HAL
open science

Predictable Multithreading of Embedded Applications Using PRET-C

Sidharta Andalam, Partha Roop, Alain Girault

To cite this version:

Sidharta Andalam, Partha Roop, Alain Girault. Predictable Multithreading of Embedded Applications Using PRET-C. International Conference on Formal Methods and Models for Codesign, MEMOCODE'10, Jul 2010, Grenoble, France. pp.159-168, 10.1109/MEMCOD.2010.5558636 . hal-00786378

HAL Id: hal-00786378

https://inria.hal.science/hal-00786378

Submitted on 8 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Predictable multithreading of embedded applications using PRET-C

Sidharta Andalam*, Partha Roop† and Alain Girault‡
Department of Electrical and Computer Engineering,
University of Auckland, New Zealand
{*sand080, †p.roop}@aucklanduni.ac.nz
‡alain.girault@inria.fr

Abstract

We propose a new language called Precision Timed C (PRET-C), for predictable and lightweight multithreading in C. PRET-C supports synchronous concurrency, preemption, and a high-level construct for logical time. In contrast to existing synchronous languages, PRET-C offers C-based shared memory communications between concurrent threads that is guaranteed to be thread safe. Due to the proposed synchronous semantics, the mapping of logical time to physical time can be achieved much more easily than with plain C, thanks to a Worst Case Reaction Time (WCRT) analyzer (not presented here). Associated to the PRET-C programming language, we present a dedicated target architecture, called ARPRET, which combines a hardware accelerator associated to an existing softcore processor. This allows us to improve the throughput while preserving the predictability. With extensive benchmarking, we then demonstrate that ARPRET not only achieves completely predictable execution of PRET-C programs, but also improves the throughput when compared to the pure software execution of PRET-C. The PRET-C software approach is also significantly more efficient in comparison to two other light-weight concurrent C variants (namely SC and Protothreads), as well as the well-known Esterel synchronous programming language.

1. Introduction

Embedded applications are reactive, concurrent, and also have strict timing requirements. The conventional approach to the design of such systems has been the use of a real-time operating system (RTOS) that executes on a speculative processor to manage both the concurrency and timing needs of the application. The problem of concurrency managed through operating system threads has been highlighted by Lee [12]: "they discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism". Understandability is lost since the programmer is burdened with ensuring correctness through complex synchronization mechanisms provided by the RTOS. Predictability is sacrificed since concurrency is emulated through RTOS scheduling that is inherently nondeterministic. More importantly, as these threads are "heavy-weight", there is a significant performance penalty to be paid. This is because each thread has to maintain its full execution context.

A move away from this direction is the concept of light weight multithreading in C — a language of choice for embedded systems. Here, through simple libraries and by light-weight context switching, threading is made feasible. Two prominent examples in this category are the recent SC [17] language and an earlier C-library called Protothreads [6]. Both languages are inspired by the concept of *coroutines* where blocks of code (called coroutines) may have multiple entry points and transfer control to other code blocks using *yield* statements. Both languages rely on C macros to generate pure C code. SC is designed mainly for directly encoding SyncCharts [3] in C. This is achieved by having a single *tick* function that manages the state transition between threads using *computed goto* statements. The main goal of SC has been to achieve reduced code size in comparison to Esterel [5] based implementations of SyncCharts. The major limitation of SC is that it has no mechanism for resolving signal status and, hence, doesn't preserve the full semantics of SyncCharts. Protothreads [6] is a light-weight C library for the programming of concurrent state-machines. The main objective is to produce minimal memory footprint for embedded applications.

In this paper, we propose a solution that is significantly different from the above-mentioned light-weight C libraries in the following way. Firstly, we propose a synchronous C language where light-weight threads communicate through shared memory. Our semantics ensures that shared memory access is thread-safe by construction; this solves the issue of *understandability* by shifting the responsibility of shared memory management from the programmer to the semantics of the language. Our threads are light-weight and are

compiled to a single function where "multithreading" is elicited through context switching using a barrier instruction called `EOT`. Due to the reliance on a new synchronous semantics [2], composition is *causal* [4] by construction and, hence, all PRET-C programs are deterministic; this solves the issue of *determinism*. Finally, we perform static timing analysis to compute the maximum tick length of our PRET-C programs. This value, called the worst case reaction time (WCRT), ensures that the execution of a PRET-C program is time-predictable; this solves the issue of *predictability*. Furthermore, our solution [15] for computing the WCRT is based on a new algorithm, tighter than earlier approaches [10], [14].

The language extensions to C that we propose are C macros, and hence the standard gcc compiler can be used for backend code generation. The optimized assembly code is represented in an intermediate format called timed concurrent control flow graph (TCCFG), which is used for the WCRT analysis [15]. PRET-C programs can be executed completely predictably by fixing the tick length to the value determined by this WCRT analysis. This is useful for applications that require full predictability or a fixed sampling rate. Alternatively, they can be executed with a variable tick length for better performances, but at the cost of predictability.

The execution of PRET-C programs can be done using either a hardware or a software approach. In the hardware approach, we extend a general processor with a hardware accelerator. That performs thread scheduling and preemption. We demonstrate the proposed method through the ARPRET processor that is designed by customizing the Xilinx MicroBlaze [19] soft-core processor. This approach achieves predictable execution without sacrificing throughput. In the software approach, however, we perform efficient thread scheduling in software similarly to CEC [8] linked-list based scheduler.

