm for WCET reduction.

### 3.3. Hyperblock Formation Algorithm

The proposed algorithm is shown in Algorithm 1. Each iteration of the algorithm consists of the following steps:



(a) Control Flow Graph

| | | |
|---|---|---|
| 1 | (BB1) | ld R1,(n) |
| 2 | | multf R2,0.5,R1 |
| 3 | | multf R2,R8,R2 |
| 4 | | blt R2,1,L1 |
| 5 | (BB2) | ld R3,(b) |
| 6 | | ld R4,(c) |
| 7 | | ld R5,(k) |
| 8 | | multf R6,R4,R5 |
| 9 | | addf R6,R6,3 |
| 10 | | multf R6,R6,R3 |
| 11 | | j L2 |
| 12 | L1: (BB3) | ld R3,(b) |
| 13 | | ld R4,(d) |
| 14 | | ld R5,(k) |
| 15 | | multf R6,R4,R5 |
| 16 | | addf R6,R6,3 |
| 17 | | multf R6,R6,R3 |
| 18 | L2: | multf R7,R5,R4 |
| 19 | (BB4) | multf R8,R7,R6 |
| 20 | L2': | multf R7,R5,R4 |
| 21 | (BB4') | multf R8,R7,R6 |

(b) Instructions

| | Mem | Br | ALU1 | ALU2 |
|---|---|---|---|---|
| | SB1 | | | |
| 1 | 6 | | | |
| 2 | 7 | | | |
| 3 | 1 | | 18 | 8 |
| 4 | 5 | | 9 | 2 |
| 5 | | | 10 | 3 |
| 6 | | 4 | 19 | |
| | SB2 | | | |
| 1 | 13 | | | |
| 2 | 14 | | | |
| 3 | 12 | | 15 | 20 |
| 4 | | | 16 | |
| 5 | | | 17 | |
| 6 | | | 21 | |

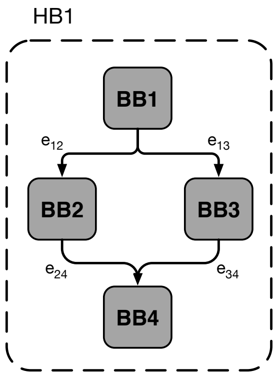
(c) Schedule Table

Fig. 3: Superblock Scheduling.

| | Mem | Br | ALU1 | ALU2 |
|---|---|---|---|---|
| | SB1 | | | |
| 1 | 1 | | | |
| 2 | 6 | | 2 | |
| 3 | 7 | | 3 | |
| 4 | 5 | 4 | 8 | 19 |
| 5 | | | 9 | |
| 6 | | | 10 | |
| 7 | | | 20 | |

Fig. 4: Non-Optimal Superblock Scheduling Example.

(1) *Timing Analysis*. In this step, the WCET of the program and the code blocks that are in the worst-case paths are determined (Line 2). Since the worst-case paths may change after each block formation, timing analysis is conducted in each iteration of the algorithm. We provide the details of this step in Section 3.5.
(2) *Block Selection*. After timing analysis and determining the critical blocks, the CFG of the program is traversed from the first block on the worst-case paths and blocks are selected according to their types for hyperblock formation (Line 3-33). We illustrate the details of this step in Section 3.4.
(3) *Block Merging*. In this step, blocks that are selected in the previous step are merged to form a hyperblock (Line 34). To merge selected blocks that are on different paths, predicated execution is exploited. Moreover, to preserve the semantics of the pro-



(a) Control Flow Graph

| | | |
|---|---|---|
| **1** | ld R1, (n) | |
| **2** | multf R8, 0.5 , R1 | |
| **3** | multf R8, R8 , R8 | |
| **4** | plt R8,1, p1 | |
| **5** | ld R2,(b) | |
| **6** | ld R3,(c ) | p1 |
| **7** | ld R3,(d) | !p1 |
| **8** | ld R11,(K) | |
| **9** | mult R4,R3,R11 | |
| **10** | add R4,R4,R2 | |
| **11** | mult R4,R4,3 | |
| **12** | multf R9,R11,R8 | |
| **13** | multf R10,R9,R4 | |

(b) Predicated Instructions

| | **Mem** | **Br** | **ALU1** | **ALU2** |
|---|---|---|---|---|
| HB1 | | | | |
| **1** | 1 | | | |
| **2** | 5 | | | 2 |
| **3** | 8 | | | 3 |
| **4** | | | 4 | 12 |
| **5** | 6 | | | |
| **6** | 7 | | | |
| **7** | | | 9 | |
| **8** | | | 10 | |
| **9** | | | 11 | |
| **10** | | | 13 | |

(c) Schedule Table

Fig. 5: Hyperblock Scheduling.

gram, whenever there are side entrances to the hyperblock, the tail duplication technique (Section 3.1) is used.

These steps are repeated until the program code size reaches the specified code size limitation or no more basic block can be selected to form new hyperblocks. Then, the algorithm finishes and the optimized CFG is returned.

### 3.4. Block Selection

As mentioned earlier, in the block selection step (Algorithm 1, Line 3-33), basic blocks are selected based on their types to form hyperblocks. We classify basic blocks into three types:

- Basic blocks that have an unconditional jump.
- Basic blocks that have a conditional jump and the both of their branch paths (taken and not-taken) have equal or nearly equal execution time (unbiased conditional jump). Indeed, we consider a conditional jump as unbiased if the WCET difference between its branch paths is less than the jump classifying threshold value ($JC\_Thr$).
- Basic blocks that have a conditional jump and the WCET difference between their branch paths is more than $JC\_Thr$.

The block selection step starts from the first critical basic block that has not been checked previously (Line 3) and in each iteration of the inner loop a basic block is added to the set of selected blocks, if it can be merged with the previously selected blocks (Line 5 and 6). Then, according to the type of the current block, the algorithm chooses a block for the next iteration (Line 7-29). If the current block does not have any conditional jump, the algorithm chooses the next block in the worst-case path to be checked in the next iteration (Line 7 and 8). This also is done for the blocks with biased conditional jumps (Line 9 and 10).
However, for the blocks with unbiased conditional jumps, at first, if possible, a temporary block is formed by merging both of its branch paths (Line 12 and 13) and then If the WCET is reduced by this block formation, all the blocks between the current block and its immediate post-dominator block (Section 3.1) are added to the set of selected

