Precision Timed Embedded Systems Using TickPAD Memory

Matthew M Y Kuo*, Partha S Roop†, Sidharta Andalam‡, Nitish Patel§
Department of Electrical and Computer Engineering, University of Auckland*†§
TUM CREATE, Singapore‡
mkuo005@aucklanduni.ac.nz*, p.roop@auckland.ac.nz†
sidharta.andalam@tum-create.edu.sg‡, nd.patel@auckland.ac.nz§

***Abstract*—There has been a great deal of impetus for the design of predictable memory such as scratchpad memory. Moreover, static analysis of cache memory is an area of intense research activity. However, there has been minimal development of caches or scratchpads (SPMs) that can exploit the inherent concurrency in synchronous languages. In this paper, we have developed a new memory subsystem, called the TPM (TickPAD memory), for predictable and efficient execution of synchronous programs. TPMs are a hybrid between conventional caches and SPMs: they support dynamic loading of cache lines (like caches) and static allocation of program points (like SPMs). However, the dynamic loading is predictably managed through statically loaded TPM commands. Extensive benchmarking comparing TPMs to SPMs and direct mapped caches reveal that TPM achieves 8.5% WCRT reduction compared to locked SPMs, 12.3% WCRT reduction compared to thread multiplexed SPM and 13.4% WCRT reduction compared to direct mapped caches, while also keeping the analysis time comparatively small.**

***Keywords*-Synchronous, Memory Hierarchy, Worst Case Analysis**

I. Introduction

Synchronous languages are gaining industrial acceptance as a paradigm of choice for the design of safety-critical systems, such as the recent A-380 control software [1]. Synchronous languages behave similarly to synchronous circuits, where computation is abstracted to complete instantaneously at each clock tick. In synchronous languages, clock ticks are discrete logical instances of time referred simply as *ticks*. During each *tick*, the program samples its inputs, computes the response and emits the corresponding outputs, also referred to as a *reaction*. The time it takes for synchronous programs to complete a *tick* is called the *reaction* time. For safety-critical systems, static computation of the longest tick length, also known as worst case reaction time (WCRT) is needed to map logical time to physical time. This is essential for deriving real-time implementations of synchronous programs where the reaction time of the program must meet design specifications. WCRT analysis has recently received a lot of research attention. Techniques range from max-plus [3], integer linear programming (ILP) [5], model checking [13] to a recently proposed approach based on graph reachability [6]. These approaches derive the WCRT, either by ignoring the impact of the underlying memory sub-system, or alternatively using direct mapped caches. However, memory architectures are one of the major bottlenecks, as they lead to wide variability of instruction execution times (depending on a cache-hit or a miss). In this paper, we focus on the memory architecture design and the associated static analysis, for concurrent synchronous programs, which has received scant research attention.

Table I
QUALITATIVE COMPARISON OF LOCAL MEMORIES

Memory	Suited for	Allocation	Analysis Time
Cache	Average Case	Dynamic	Abstract-Fast [17], [12] Tight-Slow [9]
SPM and Locked Cache	Worst Case	Static	Fast [11], [10], [15], [16]
TPM	Worst Case	Static+ Dynamic	Fast

Table I provides the trade-offs of the different memory architectures used for the design of real-time systems. The table shows, in general, there is a trade-off between the precision [9] (tightness of the estimate) and the scalability [17], [12], [9] (analysis time) for timing analysis of traditional cache memories.

To achieve timing predictability and to simplify static timing analysis, there has been a shift towards predictable memory hierarchy for hard real-time systems. One such approach is to use the cache locking mechanism [11]. In this approach, the contents of the cache are decided at compile time and are loaded prior to the execution of the program. This simplifies the analysis and provides a predictable platform. However, this significantly reduces the throughput. This simple approach has been further extended to dynamically reload and lock the cache at statically determined control points [10]. This increases the throughput at the expense of the analysis time. Another approach for caches in time predictable systems is the method cache proposed in the JOP architecture [15]. The method cache loads the entire method into the cache, before the execution of a method. Therefore, it is not possible to have any cache misses during the execution of the method. However, the approach assumes that all methods are smaller than the cache.

As an alternative, Scratchpad memories (SPMs) have been introduced for time predictable systems. In SPMs, the allocation and replacement decisions are made by software, guided by the decisions made at compile time. In contrast, the allocation and replacement of traditional caches are done dynamically, guided by the history of the cache states. Recent work on SPMs focus on developing software allocation algorithms [15], [16], [4] and/or designing tailored architectures with SPMs [15], [7].

To summarize, caches are allocated dynamically and provide better average case performance at the expense of very complex analysis of the worst case. Precise analysis techniques do not scale and the use of suitable abstractions is the key. However, this leads to a loss of precision during analysis. Locked caches and SPMs, in contrast, are allocated statically and are easier to analyze. However, they offer comparatively lower average and sometimes worst case performance compared to caches. This inspires us to propose a new hybrid approach to fill the gap between caches and SPMs. Our approach offers cache-like dynamic loading capability. However, this dynamic loading decisions are statically controlled enabling simpler analysis. Also, our approach uses static allocation of regions of the program, such as loops and other control points like the *ticks* for synchronous programs similar to SPMs. We follow the design philosophy of precision timed machines [8] to develop a predictable architecture to accelerate the execution of synchronous programs in a predictable manner. We have termed our specialized SPM controller for synchronous programs as the Tick Precise Allocation Device (TickPAD).

We have performed extensive benchmarking to compare the TPM objectively with SPMs and direct mapped caches. The results show that the TPM achieves a 8.5% WCRT reduction compared to locked SPM [16], 12.3% WCRT reduction compared to thread multiplexed SPM [16] and 13.4% WCRT reduction compared to direct mapped caches [12], [17]. These reductions are achieved while keeping analysis time comparatively smaller. Also, it was observed that the reduction in WCRT was much sharper compared to both direct mapped caches and SPMs as the size of the memory was increased (see Fig. 14 in the results section). To the best of our knowledge, the proposed approach illustrates, for the first time, a scalable technique for designing a predictable memory hierarchy for complex concurrent programs (unlike the only known SPM for concurrent programs that takes simple message sequence chart descriptions [16]).

The paper is organized as follows: Sec. II presents an overview of our approach using a motivating example. Sec. III presents the architecture of the TPM. Sec. IV illustrates the static allocation of the TPM. Sec. V presents the static timing analysis of the TPM. Benchmarking results of the TPM are presented in Sec. VI and concluding remarks in Sec. VII.