The key contributions of this paper are:

1) The design of a new light-weight and concurrent language called PRET-C, for the predictable programming. PRET-C offers a very simple mechanism for achieving thread-safe shared memory communication between light-weight C-threads, not available in earlier light-weight threading libraries for C.
2) We offer a hardware accelerator for PRET-C execution over soft-core processors so that predictable execution can be achieved without sacrificing throughput. This will be crucial for achieving precision timed implementations of real-time applications.
3) We demonstrate, that ARPRET excels in comparison to the pure software implementation of PRET-C. Interestingly, software implementation of PRET-C significantly outperforms SC, Protothreads, and Esterel in the average and worst case reaction time, while generating compact code.

The WCRT analysis is not the focus of the current paper; it can be found in [15]. Moreover, the structural operational semantics of PRET-C can be found in the companion technical report [2], along with much more detailed comparison with other synchronous programming languages (Esterel and RC). Finally, a preliminary short version of the present work appeared in [1]. Compared to this short paper, the present paper contains much more details on the PRET-C language, on the ARPRET architecture, and on the intermediate TCCFG format; it also covers the compilation into TCCFG, which what not presented in [1].

The organization of this paper is as follows. In Section 2, we present the PRET-C language through a producer-consumer example. The intermediate format is presented in Section 2.3. In Section 3, we present the ARPRET architecture. The results of the experiment, evaluating the hardware extension and performance against other concurrent programs are presented in Section 4. Finally, the conclusions are presented in Section 5.

2. PRET-C overview

The overall design philosophy of PRET-C and the associated architecture may be summarized using the following three simple concepts:

- *Concurrency*: Concurrency is logical but execution is sequential. This ensures both synchronous execution and thread-safe shared memory communication. This is the founding principle of the synchronous programming languages [4].
- *Time*: Time is logical and the mapping of logical time to physical time is achieved by the compiler and the WCRT analyzer [15].
- *Design approach*: ARPRET achieves PRET by simple customizations of general purpose processor (GPP). The extensions to C are minimal and are implemented through C-macros.

2.1. PRET-C language extensions

PRET-C extends C using the five constructs shown in Table 1. In order to guarantee a predictable execution, we impose the following four restrictions on the C language:

- Pointers can cause dynamic jumps and dynamic calls with the computed target addresses. We cur-

Statement	Description
`ReactiveInput I`	declares `I` as a reactive input coming from the environment
`ReactiveOutput O`	declares `O` as a reactive output emitted to the environment
`PAR(T1,...,Tn)`	synchronously executes in parallel the `n` threads `Ti`, with higher priority of `Ti` over `Ti+1`
`EOT`	marks the end of a tick (local or global depending on its position)
`[weak] abort P when pre C`	immediately kills `P` when `C` is true in the previous instant

Table 1. PRET-C extensions to C.

rently disallow pointers to prevent unpredictable control flow. However, in the near future, we will follow a *value analysis* technique [18] to allow the restricted use of pointers.

- Our current version does not allow dynamic memory allocation. However, some of the recent work [9] guarantees precise (de)allocation times, while focusing on reducing internal fragmentation. In the near future, we will investigate existing ideas to allow dynamic memory allocation.

- All loops must have at least one `EOT` in their body. This is needed to ensure that thread compositions are deadlock free. This restriction could be relaxed for the loops that can be statically proven to be *finite*.
- All function calls have to be non-recursive to ensure a temporal upper bound on execution time.

The C extensions are implemented as C-macros, all contained in a `pretc.h` file that must be included at the beginning of all PRET-C programs. As a result, we only rely on the `gcc` macro-expander and compiler for compiling PRET-C programs.

Like any C program, a PRET-C program starts with a preamble part (`#define` and `#include` lines), followed by global declarations (reactive inputs, reactive outputs, and classical C global variables), and finally function definitions (including the `main` function).

A PRET-C program runs periodically in a sequence of *ticks* triggered by an external clock. The inputs coming from the environment are sampled at the beginning of each tick. They are declared with the `ReactiveInput` statement. The outputs emitted to the environment are declared with the `ReactiveOutput` statement. Reactive inputs are read from the environment at the beginning of every tick and cannot be modified inside the program. Hence, the value of these variables remains fixed throughout an instant. They have a default value when the environment assigns no value. In contrast, reactive outputs may be updated by the program and can have several values within an instant. The final value of these variables (termed their steady-state value) is emitted to the environment. Reactive outputs behave exactly like normal variables in C, except that they are emitted to the environment while normal variables are for communication between threads, and are not emitted to the environment.

Programmers familiar with usual synchronous languages such as Esterel [5] or its earlier C-based extensions [11] will notice the difference with PRET-C. Unlike these languages, where the primary means of communication between threads are *signals*, we use *variables*. Signals have both a *status* and an associated *value* (when the signal is not pure). Esterel forbids the usage of variables for communication between threads for causality reasons. PRET-C allows unrestricted shared variable access across threads, and thread safe communication is achieved using static thread priority. Besides this, our reactive inputs and outputs are similar to Esterel's input and output signals, respectively.