**ALGORITHM 1:** Hyperblock formation for WCET reduction

```
Input: Initial CFG ($CFG$), and Code Size limitation ($CS\_Limit$)
Output: Optimized CFG
do
    [$Critical\_Blocks$, $WCET$] ← Timing_Analyze ($CFG$);
    $Selected\_Blocks$ ← ∅ ;
    $b$ ← Get_First_Basicblock($Critical\_Blocks$); // the first critical block that has not been checked previously
    repeat
        if $b$ can be merged with $Selected\_Blocks$ then
            Add($Selected\_Blocks$, $b$);
            if $b$ has no conditional jump then
                $b$ ← Get_Next_Basicblock($Critical\_Blocks$);
            else if $b$ has biased conditional jump then
                $b$ ← Get_Next_Basicblock($Critical\_Blocks$);
            else // unbiased conditional jump
                if two branches of $b$ can be merged together then
                    $tCFG$ ← Make_Temporal_Block($b.blocks\_of\_branches$, $CFG$);
                    $W$ ← Timing_Analyze ($tCFG$);
                    if $W \leq WCET$ then
                        Add($Selected\_Blocks$, $b.blocks\_of\_branches$);
                        $b$ ← Get_Next_Basicblock($Critical\_Blocks$);
                        continue;
                    end
                end
                if one of the branches is longer than the other and its blocks can be merged together then
                    $tCFG$ ← Make_Temporal_Block($b.blocks\_of\_longer\_branch$, $CFG$);
                    $W$ ← Timing_Analyze ($tCFG$);
                    if $W \leq WCET$ then
                        Add($Selected\_Blocks$, $b.blocks\_of\_longer\_branch$);
                        $b$ ← Get_Next_Basicblock($Critical\_Blocks$);
                        continue;
                    end
                end
                break;
            end
        else
            break;
        end
    until no new blocks can be selected;
    $CFG$ ← Block_Merging($CFG$, $Selected\_Blocks$);
while $Selected\_Blocks$ is not empty & $CFG.CS < CS\_Limit$;
return $CFG$;
```

blocks and the next critical block is selected to be checked in the next iteration (Line 14-18). Otherwise, if one of the branch paths has a higher WCET than the other and its basic blocks can be merged together, the algorithm forms a temporary block over this branch path (Line 20 and 21). If this latter block formation reduces WCET (Line 22), the blocks on this branch path are added to the set of selected blocks and the next critical block is selected to be checked in the next iteration (Line 23-26). Otherwise, if none of the mentioned block formations reduces the WCET, the block selection step finishes for this iteration of the algorithm and the set of the previously selected blocks (if any block is selected) are passed to the next step (i.e. block merging) to form a new hyperblock. The block selection step also finishes when the current block cannot be merged by the previously selected blocks. This situation occurs when the current block is in a loop body and the previously selected blocks are outside of the loop or vice versa. It should be noted that by temporarily block formation and checking the obtained WCET, we ensure that the WCET is not increased in any of the iterations of the algorithm.

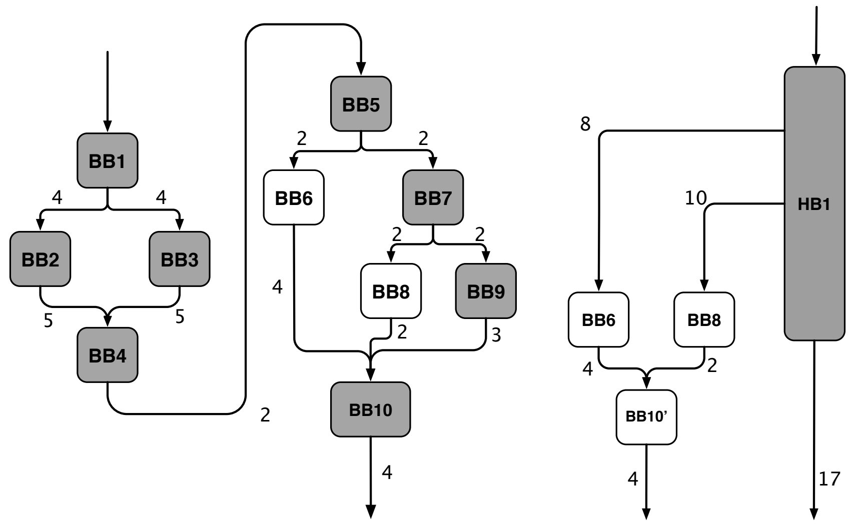
Two issues must be explained about the proposed algorithm:

— As we mentioned previously, conditional jumps for which the WCET difference between their branch paths is less than $JC\_Thr$ are considered as unbiased. For unbiased conditional jumps, in addition to block formation on the longest branch path, block formation by merging the non-longest branch path is also checked. This is because, when we have an unbiased conditional jump, if we do not merge the non-longest branch path with the longest one in the block formation, it is possible that the WCET of the non-longest branch path increases (Section 3.2) thereby increasing the whole WCET. However, when the WCET difference between the branch paths is high (biased conditional jumps), block formation on the longest branch path can achieve more WCET reduction as compared to block formation by merging both the branch paths.

— Changing the value of $JC\_Thr$ impacts the quality of the proposed algorithm. On one hand, by setting the value of $JC\_Thr$ to a small number (e.g. 1 cycle), many conditional jumps are considered as biased and hence the effect of utilizing predicated execution is only checked for a small number of conditional jumps. This reduces the number of times that the timing analysis is conducted to check the impact of the predicated execution and hence improves the speed of the proposed algorithm. However, in this case some optimization opportunities can be missed. On the other hand, by setting the threshold value to a large number (e.g. 100 cycles) many conditional jumps are considered as unbiased and hence the effect of utilizing predicated execution is checked for many conditional jumps. This in turn, provides more optimization opportunities, however in the expense of having a time-consuming algorithm (see Section 5.4). It is noteworthy that as our proposed algorithm is executed offline at design time, we believe it is reasonable to perform the proposed algorithm with large values for $JC\_Thr$ to achieve better optimizations for the runtime behavior of real-time embedded systems.

To further illustrate the algorithm (especially the block selection step) we provide an example.

**Example**
Fig. 6(a) shows the CFG of an example program. The highlighted blocks depict the worst-case paths of the CFG obtained by timing analysis. For this example, we assume $JC\_Thr$ is zero. The block selection step starts from BB1. BB1 has an unbiased conditional jump, therefore a temporary block is made over BB1, BB2, BB3, BB4 and then timing analysis is conducted to examine the impact of this temporary block formation on the WCET. For this example, the timing analysis shows that this block formation reduces the WCET. Therefore, BB1, BB2, BB3 and BB4 are added to the set of selected



(a) Original CFG (b) CFG After Hyperblock Formation

Fig. 6: An example of applying the proposed algorithm.

blocks (that will be finally merged into the same hyperblock). The block selection step continues from the next block in the worst-case path, i.e., BB5 in the example. BB5 has a biased conditional jump, therefore the next block in the worst-case path (i.e. BB7) is added to the set of selected blocks. BB7, like BB5, has a biased conditional jump, therefore BB9 which is in the worst-case path is added to the set of selected blocks. Finally, BB10 is added to the set of selected blocks and the block selection step finishes. In the next step which is block merging, the selected blocks are merged together using predicated execution and tail-duplication techniques (Sections 3.1) and a hyperblock is made. For example, as BB10 has side entrances from the blocks that are not in the previously selected blocks, it is duplicated and its side entrances are retargeted to its duplicated counterpart, i.e., BB10'. After checking BB10, since there is no more critical block to be checked in the next iteration, the algorithm finishes. Fig. 6(b) shows the CFG obtained from hyperblock formation.