II. Overview

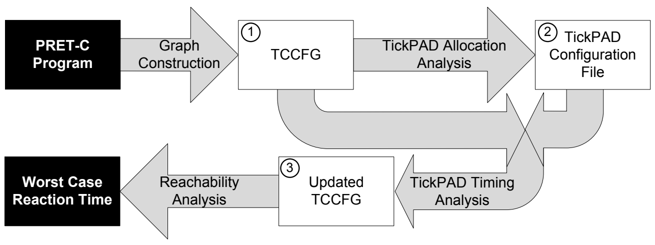

Figure 1. TickPAD design flow

We illustrate the design flow for the TPM architecture using a synchronous language called PRET-C [13]. PRET-C is a light weighted multi-threading extension to the standard C language. PRET-C extends C by introducing a minimal set of synchronous extensions. We focus on the `EOT` and the `PAR` synchronous statements. The `PAR` statement spawns concurrently executing threads of fixed priority based on

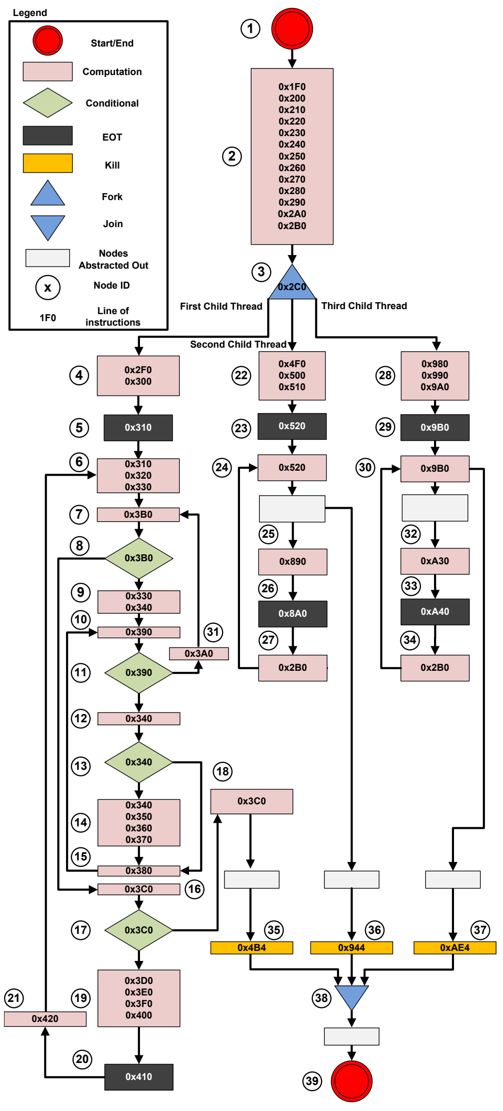

Figure 2. Abstracted TCCFG of a CCD line following robot

their order of declaration. The `EOT` statement marks the end of the *tick* of a thread, a *local tick*. *Local ticks* are context-switching points where the current thread pause execution and context switch to the next thread in the schedule. A *reaction* is defined when all threads of a program reaches their respective local ticks, also referred as a *global tick*.

Fig. 1 shows the design flow for executing hard real-time systems using the proposed TPM architecture. Step 1, the PRET-C program is compiled and analyzed to create the Timed Concurrent Control Flow Graph (TCCFG). The TCCFG is an intermediate format which describes the concurrent control flow and context switching points of a synchronous program. Fig. 2 presents a readable abstraction of a TCCFG similar to the one presented in [6]. The numbers beside each node represent the node IDs. TCCFGs have unique `Start` and `End` nodes to delimit the graph. `Computation` nodes corresponds to basic blocks where there are only a single execution path (linear code). `Condition` nodes correspond to branch statements. `FORK` nodes describes the concurrent behavior of the program. An outgoing transition from a FORK node represents the current thread (parent thread) spawning another thread (child thread). The thread priority of the child threads goes from left (highest priority) to right (lowest priority). When a thread spawns other threads, its execution pauses until all child threads terminate. A KILL node represents the thread termination. JOIN nodes marks the point where parent thread resumes execution once all child threads have terminated.

Nodes with hexadecimal numbers inside them are mapped to program instructions. These numbers represents the base addresses of the instructions fetched from main memory when the node is executed. For example, the number 0x1F0 in node 2 implies that when node 2 is reached, the instructions with the base address 0x1F0 is fetched. For the running examples in this paper, a burst transfer of four instructions is assumed. This means that when the program execution reaches node 2, the instructions 0x1F0, 0x1F4, 0x1F8 and 0x1FC are fetched. This implies that node 2 is maps to the instructions on the addresses 0x1F0, 0x1F4, 0x1F8 and/or 0x1FC. The figure has abstracted out the instruction execution costs of each node. The TCCFG at this point does not include the memory access costs. Step 2, the TCCFG is used to determine the static allocation decisions for the TPM. Step 3, the memory access cost of the TPM is modeled and mapped back onto the original TCCFG. To obtain the WCRT of the program, the updated TCCFG is analyzed using the reachability approach presented in [6].

In this paper, we have designed a TPM simulator by extending the SMMU MicroBlaze simulator [18] and we also described the TPM using VHDL and synthesized it on a FPGA. For the running example, the TPM architecture is assumed to have a main memory burst transfer cost of 38 clock cycles and each burst transfer fetches four 32 bit instructions. These four instructions correspond to a TPM line. Thus, each TPM line is represented by a base address (in hex) of a block of 4 instructions. For example, each hex number in Fig. 2 corresponds to a TPM address.. As we are focusing on the instruction memory, we assume the on-chip memory is large enough to store the program data. For ease of understanding, the execution time of each instruction is exactly 2 clock cycles for the examples presented in this paper. In the next section, we will begin by introducing the TPM architecture.

III. Tick Precise Allocation Device

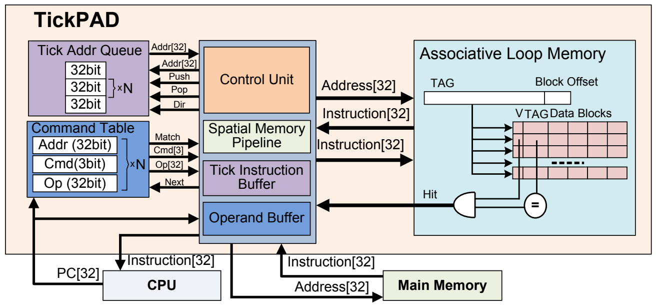

Figure 3. TickPAD Memory Architecture

Fig. 3 presents a high level architecture diagram of the TPM. The TPM is a tailored memory controller for predictable and efficient execution of synchronous programs. TPMs are designed to provide an alternative to SPMs and speculative caches for hard-real time systems. The TPM manipulates instruction memory at the granularity of a TPM line, to fully take advantage of reduced overall memory access time of burst transfers (via spatial locality). Each time an instruction is requested from main memory, every instruction on the TPM line is fetched and stored on the TPM. Therefore, every memory element in the TPM is a multiple of the TPM line, and buffers are exactly one TPM line in length.