The `PAR(T1,...,Tn)` statement spawns `n` threads that are executed in lock step. All spawned threads evolve based on the same view of the environment. However, unlike the usual $\|$ of other synchronous languages like Esterel, where threads are scheduled in each instant based on their signal dependencies, threads in PRET-C are always scheduled based on a fixed static order. This order is determined based on the order in which threads are spawned using the `PAR` construct. E.g., a `PAR(T1,T2)` statement assigns to `T1` a higher priority than to `T2`.

Parallel threads communicate through shared variables and reactive outputs. The task of ensuring mutually exclusive access is achieved by ensuring that, in every instant, all threads are executed in a fixed total order by the scheduler. When more than one thread acts as a writer for the same variable, then the semantics of the program still remains deterministic. Indeed, race conditions can happen in RTOSs when these writes are non-atomic, i.e., if one write operation can be interrupted and another thread can then modify the same variable. However, as long as these writes happen atomically in some fixed order, the value of the result will be always predictable and race conditions will be prevented. This is ensured by our ARPRET architecture thanks to the proposed multi-threaded execution. On ARPRET, once a thread starts its execution, it cannot be interrupted. The next thread is scheduled only when the previous thread reaches its `EOT`. Thus, when two or more threads can modify the same variable, they always do so in some fixed order,

ensuring that the data is consistent.

The `EOT` statement marks the end of a tick. When used within several parallel threads, it implements a *synchronization barrier* between those threads. Indeed, each `EOT` marks the end of the *local tick* of its thread. A *global tick* elapses only when all participating threads of a `PAR()` reach their respective `EOT`. In this sense, `EOT` is similar to the `pause` statement of Esterel. `EOT` enforces the synchronization between the parallel threads by ensuring that the next tick is started only when all threads have reached their `EOT`. Finally, it allows to compute precisely the WCRT of a program by computing the execution time of all the computations scheduled between any two successive `EOT` instructions. This WCRT analysis is presented in paper [15].

`EOT` is similar in spirit to the `deadi` instruction of [13] (immediate deadline). However, unlike the *low-level* `deadi` instruction that manages timing thanks to timers, `EOT` is a *high-level* programming construct. The task of ensuring precise timing of threads is not left to the programmer but is derived by WCRT analysis and is a compilation task [15]. Moreover, the `deadi` instruction is also used for achieving mutual exclusion by time-interleaving the access to shared memory. This is achieved by setting precise values to the deadlines. However, if done manually, this task can be very complex, even for simple programs. This is mainly due to arbitrary branching constructs and loops. Automating this task is non-trivial and has not been solved in [13]. Our solution to achieve mutual exclusive access to shared memory is ensured by having static thread priorities, and then scheduling the threads in every instant according to this fixed linear order.

The `abort P when pre C` construct preempts its body `P` immediately when the condition `C` is true (like `immediate abort` in Esterel). Like in Esterel, preemption can be either *strong* (`abort` alone) or *weak* (when the optional `weak` keyword is used). In case of a strong abort, the preemption happens at the beginning of an instant, while the weak abort allows its body to execute and then the preemption triggers at the end of the instant. All preemptions are triggered by the *previous* value of the Boolean condition (hence the `pre` keyword), to ensure that computations are deterministic. This is needed since the values of variables can change during an instant. The use of the `pre` ensures that preemptions are always taken based on the steady state values of variables from the previous instant. In other words, like in ReactiveC, we use a restricted form of causality compared to Esterel.

2.2. A Producer Consumer example

We present in Figure 1 a producer-consumer example adapted from [16] to motivate PRET-C. The program starts by including the `pretc.h` file (line 1). Then, reactive inputs are declared (lines 3 and 4), followed by regular global C variables (lines 5 and 6), and finally all the C functions are defined (lines 7 to 41). The `main` function consists of a single main thread that spawns two threads (line 36): a `sampler` thread that reads some data from the `sensor` reactive input and deposits this data on a global circular `buffer`, and a `display` thread that reads the deposited data from `buffer` and displays this data on the screen, thanks to the user defined function `WriteLCD` (line 29). The `sampler` and `display` threads communicate using the shared variables `cnt` and `buffer`. Also, the programmer has assigned to the `sampler` thread a higher priority than to the `display` thread. All the threads are declared as regular C functions.

During its first local tick, the `sampler` thread does nothing. During its second local tick, it checks if its data `buffer` is full (line 11): as long as `buffer` is full, it keeps on waiting until the display thread has read some data so that there is empty space in `buffer`. When it exits this while loop, it then writes the current instant's value of the `sensor` input to the next available location of the buffer (line 12) and ends its local tick (line 13). During its third and last local tick, the index `i` of the buffer and the total number `cnt` of data in the buffer are incremented (lines 14 and 15), since this is a circular buffer. Then, the `sampler` loop is restarted.

During its first local tick, the `display` thread does nothing. During its second local tick, it checks if there is any data available to read from `buffer` (line 23). If there is no data available, then it ends its local tick and keeps on waiting until some data has been sent by the producer. When this happens, it reads the next data from `buffer` (line 24) and ends its local tick (line 25). During its next local tick, `i` is incremented (line 26) and `cnt` is decremented (line 27). During its last local tick, it sends the data read from the `buffer` to a display device (line 29).

