

An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture

Aayush Prakash *Student Member, IEEE*, and Hiren D. Patel *Member, IEEE*

Abstract—This work presents a static instruction scratchpad memory allocation scheme for the precision timed architecture (PRET). 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 the timing requirements from the multiple hardware threads of the PRET architecture. We formulate the allocation problem as an integer-linear programming problem, and we implement a tool that takes compiled ARMv4 binaries, constructs a timing-requirements aware control-flow graph, performs a WCET analysis and SPM allocation, and rewrites the binaries with the allocation. We evaluate our approach using a modified version of the Malardalen benchmarks to show the benefits of the proposed approach. We also present a UAV benchmark derived from the Papabench benchmark.

Index Terms—Precision timed architecture, scratchpad memory allocation, predictability, real-time embedded systems.

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 the designer to control 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 [3], [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. Another work [5] addresses this limitation 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 execution time on the worst-case path may not be sufficient to meet the timing requirements because there may be timing requirements not on the worst-case path.

Current PRET programming practices require entire programs to fit on the SPM [1]. This limits PRET's practical use since programs are typically larger than the SPM, and manual allocation is inefficient. This brings us to the focus of our work: a static instruction SPM allocation scheme for PRET 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. The main contributions of this work are the following: 1) a structured method to specify timing requirements using timing constructs, 2) a timing-requirement aware control-flow graph (TRACFG) of the program as an intermediate representation to capture timing requirements, 3) a static instruction SPM allocation method specifically designed to meet timing requirements for PRET programs, 4) a static WCET analysis for PRET, and 5) a tool automating the analysis and allocation. We compare the proposed approach with an average-case execution time (ACET), and a WCET allocation schemes along with a hybrid of the two scheme using altered versions of the Malardalen benchmarks, and a Papabench [6] UAV benchmark. The Malardalen benchmarks consist of small programs used for WCET analysis, and the Papabench models core tasks for a UAV application.

II. BACKGROUND ON PRET ARCHITECTURE

The precision timed architecture (PRET) is a hard real-time embedded processor that has predictable and repeatable temporal behaviors [1]. PRET uses a thread-interleaved pipeline with four hardware threads. It possesses instruction-set architecture (ISA) extensions to the ARMv4 ISA that allow the user to specify temporal requirements in the form

is typically the dut. Other variants with exception handling exist, and are explained in [9]. We identify these timed blocks in the TRACFG by transforming the vertices with timing instructions to ensure that the timing instruction resides in its own basic block. Then, we annotate that basic block with *TENT* if it is a stt and with a *TEXT* if the instruction in that basic block is a dut. Hence, a timed block encapsulates a sub-graph of the TRACFG, and there can be multiple timed blocks in a program. We identify different timing instructions using their respective opcode and timing requirements using partial data-flow analysis.

d) WCET Analysis for the PRET Architecture: We implement a WCET analysis for the PRET architecture. By design, PRET's architecture provides fixed individual instruction execution costs for both non-memory, and memory instructions. These instruction costs are defined by an implementation of PRET [9] called PTARM. Before computing the WCET, a phase of the analysis computes the execution frequencies and size of basic blocks, and the overheads in terms of instructions in potentially allocating a block to the SPM. The overhead can be in the form of added instructions to maintain correct control-flow, and size by adding additional data associated with jump tables onto the SPM. We take an overly conservative approach in determining these overheads through analysis, and we do not include it in the ILP formulation. The latter would only improve the results of the allocation. The analysis iterates over all basic blocks and computes its WCET based on whether the particular block is allocated to the SPM or not. We use these basic block WCETs to compute the WCET of paths of execution within a timed block. This allows us to verify whether the timing requirement specified for a particular timed block adheres to the WCET path within the timed block.

Symbol	Description
$X_t(k)$	1 if basic block k , in thread t is allocated to SPM and 0 otherwise.
g_{pjt}	Auxiliary variable that assists in reducing the difference between the timing requirement and 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 .
N_{pjt}	The total number of basic blocks on path p
P_{jt}	Total number of paths for timed block j .
J_t	Total number of timed blocks in thread t .
H	Total number of threads.
N_t	Total number of basic blocks in thread t .