The TPM contains four main types of memory to reduce the memory access cost for different instruction memory access patterns of a synchronous program. First, the `spatial memory pipeline` is designed to reduce the memory access cost when the processor is executing linear code (single execution path). When the memory bus is idle, the `spatial memory pipeline` prefetches the next TPM line as the processor executes the instructions stored on the current TPM line. Second, the `tick address queue` and `tick instruction buffer` are designed to reduce the memory access overheads of context switching points (FORK, EOT and JOIN nodes). The `tick address queue` maintains a list of the resumption addresses of currently active threads according to the order they are scheduled. When execution of the current thread reaches a context switching point, the `tick instruction buffer` is preloaded with the instructions from the next active thread designated by the `tick address queue`. Third, the `associative loop memory` is designed to reduce the memory access cost of loops by exploiting temporal locality. Before executing each loop, the `associative loop memory` is loaded with statically determined instructions from the loop, to reduce the WCRT.

In order for the TPM determine when to load each memory element during run-time, a `command table` is implemented. The `command table` instructs the TPM when to load each memory element using special commands.

During processor execution, the TPM has two tasks. Firstly, the TPM checks if the requested instructions are available locally so that they can be provided to the processor without main memory access. To determine if these instructions are available on the TPM, tag bits are implemented alongside all TPM memory elements. Similar to caches, by comparing the tag bits with the processor request, the TPM is able to determine if these instructions are available locally. If the instructions are not stored locally, the TPM has to fetch these instructions from main memory. Secondly, the TPM the program counter (PC) values provided by the processor (during the instruction fetch stage) against each entry of the `command table` to determine which memory elements are to be preloaded.

A. The Spatial Memory Pipeline

The `spatial memory pipeline` is designed to exploit spatial locality by prefetching the next TPM line in a deterministic manner. The `spatial memory pipeline` consists of two stages: the fetch stage and the execution stage. The fetch stage prefetches the next TPM line from main memory as the processor executes the instructions stored on the execution stage of the pipeline. To ensure determinism, the `spatial memory pipeline` only prefetches the next TPM line, when the instructions on the execution stage of the pipeline does not contain branch instructions. To avoid main memory bus contention, the `spatial memory pipeline` is only enabled when there are no other TPM components accessing the main memory bus (when the bus is idle).

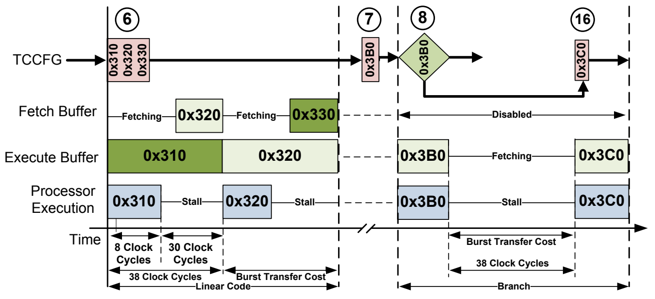

Figure 4. Execution of Spatial Memory Pipeline

Fig. 4 presents an example to illustrate the execution of the `spatial memory pipeline` as the processor executes instructions on nodes 6, 7, 8 and 16 on the TCCFG in Fig 2. As the processor executes the instructions on the execute state (TPM line 0x310 in node 6), the TPM begins to prefetch line 0x320 in parallel onto the fetch stage of the `spatial memory pipeline`. Since, node 6 is a basic block, there is only one execution path (the linear code), the `spatial memory pipeline` continues to prefetch TPM line 0x330 in parallel to the execution of instructions on line 0x320. When the processor execution reaches node 8, the TPM detects a branch instruction and disables the `spatial memory pipeline` until linear execution is restored. Once the branch is resolved, the `spatial memory pipeline` directly fetches the requested instructions onto the execution stage of the pipeline to avoid additional stall. For example, line 0x3C0 is directly fetched onto the execution stage once the branch is resolved. The figure shows that following a branch instruction a longer stall time is observed. This stall time will also be observed during TPM startup when the `spatial memory pipeline` is empty. This is equivalent to a cache miss.

The `spatial memory pipeline` memory access cost (processor stall times) is defined using Equations 1a and 1b. Equation 1a defines the memory access cost of the `spatial memory pipeline` when the processor executes linear code (basic blocks). The memory access cost of the TPM line ($Access_{cost}^{line}$) equals to the burst transfer cost ($Burst_{cost}$) minus the instruction execution cost ($InsExec_{cost}$). For example, in Fig. 4, line 0x310 takes 8 clock cycles to execute, therefore, the memory access cost of line 0x320 is 30 clock cycles ($38-8=30$). Equation 1b defines the memory access cost when the `spatial memory pipeline` is empty, for example during startup or after a branch instruction. The memory access cost equals to the burst transfer cost ($Burst_{cost}$). For example, 38 clock cycles for node 16 in Fig. 4.

$$Access_{cost}^{line} = max(Burst_{cost} - InsExec_{cost}, 0) \tag{1a}$$

$$Access_{cost}^{line} = Burst_{cost} \tag{1b}$$

Table II
TICKPAD COMMANDS AND THEIR INFORMAL SEMANTICS

Command	**Command Description**	**Operand**
0.Discard	Invalidates the Associative Loop Memory.	N/A
1.Store	Loads Associative Loop Memory with new instructions.	Start and End Address.
2.Fill Tick Instruction Buffer	Load the instructions from the address specified at the front of the Tick Address Queue onto the Tick Instruction Buffer.	N/A
3.Load Tick Address Queue	Loads an address onto the Tick Address Queue.	Resumption address

The `spatial memory pipeline` architecture is presented in Fig. 5. The `spatial memory pipeline` is constructed using buffers that are interleaved to represent the fetch and execution stages of the pipeline respectively. A toggle unit is used to control which buffer is currently the fetch stage of the pipeline so that it could be write enabled (representing the fetch stage). The use of alternating buffers eliminates the need for data propagation across stages of the pipeline. To detect branch instructions, the branch detector compares the high bits of instructions to determine if a branch instructions exists in the fetched TPM line. If a branch instruction is detected the prefetching of the `spatial memory pipeline` is disabled.