The `main` thread (main function) has an enclosing `abort` over the `PAR` construct. This preemption is taken whenever an external `reset` button has been pressed in the previous instant (line 37). In our example, when a strong preemption happens, the two threads are aborted,the `cnt` is initalized (line 38), and the `main` thread pauses for an instant before flushing the buffer and restarting the two threads again.

The execution of this code on any GPP with an RTOS to emulate concurrency will lead to race con-

```
1  #include <pretc.h>
2  #define N 1000
3  ReactiveInput (int,
       reset, 0);
4  ReactiveInput (float
       , sensor, 0.0);
5  int cnt=0;
6  float buffer[N];
7  void sampler() {
8    int i=0;
9    while(1) {
10     EOT;
11     while (cnt==N)
           EOT;
12     buffer[i]=sensor
           ;
13     EOT;
14     i=(i+1)%N
15     cnt=cnt+1;
16   }
17 }
18 void display() {
19   int i=0;
20   float out;
21   while(1) {
22     EOT;
23     while (cnt==0)
           EOT;
24     out=buffer[i];
25     EOT;
26     i=(i+1)%N;
27     cnt=cnt-1;
28     EOT;
29     WriteLCD(out);
30   }
31 }
32 void main() {
33   while(1) {
34     abort
35       flush(buffer);
36       PAR(sampler,
             display);
37     when pre (reset)
           ;
38     cnt=0;
39     EOT;
40   }
41 }
```

Figure 1. A Producer-Consumer example in PRET-C.

ditions. It is the responsibility of the programmer to ensure that critical sections are properly implemented using OS primitives such as semaphores. Race conditions will occur because of non-exclusive accesses to the shared variable `cnt`. However, on ARPRET, the execution will always be deterministic. Assume that `cnt=cnt+1` and `cnt=cnt-1` happen during the same tick. Due to the higher priority of `sampler` over `display`, `cnt` will be incremented first, and once `sampler` reaches its `EOT`, the ARPRET scheduler (see Section 3) will select `display` which will then decrement `cnt`. Thus, the value of `cnt` will be consistent without the need for enforcing mutual exclusion between the `sampler` and `display` threads.

2.3. Intermediate format

We propose an intermediate format for PRET-C programs, called Timed Concurrent Control Flow Graph (TCCFG). The TCCFG corresponding to the example of Figure 1 is shown in Figure 2. The TCCFG encodes the explicit control-flow of the threads as well as the forking and joining information of the threads. Depending on the type of aborts, we insert *checkabort* nodes. For each strong abort in the program, a checkabort node is inserted *just after* every EOT node. For each weak abort in the program, we insert a checkabort node *just before* every EOT node.

The TCCFG encodes the explicit control-flow of the threads as well as the forking and joining information of the threads. It has the following types of nodes:

- Start/End node: Every TCCFG has a unique start node where the control begins, and may have an end node if the program can terminate. Both are drawn as concentric circles.
- Fork/Join nodes: They mark where concurrent threads of control start and end. They are drawn as triangles.
- Action nodes: They are used for any C function call or data computation. We use rectangles to draw them.
- EOT nodes: They indicate the end of a local or a global tick, and are drawn as filled rectangles.

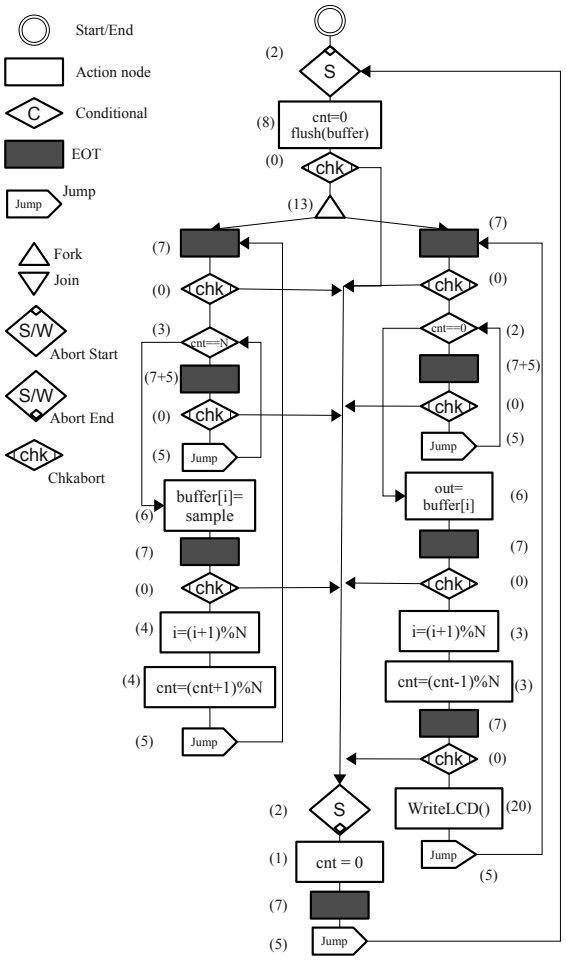


Figure 2. TCCFG of the Producer-Consumer.