TABLE II: Symbol table for variables used in allocation.

e) SPM Instruction Allocation: The objective of the instruction SPM allocation is to meet the timing requirements specified by the timed blocks. However, we want to allocate the minimum number of basic blocks such 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. Consequently, this work differs from simply reducing the WCET path as done by others [3] where the objective is to simply reduce the WCET of the program.

We use an integer-linear programming (ILP) formulation for instruction SPM allocation. We minimize a variable A that allocates just enough instructions to meet the timing requirements. The variable g_{pjt}^{abs} is an auxiliary variable that assists in reducing the difference between the timing requirement specified by timed block j , and the WCET estimate of path p in thread t . g_{pjt}^{abs} represents an absolute value and hence, the objective function appears non-linear. However, we transform the objective function to a linear variant using known techniques, by introducing g_{pjt} , a free auxiliary variable and using the constraint in Equation 4. So Equation 4 makes g_{pjt}^{abs} non-negative or absolute value of g_{pjt} . A negative value of free auxiliary variable 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 II shows the meaning of all relevant variables, we use in the formulation.

Minimize

$$A = \sum_{t=1}^H \sum_{j=1}^{J_t} g_{jt} \quad (1)$$

such that

$$\sum_{t=1}^H \sum_{k=1}^{N_t} X_t(k)S_t(k) \leq S^{spm} \quad (2)$$

and

$$\forall p \in [1, P_{jt}], \forall j \in [1, J_t], \forall t \in [1, H] \quad (3)$$

$$R_{jt} - T_{pjt} \geq g_{pjt} \quad (3)$$

$$g_{pjt}^{abs} \geq \pm g_{pjt} \quad (4)$$

where

$$g_{jt} = \sum_{p=1}^{P_{jt}} g_{pjt}^{abs}$$

$$T_{pjt} = \sum_{k \in K_{pjt}} [X_t(k)F_{jt}(k)T_t^{spm}(k) + (1 - X_t(k))F_{jt}(k)T_t^{main}(k)]$$

The first constraint (Equation 2) 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}). The second constraint (Equation 3) indicates that the difference between the requirement and the execution time is greater or equal to the auxiliary variable. 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. Note that a phase in the WCET analysis stage conservatively incorporates the overhead costs of branch instruction needed if a block is to be allocated on the SPM in $T_t^{spm}(k)$ and $T_t^{main}(k)$.

Algorithm 1: Implement the allocation in the binary.

```

Input:  $V_{TRA}, S$ 
1 Let  $S$  be the set of basic blocks to be allocated.
2 Let  $M$  be a map structure.
3 Let  $V_L$  be the ordered set of vertices according to entry point addresses.
4 foreach instruction  $ci = (pc, i)$  in basic block  $v_c \in S$  to be allocated do
5   | Insert in map  $M$  the mapping  $pc$  to the next available SPM address  $npc$ .
6   | Replace the  $pc$  for  $ci$  with  $npc$ .
7 end
8 foreach basic block  $v_c \in S$  to be allocated do
9   | Insert a  $nop$  instruction at the end of basic block  $v_c$ .
10 end
11 foreach instruction  $ci = (pc, i)$  in basic block  $v_c$  in  $V_L$  in program  $V_L$  do
12   | if  $ci$ 's opcode is of a control-transfer instruction then
13   |   | Update  $ci$  with replacement instructions to maintain control-flow semantics.
14   | end
15 end
16 foreach basic block  $v_j, v_{j+1}$  in program in  $V_L$  that are consecutive blocks do
17   | if  $v_j$  allocated to SPM and not  $v_{j+1}$  or  $v_{j+1}$  allocated to SPM and not  $v_j$  then
18   |   |  $v_j$  Update  $nop$  instruction with new target address.
19   | end
20 end
21 Write out the new binary with the allocation from  $V_L$ .