## 3.5. Timing Analysis

The approach which we use for timing analysis is based on the implicit path enumeration technique (IPET) [Li and Malik 1995]. In IPET, relations between the execution of code blocks and also execution constraints such as loop bounds are modeled by arithmetic expressions, and then an integer linear programming or constraint programming method is used to calculate an upper bound for WCET and to identify the worst-case path. To model the execution time of a program by arithmetic expressions we assume that the CFG of the program consists of set $B = \{b_i \colon 1 \leq i \leq N\}$ of blocks and set $E = \{e_{ij} \colon b_i \text{ has out edge to } b_j\}$ of edges. Also for the sake of uniformity, we assume one dummy edge connects the last node to the first node. We use the following constants and variables in the expressions:

— The set of outgoing edges from block $b_i$ ($D_{b_i}$).
— The set of ingoing edges to block $b_i$ ($S_{b_i}$).

— The number of times the program flow passes through edge $e_{ij}$ ($I_{e_{ij}}$) that is equal to the number of times block $b_j$ is executed just after block $b_i$.
— The weight of edge $e_{ij}$ ($W_{e_{ij}}$) that is equal to the WCET of block $b_i$ when it is exited from edge $e_{ij}$.

It should be noted in the original IPET, for each block a unique weight is assumed that is equal to the worst-case execution time of the block. While assuming one worst-case execution time for each block is reasonable for the programs that only consist of basic blocks, it is not reasonable for the programs that contain superblocks or hyperblocks. This is because the worst-case execution time of a superblock or hyperblock is not a single value and depends on the edge that the block is exited from. To solve this issue, we assumed multiple worst-case execution times for each block and to simplify the arithmetic expressions, we assign each WCET of the block to its corresponding outgoing edge. The arithmetic expressions that we use to calculate the WCET of programs are as follows:

— *Block equations*. For each block an equation must be satisfied which indicates that the total number of times the program flow enters to the block must be equal to the total number of times the program flow departs the block:

$$\sum_{e_{xi} \in S_{b_i}} I_{e_{xi}} = \sum_{e_{iy} \in D_{b_i}} I_{e_{iy}}$$

— *Back-edge constraints*. For each back-edge the total number of passing through it (i.e. the total number of loop iterations) must be less than the specified upper bounds:

$$I_{e_{ij}} \leq \text{Upper\_Bound}(e_{ij})$$

— *Execution time expression*. To calculate the total execution time of the program, it is sufficient to multiply the total number of passing through each edge by its weight (execution time) and then to sum them all up:

$$ExecTime = \sum_{e_{ij} \in E} I_{e_{ij}} \times W_{e_{ij}}$$

These arithmetic expressions consist a model that is solved by an integer linear programming solver such as CPLEX [Cplex 2007] to find maximum (worst-case) execution time. It should be noted that by solving the IPET model for the whole program only one of the worst-case paths is determined. However, the proposed algorithm needs to know *i)* other worst-case paths that have exactly the same execution time, *ii)* the execution time of the paths that branch out of the worst-case paths in order to classify conditional jumps as biased and unbiased. To solve these issues, in the timing analysis step, in addition to the WCET of the whole program, for each conditional jump existing in the determined worst-case path, the WCET of its branch paths up to its immediate post-dominator block (Section 3.1) is also calculated using IPET. In this way, both the above-mentioned issues are solved, and hence biased and unbiased conditional jumps are identified.

**Example**
Fig. 7 shows the CFG of expint program [Gustafsson et al. 2010] after hyperblock formation. The number on each edge represents the execution time of the source code block if it exits from that edge. Maximum iterations of the outer and the inner loops

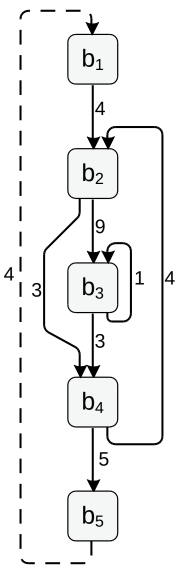


Fig. 7: The CFG of expint benchmark after hyperblock formation.

(a)
$$\begin{aligned}
b_1 &: I_{e_{12}} = I_{e_{51}} = 1\\
b_2 &: I_{e_{23}} + I_{e_{24}} = I_{e_{12}} + I_{e_{42}}\\
b_3 &: I_{e_{34}} + I_{e_{33}} = I_{e_{23}}\\
b_4 &: I_{e_{45}} + I_{e_{42}} = I_{e_{34}} + I_{e_{24}}\\
b_5 &: I_{e_{51}} = I_{e_{45}} = 1
\end{aligned}$$

(b)
$$\begin{aligned}
I_{e_{33}} &\leq 49\\
I_{e_{42}} &\leq 100
\end{aligned}$$

(c)
$$WCET = max\,(4I_{e_{12}} + 9I_{e_{23}} + 3I_{e_{24}} + 1I_{e_{33}} + 3I_{e_{34}} + 4I_{e_{42}} + 5I_{e_{45}} + 4I_{e_{51}})$$

Fig. 8: IPET model for expint benchmark.

are 101 and 50 respectively. The IPET model of expint program is shown in Fig. 8. Fig. 8(a) shows block equations which indicate that the total number of times the program flow enters to each block must be equal to the total number of times the program flow departs the block. Back-edge constraints are shown in Fig. 8(b). Since the maximum number of iterations of the inner loop is 50, the total number of passing through back-edge $e_{33}$ must be equal or less than 49 ($I_{e_{33}} \leq 49$). Similarly, since the maximum number of iterations of the outer loop is 101 the total number of passing through back-edge $e_{42}$ must be equal or less than 100 ($I_{e_{42}} \leq 100$). Finally, the objective function (maximizing the execution time) is shown in Fig. 8(c). The WCET of the program can be obtained by solving this model which yields 1674 clock cycles.