- Control flow nodes: We have two types of control flow nodes: conditional nodes to implement conditional branching (drawn as rhombuses labelled with the condition) and jump nodes for mapping unconditional branches (drawn as arrow-shaped pentagons).
- Abort nodes: We have abort start and abort end nodes to mark the scope of an abort. They are drawn as rhombuses labeled either with 's' or 'w' to indicate strong or weak aborts.
- Checkabort nodes: These are special nodes that implement the semantics of aborts. They are drawn as thin rhombuses labeled with 'chk'. More details on the structural translation of aborts are covered in [2].

2.3.1. Compiling PRET-C into TCCFG

PRET-C programs are C programs with the addition of macros that implement the macros of Table 1 to support concurrency and synchronous reactivity. Since we are compiling for MicroBlaze (MB), we use MB based gcc compiler (mb-gcc) to expand the macros and compile the resulting C program into MB assembly code. This assembly code contains a small number of assembly instructions along with lots of information to help the linker. For the timing analysis, we are only interested in the assembly instruction and some specific set of labels. So we first get rid of unwanted information, then we generate the TCCFG nodes using the ASM2TCCFG translator, developed by us. The nodes contain information about the instruction type (Figure 2 shows the node types) and the worst case execution cost.

Figure 3 shows how we translate a PAR instruction in PRET-C into a FORK node in TCCFG. The PAR instruction (shown in Figure 3a) compiles into the MB assembly code shown in Figure 3b. Line 1 of the assembly code contains the assembly comment `begin PAR`. The ASM2TCCFG translator recognizes this as the beginning of a FORK node. Then, line 2 contains the number of children for this FORK node (two in this case). Recognizing the assembly comment on line 6, ASM2TCCFG reads the next line and extracts the label $L8 as the start address of its first child. Similarly, it extracts the label $L9 on line 20 as the start address of its second child. The analyzer then creates two arrows, pointing from the FORK node respectively to the node containing the label $L8 (start of the sampler thread), and to the node containing the label $L9 (start of the display thread). Finally, the unconditional branch assembly instruction on line 16 is seen by the translator as the end of the fork node.

Following some simple rules, the ASM2TCCFG translator can generate all the nodes and the arrows connecting the nodes of a PRET-C program. To calculate the execution cost of each node, we first calculate the cost of each assembly instruction in the node. We do this by referring to a look up table containing the worst case execution cost of each assembly instruction. This data is based on the MB datasheet [19]. Secondly, by summing up the cost of all the instructions in any given node, we obtain the worst case execution time for this node. Figure 2.3.1b shows that the worst case execution cost of the FORK node is 13 clock cycles. Once we have the complete TCCFG, the WCRT analysis of PRET-C is performed as presented in [15].

3. ARPRET architecture

This section presents the hardware extension to a General Purpose Processor (GPP) called MicroBlaze (MB) [19] in order to achieve temporal predictability. We designed the PRET-C language to enable the design of PRET machines by simple customizations of GPPs. The changes needed to execute PRET-C predictably concern the support for concurrency and preemption. If concurrency is implemented purely in software, the overhead of scheduling will be proportional to the number of parallel threads. Indeed, at each `EOT`, the scheduler has to select the next thread based on the status of threads and the preemption contexts. Doing this in software will consume significant numbers of clock cycles (compared to a hardware implementation), thus reducing the overall throughput. For this reason, we do the scheduling in hardware, with a custom made Predictable Functional Unit (PFU). Figure 4 shows the basic setup of an Auckland Reactive PRET (ARPRET) platform consisting of a MB soft-core processor that is connected to a PFU.

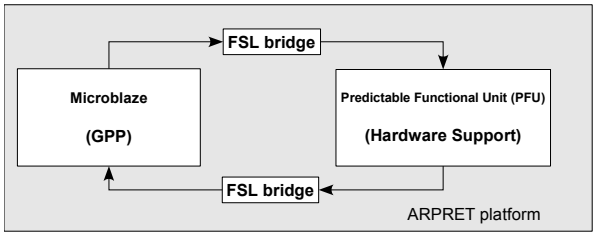


Figure 4. ARPRET platform consisting of MicroBlaze GPP and an external functional unit called PFU.

MB is a customizable RISC based soft-core processor, optimized for implementation on the Xilinx FPGA. To guarantee predictability, all instructions and data are stored in on-chip memory. Any read/write operation takes only one clock cycle, simplifying the problem of

```
PAR(sampler,display);
...
...
...
...
...
...
...
...
...
...
...
...
...
...
sampler:
 /* code for the
      sampler thread */
...
...
...
display:
 /* code for the
      display thread */
```

a) PAR in PRET-C.

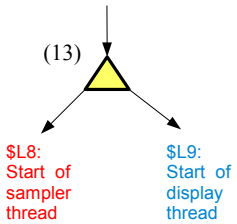
```
  # begin PAR
  # children 2
  # /*some computation in parent
      thread*/
  # /* stores the return address */

  #@ child info
  addik r4,r0,$L8
  swi   r4,r0,T1

  #@ child info
  addik r3,r0,$L9
  swi   r3,r0,T2


  # /* some computation */
  bra r7 //unconditional jump
  ...
$L8:
  #/* assembly code for sampler
      thread */
$L9:
  #/* assembly code for display
      thread */
```

b) MB assembly code.



c) FORK node in TCCFG.