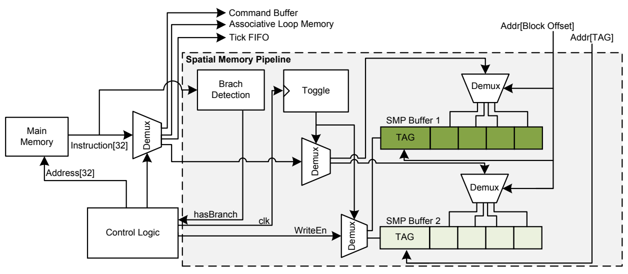

Figure 5. Spatial Memory Pipeline Architecture

B. The Command Table

The `command table` contains a set of commands used to dynamically instruct how the TPM is loaded. The set of commands stored on the `command table` are program specific and are determined though static analysis. The `command table` is implemented using a lookup table. Each entry in the `command table` has three fields: the address field, the command field and the operand field. During runtime, the TPM compares the PC value from the processor against the address fields of the `command table`. The address field is the index to the `command table` and is used to specify which TPM command to execute. When the PC matches a value stored on the address field, the TPM command stored on command field is executed. The operand part of single operand commands are directly stored in the command table while those for multi-operand commands are stored in main memory. Operands of multi-operand commands are fetched onto the `operand Buffer` before executing the TPM command. By using the PC as an index to execute TPM commands, memory operations of the TPM are synchronized with processor execution.This approach allows the TPM to be loaded without modifying to the source code. In contrast to SPMs, where modifications to the source code are required to loads its contents. This is particularly useful for large programs where the source code may not always be available (e.g. library files) or modification to the source code is undesirable.

A summery of the TPM commands along with its informal semantics are presented in Table II. The TPM has four commands: *discard*, *store*, *fill tick instruction buffer* and *load tick address queue*. The *discard* command is used to reset and invalidate the contents of the `associative loop memory`. The *store* command loads the `associative loop memory` with specified instructions from main memory. These instructions are specified using two operands, the start address and the end address of a contiguous block of instructions. The *fill tick instruction buffer* command is used to load the `tick instruction buffer` with the first TPM line of the next active thread. This TPM line is determined by reading the front of the `tick address queue`. The *load tick address queue* command is used to maintain a list of the resumption addresses of active threads. During static allocation of the `command table`, the operand field of the *load tick address queue* command is loaded with the resumptions addresses of each thread. During run-time the command loads the address stored in the operand field onto the `tick address queue`. In-depth details on the `tick instruction buffer`, `tick address queue` and `associative loop memory` are subsequently presented. The allocation of the `command table` is presented in Sec. IV.

C. The Tick Address Queue and Tick Instruction Buffer

Synchronous programs execute concurrent threads by rapidly context switching between them to emulate concurrency. During context switching, the PC is changed to the resumption address of the next active thread. This sudden change in PC may result in a cache miss in traditional cache architectures. This abrupt change in PC is, however, is determinable though static analysis of the TCCFG. For synchronous programs, both the location of context switching and the target destination can be determined at compile time by analyzing the context switching points (`EOT`, `FORK` and `KILL` nodes). The `tick address queue` and `tick instruction buffer` exploits this fact, to provide predictive reduction in memory access time for context switching points. To reduce the memory access time, three steps are required. Firstly, the TPM must have knowledge of the schedule and resumption addresses of the synchronous threads. This is achieved by the `tick address queue`. The `tick address queue` is a priority queue that stores the resumption address of the active threads according to their execution order. The address of the next active thread is stored on the front of the queue followed by the address of the subsequent threads. Secondly, the TPM must preload the instructions of the next active thread before context switching. This is achieved by the `tick instruction buffer`. During runtime, the TPM determines which instructions to fetch for the next active thread by reading the front of the `tick address queue`. Next, the TPM prefetches the first TPM line of instructions onto the `tick instruction buffer`, prior to context switching. Therefore, when the next thread becomes active after context switching, the instructions are already present in the TPM. Finally, the TPM must continuously maintain this reduced memory access cost after the instructions on the `tick instruction buffer` are executed. This is achieved by the `spatial memory pipeline`. When the thread begins executing the instructions stored on the `tick instruction buffer`, the `spatial memory pipeline` begins to fetch the next TPM line in parallel. This is similar to execution of linear code. Thus, in terms of memory access cost the execution overhead associated with context switching points is reduced.

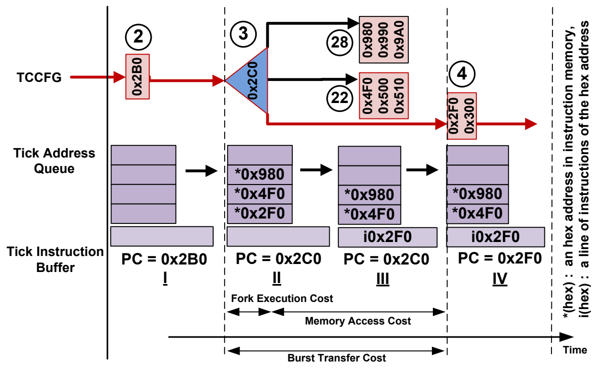

Figure 6. Illustration of the Tick Address Queue and Tick Instruction Buffer

Fig. 6 demonstrates an example of context switching of a `FORK` node where execution switches from the parent thread to the first child thread. Starting from Fig. 6-I, the PC is at 0x2B0 (node 2 of Fig 2). At this point the `tick address queue` and the `tick instruction buffer` are both empty. Next, the execution reaches the PC value 0x2C0, the `FORK` node (Fig. 6-II). At this point the TPM loads the stating address of the three child threads onto `tick address queue`. The order the addresses stored in the `tick address queue` follows the priority of the child threads. For example, the front of the `tick address queue` stores the address *0x2F0 (node 4), followed by *0x4F0 (node 22) and *0x980 (node 28). Once the `tick address queue` is loaded, the `tick instruction buffer` begins to preload the TPM line of the next active thread (i2F0) by reading the address at the front of the `tick address queue` (Fig. 6-III). Finally, when the first child thread begins to execute (node 4) the instructions (TPM line) is already present in the TPM (Fig. 6-IV). Although, in this paper PRET-C is used as the running example, the `tick address queue` and the `tick instruction buffer` can also be used for other concurrent languages such as Esterel [2].

The memory access costs of the context switching points is defined using the same equations as the `spatial memory pipeline` (Equation 1a). The memory access cost equals to the burst transfer cost ($Burst_{cost}$) minus the instruction execution cost ($InsExec_{cost}$) of the context switching node. For example, Fig. 6 shows the memory access cost for the `FORK` context switching point is the burst transfer cost minus the execution cost of the `FORK` node.

