An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture

Aayush Prakash
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
aayush.prakash@uwaterloo.ca

Hiren D. Patel
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
hdpatel@uwaterloo.ca

***Abstract*—This work presents a static instruction allocation scheme for the precision timed architecture's (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 subset of a modified version of the Malardalen benchmarks to show the benefits of performing the allocation across multiple threads.**

I. Introduction

The Precision Timed architecture (PRET) is a hard real-time embedded processor architecture [1] 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 [2]. 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 [3], [4]. SPM allocation techniques that are WCET-centric such as that proposed by Deverge and Puaut [3] and Suhendra et al. [4] 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 [5] 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 [1]. 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.

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

978-3-9810801-8-6/DATE12/©2012 EDAA

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.

II. Related Work

There are two broad areas of research in SPM allocation: reduction of average-case execution time (ACET), and reduction of worst-case execution time (WCET) [3], [4]. General purpose systems use ACET methods, and hard real-time systems typically use WCET methods. The authors in [3], [4] propose data allocation schemes that focus on reducing the WCET of the program. 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.

Whitham and Audsley [6] present a WCET-directed dynamic data and instruction allocation to SPM by introducing a time-predictable scratchpad memory management unit. Our approach performs a static allocation by rewriting the binaries, and we do not require additional hardware.

There are works on allocation of instructions across multiple threads [7], [5]. Metzlaff et al. [7] present a predictable hardware-based dynamic allocation to instruction SPM for a simultaneous multi-threaded processor. Mitra et al. [5] incorporate interference between threads due to dependencies in pre-emptive scheduling, and include this in their allocation scheme. These approaches do not incorporate explicit timing requirements, which is the focus of our work. We explored instruction allocation for PRET in the past through profiling [8]. This work extends the prior work considerably by automating the analysis, and allocation from binaries.

III. 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 [1]. This is a multi-processing architecture with a thread-interleaved pipeline that supports four hardware threads [9]. 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 [9] to control the temporal behaviour of the program. We present these timing instructions in Table I [9].

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 [10]. For more information about the PRET memory hierarchy, we forward the readers to [10].

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 I: ISA extensions with timing instructions for PRET.

A. Timed blocks

We use the timing instructions from Table I to define *timed blocks*. A timed block encloses a sequence of instructions that have a specific temporal requirement. Figure 2 shows code fragments that use macros that synthesize to timing instructions from Table I, 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.

Fig. 2: Variants of timed blocks.

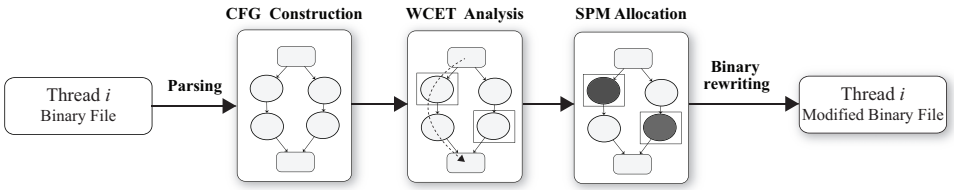


Fig. 1: Flow of the proposed tool for thread i.

IV. Multi-threaded Instruction SPM Allocation for the PRET Architecture

We prototype a tool that performs instruction SPM allocation for the multi-threaded PRET architecture. These threads can be a part of a single application or different applications. Figure 1 shows the main stages of this tool. The first stage constructs a control-flow graph (CFG) for each thread's compiled binary. We support binaries using the ARMv4 ISA. We use the CFGs to identify basic blocks, and timed blocks formed by the timing instructions in the program. Then, we perform static WCET analysis on the basic and timed blocks for each CFG. Notice that the design of the PRET architecture significantly simplifies the WCET analysis [1] because of its predictable and repeatable instruction execution. We supply the results of the WCET analysis to the SPM allocation stage, which determines the blocks to allocate on the shared SPM. Upon determining the allocation, we rewrite the binary making the necessary changes to reflect the allocation. This 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.

A. CFG Construction

The primary steps involved in this phase are parsing the binary, basic block detection, timed block detection, branch target address identification, and representing this information in a CFG. We discuss these for one thread, but the same applies to all other threads. Parsing the binary yields a list of instructions, and its assigned memory addresses as done by the compiler. We accept binaries in the form of SREC format.

We determine the basic blocks in this list of instructions by identifying control transfer instructions. PRET supports multiple control transfer instructions such as immediate branches, register-indirect branches (including instructions that alter the program counter using loads), and timing instructions with exception handling. We locate these control transfer instructions, and determine the target address for each one.