Figure 3. From PAR instruction to FORK node.

timing analysis. No parallel shifters or floating point units were employed. We used three stages in the pipeline with the branching delay slot feature disabled.

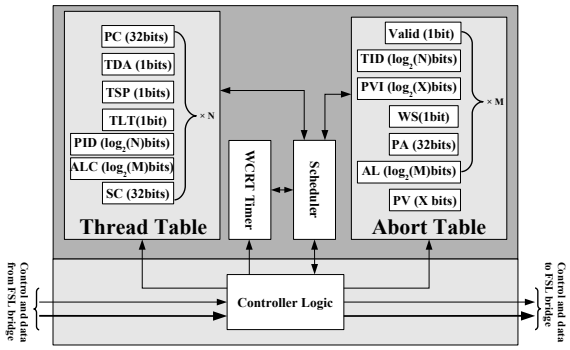


Figure 5. Predictable Functional Unit (PFU).

PFU is inspired by the STARPro processor [20], which stores multiple thread preemption contexts for executing Esterel. STARPro's hardware only allows upto four levels of nested preemptions per thread and it preallocates the hardware for each thread. This is an inefficient use of hardware. In comparison, PFU does not preallocate any hardware. Instead, all threads share the same space in the Abort Table. This is more efficient, as it allows the space consumed by each thread to change dynamically. Also, since PRET-C has fixed priorities compared to Esterel, this simplifies the hardware and reduces the footprint compared to STARPro.

The PFU stores multiple thread contexts and schedules the threads. For each thread, a thread table stores its Program Counter (PC) as a 32-bit value, its status (dead or alive, called TDA), its suspended status (TSP), its local tick status (TLT), and its priority (TP). Depending on the four thread statuses, the scheduler issues the next program counter when requested. Abort contexts are also maintained in an abort table for dealing with preemption.

MB acts as the master by initiating thread creation, termination, and suspension (this is different from Esterel suspend; in PRET-C, any thread that spawns child threads is suspended). The PFU stores the context of each thread in the thread table and monitors the progress of threads as they execute on the MB. When a given thread completes an `EOT` instruction on the MB, it sends appropriate control information to the TCB using FSL bridge. In response to this, the PFU sets the local tick bit (LTL) for this thread to 1, and then invokes the scheduler. The scheduler then selects the next highest priority thread for execution by retrieving its PC value from the thread table and sending it to MB using FSL bridge. Whenever it completes a local tick, the MB blocks to wait for the next PC value from the PFU. PFU can be configured for two different execution strategies, either with a variable execution time for throughput, or with a constant execution time for predictability. During the constant execution mode, MB awaits for the worst case tick length to expire before starting the next tick. This is stored in

WCRT timer and is calculated at compile time by static WCRT analysis of a PRET-C program as detailed [15]. We next present in detail how the tightly coupled connection between the MB and the PFU is created in ARPRET.

3.1. Communication

Communication between MB and the PFU is done by using the Fast Simplex Link (FSL) interface [19] provided by Xilinx. Two FSL bridges closely couples MB with the PFU, to provide deterministic and predictable communication. Communication with the bridge requires exchange of some common control signals such as the clock, reset, buffer status (FULL/EMPTY), read, write, and also data such as the PC value.

Function (ID)	Number of reads	Number of writes
SPAWN (10)	1	0
EOT (12)	1	1
SUSPEND (14)	1	1

Table 2. Simple lookup table is used by the PFU to decode the data from the FSL bridge.

The communication between the MB (master) and PFU (slave) is triggered when the MB executes instructions such as `PAR(T1,T2)` and `EOT`. The PFU contains a Controller Logic that refers to a look up table (LUT) as shown in Table 2 to decode the data from MB. For example, to spawn a thread, the MB writes a value of 10 followed by the start address of the thread onto the FSL bridge. Controller logic decodes 10 as SPAWN. Then, and by referring to the number of reads in Table 2, it fetches one more data element from the FSL bridge and stores it as the PC of the new thread. Also, in response to this SPAWN, other status bits such as the thread status and the suspended bits are altered appropriately. Also, the TLT bit is set to 0 to indicate that the local tick for the thread is not yet reached.

3.2. Hardware usage

We next present the results of the hardware resource usage on the FPGA device. The hardware resources in terms of Slices and Look Up Tables (LUT) are shown by Figure 6. Slices are logical blocks providing functionality such as arithmetic, ROM functions, storing, and shifting data. They contain LUTs, storage elements, and multiplexers. Four-input lookup tables are used by FPGA function generators for implementing any arbitrarily defined four-input Boolean function [19]. From Figure 6 we can see that the hardware resource consumption of ARPRET is linearly proportional to the number of threads. This is due to the fact that ARPRET mostly stores thread contexts, and only minimal datapath is required by the scheduler.

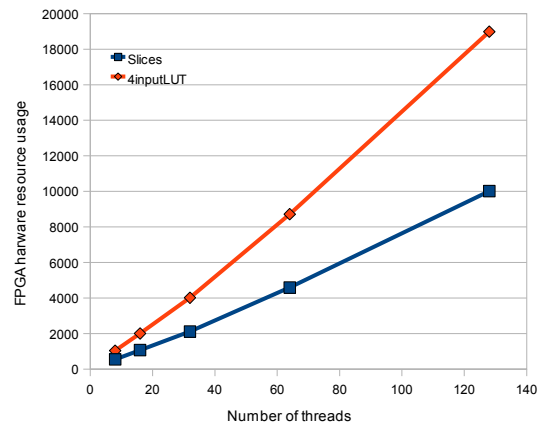


Figure 6. Number of threads versus hardware consumption in terms of LUTs.