Fig. 7 presents an implementation of the `tick address queue` using a cyclic buffer. The hexadecimal numbers in the

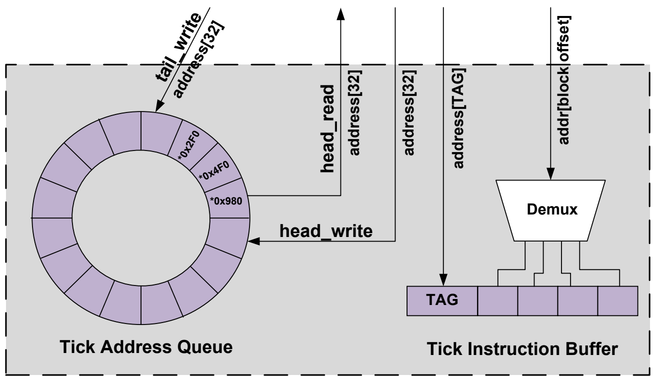

Figure 7. Tick Address Queue and Tick Instruction Buffer Architecture

figure are the same as Fig. 6-II. The resumption address of the next active thread is stored at the head (*head_read*) of the cyclic buffer. The tail (*tail_write*) points to the next free space in the queue. An extra pointer (*head_write*) which points to the empty space in front of the queue. The *head_write* pointer is used for nested threads (when a child thread spawns more threads). Newly spawned threads of nested threads are inserted into the schedule to be executed next. To correctly maintain the thread schedule, the `tick address queue` inserts the resumption addresses of these threads into the front of the `tick address queue` using the *head_write* pointer. The architecture of the `tick instruction buffer` is the same as a stage buffer of the `spatial memory pipeline`.

D. The Associative Loop Memory

Temporal locality is important to ensure a high performance memory architecture. Cache hits on subsequent iterations of a loop can reduce the memory access costs significantly. The `associative loop memory` is a fully associative memory designed to exploit temporal locality by preloading the instructions from loops between two local *tick* boundaries. The `associative loop memory` is statically allocated and during run-time the processor is stalled until all the allocated instructions are loaded. This ensures simple and tight analysis with predictable performance gain from temporal locality. Unlike caches, where the memory access cost depend on the previous iterations of the loop. Therefore, requiring complex analysis to ensure a tight bound on the execution time.

The `associative loop memory` executes in three steps. First, when execution reaches a loop, the TPM discards the previous contents of the `associative loop memory`. Second, the TPM loads the statically allocated TPM lines of the loop onto the `associative loop memory`. Finally, the contents of the `associative loop memory` are locked until another loop is encountered. Sec. IV presents the allocation of the `associative loop memory`.

Equation 2 defines the memory access cost to load a loop onto the `associative loop memory`. The memory access cost is the sum of the number of *store* commands (i) needed to load the loop. Each store command requires the memory access cost of a burst transfer ($Burst_{cost}$) multiplied by the number of TPM lines between the specified start and end

Table III
COMMAND TABLE ALLOCATION

Node	Command	Operand
1.Fork	Load Tick Address Queue$\times threads$	Child thread start addresses
	Fill Tick Instruction Buffer	
2.EOT	Load Tick Address Queue	Thread resumption address
	Fill Tick Instruction Buffer	
3.Kill	Fill Tick Instruction Buffer	N/A
4.Loop	Discard	Start & end addresses
	Store	

addresses plus the *store* command overhead ($Lines + 1$). The overhead is the cost to fetch the operands (the start address and the end address) from main memory.

$$Access_{cost}^{loop} = \sum_{i=0}^{cmds} [Burst_{cost} \times (Lines + 1)]_i \qquad (2)$$

IV. TICKPAD ALLOCATION

The `command table` of the TPM has to be statically allocated and loaded before using the TickPAD. To load the `command table` with correct commands, a TickPAD configuration file has to be generated (step 2 in Fig 1). The TickPAD configuration file is a textual file where each line directly maps to a `command table` entry.

To generate the TickPAD configuration file, firstly, the control points need to be determined. Control points are the program context switching points (`FORK`, `EOT` and `KILL` node) and the loop entry points between local ticks. To determine the control points the TCCFG transversed by analyzing each node. Secondly, the instructions to be allocated onto the `associative loop memory` are to be determined. This paper proposes a greedy heuristic to determine which TPM lines are to be loaded onto the `associative loop memory`. The heuristic first selects the TPM lines from the inner most nested loop, followed by other instructions from the same basic blocks, until the `associative loop memory` is fully utilized. TPM lines from nested loops are prioritized because they are likely to be visited more frequently than other instructions. Therefore, nested loops may reduce the memory access cost the most. Thirdly, the commands for the command field are to be decided. The commands for the command field are allocated based on the previously identified control points. These control points are the locations where the TPM loads the `tick address queue`, the `tick instruction buffer` or the `associative loop memory`. Therefore, the values stored in the address field (the address to execute the TPM commands) are the address of the control points. For example, the address field on the `command table` associated with node 3 of Fig 2 will contain the value 0x2C0.

Table III presents a summary for the allocation of the command field and the operand field for each type of control point.

1) The behavioral of `FORK` nodes is to spawn multiple child threads into the schedule followed by context switching to the first child thread. For the TPM to maintain the resumption addresses of active threads, a

load tick address queue command is required to store the starting address for each child thread onto the `tick address queue`. For example, a `FORK` node spawning five threads will require five *load tick address queue* commands. To reduce the context switching time of the `FORK` node, a *fill tick instruction buffer* is required to prefetch the instructions from the first child thread onto the `tick instruction buffer`.

2) `EOT` nodes pauses the execution of the current thread and resumes the execution of the next active thread. For the TPM to keep track of the location where the current thread will resume next, a *load tick address queue* command is used to store the resumption address of the current thread onto the `tick address queue`. To reduce the context switching time to the next active thread, a *fill tick instruction buffer* command is used to prefetch the instructions of the next active thread.
3) `KILL` nodes removes the current thread from the schedule and then context switches to the next active thread. To reduce the context switching a *fill tick instruction buffer* command is used.
4) Loops require a *discard* command to reset the `associative loop memory`, followed by, a set of *store* commands depending on the number of contiguous blocks of instruction allocated to the `associative loop memory` by the previously described greedy heuristic.

V. Timing Analysis

This section presents the static timing analysis of the TPM. First, the memory access cost of the TPM and main memory is computed using a breath first transversal algorithm (presented in Sec, V-A). Next, the memory access costs of each node of the TCCFG is added onto the instruction execution cost of the original TCCFG. Finally, the reachability approach is used to compute the WCRT. This approach has been shown in [6] to be more efficient than existing approaches for synchronous programs.

A. TickPAD Allocation Analysis

Algorithm 1 presents a breath first transversal of the TCCFG to compute the memory access cost for the TPM architecture (step 3 in Fig 1). The algorithm takes two inputs, namely the TCCFGSep and the TPConfig. TCCFGSep a TCCFG where each TPM line is placed in a separate node. Sec. V-B presents how the TCCFG is translated to TCCFGSep. The TPConfig is the TickPAD configuration file. The TickPAD configuration file contains the static allocation decisions of the TickPAD. For example, which instructions are allocated on the `associative loop memory`. The output is a cost updated TCCFG (CostUpdatedTCCFG), which includes the TPM's memory access cost.