The following issues must be considered in the timing analysis technique that we exploit in the proposed algorithm:

— To calculate the WCET of the program the WCET of each code block is required. The WCET of code blocks highly depends on the architectural features of the processor. As mentioned previously, in this paper, we assume that the target processor is a VLIW processor with no memory hierarchy. “No memory hierarchy” is a reasonable assumption as many embedded real-time systems only use one level of memory in order to provide time-predictable behavior which is important for real-time applications. Also, to avoid time-predictability problems that pipelining may cause, we assume that all pipeline timing effects [Engblom and Ermedahl 1999] between code blocks (e.g. basic block, superblock, and hyperblock) are removed by exploiting the methods like the one proposed in [Yan and Zhang 2008]. Under these assumptions, there is no context sensitivity between the code blocks of the program and hence the WCET of each code block can be obtained independently from the other code blocks. Therefore, considering these assumptions that make the processor behavior highly predictable, the timing analysis approach in this subsection which is based on IPET can derive highly safe upper bounds for the execution time of the programs. It should be noted that although these assumptions are simple, they are realistic and indeed many real-time embedded systems are based on them. Moreover, they are not necessary for our proposed hyperblock formation algorithm. Rather, they are required only for the timing analysis technique which is exploited in our algorithm. As we will discuss in Section 3.6 one can use more sophisticated timing analysis techniques to relax these assumptions without requiring any change in our proposed algorithm. It is noteworthy that, although relaxing the second assumption (i.e. no pipeline timing effects between code blocks) may decrease actual WCET, in many cases it does not have considerable effect in WCET reduction because of pessimistic assumptions required to safely model pipeline timing effects between code blocks.
— To benefit other optimizations from larger blocks, the proposed algorithm is conducted on the intermediate representation (IR) level [Cooper and Torczon 2011] before many other optimizations. At this level, the exact assembly instructions are not determined yet so the WCET of the program cannot be calculated precisely. One solution to this issue is to conduct all remained compilation levels to obtain assembly code. However, because the timing analysis is conducted in each iteration of the proposed algorithm this leads to high timing overhead. To solve this issue, we use the length of the longest dependency chain among each block’s IR level instructions as an estimation of the block’s execution time. It should be noted that we use the longest dependency chain in each block only to determine worst-case paths in IR level. For checking that a block formation reduces WCET (Section 3.4) and for the final evaluation of the proposed algorithm we use the more precise WCET for each block obtained from its exact assembly code.

### 3.6. Extending to other processors

As we mentioned previously, the proposed algorithm in addition to VLIW processors can be exploited for non-VLIW processors that support predicated execution. In this subsection, we briefly explain how this can be done. Recall from Section 3.3 that the proposed algorithm consists of three steps: timing analysis, block selection, block merging. Block selection does not depend on the architecture, because it mainly depends on the CFG of the program which is independent from architecture. Therefore, we do not need to change the block selection step when we target non-VLIW processors. To exploit the proposed algorithm for non-VLIW processors the other two steps must be modified as follows:

— *Timing analysis*: The execution time of programs highly depends on the processor architecture and to obtain a safe and tight upper bound for WCET the architecture

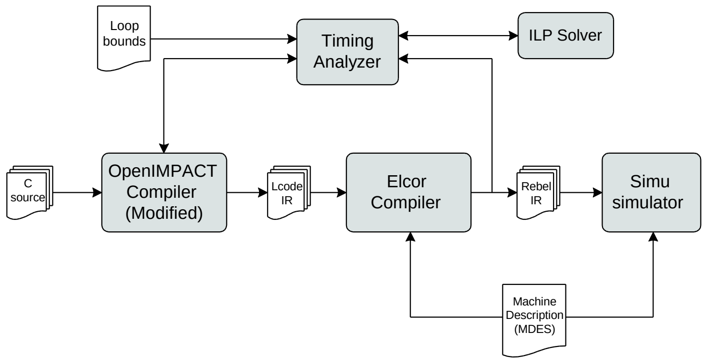


Fig. 9: The structure of modified Trimaran tool chain

must be considered in the timing analysis. The current timing analyzer that we use is for VLIW processors. However, fortunately effective timing analyzers have also been developed for non-VLIW processors (e.g., aiT [AbsInt 2015] and Chronos [Li et al. 2007]) that can be used in the proposed method. It is noteworthy that as our main contribution in this paper is not about timing analysis, here we do not address the details and refer the reader to works on this topic such as [Wilhelm et al. 2008; Li et al. 2006].

— *Block merging*: As we mentioned previously this phase is consisted of two steps: predicated execution and tail duplication. The tail duplication step (Section x) is independent of the processor architecture and hence it is not necessary to change this step for other processor architectures. However, the predicated execution step depends on the processor architecture and hence it must be modified. It is noteworthy that sophisticated compilers such as GCC [Stallman 2015] and LLVM [Lattner and Adve 2004] provides facilities which can be used to implement predicated execution.

## 4. EXPERIMENTAL SETUP

To evaluate the proposed algorithm we used Trimaran compiler/simulator infrastructure [Chakrapani et al. 2005]. Trimaran consists of three tools, front-end compiler (OpenIMPACT), back-end compiler (Elcor) and a cycle accurate simulator (Simu). The proposed algorithm was integrated into the compiler OpenIMPACT. To obtain WCET and worst-case paths, as discussed in Section 3.5, we implemented a timing analyzer based on IPET and integrated it into Trimaran. The timing analyzer is used in two different parts of our approach: i) when executing the block formation algorithm and ii) after back-end compilation in order to calculate the final WCET. Fig. 9 shows the structure of the modified Trimaran tool chain and Table I shows the parameters of the target VLIW processor and the proposed algorithm. It is noteworthy that these parameter values are used for all the experiments except for the experiment where we examine different issue-widths. For this experiment, we must change the number of functional units relative to the issue-width.

We used several benchmarks from Mälardalen [Gustafsson et al. 2010] and MiBench [Guthaus et al. 2001] benchmark suits which have been widely used in previous studies on embedded real-time systems such as [Whitham and Audsley 2010; Huang et al. 2014]. Table II shows the benchmarks used in the experiments. It should be noted that in this paper we consider real-time systems that require time-predictable behavior [Wilhelm et al. 2008]. However, some of the benchmarks in MiBench do not provide

Table I: Parameters of the target processor and the proposed algorithm

| | | |
|---|---|---|
| Target processor | Issue Width | 4 |
| | Integer Functional Units | 1 |
| | Floating Point Functional Units | 1 |
| | Load/Store Units | 1 |
| | Branch Units | 1 |
| | Register File Size | 32 |
| Proposed algorithm | Code size Limitation ($CS\_Limit$) | 40% |
| | Threshold value for classifying Conditional jumps ($JC\_Thr$) | 10 |

Table II: Benchmarks

| Name | LOC* | Description | Source |
|---|---|---|---|
| basicmath | 456 | Simple mathematical calculations | MiBench |
| qsort | 55 | Quick sort algorithm | |
| susan | 2122 | Recognizing corners and edges of an image | |
| blowfish | 1507 | A cryptographic cipher | |
| stringsearch | 108 | String search | |
| fir | 276 | Finite impulse response filter (signal processing algorithms) over a 700 items long sample | Mälardalen |
| qurt | 166 | Root computation of quadratic equations | |
| minver | 201 | Inversion of floating point matrix | |
| janne_complex | 64 | Nested loop program | |
| compress | 508 | Data compression program | |
| lcdnum | 64 | Read ten values, output half to LCD | |
| crc | 128 | Cyclic redundancy check computation on 40 bytes of data | |
| ludcmp | 147 | LU decomposition algorithm | |
| prime | 47 | Calculates whether numbers are prime | |

*Lines Of Code

the required time-predictable behavior. Therefore, for some benchmarks (e.g., susan) we replaced dynamic memory allocations with static ones as dynamic memory allocation is a known source of unpredictable timing behavior. It should be noted that to provide more comprehensive evaluation of the proposed algorithm we experimented benchmarks with various sizes (from 47 to 2122 lines of code). Nevertheless, the experimental results (Section 5.1) show that the proposed algorithm is able to reduce the WCET of both small (e.g. qurt, crc) and large (e.g. compress, basicmath) benchmarks.