```

f) *Rewriting:* The last stage rewrites each thread's TRACFG with the selected blocks for SPM allocation. Algorithm 1 uses the vertices of the TRACFG, and the basic blocks selected for allocation S as input. There are three key steps in the algorithm. The first step generates SPM addresses for the instructions, and preserves the mapping between the original address, and the new SPM address (lines 5 – 8). It also inserts a no-operation instruction at the end of the basic block to replace it later with a control-transfer instruction to honour the original control-flow semantics (lines 9 – 11). We define an instruction $ci = (pc, i)$ as a pair consisting of program counter, and the instruction bit-encoding. This step is important because any control-transfer instruction whose target address points to an allocated instruction would need to update its target address to reflect the SPM address. The map M allows us to do this. The second step (lines 12 – 16) updates the target address for every control-transfer instruction in each block, whose target address instruction is to be allocated to the SPM. Thus, we modify branch, pc-relative load/store and jump table instructions. The third step (lines 17 – 21) discovers points in the program where the control-flow semantics are violated due to SPM allocation. That is, a basic block is allocated to the SPM, but the next basic block to execute after it remains on the main memory. For this, we update the no-operation instruction with a control-transfer whose target instruction points to the first instruction of the next basic block in the main memory. This allows us to preserve the original control-flow semantics of the program for each thread. We can then create the binary with the updated vertices. The memory map of PRET [9] allows us to jump into any thread's address space from the SPM address space with a single branch instructions, and vice versa is also done through a single branch instruction.

IV. EXPERIMENTAL EVALUATION

We compare the proposed approach with three extended approaches: the ACET method [10], the WCET method [4], and a WCET-H (hybrid) approach for instruction allocation to SPM. The ACET method selects basic blocks for allocation

that provide the most savings in terms of execution time. We adapt the method to support basic blocks from multiple threads by simply introducing all blocks from every thread into a 0-1 ILP formulation [10]. To estimate the execution and frequencies of the basic blocks, we profile the execution of the benchmark. Then, we use these as input into the ACET allocation. The WCET method uses a greedy algorithm [4] that allocates basic blocks that contribute most towards the execution time on the WCET path. We use our static WCET analysis for the PRET architecture to compute the execution time of the basic blocks, and produce a WCET path for the entire program. Since the WCET path is specific for one program, we perform the allocation across multiple threads in the following way. We discover the SPM size necessary to meet all timing requirements and call this the baseline SPM size. Then, we perform the allocation for one program using the WCET approach until its terminating condition (WCET does not change). Once the terminating condition is reached, we start allocation with another thread with the remaining SPM space. We continue this for all four threads. Notice that since we report the percentage of timing requirements met, the order in which the threads are selected for allocation can affect the results. Hence, we enumerate all orderings for each configuration, and we only report one that provides the highest percentage of timing requirements met.

We acknowledge that ACET and WCET methods were not designed to incorporate explicit timing requirements. Therefore, we extend the WCET method to WCET-H based on the observation that the WCET method terminates once the WCET does not change; however, SPM space may still be available. Hence, there could be timed blocks that reside outside the WCET path that miss their timing requirements. To reduce missing timing requirements, we make a simple extension: once the WCET of the program stops changing, and there exists space on the SPM, we perform an ACET on the blocks that have not been selected for allocation. This forces the WCET-H method to fill the entire SPM by selecting blocks from other paths. Note that WCET-H is also not particularly designed to explicitly incorporate timing requirements.

We use an altered subset of the Malardalen benchmarks with timing constructs around the core functions of the benchmarks. Benchmarks that use recursions are not considered. Given that each benchmark has a different SPM size requirement to meet all its timing requirements, we set the SPM size for each configuration to be the total minimum SPM size required for all the benchmarks in that configuration. We perform the allocation using the proposed tool, and the resulting programs are executed on the publicly released PRET simulator [11] to ensure functional correctness.

g) *Results:* Figure 3a shows the percentage of timing requirements met for the three methods for dedicated (suffixed with D) and shared SPM (suffixed with S). We compute the size for the dedicated SPM by obtaining the size required to meet all the timing requirements in a configuration, and splitting it into four equal parts for each thread. The proposed method meets all timing requirements whereas other approaches do not. We attribute this to the fact that other approaches were not designed to meet timing requirements.

