An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture

by

Aayush Prakash

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Aayush Prakash

Abstract

This work presents a static instruction allocation scheme for the precision timed architectures (PRET) scratchpad memory. Since PRET provides timing instructions to control the temporal execution of programs, the objective of the allocation scheme is to ensure that the explicitly specified temporal requirements are met. Furthermore, this allocation incorporates instructions from multiple hardware threads of the PRET architecture. We formulate the allocation as an integer-linear programming problem, and we implement a tool that takes binaries, constructs a control-flow graph, performs the allocation, rewrites the binary with the new allocation, and generates an output binary for the PRET architecture. We carry out experiments on a modified version of the Malardalen benchmarks to illustrate that commonly known ACET and WCET based approaches cannot be directly applied to meet explicit timing requirements. We also show the advantage of performing the allocation across multiple threads. We present a real time benchmark controlling an Unmanned Air Vehicle as the case study.

Acknowledgements

I am grateful to Dr. Hiren Patel, who has been more than a supervisor, mentor and friend. This work would not have been possible without his guidance and encouragement. I think the learning experience and support I received throughout my masters, kept me motivated to work and perform. I would also like to thank Dr. Siddharth Garg and Dr. Mahesh Tripunitara for their support and valuable comments as my committee members. I feel incredibly fortunate to have the privilege of interacting with such great people.

I would like to thank my friends in Waterloo for making my stay memorable, enjoyable and learning experience. My thesis would not have been possible without their support.

Lastly but not the least, I would like to thank my parents and my sister for motivating and encouraging me during the tough and distressing times during my masters. I simply would not be the person that I am today without their help and unconditional love. I thank my parents for the valuable education that they have provided me and guidance through every step of my life.

Dedication

This thesis is dedicated to the loving memory of my grandfather who taught me everything in life and loved me the most.

I would also like to dedicate my thesis to my parents, Ajit & Sanju and my sister Akanksha.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The Precision Timed architecture (PRET) is a hard real-time embedded processor architecture [13] that has predictable timing behaviours. PRET achieves predictability by making instruction execution repeatable. This simplifies the complexity of determining worst-case execution time (WCET) estimates of programs executing on PRET. WCET estimates are necessary to guarantee that temporal requirements of time-sensitive applications such as those in avionics, automotive and other safety-critical systems, are always met. PRET also introduces timing instructions that explicitly state timing requirements in the program. These timing instructions allow controlling the temporal behaviours of the program. PRET's memory hierarchy favours a shared scratchpad memory (SPM) for instruction and data. Caches are not used because obtaining tight WCET estimates with caches is complex [6]. SPMs, on the other hand, use software-controlled techniques to move instructions on and off the SPM; thereby, allowing the designer to have control over the transfers on and off the SPM. This makes SPMs an attractive alternative over caches

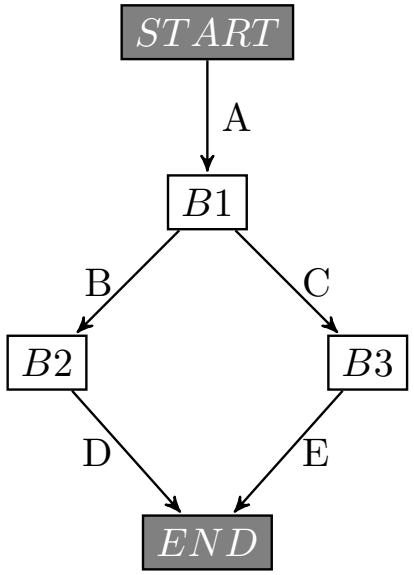
for predictability.

Although SPMs are predictable, manually performing the allocation of instruction and data is tedious, and error prone. Consequently, there are works that automatically allocate instructions and/or data onto the SPM [7, 23]. SPM allocation techniques that are WCET-centric such as that proposed by Deverge and Puaut [7] and Suhendra et al. [23] perform automatic allocation with the objective of reducing the worst-case execution path of the program. These works present innovative allocation techniques, but mainly for reducing the WCET of a single task. Hence, it cannot be directly applied to multi-threaded applications. Suhendra and Roychoudhry [24] address this by performing allocations with the goal of minimizing worst-case response time for concurrent embedded programs. Note, however, PRET programs have explicitly defined timing requirements, which means that reducing the worst-case path may not be sufficient to meet the timing requirements. For example, there may be timing requirements that do not fall on the worst-case path of the program. Then, the objective of minimizing the worst-case path for SPM allocation may entirely neglect these timing requirements.

Current PRET programming practices require entire programs to fit on the scratchpad memory [13]. This limits PRET's practical use since programs are typically larger than the SPM size, and manual allocation is inefficient. This brings us to the focus of our work: a static instruction SPM allocation scheme for the PRET architecture that is aware of timing requirements explicitly specified in the program. In particular, we identify the basic blocks enclosed within PRET's timing instructions (called a *timed block*), and schedule the basic blocks within this timed block such that it *just* meets its timing requirement. By allocating just enough instructions to meet the timing requirements, we conserve space on the shared SPM. This is important because it enables other instructions from other timed blocks in the same program, and from other threads to utilize the space for meeting requirements specified in their timed blocks. Notice that we present a static approach such that the program has the same allocation for its entire execution.

```
1  void VGA_Thread() {
2   B1:
3   img = getImage(graphics buffer);
4   snr = getSNR(Img);
5    if(snr<thresh) {
6     B2:
7     img = medianFilter(img);
8     updateGraphicsBuffer(img);
9    }
10   else {
11    B3:
12    writeImage(VGA pin,img);
13    invokeGraphicsThread();
14   }
15 }
```

(a) C code.



(b) High level CFG.

Figure 1.1: The VGA thread program with its control flow.

We will take an example highlighting the problem we identify with the WCET reduction for single thread. Using the timing instructions as described in table 2.1 we can specify the timing requirements in the program. But when we try to reduce the WCET, it may not be necessary that we meet the deadlines. For instance, 1.1b shows a control flow graph of a VGA Thread. The program (1.1a) reads image from the graphics buffer and checks the SNR of the image. If the image is noisy, it undergoes median filtering (the path (A, B, D)) which is highly computationally expensive. Else the program writes the image to the VGA pin (the path (A, C, E)). The data (image here) has to be written at a particular frequency and time to the VGA pin. Thus the VGA pin has an inherent timing requirement ((A, B, D)) and was found to be WCET path with x cycles. (A, C, E) takes y cycles. There is a timing requirement for the path (A, C, E) of z cycles, such that $z < y$. The path (A, B, D) is allocated to SPM to reduce the WCET, and assuming it consumes the entire SPM space, the path (A, C, E) will not be allocated to SPM and

hence will miss the deadline. On the other hand if (A, C, E) is allocated instead of (A, B, D), then the deadline can be met, given the fact that y becomes lower than z after allocation.

1.2 Summary of Contributions

The main contributions of this work are threefold: 1) a static instruction SPM allocation scheme to meet explicit timing requirements in the program, 2) ensuring that the allocation selects the minimum number of instruction blocks that satisfy the timing requirements, and 3) a tool that automates the analysis and allocation from ARM binaries. We perform our allocation on compiled binaries because it allows the allocation to operate on instructions that are a part of a library, whose source might be unavailable, and to make the allocation scheme compiler agnostic such that any pre-compiled binary following the ARMv4 ISA can be used. We implement a tool that accepts binaries as input, parses them and constructs control-flow graphs (CFG)s for each thread, identifies timed blocks and control-flow subgraphs encompassed within those timed blocks, computes WCET estimates of the timed blocks, and performs the instruction SPM allocation. We formulate the allocation as an integer-linear programming (ILP) problem that allocates the minimum number of instruction blocks necessary to meet the timing requirements of the timed blocks. We perform experiments on a subset of modified Malardalen benchmarks to show the benefits of performing the allocation across multiple threads when using a shared SPM.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 1 presents the motivation behind the work and the contributions. We discuss the background, PRET, timing instructions and related work in chapter 2. Chapter 3 talks about CFG (control flow graph) construction, timing and WCET analysis

of a given binary. We present instruction allocation in chapter 4 and re-writing of the binary after allocation in chapter 5. Results are present in chapter 6 and the UAV case study in chapter 7. We conclude the thesis with chapter 8.

Chapter 2

General Background: Related Work and PRET Architecture

2.1 Related Work

There are two broad areas of research in SPM allocation: reduction of average-case execution time (ACET) [3, 27], and reduction of worst-case execution time (WCET) [7, 23]. General purpose systems use ACET methods, and hard real-time systems typically use WCET methods. Avissar et al. [3] present a static SPM allocation at compile time. They formulate the allocation as an ILP problem, allocating the most profitable instructions and data. Udayakumaran et al. [27], propose a IPET(Implicit Path Enumeration Technique)based dynamic allocation of instructions & data to SPM using compile time decisions. They divide the whole program into regions (with fixed allocation) guarded by program points and associate time stamps at each program points. These program points can be functions, loop entry or exit points, start or end of if/else, switch statements, etc. The level of allocation for instructions are these regions, while for us, its ba-

sic blocks. These time stamps help in collecting data access and instruction frequency during profiling. The time stamps help decide the potential candidates for allocation and eviction. The candidates for allocation are the ones giving higher latency gain and lower transfer cost. But, both [3, 27] cannot be used for real time systems, as opposed to our work which deals with hard real time systems.

The authors in [7, 23] propose data allocation schemes that focus on reducing the WCET of the program They identify a unique problem that is the WCET path may change with a single allocation of data. They first formulate the problem of static allocation of data to SPM as 0-1 ILP problem. Then they iteratively allocate one variable to SPM and perform the WCET analysis to determine the new WCET path in each iteration. Deverge [7], also identify the same problem, and come up with an iterative scheme for dynamic allocation of data to SPM. However, by reducing the execution time of the worst-case path of the program, the allocation does not account for timing requirements that may be embedded via timing instructions. We focus on meeting timing requirements that are explicitly specified in the program. For example, a VGA I/O operation that must occur at specific pre-defined rates may not lie on the worst-case path. Minimizing the worst-case path may neglect this explicit timing requirement, which is the focus of our work. Note that while our approach focuses on the instruction SPM allocation part of the problem, the deficiency with WCET reduction exists for both data and instruction.

Other research efforts by Whitham and Audsley [29] present a hardware based WCET-directed dynamic data and instruction allocation to SPM by introducing a time-predictable scratch-pad memory management unit (SMMU). They point out the problem of pointer aliasing and pointer invalidation when using when supporting dynamic structures. For this, they propose a memory management unit that translates logical addresses to physical addresses. They later present an allocation scheme that uses the SMMU [30]. This is also a WCET-centric allocation that focuses on reducing the execution time on the WCET path.

Recent works have focused on allocation of instructions across multiple threads [15, 24]. Metzlaff et al. [15] present a predictable hardware-based dynamic allocation to instruction SPM for a simultaneous multi-threaded processor. Their proposal is to dynamically allocate the SPM at the granularity of functions, which is similar to method caches by Schoeberl et al. [21]. The loading of the SPM is done at runtime.

Mitra et al. [24] incorporate interference between threads due to dependencies in pre-emptive scheduling, and include this in their allocation scheme as presented in [23]. They propose a metric called worst-case response time of a concurrent application based on [23] and model the application as message sequence chart to capture thread interactions. Their SPM allocation method is similar to their earlier work [23] described eariler.

Schoeberl et al. [22, 10] and Andalam et al. [1] both introduce a PRET like architecture, named Patmos and ARPRET respectively. Schoeberl et al. discuss how compiler transformations, and optimizations can impact the WCET paths where the analysis uses an iterative implicit path enumeration technique approach [10]. Andalam et al. [1] present a concurrent language PRET-C coupled with a microblaze-based platform called ARPRET, to ensure concurrency and predictability in programs. PRET-C uses an EOT instruction that is capable of making threads wait for a specified time, and can achieve some of the functionalities like repeatability and mutual exclusion. While EOT provides an implicit notion of time, it is different than PRET [13] in that PRET requires explicit timing instructions. As of now, the authors have not concentrated on the SPM allocation problem. In summary, most existing allocation approaches are WCET-centric, which does not directly apply to the PRET architecture with explicit timing requirements. Our own previous efforts used profiling for SPM allocation for PRET [19]. The work in this paper extends this work by performing static analysis, and an allocation; thereby, completing the entire automation tool as well.

2.2 PRET Architecture and the timing instructions

The precision timed architecture (PRET) is a hard real-time embedded processor that has predictable and repeatable temporal behaviors [13]. This is a multi-processing architecture with a thread-interleaved pipeline that supports four hardware threads [5]. PRET also proposes instruction-set architecture (ISA) extensions to the ARMv4 ISA, which allow the user to specify temporal requirements in the form of timing instructions [5] to control the temporal behaviour of the program. We present these timing instructions in Table 2.1 [5].