4. Benchmarks and results

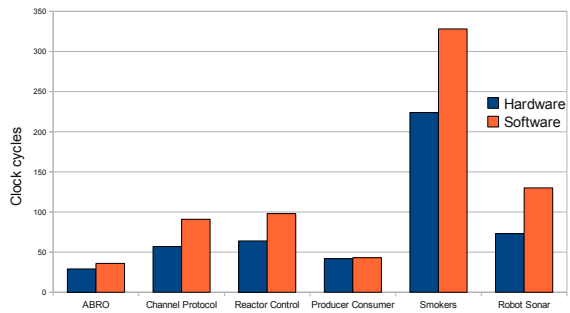
To assess the efficacy of the proposed hardware acceleration, we first compare the hardware execution of PRET-C on ARPRET with the software execution on MicroBlaze. We then compare the software execution of PRET-C with that of Protothreads, SC, and Esterel. Comparison is carried out over both execution time and memory usage of a set of benchmark programs with high degree of concurrency and preemption. Some of the benchmarks are adaptations of programs from the Estbench [7] suite.

To preserve behavioral equivalence when translating PRET-C programs into Protothreads, we made Protothreads synchronous by using the *yield* construct, which is similar to EOTs, and also by forcing tick synchronization to facilitate a synchronous execution as in PRET-C and Esterel. Preemptions in Protothreads were emulated using a software-like approach based on the placement of checkaborts. For Esterel, all non-immediate aborts were replaced by the immediate counterparts.

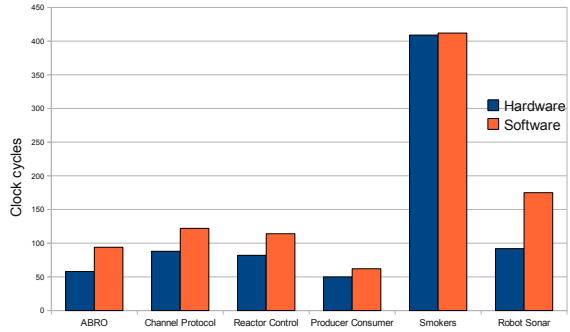
4.1. Benchmarking

The benchmarking process was carried out as follows. Firstly, we generated code for ARPRET. Then, for the same benchmarks, we generated C code for execution on MicroBlaze alone (that is, with no PFU). To enable a fair comparison with the hardware scheduler, thread scheduling was done very efficiently in software thanks to CEC-like [8] linked-list based scheduler[1]. We call this approach the *software* compilation approach for

1. CEC is the most efficient compiler for Esterel to C. It has very efficient mechanisms for thread management in software, which ensure minimal context switching

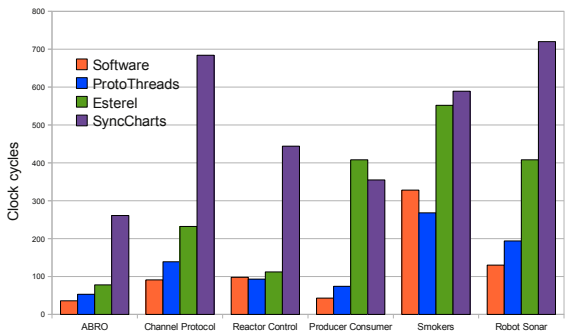


(a) Average Case Reaction Time.

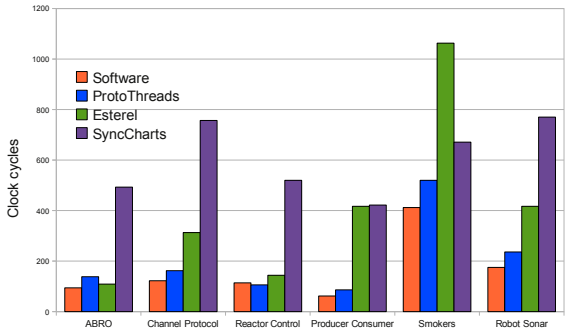


(b) Worst Case Reaction Time.

Figure 7. Comparing the reaction time on hardware and software.



(a) Average Case Reaction Time.



(b) Worst Case Reaction Time.

Figure 8. Comparing the Reaction time of PRET-C (software) with Protothreads, Esterel, and SC.

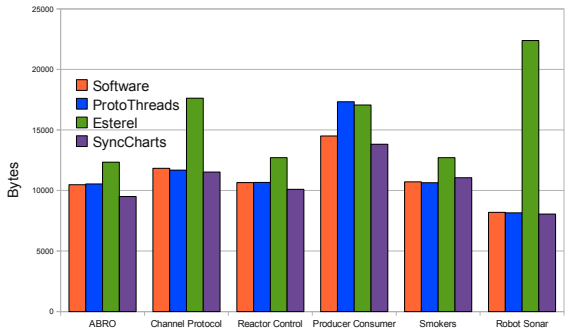


Figure 9. Comparing the memory usage of PRET-C (software) with Protothreads, Esterel, and SC.