The target address for an immediate branch is straightforward to compute because the effective address is a function of the immediate value and the program counter (PC) of the current branch instruction. Jump tables commonly use register-indirect branches, which use registers to compute the offset for computing the target address. While in general, determining the target address of a register-indirect branch requires a form of binary-level data-flow analysis [11], we perform a partial data-flow analysis to compute the possible targets for jump tables. Currently, we do not support general register-indirect branches in our tool. Observe that jump tables are typically of the following form:

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

The instruction `ldrls` loads the address $pc + 4 + r1 \times 4$ into pc, only when $r1$ is less than or equal to n. Note that $\#n$ denotes a known immediate value n, and the branch is exclusively either forward or backward. Suppose that n is forward, then we can compute the possible values for $r1$. For example, if n is 2, then the possible offsets of n are 0, 1 and 2. Hence, pc may hold an address between $pc + 4$ and $pc + 12$. We conservatively construct edges from the end of the basic block with the register-indirect instruction to each of the possible target addresses.

Another use of register-indirect branches is to return from procedure calls. To return from a procedure P, to a procedure Q_k the link register with Q_k's address is loaded into the pc. We accomplish this by making a callback list for P with all the procedures $Q_1, Q_2, ..., Q_k$ that invoked P at some point of time using static analysis. The targets of register-indirect branches from P are the program points in $Q_1, Q_2, ..., Q_k$ that invoke P.

Control transfers as a result of exceptions in the timing instructions are handled similar to immediate branch instructions. We extend the definition of the CFG with special nodes that identify the beginning (set_time) and end (delay_until or branch_expired) of a timed block based on the timing instructions. Every timed block specifies a timing requirement. A timed block, however, may enclose multiple subpaths of the CFG. We discover these subpaths through target analysis.

B. WCET Analysis for the PRET Architecture

PRET significantly simplifies WCET analysis. We assume known loop bounds, and individual instruction execution costs. The latter is possible due to the predictable and repeatable temporal behaviours of the instructions. We know that every instruction always takes the same number of execution cycles. The WCET of the basic block starts by assuming that all instructions are on the main memory T^{main}. We perform this analysis for every thread since PRET is a multi-threaded architecture. For the blocks of code enclosed within timed blocks, we compute their WCET estimates. 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 the timed blocks always meet the 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 [3].

C. SPM Instruction Allocation

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

We present an integer-linear programming (ILP) formulation for allocating instructions from multiple threads. The variable $X_t(k)$ (Equation 1) 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 2) 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 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. 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 3) states that the sum of the size of all basic blocks allocated to the SPM must not exceed the maximum size of the SPM (S^{spm}). $S_t(k)$ is the size of basic block k in thread t. The second constraint (Equation 4) is for all the three variants of the timed blocks discussed in Section III-A. 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 $X_{pjt}(k)$, $F_{pjt}(k)$, $T^{spm}_{pjt}(k)$, $T^{main}_{pjt}(k)$ represent the indicator variable, frequency of occurrence, execution time on SPM and execution time on main memory, respectively. Extraction of frequency $F_{pjt}(k)$, involves partial data-flow analysis on the binary.

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

Minimize

$$A = \sum_{t=1}^{H}\sum_{j=1}^{J_t}\sum_{p=1}^{P_{jt}} |g_{pjt}| \quad (2)$$

such that

$$\sum_{t=1}^{H}\sum_{k=1}^{N_t} X_t(k)S_t(k) \leq S^{spm} \quad (3)$$

and

$$\forall p \in [1, P_{jt}], \forall j \in [1, J_t], \forall t \in [1, H]$$
$$R_{jt} - T_{pjt} \geq g_{pjt} \quad (4)$$

where

$$T_{pjt} = \sum_{k=1}^{N_{pjt}} [X_{pjt}(k)F_{pjt}(k)T^{spm}_{pjt}(k) + (1 - X_{pjt}(k))F_{pjt}(k)T^{main}_{pjt}(k)]$$

Figure 3 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 $A \rightarrow B$ and $C \rightarrow D$, 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 3, 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.