Instruction	Semantics
set_time %r, offset	Load the currentTime+ offset into register %r
delay_until %r	Stall pipeline until currentTime >= [%r].
branch_expired %r, target	Conditionally branch to the target if the currentTime > [%r].
exception_on_expire %r, id	Processor throws an exception with id when currentTime > [%r].
deactivate_exception id	Disable exception handler for exception id.

Table 2.1: ISA extensions with timing instructions for PRET.

PRET puts forward a memory hierarchy that shares one SPM between the four threads for both instructions and data. However, the off-chip main memory assigns a particular DRAM bank to a hardware thread [20]. For more information about the PRET memory hierarchy, we forward the readers to [20].

Figure 2.1 describes the memory map of the PRET architecture. The address spaces 0x00000000-0x00000020, 0x00000020-0x00004000 and 0x00004000-0x00004100 are reserved for exception vector table, supervisor mode stack and IRQ mode stack respectively. The address space 0x40000000-0x40400000 corresponds to four hardware threads. The address space 0x40400000-0x40410000 represents the shared SPM among all the threads.

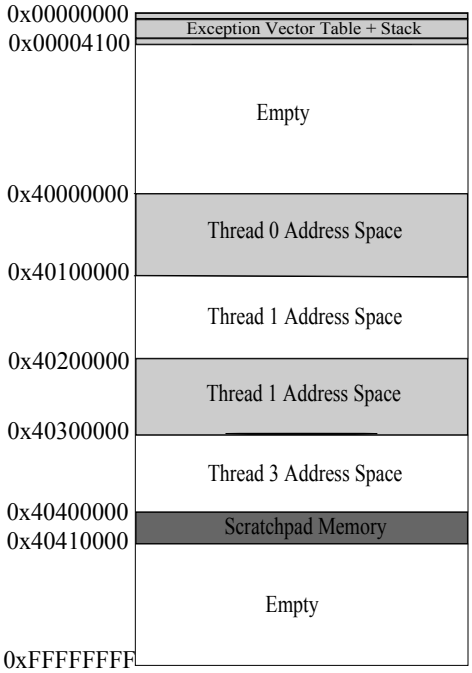


Figure 2.1: The Memory Map of PRET

2.2.1 Timed blocks

We use the timing instructions from Table 2.1 to define *timed blocks*.

A timed block encloses a sequence of instructions that have a specific temporal requirement. Figure 2.2 shows code fragments that use macros that synthesize to timing instructions from Table 2.1, and define timed blocks. For example, the $v1$ timed block (line 8–10). This timed block specifies that the enclosed code must always take 100ns. Notice that there are three variants $v1$, $v2$ and $v3$, each with unique semantics. Variant $v1$ specifies that the enclosed code must always take at least the specified amount of time. If the program runs faster, then it is padded with no-operations until the specified timing requirement. There is no exception if the execution exceeds the timing requirement. For variant $v2$, the semantics are the same as $v1$ except that violations of the timing requirements cause a branch to the $patchup()$ function at the end of the

timed block. Variant $v3$ provides immediate miss detection, which branches to the $patchup()$ function as soon as the timing requirements are violated.

```
1  int main() {
2    int i,j;
3    char inbuf[M][N];
4    char outbuf[M][N];
5    for(i=0;i<M,i++){
6     for(j=0;j<N;j++){
7       //v1: r1,100ns
8       SET_TIME(1,100);
9       inbuf[i][j]=inport;
10      DELAY_UNTIL(1);
11    }
12   }
13   //v2: r1,100000ns
14   SET_TIME(1,100000);
15   outbuf=filter(inbuf);
16   BRANCH_EXPIRE(1,patchup());
17   DELAY_UNTIL(1);
18   display(outbuf);
19   return 0;
20 }
```

(a) Variants 1 and 2 of timed block.

```
1 void display(char outbuf[][]) {
2   //v3:r1,100ns
3   SET_TIME(1,100);
4   EXCEPTION_EXPIRE(1,2);
5    writeToVGA(outbuf);
6   DEACTIVATE_EXCEPTION(2);
7   DELAY_UNTIL(1);
8 }
```

(b) Variant 3 of timed block.

Figure 2.2: Variants of timed blocks.

Chapter 3

CFG Construction, timing and WCET Analysis

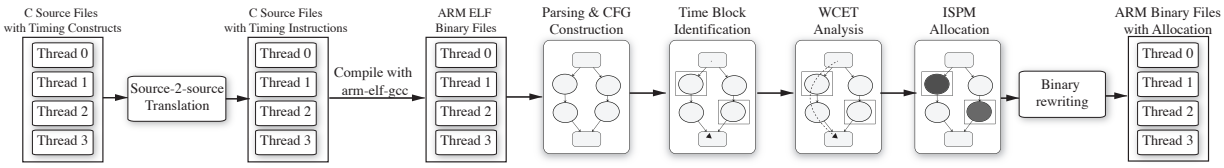


Figure 3.1: Proposed tool flow for instruction SPM allocation for PRET.

We prototype a tool that performs instruction SPM allocation for the multi-threaded PRET architecture [13]. As indicated by Lickly et al. [13], these threads can be a part of a single application or they can be executing different applications. Figure 3.1 shows the main stages of the SPM allocation tool. The first stage performs a source to source translation of the original source program with the timing constructs to its representative source program with timing instructions. This is purely a source translation where we embed the appropriate timing instructions in place of the timing constructs. We use the GNU ARM GCC compiler to compile each of the threads

into its respective binary files. Notice that no SPM allocation is done in this stage. After having generated the binary, the second stage parses the binary, and constructs a control-flow graph (CFG) for each thread. For this work, we support binaries using the ARMv4 ISA. In the third stage, we use the constructed CFGs to identify basic blocks, and timed blocks formed by the timing instructions in the program. This allows us to perform static WCET analysis on the basic and timed blocks for each CFG in the fourth stage. Notice that the design of the PRET architecture significantly simplifies the WCET analysis [13] because of its predictable and repeatable instruction execution. The fifth stage uses the results of the WCET analysis and provides it to the SPM allocation stage, which determines the blocks to allocate on the shared SPM given the timing constraints imposed by the timing instructions, and the size of the SPM. Upon determining the allocation, we rewrite the binary making the necessary changes to reflect the allocation in stage six. The rewriting requires adding instructions to the binary for preserving the correct program flow semantics. Finally, we output the rewritten binaries that are executable on the PRET architecture.

3.1 Source to Source Translation

```
1 tryfor(100, P_NS) {
2   // ...
3 }
```

(a) tryfor

```
1 tryin(100000, P_NS) {
2   // ...
3 } immediateCatch (...) {
4   patchup();
5 }
```

(b) tryin with patchup()

```
1 tryin(100, P_NS) {
2   // ...
3 } catch (...) {
4   patchup();
5 }
```

(c) tryin

Figure 3.2: Timing constructs for specifying timing requirements.

Timing constructs provide a structured approach to encoding timing requirements in the program. These are essential because they ensure that the description of timing requirements are

wellformed. An example of a malformed timing requirement is one that has a set_time to indicate the start of the timed block, but one that does not have a terminating delay_until. For the three variants of timing instructions described in Section 2.2, we provide three timing constructs as illustrated in Figure 3.2 [19]. We provide C constructs to specify the three variants of the timed block ($v1$, $v2$ and $v3$) shown in figure 2.2. We convert these timing constructs into timing instructions from Table 2.1. Listing 3.2a synthesizes to the first variant, Listing 3.2b converts to the second variant with a patchup() invocation when the timer expires, and Listing 3.2c synthesizes to the third variant supporting immediate exception handling once the timer expires. The example shown in Figure 2.2 with the corresponding timing constructs is shown in Figure 3.3.

```
1  int main() {
2    int i,j;
3    char inbuf[M][N];
4    char outbuf[M][N];
5    for(i=0;i<M;i++){
6     for(j=0;j<N;j++){
7       //v1: r1,100ns
8       tryfor(100){
9        inbuf[i][j]=inport;
10      }
11    }
12   }
13   //v2: r1,100000ns
14   tryin(100000){
15    outbuf=filter(inbuf);
16   }
17   expire{
18    patchup();
19   }
20   display(outbuf);
21   return 0;
22 }
```

(a) Timing constructs for $v1$ and $v2$.

```
1 void display(char outbuf[][]) {
2   //v3:r1,100ns
3   tryin(100){
4    writeToVGA(outbuf);
5   }
6   catch(exception id){
7    patchup();
8   }
9 }
```

(b) Timing constructs for $v3$.

Figure 3.3: Example with timing constructs.

3.2 Stage 2: Parsing and CFG Construction

We parse the binaries, and construct the CFG representation for each thread from its compiled binary. While we discuss the details of the CFG construction algorithm for a single thread, the same applies to all other threads. The key steps involved in this stage are parsing the binary, detecting the basic blocks, discovering the blocks that form timed blocks, identifying the target addresses for branch instructions, and representing this information in a control-flow graph (CFG) data structure. These key steps form the crux of Figure 3.4, which we explain next.

3.2.1 Parsing the binary

Using the GNU ARM GCC toolchain, we generate a binary in the SREC format. This is a textual representation of its equivalent ELF binary. Note that this is a compiled binary, which consists of a sequence of instructions with each instruction being assigned a program counter (pc) by the compiler as described in Definition 1. We use the domain $\mathbb{B} = \{0, 1\}$ to represent binary values. We use a superscript to denote the bit-width of the particular field. For example, $\mathbb{B}^{32}$ indicates a binary string that is 32 bits wide.

Definition 1. *A compiled program is a sequence of program counter (pc) and instruction pairs* $pi = [(pc_1, i_1), (pc_2, i_2), ..., (pc_{|p|}, i_{|p|})]$ *where* $|pi|$ *is the number of static instructions in the program,* $pc_m \in \mathbb{B}^{32}$ *is a 32-bit program counter for instruction* i_m*, and* $i_m \in \mathbb{B}^{32}$ *is a 32-bit encoding of the* m^{th} *instruction. The set of all compiled programs is denoted by* P*.*

An instruction in the compiled program is encoded in binary. This encoding contains the operation to be performed by the instruction, the source and destination registers used by the instruction, and offsets for control-transfer instructions. We decode the binary representation of instructions, and represent the instruction as shown in Definition 2.

Algorithm 1: Control-flow graph construction.

Input: $P = \langle ci_1, ci_2, \ldots, ci_f \rangle$
Output: $G = \langle V, E \rangle$

Let ControlTransfer be the set of opcodes for control-transfer instructions
Let $V \leftarrow \emptyset$ and $E \leftarrow \emptyset$
Let $L_I \leftarrow []$ be an empty sequence of compiled instructions
Let v_p, v_c be vertices
Let $C \leftarrow \{t \in B^{32} : \forall c \in P, t \leftarrow \mathsf{getTargetInstruction}(c)\}$ be the set of target instructions for control-transfer in P
foreach *ci in P* **do**
    if $\{ci\} \cap C \neq \emptyset \wedge \mathsf{isNotEmpty}(L_I)$ **then**
        if $\mathsf{isBlockPresent}(\mathsf{getAddress}(\mathsf{peek}(L_I)), V)$ **then**
            $v_p \leftarrow \mathsf{findBlock}(\mathsf{getAddress}(\mathsf{peek}(L_I)), V)$
            $v_p \leftarrow \mathsf{updateBlock}(v_p, L_I)$
        else
            $v_p \leftarrow \mathsf{createBlock}(L_I)$
        end
        if $\mathsf{isBlockPresent}(\mathsf{getAddress}(ci), V)$ **then**
            $v_c \leftarrow \mathsf{findBlock}(\mathsf{getAddress}(ci), V)$
        else
            $v_c \leftarrow \mathsf{createEBlock}(\mathsf{getAddress}(ci))$
        end
        $E \leftarrow E \cup \{(v_p, v_c)\}$
        $V \leftarrow V \cup \{v_p, v_c\}$
        $L_I \leftarrow []$
    end
    $\mathsf{add}(L_I, ci)$
    if $\{\mathsf{getOpcode}(ci)\} \cap \mathsf{ControlTransfer} \neq \emptyset$ **then**
        if $\mathsf{isBlockPresent}(\mathsf{getAddress}(\mathsf{peek}(L_I)), V)$ **then**
            $v_p \leftarrow \mathsf{findBlock}(\mathsf{getAddress}(\mathsf{peek}(L_I)), V)$
            $v_p \leftarrow \mathsf{updateBlock}(v_p, L_I)$
        else
            $v_p \leftarrow \mathsf{createBlock}(L_I)$
        end
        $V_c \leftarrow \emptyset$
        foreach $ti \in \mathsf{getComputedTargets}(ci, P)$ **do**
            if $\mathsf{isBlockPresent}(\mathsf{getAddress}(ti), V)$ **then**
                $V_c \leftarrow V_c \cup \{\mathsf{findBlock}(\mathsf{getAddress}(ti), V)\}$
            else
                $V_c \leftarrow V_c \cup \{\mathsf{createEBlock}(\mathsf{getAddress}(ti))\}$
            end
        end
        $E \leftarrow E \cup \{(v_p, v_c) : \forall v_c \in V_c\}$
        $V \leftarrow V \cup V_c$
        $L_I \leftarrow []$
    end
end

Figure 3.4: Control-flow graph construction.