To compare our proposed algorithm with other block formation algorithms proposed in the previous related works, we also experimented on the following algorithms:

— The superblock based WCET reduction algorithm, which is based on WCET-aware superblock formation algorithms proposed in [Lokuciejewski et al. 2010] and [Zhao et al. 2005a]. In this algorithm, similar to the proposed algorithm in each iteration, at first the worst-case path is determined. Then the blocks on the worst-case path are selected to form superblocks if it reduces WCET. This algorithm finishes when no new superblock can be formed or the code size limit is reached. It should be noted that unlike our proposed algorithm, the superblock formation algorithm does not take advantage of predicated execution.

— The full hyperblock formation algorithm, which is proposed in [Yan and Zhang 2008]. In this algorithm, all of the execution paths are merged together by predicated execution. Unlike our proposed algorithm, in this algorithm merging the execution paths is conducted without considering and determining worst-case paths.
— Trimaran's hyperblock formation algorithm that is aimed at reducing ACET. In this algorithm, at first, the most frequently executed path are determined by profiling and then by a heuristic approach the path blocks are merged together to form hyperblocks [Mahlke et al. 1992]. It should be noted that unlike our proposed algorithm, this algorithm is focused on ACET and not WCET.

## 5. RESULTS AND DISCUSSION

In this section, we present the impact of different block formation algorithms on the WCET, the ACET and the code size of the benchmarks. Moreover, we report the compile-time overhead of the proposed algorithm and the algorithms from the previous works.

### 5.1. WCET

Fig. 10 shows the WCET of the benchmarks obtained from applying different block formation algorithms. The WCETs for each benchmark are normalized to the WCET of its basic block scheduling (compiling without block formation). The reported results in Fig. 10 show that the proposed algorithm in most of the cases reduces WCET and in the others, does not have negative impact on WCET. On average, the proposed algorithm reduces WCET by 19%, which is a significant improvement over the other algorithms (8% compared to full hyperblock formation, 12% compared to Trimaran's hyperblock formation and 10% compared to superblock formation). In the best case, the proposed algorithm reduces the WCET of qurt program by 59%. This program consists of a loop with an unbiased conditional jump. The proposed algorithm removes this jump instruction from the body of this loop and hence increases the effectiveness of software pipelining (Section 3.2) that results in considerable WCET reduction. The full hyperblock formation algorithm also does the same for qurt program and hence achieves the same result. However, in the programs for which the predicated execution of the conditional jumps does not provide considerable optimization opportunity (e.g. *blowfish* and *basicmath*) the full hyperblock formation algorithm increases WCET. This is because, the full hyperblock formation algorithm converts all conditional jump instructions to predicated counterpart without considering its impact on the WCET. However, in such cases, the proposed algorithm achieves better results than the full hyperblock formation algorithm, because the proposed algorithm considers the impact of the predicated execution on the WCET and converts a conditional jump to the predicated counterpart if it is beneficial. It is noteworthy that for susan and lcdnum programs, the full hyperblock formation algorithm is slightly (3% and 1% respectively) better than the proposed algorithm. This is because in these benchmarks there are some biased jumps that converting them to predicated ones can give more WCET optimization, however, the proposed algorithm does not convert them to predicated ones (indeed this depends on the value of the parameter $JC\_Thr$ explained in Section 3.4) and misses this optimization. Nevertheless, it must be emphasized that on average (as shown in Fig. 10) the proposed algorithm is superior to the full hyperblock formation algorithm.

Compared to the superblock formation algorithm, the proposed algorithm achieves better or at least the same results in all of the programs. This is mainly because the superblock formation algorithm is not able to reduce the WCET of unbiased conditional

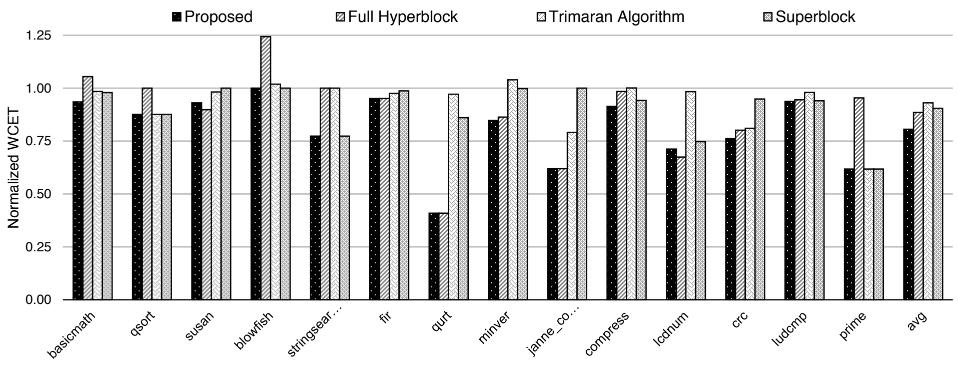


Fig. 10: Worst-Case Execution Time.

jumps as efficiently as the proposed algorithm. However, the proposed algorithm, can reduce the WCET of these jumps by predicated execution.

Compared to the Trimaran's hyperblock formation algorithm, the proposed algorithm also achieves better or at least the same results in all of the programs. This is because the Trimaran's algorithm can significantly reduce the WCET only in the cases that the worst-case path is mostly overlapped with the frequently executed path (e.g. prime). However, in the cases where there is little or no overlap between the worst-case and the frequently executed paths (e.g. minver), using the Trimaran's algorithm can even increase WCET. However, the proposed algorithm is aware of the worst-case paths and optimizes these paths instead of the frequently executed paths and hence achieves better WCET reduction.