D. Rewriting

After determining the allocation, we rewrite the binary to reflect the allocation. The rewrite phase moves allocated blocks to the SPM address space, and inserts appropriate control transfer instructions to preserve the correct semantics of the program. Algorithm 1 explains the rewriting. The inputs are the CFG G, and a set of basic blocks S that are to be allocated to the SPM. We represent every instruction e as a tuple $\langle o, i \rangle$ where o is the address, and i is the instruction encoding in binary. For the basic blocks $b \in S$, we obtain the new SPM address using $getSPMAddr()$, and rewrite the address in e. Changing the addresses of instructions leads to the problem of incorrect control flow. This requires us to modify the targets of other instructions that use the PC such as branches, loads, and stores. Lines 9–11 in Algorithm 1 describe the modification to immediate branch instructions. We use $newTarget()$ to compute the updated target address, and rewrite the immediate branch with this target address. 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

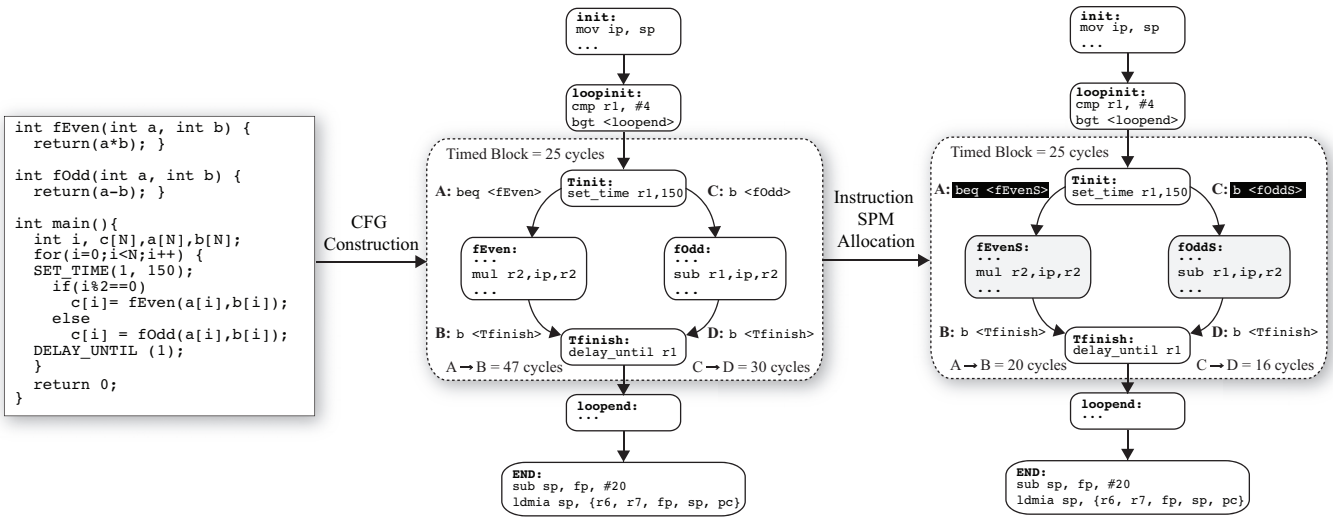


Fig. 3: Example allocation. Note that it shows a fragment of the binary code.

Algorithm 1: Rewriting the binary.

Input: G, S
Output: re-written binary

```
1  Let B ← basicBlocks(G) be the set of m basic blocks
2  foreach b ∈ S do
3      foreach instruction e = ⟨o, i⟩ in b do
4          e ← updateAddr(⟨getSPMAddr(), i⟩)
5      end
6  end
7  foreach b ∈ B do
8      foreach instruction r = ⟨o, i⟩ ∈ b do
9          if r is an immediate branch then
10             r ← ⟨o, rewriteBranch(i, newTarget(i))⟩
11         else if r is a PC relative load then
12             r ← ⟨o, rewriteLoad(i, newTarget(i))⟩
13         end
14     end
15 end
16 B_list ← sort(B)
17 foreach b_j, b_{j+1} ∈ B_list do
18     if (b_j ∈ S ∧ b_{j+1} ∉ S) ∨ (b_j ∉ S ∧ b_{j+1} ∈ S) then
19         addTo(b_j, ⟨o, newBranch(startAddr(b_{j+1}))⟩)
20     end
21 end
22 dumpBinary(B_list)
```

branches as shown in the lines 11–13. However, the ARM ISA only permits an immediate field of 12 bits. In order to load from a 32 bit immediate value, we encode e 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 IV-A). A precise control flow (since all the n instructions are potential targets and hence it is not an over-approximation [12]) 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. We add branch instructions to maintain the correct control flow of the whole program. For example :