Definition 2. *A three operand instruction is a 5-tuple* $i = \langle op, sReg_1, sReg_2, dReg, tOffset \rangle$ *where* $op \in OPCODES$ *defines the set of all operations an an instruction can perform,* $sReg_1$*,* $sReg_2 \in \mathbb{B}^5$ *are source registers,* $dReg \in \mathbb{B}^5$ *is the destination register, and* $tOffset \in \mathbb{B}^{16}$ *is the offset for control-transfer instructions.*

Figure 3.4 takes as input the sequence of instructions of the compiled program, and outputs a control-flow graph (CFG) of the main function. We define the CFG in Definition 3.

Definition 3. *A control-flow graph (CFG) is a directed graph* $G = \langle V, E \rangle$ *where* $V \subseteq Z^+ \times P$ *is the set of basic blocks, and* $E \subseteq V \times V$ *is the set of edges dictating the control-flow of the program. A vertex* $v = \langle k, p' \rangle$ *is a tuple where* k *is a unique identifier for the vertex, and* p' *is the compiled sequence of instructions forming the basic block.*

3.2.2 Identifying basic blocks

A key component in constructing the CFG is identifying basic blocks of the program. A basic block is a part of a program that is a sequence of instructions with a single entry point, and a single exit point. This means that none of the instructions within the basic block are target addresses of control-transfer instructions of any instruction in the program. In addition, the basic block's last instruction is a control-transfer instruction. These basic blocks are the vertices of the CFG uniquely identified via k with instructions contained in the basic block as a sequence of instructions p'. We locate control-transfer instructions in the program, and determine the target address of the control-transfer instruction by as indicated by the ARM reference manual [2].Note that the method of computing the target address depends on the type of control-transfer instructions. We show this in line 5 of Figure 3.4.

Figure 3.4 uses $ci = (pc, i)$, as a compiled instruction i with program counter pc.. The algorithm iterates through the instructions in the compiled program, and constructs vertices based

on two conditions. The first condition is when when ci's pc is a target of some other control-transfer instruction (line 7–2), and the second is when ci is a control-transfer instruction (line 24 – 42) itself. In the first condition, we collect the sequence of instructions that are neither control-transfer or targets of control-transfer instructions in L_I. Lines 8–13 check if a basic block for the instructions in L_I already exists or not. If it does, then that vertex is denoted v_p, but if it does not, then it is created. This is the parent vertex. Then, lines 14–18 performs the same check for ci, and finds the vertex to which it belongs if one exists; otherwise, it creates a new one. Since these two vertices follow one after another, we create an edge between the two and insert the edges, and nodes into the CFG in lines 19–20. The second condition (lines 24–42) addresses the situation when ci is a control-transfer instruction. Similar to the first condition, lines 25–30 find or create the parent vertex. Notice that if a vertex is found then we update the instructions that belong to it using updateBlock(...). Since control-transfer instruction can have multiple targets (consider conditional branches and jump tables), we retrieve the set of possible targets using computeTargets(...). For each one of these possible targets, we either create a vertex, or we retrieve the vertex associated with the target. Later, we insert edges to all these targets in line 39, and add the vertices to the set of vertices V.

We describe how we identify the targets for control-transfer instructions next (line 32). By targets, we mean the address of the children to add and preserve edges from the parent to children (line 39). PRET supports multiple control-transfer instructions that include immediate branches, register-indirect branches (including instructions that alter the program counter using loads), and timing instructions with exception handling.

We support immediate branches, timing instructions with exception handling, and a subset of register-indirect branches. In particular, we focus on the register-indirect branches that implement jump tables. Immediate branches encode the offset in the immediate field $tOffset$ of the instruction. Hence, it is straightforward to compute the target address, which is a function of the

immediate value and the program counter of the instruction itself.

Identifying the target addresses of register-indirect branches, on the other hand, are difficult in general. This is because the target address is contained within a register. Determining the potential target addresses of register-indirect branches in general requires a form of binary-level data-flow analysis [9] such that expected data values of the registers are known through the analysis. Our approach is different in that we perform a partial data-flow analysis to compute the potential target addresses for commonly used register-indirect branches. The most common of these is the use of jump tables that are used to implement switch statements, and procedure calls. Figure 3.5 presents a simple example with a jump table where the `ldrls` instruction conditionally executes only if the `cmp` evaluates to true. Note that the ARMv4 ISA supports conditional execution of all instructions. The instruction `ldrls` loads the address $pc+4+r1\times 4$ into the program counter register only when $r1$ is less than or equal to 2. By inspecting this simple program, one can determine that the possible offsets (value of $r1$) for which the `ldrls` executes are 0, 1 and 2. Hence, the target address can be $pc+4$, $pc+8$ or $pc+12$. Since we do not know the actual value of $r1$, we conservatively construct edges from the end of the basic block with the register-indirect instruction to each of the possible target addresses.

```
0x0000bc48:  cmp r1, #2
0x0000bc4c:  ldrls pc, [pc, r1, lsl #2]
```

Figure 3.5: A simple example of a jump table.

Another common use of register-indirect branches is to return from procedure calls. A call is made from a procedure Cr_k (caller) to procedure Ce (callee), using the branch and link instruction. To return from a procedure Ce to the procedure Cr_k, the link register with Cr_k's address is loaded into the pc. Our partial analysis correctly handles register-indirect branches used in procedure calls. We accomplish this by making a callback list for Ce with all the procedures

$Cr_1, Cr_2, ..., Cr_k$ that invoked Ce at some point of time using static analysis. The targets of register-indirect branches from Ce are the program points in $Cr_1, Cr_2, ..., Cr_k$ that invoke Ce. Control transfers as a result of exceptions in the timing instructions are handled similar to immediate branch instructions. However, we acknowledge that a limitation of our work is that we do not support general register-indirect branches at the moment.

Figure 3.6a shows a binary corresponding to a simple loop program with timing instructions. Figure 3.6b shows how we construct CFG given the binary. We add all the instructions sequentially from lines 2–19 in L_I (using line 23 in Figure 3.4). When we reach line 21, we see its a target of branch instruction on line 37. So we create a basic block $B1$, update it with L_I, and then create an empty block $B2$ using address of instruction on line 21, and add edge from $B1$ to $B2$ (using lines 7-22 in Figure 3.4). We set L_I as empty.

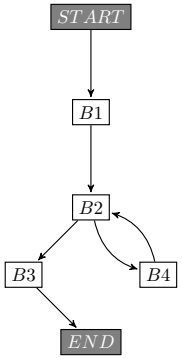
We then add sequence of instructions from lines 21 to 23 in L_I until we encounter a control transfer instruction. We try to find if we created an empty block corresponding to L_I before, and we did as the block is $B2$. We do this search using address of first instruction on L_I, i.e. line 21. We update the sequence of instructions in $B2$. $B2$ has two children, so we create empty blocks $B3$ and $B4$ with addresses on line 39 and 25 respectively and add directional edges from $B2$ to $B3$ and $B4$. We set L_I as empty. Similarly we update the instructions on $B3$ and $B4$, whenever we encounter their control transfer instructions on line 46 and 37 respectively. We also add an edge from $B4$ to $B2$. This is done using lines 24–42 of Figure 3.4.

3.3 Identifying timed blocks

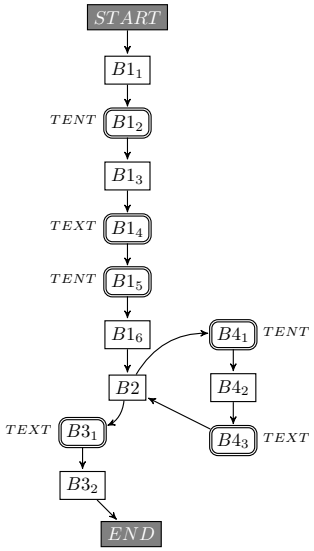
Timed blocks enclose program code between two timing instructions. The first timing instruction is the set_time and the second is typically the delay_until. The program code within these two

```
1 B1:
2 40000024: mov ip, sp
3 40000028: stmdb sp!, {fp, ip, lr, pc}
4 ........
5 40000034: mov r2, #8000
6 40000038: mov r3, #0
7 4000003c: stt r2, r3, r3, 0
8 40000040: mov r1, #3
9 40000044: str r1, [fp, #-16]
10 ........
11 40000060: mov r2, #8000
12 40000064: mov r3, #0
13 40000068: dut r2, r3, r3, 0
14 4000006c: mov r2, #9984
15 40000070: add r2, r2, #16
16 40000074: mov r3, #0
17 40000078: stt r2, r3, r3, 0
18 4000007c: mov r1, #0
19 40000080: str r1, [fp, #-16]
20 B2:
21 40000084: ldr r1, [fp, #-16]
22 40000088: cmp r1, #9
23 4000008c: bgt 400000c4
24 B4:
25 40000090: mov r2, #8000
26 40000094: mov r3, #0
27 40000098: stt r2, r3, r3, 0
28 4000009c: ldr r1, [fp, #-20]
29 400000a0: sub r1, r1, #3
30 400000a4: str r1, [fp, #-20]
31 400000a8: mov r2, #8000
32 400000ac: mov r3, #0  ; 0x0
33 400000b0: dut r2, r3, r3, 0
34 400000b4: ldr r1, [fp, #-16]
35 400000b8: add r1, r1, #1
36 400000bc: str r1, [fp, #-16]
37 400000c0: b 40000084
38 B3:
39 400000c4: mov r2, #9984
40 400000c8: add r2, r2, #16
41 400000cc: mov r3, #0
42 400000d0: dut r2, r3, r3, 0
43 400000d4: mov r1, #0
44 400000d8: mov r0, r1
45 400000dc: sub sp, fp, #12
46 400000e0: ldmia sp, {fp, sp, pc}
```

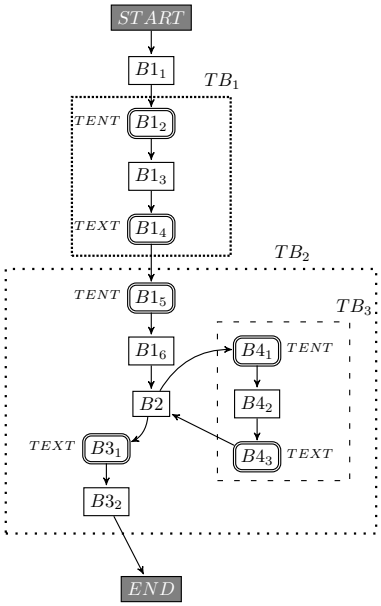
(a) Source binary.



(b) CFG.



(c) TRACFG.



(d) Timed blocks.

Figure 3.6: Timed block identification.

timing instructions form a timed block. For example, Figure 3.7 presents a fragment of the program that is a timed block. Notice that the set_time and delay_until instructions are implemented by PRET using ARM coprocessor instructions, which we annotate on the side with comments.

Notice that `stt` denotes a set_time instruction, and `dut` a delay_until. We identify different timing instructions using their respective opcode. For instance, delay_until in assembly is encoded as `dut`. The timing requirement is specified within the register, which we extract using the partial flow-analysis. In this case, the `mov r2, #8000` provides us with the timing requirement for this timed block.

Definition 4. *A timing-requirement aware CFG (TRACFG) is a CFG* G_{TRA}= $\langle V_{TRA}, E_{TRA} \rangle$ *where* $V \subseteq Z^+ \times V_{TRA} \times P$ *is the set of annotated basic blocks, and* $E \subseteq V_{TRA} \times V_{TRA}$ *is the*

set of control-flow edges. A vertex $v = \langle k, a, p' \rangle$ is a 3-tuple where k is a unique identifier for the vertex, a is an annotated type such that $V_A = \{REG, TENT, TEXT\}$, and p' is the compiled sequence of instructions forming the basic block.