Fig. 11 shows how WCET changes as the issue-width of the VLIW processor varies. The results are normalized to the WCET obtained from the basic block scheduling (i.e. no-block formation) when used for the single-issue case. The figure shows that the proposed algorithm reduces the WCET even if we reduce the issue-width of the processor (and even for the single-issue case). This is mainly because many jump instructions are removed from the worst-case paths of the programs by the proposed hyperblock formation. This in turn, eliminates pipeline stalls and hence reduces the (worst-case) execution time. For the processors with higher issue-width, the proposed hyperblock formation can achieve even more WCET reduction. This is because hyperblock formation increases the instruction-level parallelism which can be exploited for parallel instruction execution when the issue-width of the processor increases. Fig. 11 also shows that in all experimented issue-widths the proposed algorithm is superior to the other algorithms.

To explore how the optimization ability of the compiler influences the proposed and the previous algorithms, we examine the algorithms in two cases. In one case, ordinary performance optimization options of the compiler (e.g., software pipelining, dead block elimination, constant propagation, etc.) are turned off and in the other case, these optimizations are all turned on. Fig. 12 shows the averaged results of the proposed and the previous algorithms in these two cases. The results are normalized to the basic block scheduling (i.e., when there is no block formation) in the case the optimizations are turned off. The figure shows that by turning the optimizations on, all of the algorithms can achieve more WCET reduction (15% to 19%). Moreover, the figure shows that in both of the cases the proposed algorithm achieves better results than the previous algorithms and the quality of compiling does not have considerable impact on the relative effectiveness of the proposed algorithm. It is noteworthy that the safety of

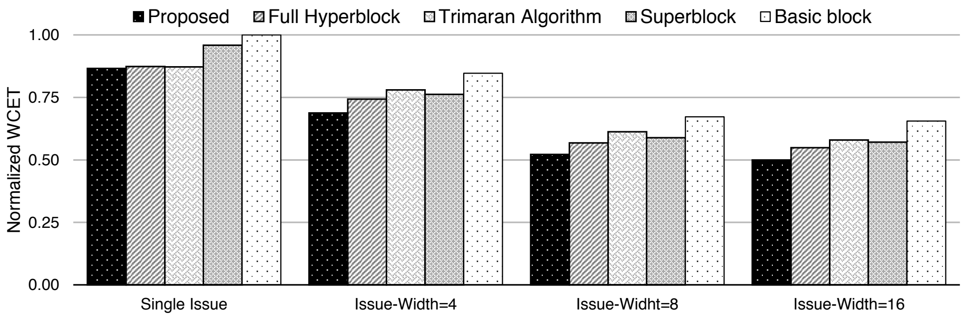


Fig. 11: Impact of changing issue-width.

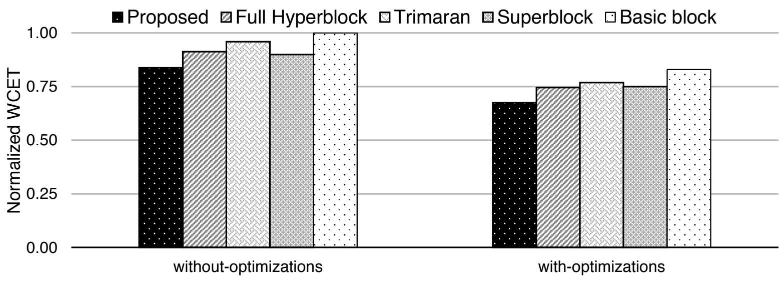


Fig. 12: Impact of compiler optimizations.

the obtained WCETs for the proposed and the other methods are equal since we used same method to calculate WCET (Section 3.5).

## 5.2. ACET

Fig. 13 shows the ACET of the benchmark programs normalized to the ACET of basic block scheduling. The figure shows that while the full hyperblock formation algorithm increases ACET by 83% on average, the proposed algorithm has less negative impact on ACET (35% on average). It is noteworthy that this overhead arises from the lack of resources (in particular issue-width) to simultaneously execute the multiple paths of the program.

The ACET results of the proposed method is almost equal to the results of Trimaran's algorithm in the programs where the worst-case path mostly overlaps with the frequently executed path (e.g. prime). It should be noted that since the Trimaran's algorithm does not predict the adverse effects of hyperblock formation on ACET, in some benchmarks (e.g. susan, lcdnum, minver) it can even increase ACET.

## 5.3. Code Size

Fig. 14 shows the code size of the benchmarks obtained from applying different block formation algorithms. The numbers are normalized to the code size of the benchmarks in the case of basic block scheduling. The results show that the superblock formation algorithm has the most code size overhead (about 5% on average). This is because in the superblock formation algorithm, usually more tail duplication is required. Indeed, since a superblock can include only one path of the program, the number of its side entrances increases and hence it needs more tail duplications. In contrast to the su-

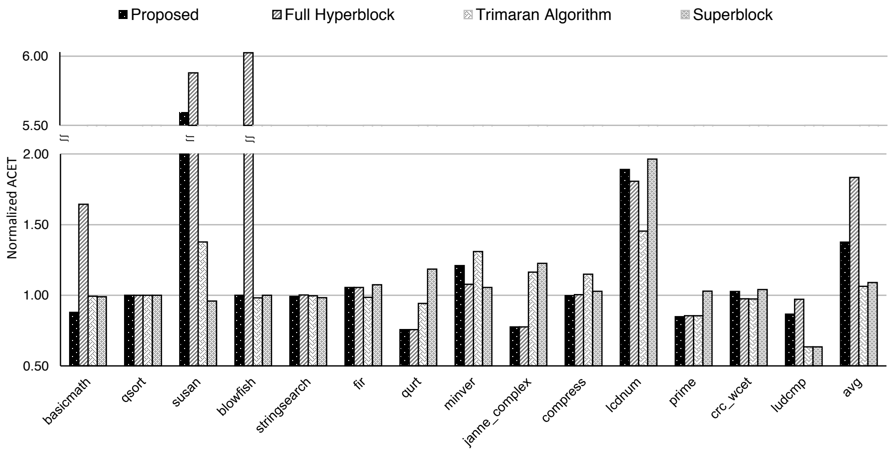


Fig. 13: Average-Case Execution Time.

perblock formation algorithm, the proposed algorithm because of using hyperblocks, can easily include two branch paths of a conditional jump in a hyperblock and obviates the need for tail duplication. Therefore, the proposed algorithm has considerably less code size overhead (less than 1% on average) than the superblock formation algorithm. However, in the programs where the worst-case path has several biased conditional jumps (e.g. stringsearch) the proposed algorithm, as discussed in Section 3.4, includes only one branch path and as a result the number of side entrances to the hyperblocks increases which consequently leads to more tail duplication and code-size growth.

As the Fig. 14 shows the full hyperblock formation algorithm in many cases not only does not increase the code size but also decreases it. This is because aggressive hyperblock formation increases the opportunity of the instruction merging optimization [Mahlke et al. 1992] that combines two instructions with complementary predicates into a single instruction. Moreover, the full hyperblock formation algorithm merges all execution paths of the program into a single path and hence there is no need to use tail duplication in this algorithm. However, as we mentioned previously in Section 5.1 aggressive merging all execution paths can even increase WCET.