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

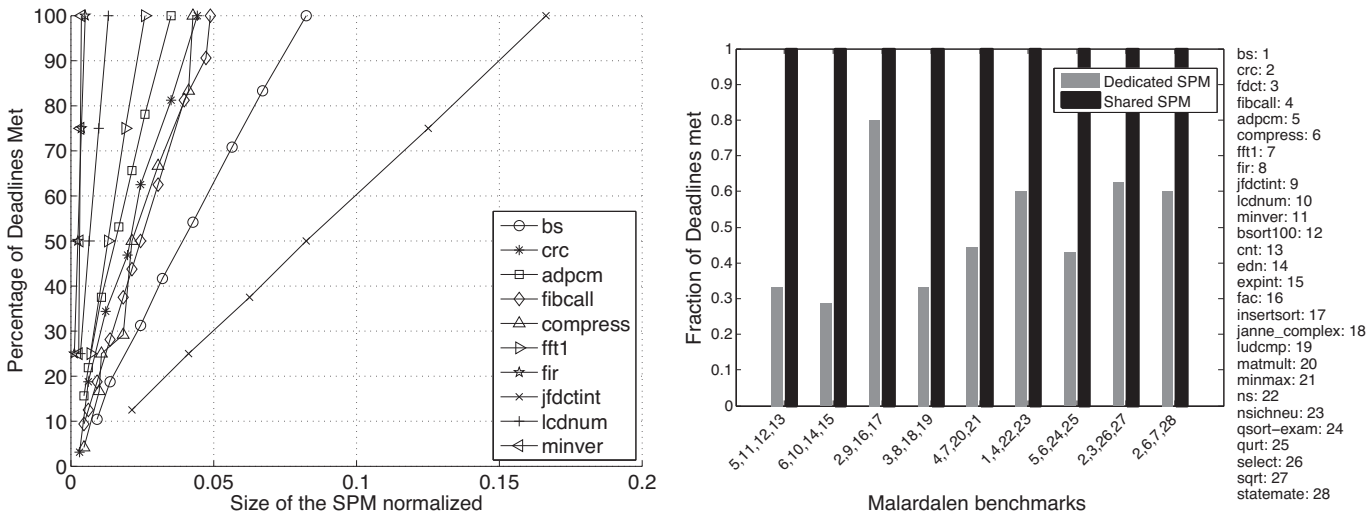
Assume the instruction block with the `bgt` instruction gets allocated to the SPM. To maintain the correct control flow, we insert a branch instruction after the `bgt` instruction such that it branches back to main memory location of the previous fall through path. The modified instructions are shown below:

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

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.

V. Results

We modify the Malardalen benchmark by adding timing instructions to a subset of its programs. We ran two sets of experiments on these benchmarks. The first experiment shows that increasing the SPM size increases the fraction of timed blocks that meet their timing requirements as shown in



(a) Timing requirements met versus SPM size.

(b) Shared SPM versus dedicated SPM for each benchmark.

Fig. 4: Results of the experiments on Malardalen benchmark.

Figure 4a. Note that some of the benchmarks meet their timing requirements earlier than others. This happens because of the different timing requirements in the benchmarks. We ran the experiments from 0% to 20% of the total SPM size available on PRET, and executed it on the PRET simulator [13].

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. Figure 4b shows the results. Each benchmark is denoted by a number given on the right of the figure. The four numbers under the bars denote the four benchmarks used for the four threads. Note that for each set of experiments we use different SPM size. This is because every program has different requirements of SPM size for meeting its timing requirements. The black bar shows the fraction of timing requirements met for shared SPM, and gray bar shows the deadlines met when each thread has its own dedicated SPM segment. The total SPM size for all threads is the same as the shared SPM. Figure 4b 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.

VI. Conclusions

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.

References

[1] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 137–146.

[2] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," *Ingénieurs de l'Automobile*, vol. 807, pp. 36–42, 2010.

[3] J.-F. Deverge and I. Puaut, "WCET-directed dynamic scratchpad memory allocation of data," in *Proc. of the Euromicro Conference on Real-Time Systems*. IEEE Computer Society, 2007, pp. 179–190.

[4] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proc. of the IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 2005, pp. 223–232.

[5] V. Suhendra, A. Roychoudhury, and T. Mitra, "Scratchpad allocation for concurrent embedded software," in *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2008, pp. 37–42.

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

[7] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer, "Predictable dynamic instruction scratchpad for simultaneous multithreaded processors," in *Proc. of the Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*. ACM, 2008, pp. 38–45.

[8] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee, "A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-115, 2008.

[9] D. N. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Proc. of Design Automation Conference*, 2011, pp. 274–279.

[10] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. 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*. ACM, 2011, pp. 99–108.

[11] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *Proc. of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2005, pp. 291–302.

[12] L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," University of California, Davis, Tech. Rep., 2009.

[13] The CHESS PRET team. (2011) The pret simulator ptarm. [Online]. Available: http://chess.eecs.berkeley.edu/pret/src/ptarm-1.0/