In order to detect timed blocks, we first identify timing instructions in the CFG by transforming the CFG to a timing-requirement aware CFG (TRACFG) as in Definition 4. The TRACFG separates a basic block that has a timing instruction in it into multiple blocks. The TRACFG also annotates the vertices (basic blocks) with a vertex type. REG means the vertex is a regular basic block, $TENT$ identifies the set_time instruction, and $TEND$ annotates the vertex that has the delay_until instruction. We use a helper function type(v) to retrieve the attribute of the vertex.

We construct the TRACFG from the CFG using Algorithm 1. This algorithm identifies basic blocks that have timing instructions within them, and splits them such that the timing instruction forms a vertex in the TRACFG with a specific type. The input to the algorithm is the CFG $G = \langle V, E \rangle$ obtained from Figure 3.4 and the output is a TRACFG $G_{TRA} = \langle V_{TRA}, E_{TRA} \rangle$. E_{in} and E_{out} are the set of incoming and outgoing edges for a certain vertex v (lines 3–4). The algorithm starts by identifying the vertices in V that contain timing instructions using the hasTimingInsn(v) (line 6). Note that there can be multiple timing instructions within the same basic block as shown in the Figure 3.6 ($B1$), and each timing instruction resides in a unique basic block. splitVertex(v) performs the splitting of the vertex v, and produces a sequence of newly

```
40000034: mov r2, #8000
40000038: mov r3, #0
4000003c: stt r2, r3, r3, 0
40000040: mov r1, #3
.........
40000068: dut r2, r3, r3, 0
```

Figure 3.7: Assembly program fragment for timed block.

Algorithm 1: Constructing the timing-aware CFG.

Input: $G = \langle V, E \rangle$ be the CFG.
Output: $G_{TRA} = \langle V_{TRA}, E_{TRA} \rangle$ be the TRACFG

1 Let $V_{TRA} \leftarrow V$ and $E_{TRA} \leftarrow E$.
2 **foreach** $v \in V$ **do**
3 $\quad$ type$(v) \leftarrow REG$
4 $\quad$ Let $E_{in} \leftarrow \{(v_i, v_j) \in E_{TRA} : v_j = v\}$
5 $\quad$ Let $E_{out} \leftarrow \{(v_i, v_j) \in E_{TRA} : v_i = v\}$
6 $\quad$ **if** hasTimingInsn(v) **then**
7 $\quad\quad$ $\langle [v_1, v_2, .., v_l] \rangle \leftarrow$ splitVertex(v)
8 $\quad\quad$ $E'_{in} \leftarrow \{(v_i, v_1) : \forall (v_i, v_j) \in E_{in}\}$
9 $\quad\quad$ $E'_{out} \leftarrow \{(v_l, v_j) : \forall (v_i, v_j) \in E_{out}\}$
10 $\quad\quad$ $V_{TRA} \leftarrow V_{TRA} \cup \{v_1\} .... \cup \{v_l\} - \{v\}$
11 $\quad\quad$ **foreach** v_i *in* $[v_1, .., v_{l-1}]$ **do**
12 $\quad\quad\quad$ $E_{TRA} \leftarrow E_{TRA} \cup \{(v_i, v_{i+1})\}$
13 $\quad\quad$ **end**
14 $\quad\quad$ $E_{TRA} \leftarrow E_{TRA} \cup E'_{in} \cup E'_{out} - E_{in} - E_{out}$
15 $\quad$ **end**
16 **end**
17 **return** $\langle V_{TRA}, E_{TRA} \rangle$

created vertices L_V as the output. It first identifies if the instruction ci is set_time (using function isSetTime(), line 4) or delay_until (using function isDelayUntil(), line 10). If the condition is true, then it first creates a vertex with REG attribute using the sequence of instructions L_I, given its not empty, (lines 5–7, 11–13, 20–22). Then, the algorithm creates a new vertex with either $TEXT$ or $TENT$ attributes for the instruction ci. We then add both the vertices to L_V (lines 8, 14), and build the sequence of instructions L_I by adding the instruction in each iteration (line 17) until we encounter a timing instruction (lines 9, 15). For example, if the vertex has exactly one set_time (or delay_until), it will be split into three vertices, $\{v_1, v_2, v_3\}$. v_1 will contain the instructions before set_time, v_2 will contain the set_time instruction, and v_3 will contain instruction after set_time. Note that v_1 or v_3 may not exist too, as the actual vertex v, can only contain set_time. If v contains multiple set_time or delay_until , it can be split into

Algorithm 2: Function splitVertex().

```
   Input: Vertex v
   Output: L_V = [v_1, v_2, .., v_l]
 1 Let L_V ← [] be an empty sequence of vertices
 2 Let L_I ← [] be an empty sequence of compiled instructions
 3 foreach ci in getCompiledInsnSequence(v) do
 4     if isSetTime(ci) then
 5         if isNotEmpty(L_I) then
 6             add(L_V, ⟨getUniqueId(), REG, L_I⟩)
 7         end
 8         add(L_V, ⟨getUniqueId(), TENT, [ci]⟩)
 9         L_I ← []
10     else if isDelayUntil(ci) then
11         if isNotEmpty(L_I) then
12             add(L_V, ⟨getUniqueId(), REG, L_I⟩)
13         end
14         add(L_V, ⟨getUniqueId(), TEXT, [ci]⟩)
15         L_I ← []
16     else
17         add(L_I, ci)
18     end
19 end
20 if isNotEmpty(L_I) then
21     add(L_V, ⟨getUniqueId(), REG, L_I⟩)
22 end
23 return L_V
```

multiple vertices such that each timing instruction gets a new vertex, with either $TEXT$ or $TENT$ attribute (line 7–14). For example, vertex $B1$ in figure 3.6 is split into six basic blocks $\{B1_1, B1_2, .., B1_6\}$.

After we obtain the sequence of vertices from algorithm 2, we proceed with Algorithm 1. We augment V_{TRA} gets with the new set of basic blocks (vertices) $\{v_1, v_2, .., v_l\}$, and v is removed (line 10). We create $l - 1$ edges from vertex v_1 to v_2, v_2 to v_3, and so on in the sequence $[v_1, v_2, ..., v_l]$, and insert them in E_{TRA} (lines 11–13). We also create appropriate edges between

parents of v, and the first vertex v_1, and also between the last vertex v_l and the children of v (lines 8, 9). We insert these edges in E_{TRA},after removing E_{in} and E_{out} (line 14). Figure 3.6c shows the TRACFG of the CFG in Figure 3.6b. Given the CFG, in figure 3.6b, we traverse its vertices. As mentioned, before, we split vertex $B1$ into six vertices as it has multiple timing instructions. We then add edges among the newly created vertices, that is we create $(B1_1, B1_2), ..., (B1_5, B1_6)$. We also create edge between parent of $B1$, $START$ & $B1_1$ and $B1_6$ & child of $B1$, $B2$. Likewise, we split $B3$ and $B4$ into two ($\{B3_1, B3_2\}$) and three ($\{B4_1, B4_2, B4_3\}$) vertices respectively as they both contain one timing instruction each. We then add edges among the newly created sequential vertices, that is we create $(B1_1, B1_2), ..., (B1_5, B1_6)$. Similarly we create edges $(B2, B4_1), (B4_1, B4_2), (B4_2, B4_3), (B4_3, B2), (B2, B3_1)$, $(B31, B3_2)\, and\, (B3_2, END)$.

Definition 5. *A timed block starting at vertex v_1 and ending at v_2 is a subgraph $G'_{TRA} = \langle V'_{TRA}, E'_{TRA} \rangle$ of a function's TRACFG $G_{TRA} = \langle V_{TRA}, E_{TRA} \rangle$ where $V'_{TRA} \subseteq V_{TRA}$ and $E'_{TRA} \subseteq E_{TRA}$. Vertices $v_1, v_2 \in V'_{TRA} \subseteq V_{TRA}$ such that $v_1 \neq v_2 \wedge \mathsf{type}(v_1) = TENT \wedge \mathsf{type}(v_2) = TEXT \wedge \mathsf{hasPath}(v_1, v_2, G_{TRA})$.*

Algorithm 3 describes our approach to identifying timed blocks. Notice that a timed block is in itself a sub-graph of the TRACFG (Definition 5). However, there can be multiple timed blocks within a single TRACFG. We denote this as a set of timed blocks TB. The algorithm takes the TRACFG G_{TRA} constructed using Algorithm 1, an empty stack S, and root vertex of G_{TRA} (v_{root}) as inputs. The function $\mathsf{timedBlockFinder}(...)$ is a recursive DFS traversal, and it is invoked with the root vertex of the graph v_{root}. During the traversal, we identify the timed blocks by matching the $TEXT$ block to the corresponding $TENT$ block. In each function invocation, we identify whether the current node is timed block entry ($TENT$), and then add it to the stack S (lines 5–6). Upon encountering a timed block exit vertex ($TEXT$), we pop the top node on the S, and create the entry and exit pair (line 7–8). The function $\mathsf{copyGraph}(G_{TRA}, v, v_1)$, copies the sub-graph from G_{TRA}, starting at v_1 and ending at v, which we add into TB (line 9).

Algorithm 3: timedBlockFinder(...): Identifying timed blocks.

Input: $G_{TRA} = \langle V_{TRA}, E_{TRA} \rangle$, S is an empty stack, TB is an empty set of timed blocks, v_{root} is the root vertex of the G_{TRA}

Output: TB

```
1  if isVisited()(v) then
2  |   return
3  end
4  isVisited()(v) ← true
5  if type(v) = TENT then
6  |   push(S, v)
7  else if type(v) = TEXT then
8  |   v1 ← pop(S)
9  |   TB ← TB ∪ {copyGraph(G_TRA, v, v1)}
10 end
11 foreach v2 in getChildren(v) do
12 |   timedBlockFinder(G_TRA, S, v2)
13 end
```

Figure 3.6d shows how we identify the timed blocks. We invoke timedBlockFinder(...) on $START$, and we traverse through the path ($B1_1$, $B1_2$, $B1_3$, $B1_4$, $B1_5$, $B1_6$, $B2$, $B3_1$, $B3_2$, END) on successive invocations. We traverse $B3_1$ before $B4$ (line 12). When we reach $B1_2$, put that on S and when we reach $B1_4$, pop $B1_2$ out, and create the timed block $TB1$. Similarly we create $TB2$, by identifying $B1_5$ and $B3_1$. We invoke timedBlockFinder(...) on the other child of $B2$, and create $TB3$, by identifying $B4_1$ and $B4_3$. The algorithm then terminates as we have visited all the nodes. Please note that, the selection of $B4$ over $B3_1$ will yield the same result. timedBlockFinder(...) creates the timed block because of three reasons. First, it does not travel the same vertex twice (the function isVisited(), lines 1–3). There are no malformed timed blocks (timed blocks do not partially overlap, and by definition they are singly entry (entry block) and single exit (exit block) sub graph). timedBlockFinder(...) travels the node in the program order. Notice that we need to discover paths from entry to exit in timed blocks. The enumeration is used for further analysis in section 4.1. In the future, we plan to use satisfiable

modulo theories to assist in discovering paths between vertices [11]. We also leverage the fact that timing requirement of ($B2$, $B3_1$, $B3_2$, END) in $TB2$ will always be met, whenever the timing requirement of ($B2$, $B4_1$, $B4_2$, $B4_3$, $B3_1$, $B3_2$, END) is met. Hence, we can ignore thee former path.

3.4 WCET Analysis for the PRET Architecture

A key objective of the PRET architecture [13] is to simplify the complexity of the WCET analysis. This is possible due to the predictable microarchitecture of PRET, and its repeatable temporal behaviours for instruction execution. Hence, our WCET analysis uses fixed individual instruction execution costs, and it assumes known loop bounds. We compute the WCET of a basic block by assuming that all instructions in the basic block are on allocated to the main memory T^{main}. We perform this analysis for every thread since PRET is a multi-threaded architecture. However, we only compute the WCET of basic blocks enclosed within timed blocks. This allows us to verify whether the timing requirement specified in the timed block adheres to the WCET of the instructions within the timed block. Since our objective is to ensure that all executions of a timed block always meets timing requirement, we must guarantee that every path's execution is less than or equal to the timing requirement specified by the timing instructions. Consequently, this work differs from simply reducing the WCET path as done by others [7] where the objective is to simply reduce the WCET of the program. By having a timing requirement, we only allocate the necessary number of instructions to the SPM to meet the timing requirement, and use the remaining space on the SPM for other instructions for other timed blocks or hardware threads.