The memory access times of TickPAD depend on the previous TPM line (prefetching of the `spatial memory pipeline`) and the static memory allocations. This is unlike caches, where the memory access time is dependent on the complete history of execution. Therefore, to compute the memory access cost of the TPM without any abstraction (no overestimations), only the previous and current TPM lines needs to be considered. The current line is a node in TCCFGSep and the previous lines are the source nodes of this particular node. The algorithm stores the computed memory access costs onto the source transitions. Memory access costs are stored on transitions to preserve the instruction execution costs of each blocks during analysis. A transition from a source node to a destination node is denoted as $trans^{(source\ node)}_{(destination\ node)}$ and the costs on this transition is denoted as ($trans^{(source\ node)}_{(destination\ node)}$.cost). These costs are later remapped onto the cost updated TCCFG as empty nodes with timing costs. To model the instructions stored on the TPM, all nodes have a flag called 'cached' and is denoted as $(node).cached$. The cached flag indicates if the TPM line of the node is absent or present in the TPM.

Algorithm 1 Update Cost of TCCFG

```
Require: $TCCFGSep, TPConfig$
Ensure: $CostUpdatedTCCFG$
 1: $visited = \emptyset$
 2: $pending = \{TCCFGSep.startNode\}$
 3: while $pending \neq \emptyset$ do
 4:     $n = getNextNode(pending)$
 5:     if $n \cap visited \neq \emptyset$ then
 6:         continue
 7:     end if
 8:     $visited = visited \cup n$
 9:     if $TPConfig.isPreloaded(n)$ then
10:         $n.cached = true$
11:     else if $n.cached = false$ then
12:         $n.cached = true$
13:         $trans_n^{src(n)}.cost{+} = buscost$
14:     end if
15:     for $destNode : n.dest$ do
16:         $destNode.cached = false$
17:         $cmds = TPConfig.get(destNode)$
18:         if $cmds \neq \emptyset$ then
19:             for $cmd : cmds$ do
20:                 if $cmd = ContextSwitchCommand$ then
21:                     $trans^n_{destNode}.cost{+}{=}max(buscost - n.cost, 0)$
22:                 else
23:                     $trans^n_{destNode}.cost{+}{=}buscost\times\ (cmd.lines{+}1)$
24:                 end if
25:             end for
26:         else if $destNode.line = \emptyset$ then
27:             $destNode.cached = true$
28:         else if $TPConfig.isPreloaded(destNode)$ then
29:             {no statement}
30:         else
31:             if $destNode.line = n.line$ then
32:                 $destNode.cached = true$
33:             else if $containsBranch(n){=}false$ then
34:                 $destNode.cached = true$
35:                 $trans^n_{destNode}.cost{+}{=}max(buscost - n.cost, 0)$
36:             else
37:                 $destNode.cached = true$
38:                 $trans^n_{destNode}.cost{+}{=}buscost$
39:             end if
40:         end if
41:     end for
42: end while
43: return $translate(TCCFGSep)$
```

Lines 9 to 10 determine if the TPM line of the node are statically allocated on the `command table`. Nodes that are statically allocated to the TPM do not have memory access costs. The memory access costs to preload these instructions are associated with the nodes that preload them. Lines 11 to 13 compute the cost to fetch instructions onto the `spatial memory pipeline` when the TPM is empty (for example, during start up or after a branch statement). Line 13 shows that the cost penalty equals to the cost of a burst transfer (Equation 1b).

The memory access cost of the TPM can be dependent on the source node due to prefetching. Therefore, from line 15 onwards, the algorithm processes destination nodes (destNode) instead of the current node (n), the current node being the source node. Line 16 resets the cached state of each destination node to compute the different memory access cost for each source node. Line 17 determines the commands associated to the addresses of each destination node with memory access costs. Commands that have memory access costs are the *fill Tick Instruction Buffer* command (context switching commands) and the *store* command.

Lines 18 to 24 compute the memory access costs of TPM commands. Line 19 contains a loop because there can be multiple commands that are matched to the same node (same TPM line). For example, there can be multiple store commands for a segmented loop. However, there can only be one command to access the `tick instruction buffer` per node because there is at most one context switching point per TCCFG node. Line 20 checks if the command is a context switching command. In this event, line 21 calculates the memory access cost according to Equation 1a. If the command is a *store* command, the memory access cost is computed using Equation 2 (Line 22- 23). Lines 26 to 27 compute the memory access costs of destination nodes with no instructions, such as, `JOIN` nodes. If there are no instructions in the destination node, the node is set as cached. Lines 28 to 29 compute the memory access costs of nodes that are already preloaded. Since the node will be preloaded nothing needs to be assigned. Lines 31 to 32 compute the memory access costs of nodes where the TPM line of the current node (n.line) matches the TPM line of the destination node (destNode.line). In this case, the instruction is already fetched onto TPM by the source node. Therefore, there are no timing penalties and the state is set as cached. Lines 33 to 35 computes the memory access costs of node that are to be prefetched by the `spatial memory pipeline` during execution of linear code. The cost of the `spatial memory pipeline` is computed according to Equation 1a. Lines 36 to 38 models the memory access costs when a branch instruction is detected. The cost to fetch the next line of instructions after a branch instruction follows Equation 1b.

Once all nodes in the TCCFG are transversed, line 43 translates the costs on the source transitions to empty nodes with timing costs (see Section V-B2) and returns the CostUpdatedTCCFG. To obtain the `WCRT`, the CostUpdatedTCCFG can be passed to existing timing analysis tools such as reachability [6].

B. TCCFG Transformation

Two transformations are performed on the TCCFG for the TPM architecture. First transformation is needed to construct the TCCFGSep where each TPM line is in a separate node. Second transformation is needed to map the memory costs stored on the transitions back onto nodes of the TCCFG.

1) Construction of TCCFGSep: Fig. 8 illustrates the process starting from the constructing of the TCCFG from binary to the construction of TCCFGSep from an existing TCCFG. Fig. 8a shows a snippet of assembly instructions reconstructed from the program binary. To construct the TCCFG, each instruction is mapped to one of the TCCFG block types (Fig. 8b). Next, basic blocks on the same TPM line are merged together (Fig. 8c). For example, instructions on the addresses 0xA60, 0xA64, 0xA68 and 0xA6c are

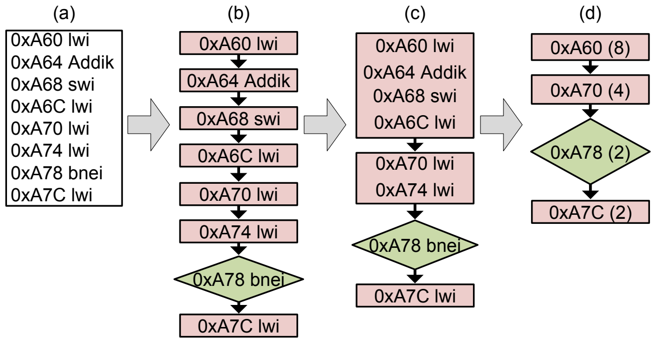

Figure 8. Transformation of Basic Blocks

merged to a single basic block because they all belong to TPM line 0xA60. However, address 0xA78 is not merged with 0xA70 and 0xA74 because it is a conditional statement. Once all the blocks are merged, the TPM line and the instruction execution cost are stored on the TCCFG nodes (Fig. 8d). Here, 0xA60 (8) represents a node containing instructions belonging to TPM line 0xA60 and has a execution cost of 8 clock cycles.