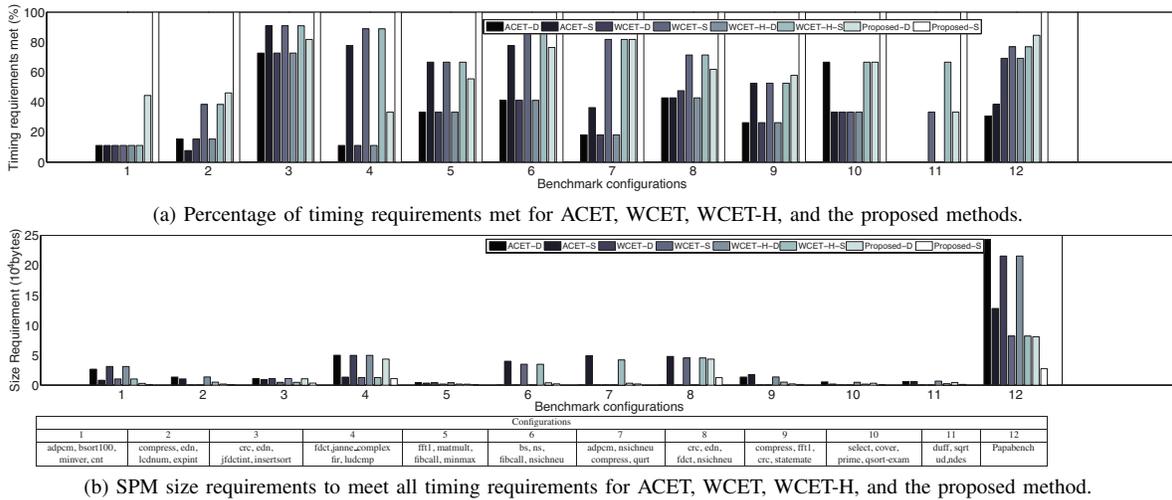


Fig. 3: Experimental results.

We notice that WCET-H has a higher percentage of timing requirements met than WCET for configurations 10 and 11 (shared case). This is because WCET terminates while SPM space is available, but WCET-H takes advantage of the available space. For the remaining configurations, WCET-H performs the same as WCET. WCET methods outperform ACET in configurations 4, 6, 7, 8 and 12 because ACET methods allocate based on frequency and execution cost, but are agnostic to whether the basic block belongs within a timed block. Configuration 12 shows the results We perform additional experimental evaluation using the GNU Paparazzi UAV project known as PapaBench [6] as the 12th configuration. This is a real-time benchmark with two main functional components: *flybywire* that manages servo/radio commands, and *autopilot* that controls the flight movement. Figure 3a also shows that having a shared SPM assists in meeting timing requirements when compared to dedicated SPM.

Figure 3b shows the total SPM size required to meet **all** timing requirements, if possible. For all benchmarks, the proposed method requires the least amount of SPM space in all configurations. Note that it is possible for ACET and WCET-H to meet all the timing requirements in the program. However, the worst-case space requirement for this could be the entire program. For example, configuration 7 requires 23x more SPM space for the ACET to meet all timing requirements. Similarly, for WCET-H, it is 20x. Notice that when using WCET, the total size requirement in configurations 2, 7, 9, 10 and 11 is zero. This is because there does not exist an allocation using WCET that will meet all timing requirements because some timing requirements are outside the path with the WCET. It is evident that shared SPM performs better than dedicated SPM. For example, in all the configurations we meet higher or equal percentage of timing requirements for shared SPM than dedicated SPM.

V. CONCLUSIONS

This work presents a method to statically allocate instructions to the SPM for the PRET architecture. The PRET

architecture is unique in that it allows programs to specify explicit timing requirements. The proposed allocation scheme attempts to meet these timing requirements by identifying them, and allocating the minimum number of basic blocks necessary to satisfy them. We automate the allocation process in a tool. Our experiments show the benefit of the proposed approach for the PRET architecture.

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*, 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*, 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] F. Nemer, H. Cass, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "Papabench : A free real-time benchmark," in *Proc. of Workshop on Worst-Case Execution Time Analysis*, 2006.
- [7] 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.
- [8] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD Conference, Ottawa, Canada*, 2008.
- [9] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A pret microarchitecture implementation with repeatable timing and competitive performance," in *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*, October 2012.
- [10] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, pp. 6–26, 2002.
- [11] The CHESS PRET team. (2011) The pret simulator ptarm. [Online]. Available: <http://chess.eecs.berkeley.edu/pret/src/ptarm-1.0/>