Chapter 4

SPM Instruction Allocation

The allocation of instructions to SPM is a 0-1 knapsack problem. Given a set of n items and a knapsack with a capacity c with p_j and w_j as the profit and weight of item j respectively. The knapsack problem is selection of items, such that we maximize the total profit Pr :

$$Pr = \sum_{j=1}^{n} p_j * x_j \tag{4.1}$$

such that

$$\sum_{j=1}^{n} w_j * x_j \leq c \tag{4.2}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

w_j and p_j are independent of w_i and p_i respectively, where $j \neq i$,

For allocation of instructions to SPM, the item j is the basic block of instruction. So we have n basic blocks or items, and the capacity c is the total SPM size. x_j is set to 1, if the item j

(basic block) is selected (for allocation) and 0 otherwise. Profit $p_j = frequency_j * (time_j^{main} - time_j spm)$, for each block j. Please note that w_j and p_j are distinct for each basic blocks. We want to maximize Pr, which means the instruction saved/gain by allocating basic blocks to SPM.

4.1 Problem Formulation

Symbol	Description
g_{pjt}	Auxiliary variable that assists in reducing the difference between the timing requirement and the actual execution time of path p in timed block j of thread t.
g_{pjt}^{abs}	The absolute value of g_{pjt}.
$F_{jt}(k)$	Frequency of execution of basic block k, with respect to timed block j in thread t.
$T_t^{main}(k)$	WCET of basic block k, when executed on main memory.
$T_t^{spm}(k)$	WCET of basic block k, when executed on SPM.
K_{pjt}	Set of all basic blocks forming path p in timed block j of thread t.
$S_t(k)$	The size of basic block k in thread t.

Table 4.1: Symbol table for variables used in allocation.

The objective of the instruction SPM allocation is to meet the timing requirements specified in the timed blocks. However, we want to allocate the minimum number of basic blocks so that we *just* meet our requirements. This allows us to utilize the remaining SPM space for other timed blocks from the same thread, and from other threads.

$$X_t(k) = \begin{cases} 1 & \text{if basic block } k \text{ in thread } t \text{ is allocated} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Minimize

$$A = \sum_{t=1}^{H} \sum_{j=1}^{J_t} \sum_{p=1}^{P_{jt}} g_{pjt}^{abs} \tag{4.5}$$

such that

$$\sum_{t=1}^{H} \sum_{k=1}^{N_t} X_t(k) S_t(k) \leq S^{spm} \tag{4.6}$$

and

$$\forall p \in [1, P_{jt}], \forall j \in [1, J_t], \forall t \in [1, H]$$

$$R_{jt} \geq g_{pjt} + T_{pjt} \tag{4.7}$$

$$g_{pjt}^{abs} \geq \pm g_{pjt} \tag{4.8}$$

where

$$T_{pjt} = \sum_{k \in K_{pjt}} [X_t(k) F_{jt}(k) T_t^{spm}(k) +$$

$$(1 - X_t(k)) F_{jt}(k) T_t^{main}(k)]$$

and

$$g_{pjt}\, is\, free\, integer.$$

We present an integer-linear programming (ILP) formulation for allocating instructions from multiple threads. The variable $X_t(k)$ (Equation 4.4) is the boolean variable representing the basic block of instruction k in thread t. It is equal to 1 only when the basic block is allocated to SPM. We minimize a variable A that allocates just enough instructions to meet the timing requirements. The variable g_{pjt} in the objective function (Equation 4.5) is an auxiliary variable that assists in

reducing the difference between the timing requirement specified by the timed block j, and the WCET estimates of path p in thread t. A negative value of g_{pjt} suggests a violation of the timing requirements. By minimizing the sum of the absolute value of this variable (g_{pjt}^{abs}) for all paths p in timed block j and in thread t, we reduce the difference between the timing requirements of the timed blocks and the WCETs of the enclosed paths. Table 4.1 shows the meaning of all relevant variables, we use in the formulation. N_{pjt} is the total number of basic blocks on path p, P_{jt} is the total number of paths for timed block j, and J_t is the total number of timed blocks in thread t. H is the total number of threads. N_t is the total number of basic blocks in thread t.

The first constraint (Equation 4.6) states that the sum of the sizes of all basic blocks allocated to the SPM must not exceed the maximum size of the SPM (S^{spm}). The second constraint (Equation 4.7) is for all the three variants of the timed blocks discussed in Section 2.2.1. By minimizing g_{pjt}, we reduce the WCET of path p by allocating basic blocks from path p to the SPM. R_{jt} denotes the timing requirement for timed block j in thread t, and T_{pjt} is the computed WCET of path p within timed block j in thread t. There are a total of $\sum_{t=1}^{H}\sum_{j=1}^{J_t} P_{jt}$ such constraints. The variables $X_t(k)$, $F_{jt}(k)$, $T_t^{spm}(k)$, $T_t^{main}(k)$ represent the indicator variable, frequency of occurrence, execution time on SPM and execution time on main memory, respectively of basic block k. K_{pjt} denotes the set of basic blocks in the path p within timed block j in thread t.

Lemma 4.1.1. *The auxiliary variable $g_{pjt} \leq 0$, for a given path p, timed block j and thread t.*

Proof. Given $R_{jt} - T_{pjt} \geq g_{pjt}$. We want to minimize $|g_{pjt}|$, and the best possible value for that is zero as the absolute auxiliary variable is always non-negative. At any given point of time before, during or after the allocation, we will have two cases:

Case 1: When $R_{jt} \geq T_{pjt}$

In this case, g_{pjt} will always be zero, so the above condition holds true and we achieve the best possible value for objective function.

Case 2: When $R_{jt} \leq T_{pjt}$

In this case g_{pjt} will be a negative integer.

Hence $g_{pjt} \leq 0$. □

Extraction of frequency $F_{jt}(k)$ uses our partial data-flow analysis on the binary. The loop detection can be done by using any of the standard graph based techniques. Most common among them being the approach based on dominator trees. We use a straightforward graph based cycle detection technique to identify them. As we assume known loop bounds, we extract them automatically from the binary by identifying the loop header, and looking at the compare instruction. For instance in Figure 4.1:

```
40400050: cmp     r1, #n
40400054: bgt     404000a4
```

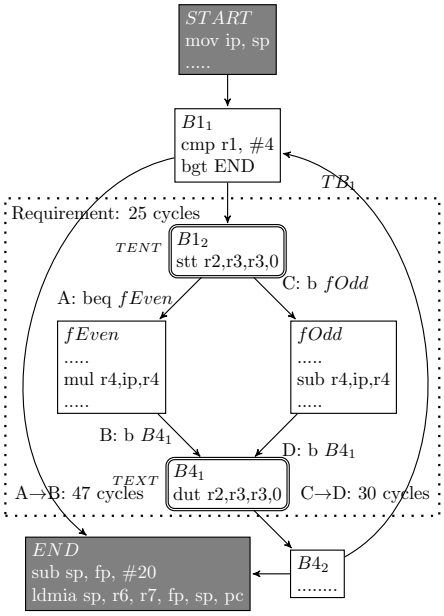
Figure 4.1: Identifying the frequency.

By looking at $cmpr1, \#n$ instruction in the loop header, we find that $r1$ is the loop induction variable, and $n + 1$ is the loop bound. For backward loops, the comparison looks like $movr1, \#n; cmpr1, \#0; ble < exitLoopTarget >$. The comparison can also be done relative to a register, after storing loop bound in the same ($cmpr1, r8$). A value analysis on $r8$, may yield the loop bound, but that is not the focus of the paper, as we said we assume known loop bounds. So the value of $r8$ is known. Also, please note that frequency $F_{jt}(k)$ is a relative frequency of execution of the block, not an absolute. By $F_{jt}(k)$, we mean how frequent block k executes inside the timed block j, not how frequent it executes in the program. For instance in figure 3.6d, the loop bound of the loop ($(B2, B4_1, B4_2, B4_3)$ is 10. From the perspective of timed block T3, the frequency of execution of block $B4_2$ is 1. However, from the perspective of T2, the frequency of execution is 10. Hence, $F_{31}(k)$ equals 1, and $F_{21}(9)$ equals 10.

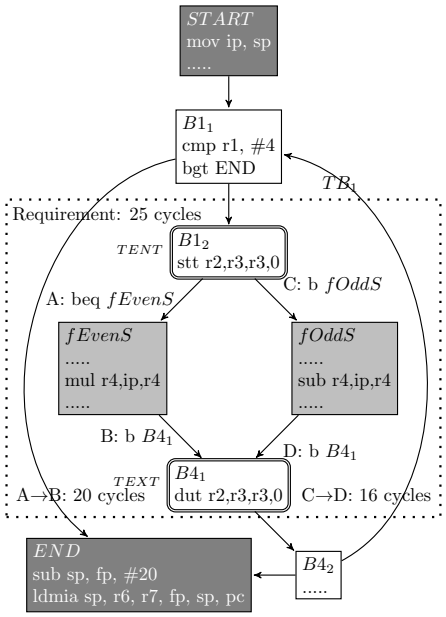
Figure 4.2 shows an example program, its CFG from binary, and the result of the allocation. The code consists of a simple loop with a timed block with a timing requirement of 25 cycles. The timed block consists of two paths (A, B) and (C, D), which take 47 and 30 cycles, respectively. We allocate blocks from all paths within the timed block such that the WCETs of each path is less than the timing requirement of the timed block. Hence, for the example in Figure 4.2, we allocate both paths to meet the timing requirement. Blocks `fEven` and `fOdd` are allocated to the SPM. Note that after allocation, we alter the branch instructions (`beq <feven>` and `b <fodd>` to `beq <fevenS>`, and `b <foddS>`, respectively) in the binary, to correctly branch to the SPM locations, and maintain the correct control flow.

```
        int fEven(int a, int b) {
          return(a*b);
        }
        int fOdd(int a, int b) {
          return(a-b);
        }
        int main(){
START:    int i, c[N],a[N],b[N];
B11, B42:   for(i=0;i<N;i++) {
B12, B41:    tryfor(150, P_NS){
              if(imodulo2 ==0 )
fEven:      c[i]= fEven(a[i],b[i]);
              else
fOdd:       c[i] = fOdd(a[i],b[i]);
             }
           }
END:       return 0;
        }
```

(a) C code fragment.



(b) TRACFG with timed block identification.



(c) After allocation.

Figure 4.2: Example allocation. Note that it shows a fragment of the binary code.

Chapter 5

Re-writing

5.1 Re-writing the PRET executable binary

After determining the allocation, we rewrite the binary to reflect the allocation. The rewriting stage moves allocated blocks to the SPM address space, and inserts appropriate control-transfer instructions to preserve the correct semantics of the program. Algorithm 4 explains the rewriting.

The inputs for Algorithm 4 include the set of vertices from the TRACFG V_{TRA}, and the set of basic blocks S selected for allocation to the SPM. The output is a sequence of compiled instructions P'. For each vertex $v_c = \langle k, a, p \rangle$, we obtain the new SPM address using $\mathsf{getNextSPMAddr}()$, and rewrite its pc. We also keep a mapping of the old address (pc) and the new SPM address pc' (lines 4–10). Changing the addresses of an instruction can result in incorrect control flow of the program. For example, changing the address of an instruction ci whose address was the target of some other branch instruction. To address this, we modify the targets of other control-transfer instructions to reflect the allocation changes using the mapping. Notice that we also insert a no-operation instruction to the end of the basic block in v_c. We replace this

no-operation instruction with a branch instruction to the next basic block in order to correctly reflect the control flow.

```
B1:
0x4000004c: cmp     r1, #2 ; 0xf
0x40000050: ldrls   pc, [pc, r1, lsl #2]
B2:
0x40000054: b       40000158
B1':
0x40000058: 40000098
0x4000005c: 400000a4
B3:
0x40000098: mov     r1, #0
0x4000009c: str     r1, [fp, #-20]
0x400000a0: b       4000016c
B4:
0x400000a4: mov     r1, #36
0x400000a8: str     r1, [fp, #-20]
0x400000ac: b       4000016c
```

Figure 5.1: Simple example of jump table with targets.

Lines 13–23 in algorithm 4 describes the modification to branch instructions, pc-relative load/stores and jump tables. We traverse each instruction in the basic block for v_c denoted by p, and we use newTarget() to compute the updated target address. If ci is a branch instruction then we use rewrite ci with the new target address (lines 15–16). Notice that newTarget() uses the map between old addresses and new address from line 5. When we allocate PC-relative load and store instructions (i.e. `ldr r1, [pc,#n]`) to SPM, they also suffer from the problem of pointing to incorrect targets. We correct these the same way we corrected immediate branches, as shown in the lines 15–16. However, the ARM ISA only permits an immediate field of 12 bits. In order to load from a 32- bit immediate value, we encode ci as a sequence of `eor` and shift operations followed by the relevant load instruction. The register-indirect branch instructions used for jump tables have unknown targets, but we resolve these as before (see Section 3.2). We

illustrate these key concepts in rewriting using a simple example in Figure 5.1.