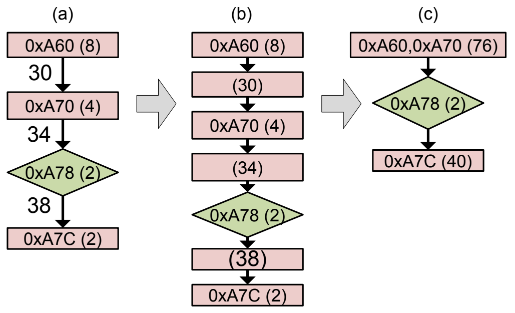

Figure 9. Cost mapping

2) Cost Mapping: During static timing analysis, the memory access costs are temporally stored on the transitions of the TCCFG. Fig. 9 shows how the costs are mapped back to the nodes of the TCCFG. Fig. 9a shows a TCCFG with the memory access cost stored on the transitions. First, empty nodes are created at source node with only the memory access costs fmrom the transitions (Fig. 9b). These blocks represent the TPM memory access costs of the destination block. Next, to reduce the state space for timing analysis, basic blocks are merged together (Fig. 9c).

VI. Results

In this section, we first present hardware synthesis results for the TPM architecture followed by experiments comparing the WCRT and the WCRT analysis time for the proposed TPM architecture against SPMs (locked SPM and thread multiplexed). Locked SPMs are allocated at compile time and this allocation does not change for the entire duration of the program's execution. Thread multiplexed SPMs are statically allocated and dynamically loaded at specific program points. For example, loading specific instructions of a thread before its execution. The contents of SPMs are selected using the ILP based selection algorithm presented in [16]. We have also compared with direct mapped caches,

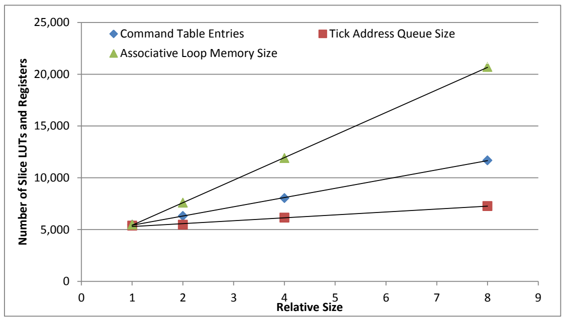

Figure 10. LUT and Register Utilization vs TickPAD Size

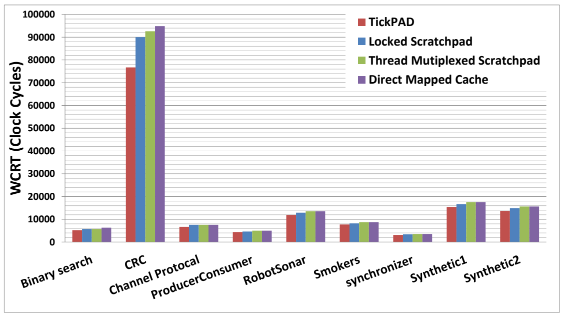

Figure 11. Worst Case Reaction Time

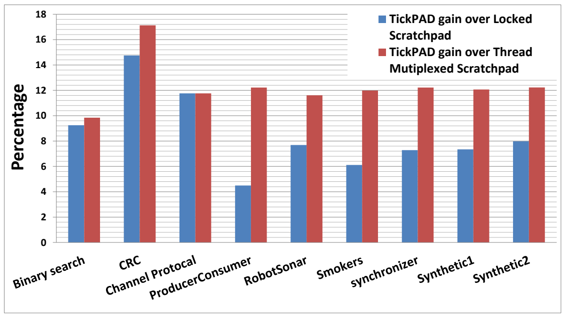

Figure 12. WCRT gain of TickPAD vs Scratchpad

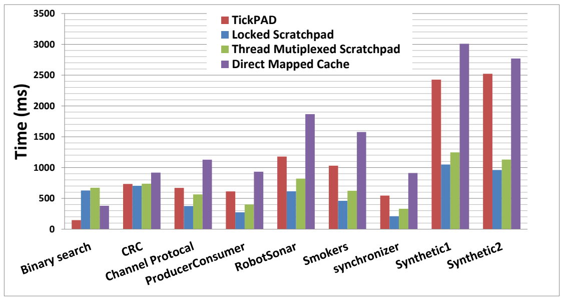

Figure 13. Analysis time

where the worst case misses in each basic block is analyzed using the well known Absint approach [17], [12].

For our experiments, we have selected a set of benchmarks based on the examples presented in [13]. In addition, we have also included two computation intensive benchmarks based on the binary search and the CRC algorithms [14]. The size of on-chip memory is set at 5% of the program size, unless otherwise specified. The benchmarking is done

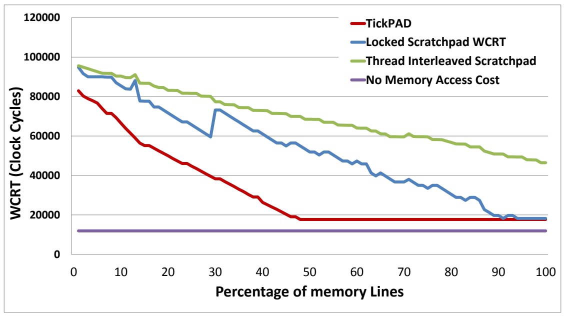

Figure 14. Effect on WCRT as memory size increases for the CRC example

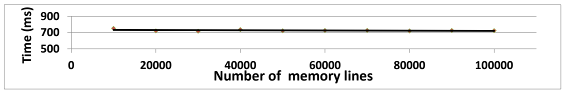

Figure 15. Effect on analysis time for TickPAD as memory size increases for the CRC example

on a Intel Core i7 920, executing the Windows platform.