## 5.4. Compilation Run Time

The compilation run time of the proposed and the other block formation algorithms were measured on an Intel® Xeon® system (E5-2650, 2.00GHz, 3GB RAM). Table III shows the measured compilation run times of the algorithms. The table shows that the proposed algorithm in some programs that have many conditional jumps suffers from a large compilation run time (e.g. in the worst-case, about 851 minutes for the susan program). This is because in each iteration of the proposed algorithm we need to provoke the timing analyzer at least one time (Section 3.3) and the timing analyzer uses linear programming which is known for being very time-consuming, especially when the input size (in our case, the number of CFG blocks) is large. Nevertheless, as mentioned previously in Section 3.4, since this time-consuming compilation is performed offline at design time, we believe it is beneficial to consume more time at design time to achieve a better system runtime behavior (more WCET reduction in our case). Some methods can be used to reduce this compilation time, such as restricting the number of the iterations of the algorithm, or setting $JC\_Thr$ value to a small number to reduce the number of conditional jumps that are considered as unbiased (Section 3.4). How-

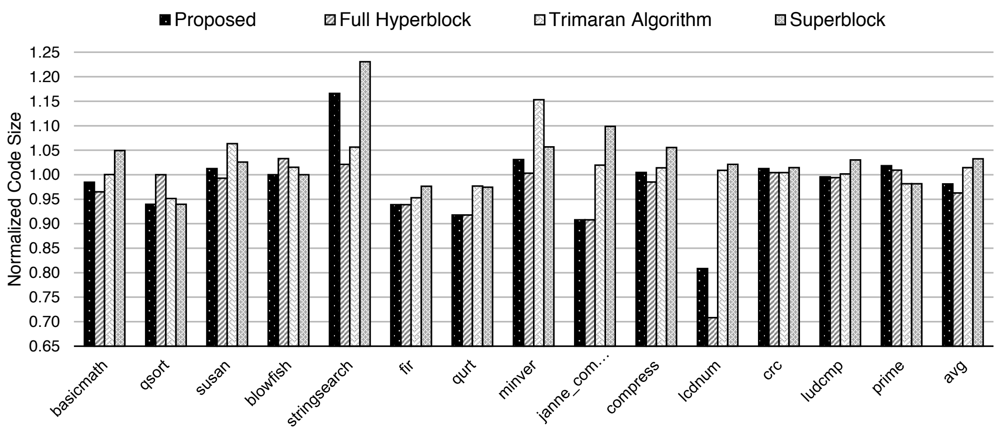


Fig. 14: Code size.

Table III: Compilation Run Time

| Benchmark | Proposed | Full Hyperblock | Trimaran | Superblcok | Basic block |
|---|---|---|---|---|---|
| **basicmath** | 7m24s | 0m24s | 0m24s | 1m84s | 0m24s |
| **qsort** | 1m41s | 0m14s | 0m14s | 1m11s | 0m14s |
| **susan** | 851m8s | 6m39s | 7m14s | 82m54s | 6m10s |
| **blowfish** | 96m28s | 2m57s | 2m48s | 17m21s | 2m35s |
| **stringsearch** | 2m15s | 0m17s | 0m17s | 0m43s | 0m17s |
| **fir** | 1m1s | 0m13s | 0m13s | 0m23s | 0m13s |
| **qurt** | 2m17s | 0m15s | 0m15s | 0m39s | 0m15s |
| **minver** | 6m30s | 0m19s | 0m20s | 2m1s | 0m20s |
| **janne_complex** | 1m28s | 0m13s | 0m13s | 0m41s | 0m13s |
| **compress** | 12m26s | 0m31s | 0m32s | 3m51s | 0m31s |
| **lcdnum** | 2m24s | 0m13s | 0m14s | 0m49s | 0m14s |
| **crc** | 2m33s | 0m15s | 0m15s | 1m12s | 0m15s |
| **ludcmp** | 4m6s | 0m16s | 0m16s | 1m17s | 0m16s |
| **prime** | 0m52s | 0m13s | 0m13s | 0m29s | 0m13s |

ever, the use of these methods comes at the cost of missing some achievable WCET reduction.

## 6. CONCLUSION

In this article we proposed a novel block formation method which is more effective than previous approaches that have addressed the same problem. Indeed, the previous approaches lay in three different categories:

— The works that use superblocks[Lokuciejewski et al. 2010; Zhao et al. 2005a]. which means that they do not consider issues like predicated execution, merging of execution paths, considering the issue-widths of processor, unbiased and biased jumps, etc., while we have considered hyperblock formation and hence addressed all the mentioned issues,

— The work that uses full hyperblock formation [Yan and Zhang 2008]. This work does not involve issues like unbiased and biased jumps, the use of timing analyzer, tail duplication, issue-widths of processor, finding a path among multiple path branches, etc., while in the proposed hyperblock formation algorithm we have addressed all these issues, so that the proposed algorithm achieves more WCET reduction (the work in [Yan and Zhang 2008] might even increase WCET).
— The work that uses hyperblock formation to reduce ACET [Mahlke et al. 1992]. Evidently WCET is not the main concern for this work and hence it is not suitable for real-time applications. This work also does not consider issues like unbiased and biased jumps, the use of timing analyzer, WCET, real-time operation, etc., while in the proposed hyperblock formation algorithm we have addressed all these issues.

The experimental results showed that the proposed method can significantly reduce WCET (19% on average) while has considerably less adverse impacts than the other block formation algorithms on ACET and code size. To extend the proposed method, an interesting future work is integrating other compile-time optimizations such as loop unrolling with hyperblock formation aimed at reducing WCET. The other interesting future work is studying the impacts of block formation algorithms on the processors that have memory hierarchy.

## ACKNOWLEDGMENTS

The authors would like to thank Mr. Mohammad Mesbah of Sharif University of Technology for extracting the loop bounds of the benchmarks.

## REFERENCES

AbsInt 2015. http://www.absint.com/ait/. http://www.absint.com/ait/. (2015). Accessed: 2015-05-29.

Hakan Aydin, Rami Melhem, Daniel Mosse, and Pedro Mejia-Alvarez. 2004. Power-aware scheduling for periodic real-time tasks. *Computers, IEEE Transactions on* 53, 5 (May 2004), 584–600. DOI:http://dx.doi.org/10.1109/TC.2004.1275298

LakshmiN. Chakrapani, John Gyllenhaal, Wen-meiW. Hwu, ScottA. Mahlke, KrishnaV. Palem, and RodricM. Rabbah. 2005. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. In *Languages and Compilers for High Performance Computing*, Rudolf Eigenmann, Zhiyuan Li, and SamuelP. Midkiff (Eds.). Lecture Notes in Computer Science, Vol. 3602. Springer Berlin Heidelberg, 32–41. DOI:http://dx.doi.org/10.1007/11532378_4

Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. 1991. Using profile information to assist classic code optimizations. *Software: Practice and Experience* 21, 12 (1991), 1301–1321. DOI:http://dx.doi.org/10.1002/spe.4380211204

Keith Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.