A precise control flow (since all the n instructions are potential targets and hence it is not an over-approximation [31]) is maintained by either allocating all the instruction blocks containing the n instructions and instruction block containing the indirect jump to SPM, or simply not allocating the blocks. In this case, if $B1$ is allocated to SPM, we need to allocate $B1'$ to SPM too and vice versa. If any of the targets themselves, are allocated to SPM ($B3$ or $B4$), we need to change the entries of the jump table ($B1'$) to match the new target(s). The function updateJumpTable() in Algorithm 4, lines 17–19, achieves the same. So if we allocate $B3$ to SPM, the resulting binary will look like figure 5.2:

```
B1':
  0x40000058: 40000098
  0x4000005c: 40400000
B3:
  0x40400000: mov     r1, #0
  0x40400004: str     r1, [fp, #-20]
  0x40400008: b       4000016c
```

Figure 5.2: Re-written jump table with targets.

We add branch instructions to maintain the correct control flow of the whole program. These branch instructions use the space created by the `nop` instruction on line 8. This is shown in Figure 5.3.

```
  0x400082ac: cmp r3, #10
  0x400082b0: bgt 0x400082dc
  0x400082b4: ldr r1, [pc, #76]
```

Figure 5.3: Illustration of rewriting.

Assume the instruction block with the `bgt` instruction gets allocated to the SPM. To maintain the correct control flow, we insert a branch instruction after the `bgt` instruction such that it

branches back to main memory location of the previous fall-through path. The modified instructions are shown in Figure 5.4.

```
0x40400004: cmp r3, #10
0x40400008: bgt 0x400082dc
0x4040000c: b 0x400082b4
0x400082b4: ldr r1, [pc, #76]
```

Figure 5.4: Adding branch instructions to maintain control flow.

These added branch instructions also help in maintaining the correct control flow in case of register-indirect branch instructions. For example, the instruction `mov pc,lr` loads the pc with the contents of the link register. However, the instruction identified by the target address in the link register lr can reside in the SPM or the main memory. If the target is in main memory, correct control flow is maintained as it branches back to the correct location. If the target is in SPM, then we insert a branch instruction to preserve correct control flow. We do not currently address the issue of fragmentation caused by adding these additional branch instructions. However, we do incorporate these added branch instructions in the execution cost of each block on the SPM. Note that in the examples, we use branch instructions with target addresses. However, in binary, the target offset is encoded instead of address.

Algorithm 4: Rewriting the binary.

Input: V_{TRA}, S
Output: P', a re-written sequence of compiled instructions.

Let BRANCHOPS, PCLOADOPS and JTABLESTARTOPS be the set of opcodes for branch instructions, pc relative load/stores and starting instruction of jump table respectively
Let S is the set of basic blocks to be allocated
Let M be a Map
foreach $v_c = \langle k, a, p \rangle \in S$ **do**
    foreach $ci = (pc, i)$ *in* p **do**
        Let $pc' \leftarrow$ getNextSPMAddr()
        putToMap(M, getAddress(ci), pc')
        $ci \leftarrow (pc', i)$
    end
    addToEnd(p, (getNextSPMAddr(), nop()))
end
$V_L \leftarrow$ order(V_{TRA})
foreach $v_c = \langle k, a, p \rangle \in V_L$ **do**
    foreach $ci = (pc, i)$ *in* p **do**
        if getOpcode(ci) $\cap$ BRANCHOPS $\neq \emptyset$ **then**
            $ci \leftarrow (pc,$ wrtBranch(i, newTarget(i, M)))
        else if getOpcode(ci) $\cap$ PCLOADOPS $\neq \emptyset$ **then**
            $ci \leftarrow (pc,$ wrtLoad(i, newTarget(i, M)))
        else if getOpcode(ci) $\cap$ JTABLESTARTOPS $\neq \emptyset$ **then**
            updateJumpTable(ci, M)
        end
    end
end
foreach $v_j = \langle k, a, p \rangle, v_{j+1} = \langle k_{j+1}, a_{j+1}, p_{j+1} \in V_L$ **do**
    if $(v_j \in S \wedge v_{j+1} \notin S) \vee (v_j \notin S \wedge v_{j+1} \in S)$ **then**
        $pc' \leftarrow$ getAddress(getNop(p))
        updateNop(p, (pc', wrtUBranch(peek(p_{j+1})))
    end
end
$P' \leftarrow$ createBinary(V_L)

Chapter 6

Results

We experimentally compare the proposed approach with an approach that uses the ACET method [3], and one that uses the WCET method [23]. Since there are a limited number of PRET programs, we perform the evaluation on altered Malardalen benchmarks. We alter them by inserting timing constructs in the benchmarks. The only benchmarks we do not use are the ones that use recursions because our analysis does not support that as of now.

The first experiment shows that increasing the SPM size increases the fraction of timed blocks that meet their timing requirements as shown in Figure 6.4. Note that some of the benchmarks meet their timing requirements earlier than others. This happens because of the different timing requirements in the benchmarks.

In this section we present two main ideas. First the comparison of our allocation strategy with ACET [3] and WCET [23] reduction based SPM allocation. We show that either the common allocation techniques cannot be directly applied and need some adaptation or they use more SPM size. Second the improvement with respect to PRET having shared memory among threads as opposed to dedicated SPM memory banks for each threads.

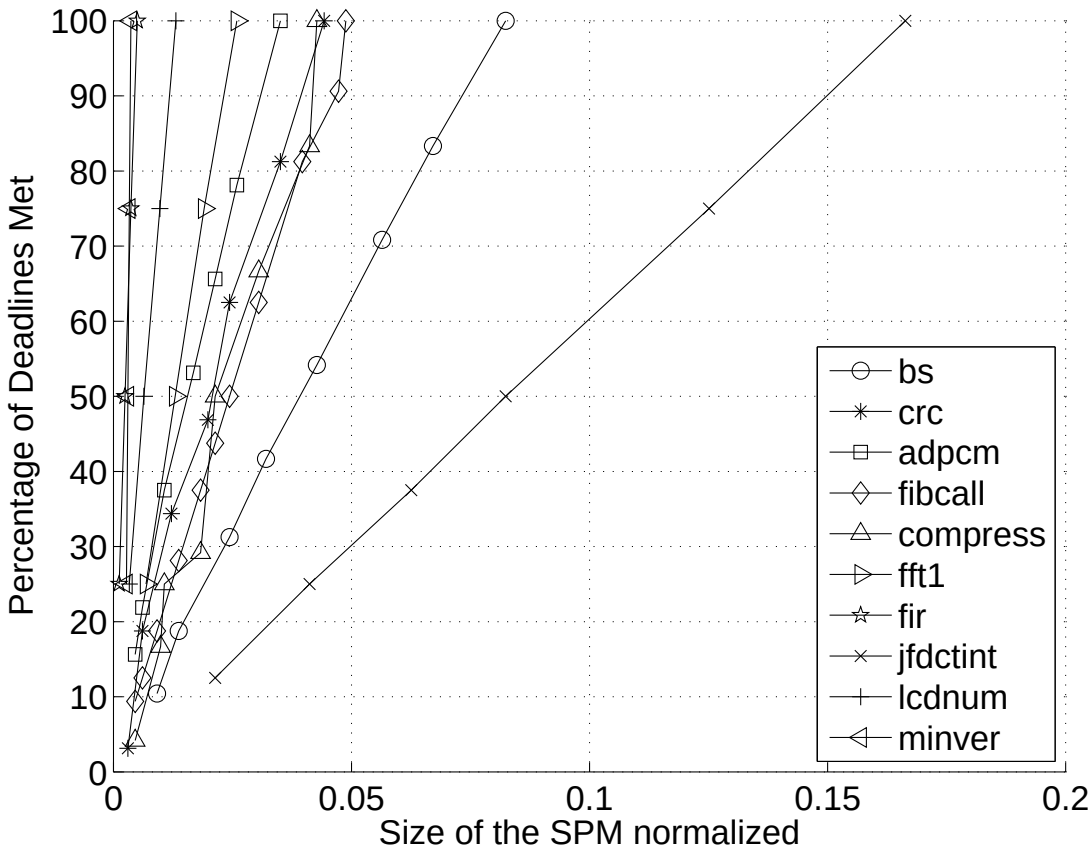


Figure 6.1: Timing requirements met versus SPM size.

The table 6.1 shows each program with their respective number of timed blocks. The table also shows the individual SPM space requirements to meet all the timing requirements of each program for our scheme and WCET based scheme. We ran the experiments from 0% to 20% of the total SPM size available on PRET, and executed it on the PRET simulator [26]. We will talk about how we implement ACET & WCET approaches next.

6.1 Comparison of our scheme vs ACET and WCET based approaches

Benchmark	# Timed Blocks	SPM Size Requirements (bytes)	
		WCET	Our Scheme
adpcmc	8	2628	760
bs	5	3088	1000
compress	10	3564	480
crc	8	956	560
edn	1	1144	300
fdct	8	12440	10900
fibcall	8	1152	430
fir	1	5824	80
fft1	1	728	200
jfdctint	2	2764	2680
lcdnum	2	516	240
minver	1	6192	20
nsichneu	4	33436	840
duff	1	664	480
prime	1	492	100
select	1	948	840
ud	1	1716	1140

Table 6.1: Malardalen Benchmarks.

Figure 6.2 shows the comparison between ACET based, WCET based static allocation and our allocation. Like Figure 6.4, the numbers on x axis denote the respective configuration. We have a total of 9 configurations, made after randomly selecting four Malardalen Benchmarks. Each benchmark is denoted by a number given on the right of the figure 6.4. The four numbers under the bars denote the four benchmarks (comprising each configuration) used for the four threads. Note that for each set of experiments we use different SPM size. This is because every

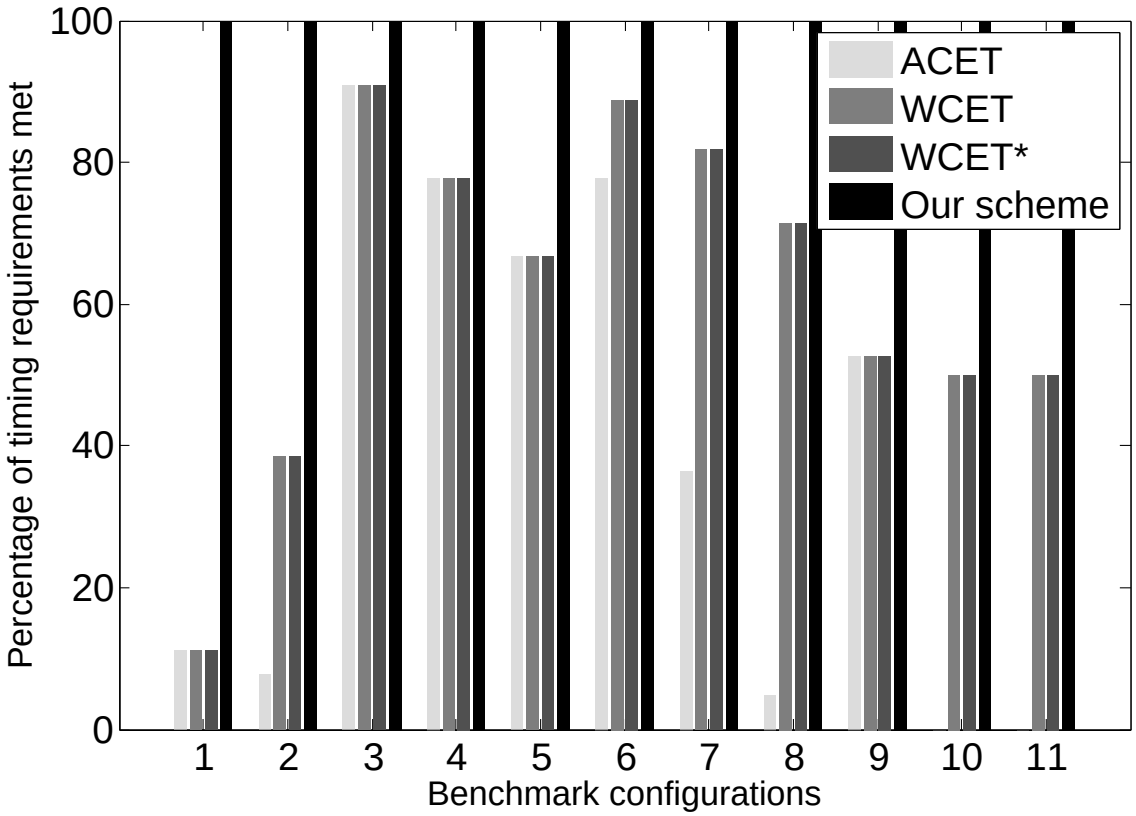


Figure 6.2: Percentage of Timing requirements Met.

program has different SPM size requirements for meeting its timing requirements. The figure shows the percentage of the total timing requirements met at the point (SPM space), where our scheme met all of them. This point (SPM space) is different for different configurations as different threads have different minimum space requirements. As evident from the figure, our scheme excels over both the commonly known approaches. In ACET, we select only those basic blocks for allocation which give higher cost which is product of frequency and execution time. We formulate the problem as 0-1 ILP problem with size constraint, to enable an optimal selection of basic blocks. The formulation is similar to the one in [3]. We try to maximize the gain after allocation to SPM and Avissar et al [3], who try to minimize the loss of having instructions on

main memory. By its definition, we cannot use ACET based approach, in real time system as they don't care about the timing requirements. This is evident from our experiments, where we miss timing requirements (figure 6.2). In configuration 3, 4, 5 & 6, ACET meets higher percentage of timing requirements ($> 70\%$). The reason is two-fold. First, the included benchmarks (crc, edn, $lcdnum$, $fft1$ & $fibcall$) have low SPM size requirement to meet timing constraints individually (shown in table 6.1). Second, the timed blocks mostly include expensive basic blocks.