PRET-C. We present the results of the hardware versus software execution of PRET-C in Figure 7.

For WCRT comparison, we used random test vectors and measured the WCRT over one million reactions. The worst case (shown in Figure 7a) is the maximum of the measured values, while the average case (shown in Figure 7b) is obtained by averaging over all samples. As detailed in Section 3, for predictability, the hardware can be configured with constant tick length, obtained through the static analysis approach [15]. The hardware approach is 28% more efficient than the software approach for the average case, and 26% for the worst case.

Since there are no hardware acceleration platforms for SC, Protothreads, and Esterel, we compared the software execution of PRET-C with the software execution of these languages. The results are presented in Figure 4.1. Code was generated on MicroBlaze for SC, Protothreads, Esterel, and the PRET-C software approach. We used the CEC compiler for Esterel code generation since it consistently generated the most efficient code compared to all other Esterel compilers.

Figure 4.1a shows the average case reaction times and Figure 4.1b shows the WCRT. On average, PRET-C consumes about 18%, 68%, and 77% less computation time when compared to Protothreads, Esterel, and SC respectively. Similarly for the WCRT, PRET-C consumes about 20%, 50%, and 74% less time respectively. In comparison SC achieves the least throughput. This is due to its implementation of the thread scheduling by calling a function, where the overhead of the function call is proportional to the number of threads. In contrast, the context-switch in PRET-C takes constant time that does not involve any function calls.

The memory usage of these languages is presented in Figure 4.1. PRET-C generates significantly more compact code compared to Esterel, while being slightly

inferior to SC (by only 4%) and almost equivalent to Protothreads. In summary, these results reveal that PRET-C yields significantly more efficient code compared to all others in both the average and worst case. Also, the memory usage of PRET-C is superior to Esterel and comparable to SC and Protothreads.

5. Conclusions and future work

Precision Timed (PRET) architectures are a recent attempt to design processors that guarantee predictable execution of code without sacrificing throughput. Researchers from Berkeley and Columbia proposed a tailored processor with a thread interleaved pipeline and a low-level instruction to introduce precise timing in C code [13]. In contrast, this paper proposes the customization of embedded soft-core processors to design PRET architectures with minimal hardware requirements. We also propose a new language for programming PRET machines, called PRET-C, by simple synchronous extensions to the C language. PRET-C has constructs for expressing logical time, preemption, and concurrency. Concurrent threads communicate using the shared memory model (regular C variables) and communication is thread-safe by construction. We have designed a new PRET machine, called ARPRET, by customizing the MicroBlaze soft-core processor. We have benchmarked the proposed design by comparing the execution of PRET-C on ARPRET with the execution of Esterel on its speculative counterpart (MicroBlaze). Benchmarking results reveal that the proposed approach achieves predictable execution without sacrificing throughput.

In the future, we will explore more scalable architecture design, consider memory hierarchy, and the multicore execution of PRET-C.

References

[1] S. Andalam, P. Roop, and A. Girault. Deterministic, predictable and light-weight multithreading using PRET-C. In *DATE'10*, Dresden, Germany, March 2010.

[2] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. Technical Report 6922, INRIA Grenoble Rhône-Alpes, 2009. www.ece.auckland.ac.nz/∼roop/pub/2009/andalam09.pdf.

[3] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR-2003-24-FR, Sophia Antipolis, 2003.

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

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

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *SenSys'06*, Boulder, Colorado, USA, Nov. 2006. ACM.

[7] S. A. Edwards. *EstBench Esterel Benchmark Suit*. http://www1.cs.columbia.edu/~sedwards/software.html (Last Accessed: 8/6/2009).

[8] S. A. Edwards and J. Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007. Article ID 52651.

[9] J. Herter, J. Reineke, and R. Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. In *Work-In-Progress Session ECRTS*, pages 24–27, July 2008.

[10] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of esterel specifications. In *CODES+ISSS'08*, pages 173–178, New York, NY, USA, 2008. ACM.

[11] L. Lavagno and E. Sentovich. Ecl: a specification environment for system-level design. In *DAC '99*, pages 511–516, New York, NY, USA, 1999. ACM.

[12] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[13] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *CASES'08*, pages 137–146. ACM, 2008.

[14] M. Mendler, R. von Hanxleden, and C. Traulsen. Wcrt algebra and interfaces for esterel-style synchronous processing. In *DATE*, pages 93–98. IEEE, 2009.

[15] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight wcrt analysis of synchronous c programs. In *CASES'09*, pages 205–214, New York, NY, USA, 2009. ACM.

[16] F. Vahid and T. Givargis. *Embedded System Design*. John Wiley and Sons, 2002.

[17] R. von Hanxleden. Synccharts in c: a proposal for light-weight, deterministic concurrency. In *EMSOFT'09*, pages 225–234, New York, NY, USA, 2009. ACM.

[18] R. Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[19] Xilinx. *MicroBlaze Processor Reference Guide*, 2008.

[20] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic. Starpro — a new multithreaded direct execution platform for esterel. *Electron. Notes Theor. Comput. Sci.*, 238(1):37–55, 2009.