Fig. 10 shows the resource utilization of the TPM architecture. The x-axis is the relative size of the TPM (the multiplier). The baseline parameters of the TickPAD (value of x-axis equals to one) is 4 spaces in the `tick address queue`, 8 `command table` entries and 128 Bytes of memory in the `associative loop memory`. A value of two in the x-axis represents the parameters of 8 spaces in the `tick address queue`, 16 entries in the `command table` and 256 Bytes of memory in the `associative loop memory` etc. The results show that, as each TPM component increases, the LUT and register utilizations increase linearly. This shows that the TPM is scalable because increasing the parameters of the TPM has little overhead for all three types of memory elements.

Fig. 11 presents the computed `WCRT` of the TPM, the SPM and the direct mapped cache. For all example the TPM compute a smaller `WCRT` compared to SPM and direct mapped caches. The TPM has an average `WCRT` reduction of 8.5% compared to locked scratchpad memory and 12.3% compared to thread multiplexing of scratchpad memories, highlighted in Fig. 12. The TPM has a smaller `WCRT` value because the `spatial memory pipeline` fully utilizes the main memory bus, resulting in smaller bus idle times compared to SPMs and direct mapped caches. In addition, for the direct mapped cache, the Absint approach [17], [12] introduces an abstraction that makes the approach scalable while some precision is lost. A smaller `WCRT` for direct mapped caches maybe possible using more precise analysis approaches, however, this is at the expense of considerable analysis time.

Fig. 13 presents the analysis time required to model the memory behavior of the TPM, SPM and the direct mapped cache. For all benchmarks the analysis time for TPM lies between the other approaches. Compared to SPMs, the TPM has a longer analysis time because static analysis needs to compute the memory access time of the dynamic features. Compared to direct mapped caches, the TPM require less analysis time because timing analysis of the TPM only require analysis of the previous and current TPM lines. Where cache analysis, may require analysis of multiple previous cache lines to determine the cache state.

Fig. 14 presents the relationship between the `WCRT` and memory size for the TPM and the SPM. As the memory increases, the `WCRT` for both approaches decreases and eventually reaches a steady state value. For the TPM, this steady state value is reached when all loops between local ticks fit on the memory. For SPM, the stead state value is reached when the allocation algorithm allocates all the instructions. The graph shows that the `WCRT` for TPMs reduces more sharply compared to both SPM techniques because only the instructions in the loop are stored in the `associative loop memory`. The thread multiplexed SPM has a higher steady state value due to the thread reloading overhead. The spikes for the SPM occur because the allocation algorithm [16] does not take synchronous state alignment into consideration.

Fig. 15 presents the relationship between the analysis time and memory size increase for the TPM. The analysis time for the TPM is mostly constant. This is because the complexity of modeling the TPM is proportional to the size of the program.

The TPM offers a very scalable approach for the design of predictable memory hierarchy for embedded systems. The above results show that the analysis time for TPMs are similar to SPM and lower compared to direct mapped caches, and at the same time the TPMs offers much better WCRT values.

VII. Conclusions

Design of real-time systems must rely on suitable memory sub-system that is comparatively simple to analyze while providing good worst case performance. So far, the memory sub-system is either designed using direct mapped caches or SPMs. Caches offer better performance at the expense of complex and less precise static analysis. SPMs offer simpler analysis at the expense of lower performance compared to caches. Also, SPMs and caches have, so far, not been designed to directly execute concurrent programs.

In this paper we propose, for the first time, a predictable hybrid architecture called TPM that exploits the benefits of both approaches without sacrificing predictability. TPMs offer better worst case performance compared to SPMs and caches while keeping analysis time better than caches and slightly inferior to SPMs. Also, TPMs for the first time, exploit the features of concurrent programming for the acceleration of such programs. In particular, TPMs have been designed for exploiting the synchronous concurrency. We believe that TPMs can be used with minor customizations for the execution of other types of concurrent/parallel programs and this will be one of the avenues for our future investigation.

References

[1] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91*, 1 (January 2003), 64–83.

[2] Berry, G., and Gonthier, G. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program. 19*, 2 (Nov. 1992), 87–152.

[3] Boldt, M., Traulsen, C., and von Hanxleden, R. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science 203*, 4 (June 2008), 65–79.

[4] Falk, H., and Kleinsorge, J. Optimal static wcet-aware scratchpad allocation of program code. In *Design Automation Conference, 2009. DAC 2009. 46th ACM/IEEE* (July 2009), pp. 732 –737.

[5] Ju, L., Huynh, B. K., Roychoudhury, A., and Chakraborty, S. Performance debugging of Esterel specifications. *Real-Time Systems 48*, 5 (2012), 570–600.

[6] Kuo, M., Sinha, R., and Roop, P. Efficient WCRT analysis of synchronous programs using reachability. In *Proceedings of the 48th Design Automation Conference* (2011), DAC 2011, ACM, pp. 480–485.

[7] Lickly, B., Liu, I., Kim, S., Patel, H. D., Edwards, S. A., and Lee, E. A. Predictable programming on a precision timed architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems (CASES), Atlanta, Georgia* (October 2008), E. R. Altman, Ed., ACM, pp. 137–146.

[8] Liu, I., Reineke, J., Broman, D., Zimmer, M., and Lee, E. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on* (2012), pp. 87–93.

[9] Negi, H. S., Mitra, T., and Roychoudhury, A. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (New York, NY, USA, 2003), CODES+ISSS '03, ACM, pp. 201–206.

[10] Puaut, I. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Real-Time Systems, 2006. 18th Euromicro Conference on* (2006), pp. 10 pp.–226.

[11] Puaut, I., and Pais, C. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, automation and test in Europe* (San Jose, CA, USA, 2007), DATE '07, EDA Consortium, pp. 1484–1489.

[12] Reineke, J. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008.

[13] Roop, P. S., Andalam, S., von Hanxleden, R., Yuan, S., and Traulsen, C. Tight WCRT analysis of synchronous c programs. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems* (New York, NY, USA, 2009), CASES '09, ACM, pp. 205–214.

[14] SATABS. Snu real-time benchmarks, 2012. http://www.cprover.org/goto-cc/examples/snu.html (Last Accessed: 04/02/12).

[15] Schoeberl, M. A time predictable instruction cache for a java processor. In *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* (2004), Springer, pp. 371–382.

[16] Suhendra, V., Roychoudhury, A., and Mitra, T. Scratchpad allocation for concurrent embedded software. *ACM Trans. Program. Lang. Syst. 32*, 4 (Apr. 2010), 13:1–13:47.

[17] Theiling, H., Ferdinand, C., and Wilhelm, R. Fast and precise wcet prediction by separated cache and path analyses. *REAL-TIME SYSTEMS 18* (1999), 157–179.

[18] Whitham, J., and Audsley, N. Implementing Time-predictable Load and Store Operations. In *Proc. EMSOFT* (2009), pp. 265–274.