In WCET, we follow the scheme described as greedy heuristic algorithm in [23] with code object as basic block. We allocate the most impacting basic block (highest contribution towards execution time) on the WCET path. Please note that the allocation, may change the WCET path, so we recompute the WCET path after allocation. We keep allocating until we reach the size limit or WCET path stops changing and all the blocks in that path are allocated. We raised a concern before, that all the timing requirements may not be met, because of this WCET approach. There may be some timed blocks outside this WCET path, awaiting allocation. So even if we have unused space left on SPM, we may miss timing requirements on such timed blocks. This actually happened in our benchmark $compress$ as we miss one timing requirement on the function $writebytes$ as it lies outside the WCET path. So our scheme meets all the timing requirements for a given space in each configurations, while WCET-based static allocation misses timing requirements. That is why we extend the WCET approach to WCET-H approach, to use the unused space to meet the timing requirements by allocating expensive blocks. This scheme meets all the timing requirements in $compress$ benchmark.

We see that the WCET and ACET meet equivalent percentage of timing requirements in some configurations. This is because majority of basic blocks in our experiments come in the WCET path of the entire program and ACET also selects the most impacting (expensive) basic block among all the basic blocks. WCET meets higher timing requirements in 3, 4, 5 & 6 because of the same reasons as ACET approach mentioned above. It also meets $> 50\%$ timing requirements

in configurations 7, 8 & 9 because of the benchmark $nsichneu$ which has huge individual space requirement to meet its timing requirements (shown in table 6.1). WCET on the other hand, chooses to rather meet all other timing requirements instead of $nsichneu$ in the available space. Our scheme has high minimum space requirement to meet all the timing requirements in 7, 8 & 9 including $nsichneu$.

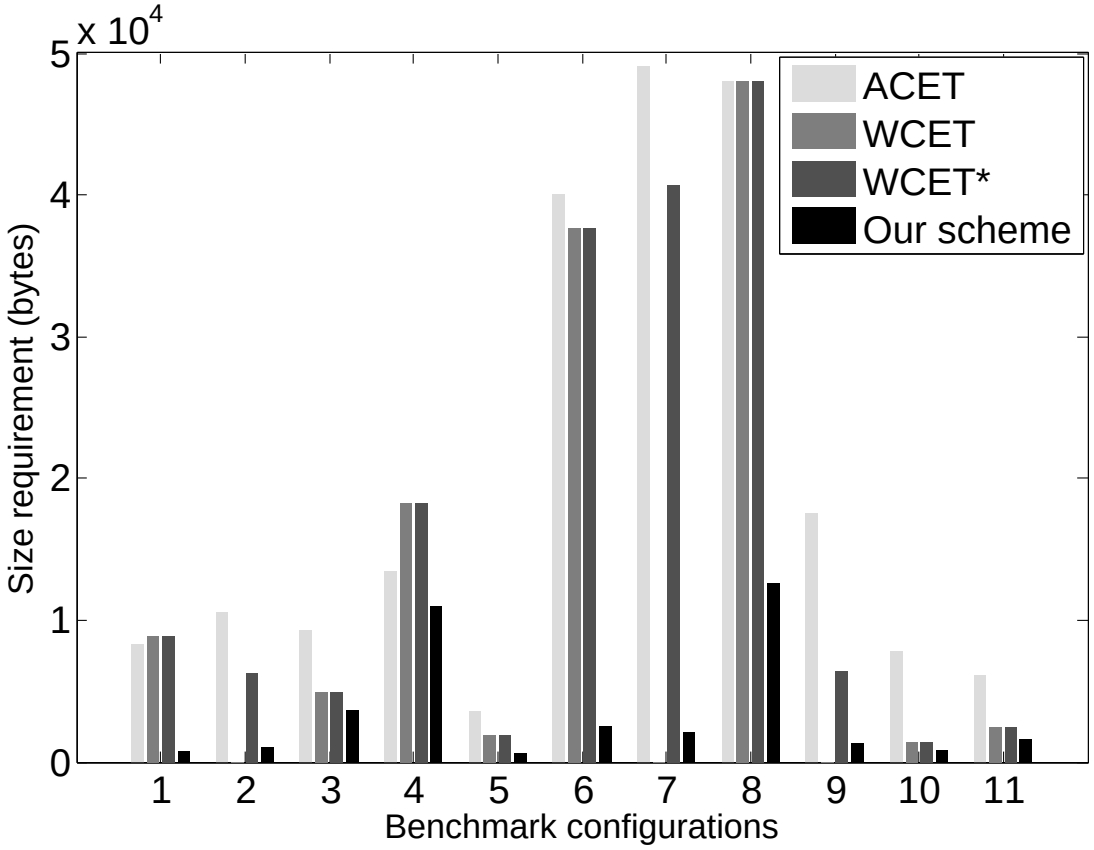


Figure 6.3: Size requirements.

Figure 6.3 shows the total minimum SPM size required by ACET, WCET and our scheme to meet all the timing requirements in each configurations. The results observed are in accordance with the one in Figure 6.2. The total size requirement of ACET based approach to meet all the

deadlines, is as high as 23x as compared to our scheme (configuration 7) with the average 9x.

Our scheme requires less SPM space in all the configurations, as we specifically target timed blocks and its constituent basic blocks. Important point to note is that, ACET will meet all the timing requirements eventually, the worst case size requirement being the whole program.

In WCET, the total size requirement in the configuration 2, 7 & 9 is zero because we will always miss one time requirement in the benchmark $compress$. So we use WCET-H approach, to meet all the timing requirements minus this missed timing requirement in configurations 2, 7& 9. So we have presented the total size required to meet all the timing requirements minus this missed timing requirement. The reason why WCET works better than ACET in most configurations is two-fold. First non-inclusion of expensive basic blocks in WCET path. Second not all threads have timing requirements, for instance threads $bsort$, cnt, $expint$, $insertsort$, $janne\_complex$, $ludcmp$, $matmult$, $minmax$, ns, $qurt$ and $statemate$ do not have any timing requirements. WCET also performs worse than ACET in some configurations (1 & 4) of figure 6.3. This is because WCET selects the most impacting basic block with no regard to its size or bulkiness. ACET is slightly smarter in this regard as it leverages ILP.

In our benchmarks, as we pointed out before, the improvement observed as compared to ACET and WCET is also because we specifically target, identify and allocate timed blocks. We will also like to point out that, even in case of straight line program with a single time requirement on the entire program, we perform better than ACET and WCET. For instance, consider a simple program with just four basic blocks (a, b, c, d), with execution times t_a^{main}, t_b^{main}, t_c^{main} & t_d^{main} respectively on main memory, t_a^{spm}, t_b^{spm}, t_c^{spm} & t_d^{spm} respectively on SPM and sizes s_a, s_b, s_c & s_d respectively. Its given that $t_d > t_c > t_b >> t_a$ and $s_d > s_c > s_b >> s_a$. Suppose if the timing requirement is $t_a^{main} + t_b^{main} + t_c^{main} + t_d^{main} + \epsilon$ where $\epsilon < t_a^{main} - t_a spm$. So while ACET and WCET will both select block d as the allocation candidate, our scheme will select block a as the candidate. So clearly we meet the timing requirement with least space requirement (s_a) among

the other schemes (s_d).

6.2 Experiments on Shared SPM vs Dedicated SPM

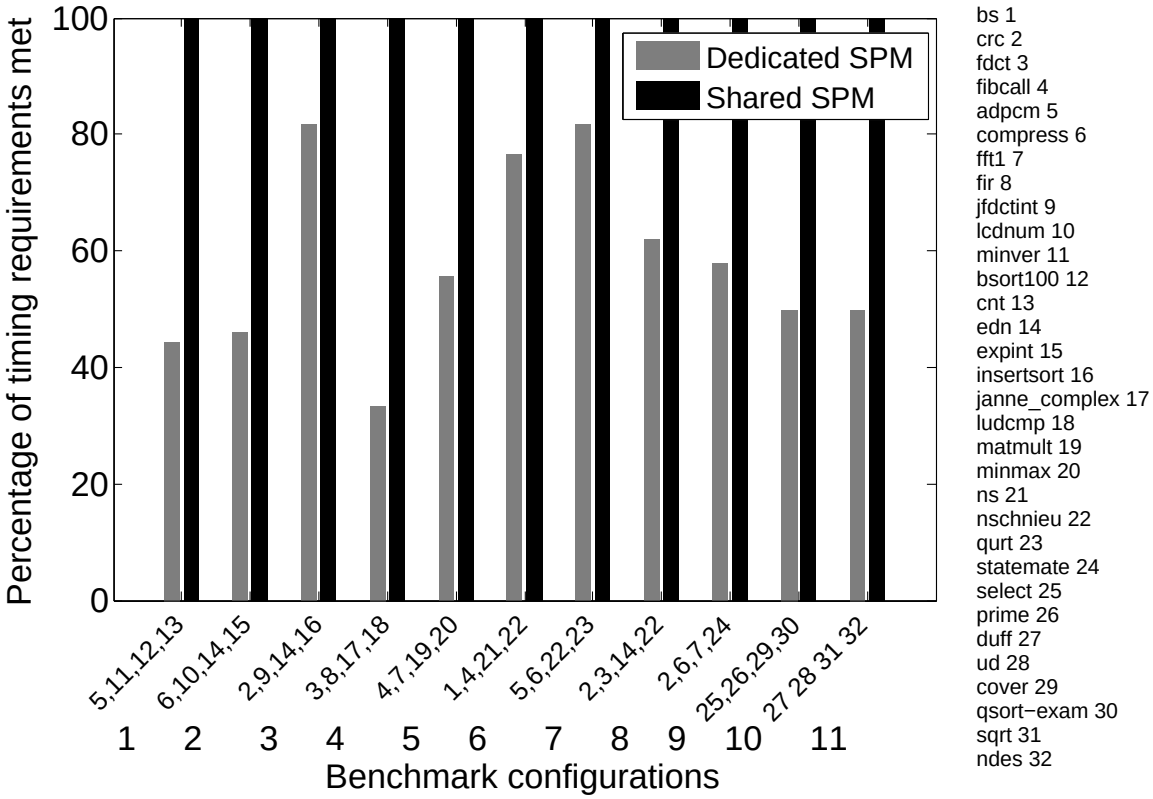


Figure 6.4: Percentage of timing requirements met

The second set of experiments indicate the advantage of using the shared SPM, and our multi-threaded SPM allocation over having dedicated SPM segments for each thread. Figures 6.5, 6.4 show the results. We use the same configuration for figures 6.2 and 6.3. The black bar shows the fraction of timing requirements met for shared SPM, and gray bar shows the timing requirements met when each thread has its own dedicated SPM segment. The total SPM size for all threads