ILOG Cplex. 2007. *11.0 Users manual*. Report.

Jakob Engblom and Andreas Ermedahl. 1999. Pipeline timing analysis using a trace-driven simulator. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*. IEEE, 88–95. DOI:http://dx.doi.org/10.1109/RTCSA.1999.811197

Heiok Falk. 2009. WCET-aware register allocation based on graph coloring. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. IEEE, 726–731.

Heiko Falk and Jan C. Kleinsorge. 2009. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*. ACM, New York, NY, USA, 732–737. DOI:http://dx.doi.org/10.1145/1629911.1630101

Heiok Falk, Paul Lokuciejewski, and Henrik Theiling. 2006. Design of a WCET-Aware C Compiler. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTMED '06)*. IEEE Computer Society, Washington, DC, USA, 121–126. DOI:http://dx.doi.org/10.1109/ESTMED.2006.321284

Heiok Falk, Norman Schmitz, and Florian Schmoll. 2011. WCET-aware Register Allocation Based on Integer-Linear Programming. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*. IEEE, 13–22. DOI:http://dx.doi.org/10.1109/ECRTS.2011.10

Heiko Falk and Martin Schwarzer. 2006. Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In *Embedded Systems for Real Time Multimedia, Proceedings of the 2006 IEEE/ACM/IFIP Workshop on*. IEEE, 115–120.

Joseph A. Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* C-30, 7 (1981), 478–490.

Joseph A. Fisher, Paolo Faraboschi, and Clifford Young. 2005. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier.

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Bjrn Lisper. 2010. The Mlardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Björn Lisper (Ed.), Vol. 15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 136–146. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, Austin, TX, USA, 3–14.

Jörg Henkel and Sri Parameswaran. 2007. *Designing Embedded Processors: A Low Power Perspective*. Springer Publishing Company, Incorporated, New York.

Yazhi Huang, Liang Shi, Jianhua Li, Qingan Li, and C.J. Xue. 2014. WCET-Aware Re-Scheduling Register Allocation for Real-Time Embedded Systems With Clustered VLIW Architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 22, 1 (Jan 2014), 168–180. DOI:http://dx.doi.org/10.1109/TVLSI.2012.2236114

Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. 75–86. DOI:http://dx.doi.org/10.1109/CGO.2004.1281665

Rainer Leupers. 1999. Exploiting Conditional Instructions in Code Generation for Embedded VLIW Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '99)*. ACM, New York, NY, USA, Article 23. DOI:http://dx.doi.org/10.1145/307418.307462

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. 2007. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming* 69, 1-3 (2007), 56–67. http://www.comp.nus.edu.sg/ rpembed/chronos.

Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006), 195–227. DOI:http://dx.doi.org/10.1007/s11241-006-9205-5

Yau-Tsun Steven Li and Sharad Malik. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *SIGPLAN Not.* 30, 11 (Nov. 1995), 88–98. DOI:http://dx.doi.org/10.1145/216633.216666

Paul Lokuciejewski, Heiko Falk, and Peter Marwedel. 2008a. WCET-driven Cache-based Procedure Positioning Optimizations. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*. IEEE, 321–330. DOI:http://dx.doi.org/10.1109/ECRTS.2008.20

Paul Lokuciejewski, Heiko Falk, Peter Marwedel, and Henrik Theiling. 2008b. WCET-driven, Code-size Critical Procedure Cloning. In *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems (SCOPES '08)*. ACM, New York, NY, USA, 21–30. http://dl.acm.org/citation.cfm?id=1361096.1361100

Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. 2009. Automatic WCET reduction by machine learning based heuristics for function inlining. In *3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*. 1–15.

Paul Lokuciejewski, Timon Kelter, and Peter Marwedel. 2010. Superblock-Based Source Code Optimizations for WCET Reduction. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 1918–1925. DOI:http://dx.doi.org/10.1109/CIT.2010.327

Paul Lokuciejewski and Peter Marwedel. 2009. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*. 35–44. DOI:http://dx.doi.org/10.1109/ECRTS.2009.9

Paul Lokuciejewski and Peter Marwedel. 2010. *Worst-case execution time aware compilation techniques for real-time systems*. Springer Science & Business Media.

Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. *SIGMICRO Newsl.* 23 (1992), 45–54.

Dhiraj K Pradhan. 1996. *Fault-tolerant computer system design*. Prentice-Hall.

Peter Puschner. 2002. Is worst-case execution-time analysis a non-problem?-Towards new software and hardware architecture. In *Proc. 2nd Euromicro International Workshop on WCET Analysis*.

Martin Schoberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, Andr Rocha, Cludio Silva, Jens Spars, and Alessandro Tocchi. 2015. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture* 0 (2015).

Martin Schoberl, Peter Puschner, and Raimund Kirner. 2009. Single-path programming on a chip-multiprocessor system. In *In: Procs of the Workshop on Reconciling Performance with Predictability (RePP)*.

Martin Schoberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W Probst, Sven Karlsson, Tommy Thorn, and others. 2011. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, Vol. 18. 11–21.

David Seal. 2000. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc.

Richard M. Stallman. 2015. *Using the GNU Compiler Collection*. Free Software Foundation. https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc.pdf

Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. 10 pp.–232. DOI:http://dx.doi.org/10.1109/RTSS.2005.45

Jack Whitham and Neil C. Audsley. 2010. Time-Predictable Out-of-Order Execution for Hard Real-Time Systems. *Computers, IEEE Transactions on* 59, 9 (Sept 2010), 1210–1223. DOI:http://dx.doi.org/10.1109/TC.2010.109

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3 (2008), 36:1–36:53.

Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal Instruction Scheduling Using Integer Programming. *SIGPLAN Not.* 35, 5 (2000), 121–133.

Hui Wu, Jingling Xue, and Sri Parameswaran. 2010. Optimal WCET-aware Code Selection for Scratchpad Memory. In *Proceedings of the Tenth ACM International Conference on Embedded Software*. ACM, 59–68.

Jun Yan and Wei Zhang. 2008. A time-predictable VLIW processor and its compiler support. *Real-Time Systems* 38, 1 (2008), 67–84.

Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. 2005a. Improving WCET by optimizing worst-case paths. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*. 138–147.

Wankang Zhao, David Whalley, Christopher Healy, and Frank Mueller. 2005b. Improving WCET by Applying a WC Code-positioning Optimization. *ACM Trans. Archit. Code Optim.* 2, 4 (2005), 335–365.