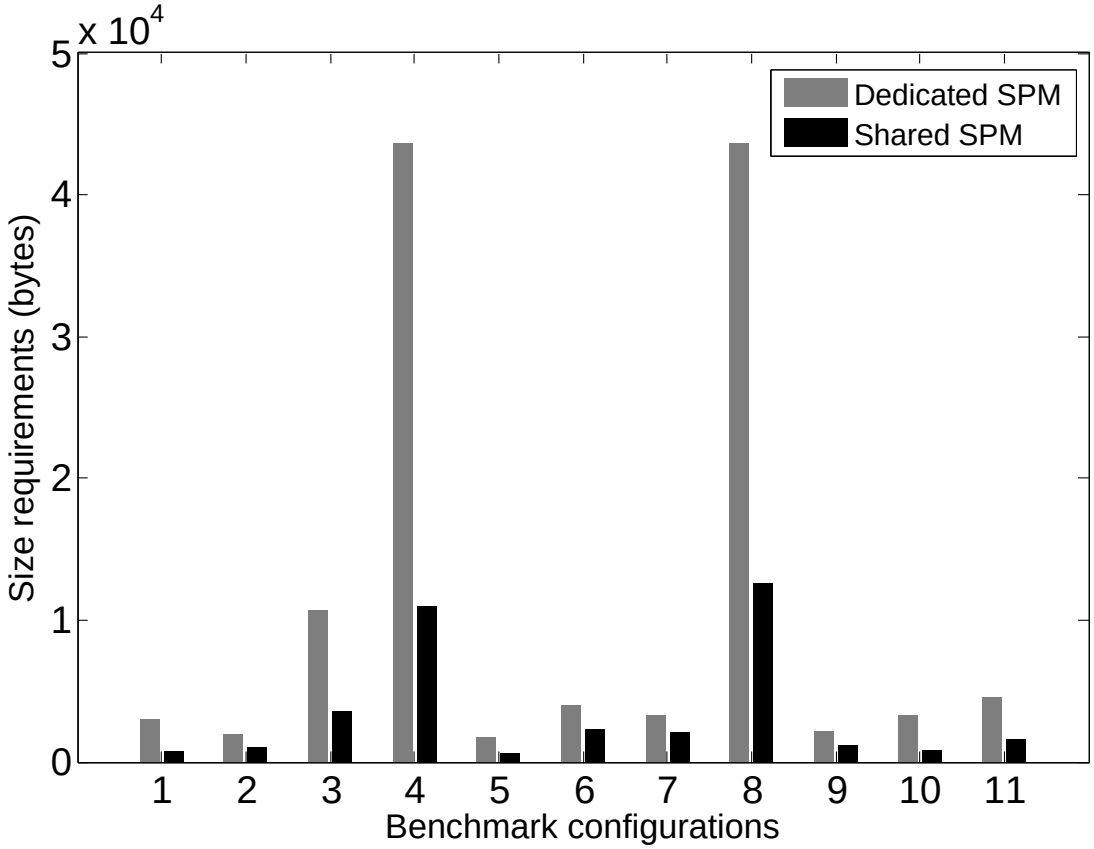


Figure 6.5: Total Size requirements

is the same as the shared SPM. Figure 6.4 shows that the shared SPM clearly helps in meeting timing requirements, and our allocation leverages the shared SPM by using the unused space of one thread for another thread's timed blocks. Figure 6.5 shows the minimum size requirements, to meet all the deadlines for all benchmarks in each configuration. Clearly shared SPM space excels over dedicated memory banks for each thread. Note that if we simply sum up the individual size requirements for each thread from table 6.1, it will be same as size requirement with shared space. The reason we get improvement is because, we take thread in each configuration with maximum size requirement, then give the other three threads the same size requirement,

and then compute the total size requirement for each configuration. The rationale behind this is that if each thread has its own SPM bank, they all need to be of equal size.

Chapter 7

Case Study

7.1 Unmanned Air Vehicle

We present a case study taken from PapaBench [16]. PapaBench is a real time benchmark derived from GNU Paparazzi UAV project.It has two main functional units executing on different processors, namely fly_by_wire (managing servo/radio commands) and $autopilot$ (controlling the flight movement). fly_by_wire incorporates five tasks r1–r5 shown in table 7.1 with their respective timing requirements. We run fly_by_wire on first two threads and $autopilot$ on the next two threads. Nemer et al. [16] launch tasks at the respective frequencies/periodicity, without explicitly specifying the amount of time these tasks should take. They assume that these tasks will always be finished within the time bound of their respective period. Their assumption might hold when WCET of these tasks are low as compared to their respective time periods. That is why scale the frequencies to make the case study more interesting. Also, in order to specify the time constraints on these tasks, instead of writing a scheduler, a simpler and infallible way is use of timed blocks. For example, if we want altitude_control_task to take x cycles periodically, we

can simply specify in the program (figure 7.1):

```
tryfor(x){
altitude_control_task()
}
```

Figure 7.1: Task from Papabench

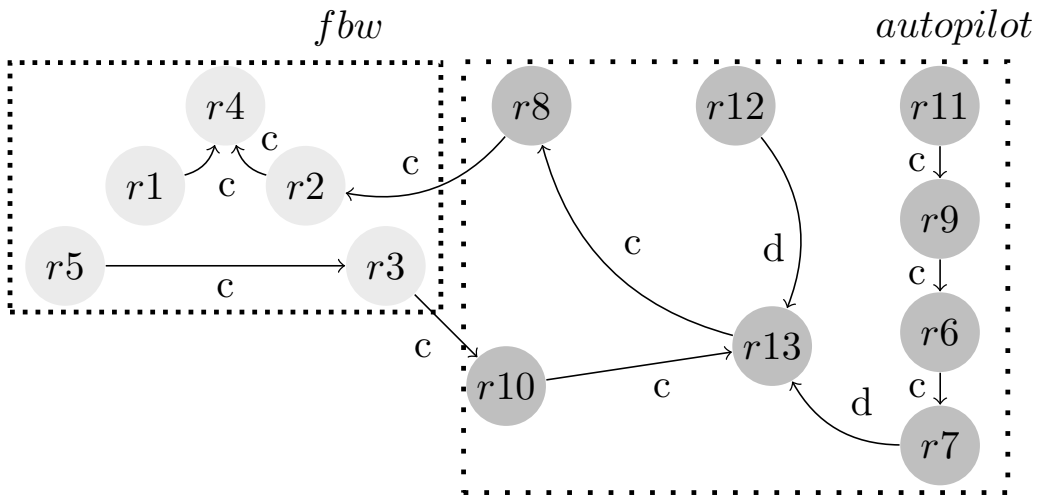


Figure 7.2: Inter-task interactions in Papabench with control and data dependency.

Figure 7.2 shows the interaction between different tasks and dependencies c, for control and d for data. fly_by_wire receives radio orders using task $r5$, and transmits to $autopilot$ via task $r3$. $autopilot$ performs tasks $r10$, $r13$ and $r8$ to analyse the orders, stabilize and transmit back to fly_by_wire. $r2$ receives the data, and $r4$ transmits it to servos. Task $r11$ helps in receiving the gps data and tasks $r9$, $r6$ & $r7$ help in the stabilization task based on gps data.

In table 7.1, we have scaled the timing requirements from the actual benchmark by a factor of 195 in order to make the case study more interesting. This is equivalent to slowing down the processor. We run tasks $r1$, $r2$ & $r5$ on the first thread , $r3$ & $r4$ on the second thread, $r6$, $r7$, $r8$, $r9$, $r12$ & $r13$ on the third thread and $r10$ & $r11$ on the fourth thread. We also change the timing

requirements of the task $r10$ and $r12$ by a factor of 4 and 2 respectively to again make the case study more interesting.

We observe, that our approach takes around 27.5 k of SPM to meet all the timing requirements in the 13 tasks. ACET and WCET as discussed in the chapter 6 take 128k and 82k SPM space respectively to meet all the timing requirements, making us 4.7x and 3.0x better than ACET and WCET respectively in space requirements. For a given SPM size of 27.5k, we meet all the timing requirements using our approach. However, ACET and WCET approaches miss 8 and 4 timing requirements out of the 13 tasks/timing requirements. Also please note that, some of the tasks including $r4$, $r6$, $r7$ and $r8$, do not need any allocation, as their time of execution across all paths on main memory is lower than the respective timing requirement.

TaskID	Tasks	Timing Requirements (kHz)
r1	check_failsafe_task	62.50
r2	check_mega128_values_task	62.50
r3	send_data_to_autopilot_task	31.25
r4	servo_transmit	62.50
r5	test_ppm_task	31.25
r6	altitude_control_task	312.50
r7	climb_control_task	312.50
r8	link_fbw_send	62.50
r9	navigation_task	312.50
r10	radio_control_task	125.00
r11	receive_gps_data_task	312.50
r12	reporting_task	250.00
r13	stabilisation_task	62.50

Table 7.1: Tasks and their timing requirements in Papabench.

Chapter 8

Conclusion

We present a tool that statically allocates instructions from multiple threads to a shared SPM for the PRET architecture. Our allocation has three key novelties: 1) it attempts to meet timing requirements explicitly specified in the program, 2) it allocates the minimum number of basic blocks necessary to satisfy these timing requirements, and 3) it performs its allocation across multiple threads, which benefits overall application allocation. Our results indicate that we successfully meet our timing requirements, and that we leverage the shared SPM for the multi-threaded PRET architecture over dedicated SPMs. We are currently investigating the allocation of data and synchronization variables across multiple threads.

8.1 Future Work

The approach does not discuss synchronization between the threads, and the delay introduced because of the synchronization. If there is a wait within the timed block on an event of another thread, the allocation should change to reduce the response time. The work in [23], talks

about the worst case response time between the threads and how they minimize it. We plan to investigate this issue in the future work

We plan to investigate dynamic allocation and feasibility of the approach with regard to binary re-writing. Re-writing dynamically is challenging as the binary changes at run time. So we plan to investigate the feasibility of the approach and the overheads involved.

We are also working on size agnostic implementation of our approach. By size agnostic we mean, we will prepare the best allocation strategy for each possible SPM size off line. Then we retrieve the allocation based on SPM size. We also want to run one re-written binary, on all platforms with different SPM sizes.

References

[1] S. Andalam, P. Roop, and A. Girault. Predictable multithreading of embedded applications using pret-c. In *Proc. of International Conference on Formal Methods and Models for Codesign*, pages 159–168. ACM, 2010.

[2] ARM. ARM technical reference manual, 2001.

[3] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1:6–26, 2002.

[4] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, 2010.

[5] Dai Nguyen Bui, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Jan Reineke. Temporal isolation on multiprocessing architectures. In *Proc. of Design Automation Conference*, pages 274–279, 2011.

[6] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Ingénieurs de l'Automobile*, 807:36–42, 2010.

[7] Jean-Francois Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 179–190. IEEE Computer Society, 2007.

[8] GNU. GNU ARM cross compiler toolchain, 2006.

[9] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and accurate low-level pointer analysis. In *Proc. of the International Symposium on Code Generation and Optimization*, pages 291–302. IEEE Computer Society, 2005.

[10] T. Harmon, M. Schoeberl, R. Kirner, R. Klefstad, K.H.K. Kim, and M.R. Lowry. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Transactions on Industrial Informatics*, 8:366–377, 2012.

[11] W.R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *ACM Sigplan Notices*, volume 45, pages 71–82. ACM, 2010.

[12] C. Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference, Ottawa, Canada*, 2008.

[13] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 137–146, 2008.

[14] Isaac Liu, Jan Reineke, and Edward A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *Asilomar Conference on Signals, Systems, and Computers*, 2010.

[15] Stefan Metzlaff, Sascha Uhrig, Jörg Mische, and Theo Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proc. of the Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, pages 38–45. ACM, 2008.

[16] Fadia Nemer, Hugues Cass, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench : A free real-time benchmark. In *Proc. of Workshop on Worst-Case Execution Time Analysis*, 2006.

[17] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Scratch-pad memory allocation without compiler support for java applications. In *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 85–94. ACM, 2007.

[18] Jongsoo Park, James Balfour, and William James Dally. Fine-grain dynamic instruction placement for l0 scratch-pad memory. In *Proc. of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 137–146. ACM, 2010.

[19] Hiren D. Patel, Ben Lickly, Bas Burgers, and Edward A. Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. Technical Report UCB/EECS-2008-115, EECS Department, University of California, Berkeley, 2008.

[20] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. Pret dram controller: bank privatization for predictability and temporal isolation. In *Proc. of the International Conference on Hardware/software Codesign and System Synthesis*, pages 99–108. ACM, 2011.

[21] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time*

and Embedded Systems (JTRES 2004), volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[22] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *In Proc. of Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, 2011.

[23] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Proc. of the IEEE International Real-Time Systems Symposium*, pages 223–232. IEEE Computer Society, 2005.

[24] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad allocation for concurrent embedded software. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 37–42. ACM, 2008.

[25] The Paparazzi team. The paparazzi project, 2003.

[26] The CHESS PRET team. The pret simulator ptarm, 2011.

[27] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5:472–511, 2006.

[28] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 104–109, New York, NY, USA, 2004. ACM.

[29] J. Whitham and N. Audsley. Studying the applicability of the scratchpad memory management unit. In *Real-Time and Embedded Technology and Applications Symposium*, pages 205 –214, 2010.

[30] Jack Whitham and Neil Audsley. Studying the applicability of the scratchpad memory management unit. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 205–214, Washington, DC, USA, 2010. IEEE Computer Society.

[31] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. Technical report, University of California, Davis, 2009.