

# PRET-C: A New Language for Programming Precision Timed Architectures (extended abstract)

Sidharta Andalam <sup>1</sup>, Partha S Roop <sup>1</sup>, Alain Girault <sup>2</sup>, and Claus Traulsen <sup>3</sup>

<sup>1</sup> University of Auckland, New Zealand

<sup>2</sup> INRIA Grenoble, France

<sup>3</sup> CAU Kiel, Germany

## 1 Introduction and related work

The idea of predictable architectures has recently obtained more attention with the concept of a Precision Timed Machine (PRET) [5]. The two goals are to guarantee precise timing without sacrificing throughput, and to simplify the worst case timing analysis of code executing on PRET machines. Such a PRET machine, based on the SPARC ISA, has been developed by the UC Berkeley and Columbia groups [7] (hereafter called the Berkeley-Columbia approach). They use a thread-interleaved pipeline executing with the support of a memory wheel so that each stage of the pipeline feeds from a new thread. They use scratchpad memories and a static allocation mechanism. A new instruction (`deadi`) enforces a precise timing upper bound on the execution of a code segment. The programmer must insert `deadi` instructions at suitable points and with appropriate deadline values to ensure mutual exclusive access to shared memory.

PRET machines were also developed at Auckland University and Kiel University under the banner of reactive processors. Reactive processors [10] were developed with Esterel-like ISA so as to eliminate the imprecise environment interaction mechanism of interrupts used in modern processors. Subsequently, they have been used to directly execute Esterel code both efficiently [8] and precisely [2]. Also, PRET architectures for Java were developed by a group in Vienna and in [11] a nice survey of predictable architectures is presented.

Yet, C remains a language of choice for programming embedded systems: the system is split into interacting hardware and software components where the hardware components are synthesized on FPGA and the software parts are executed on an RTOS executing multi-threaded C code. The problem of this approach is that RTOSs add unnecessary overhead while the overall execution remains non-deterministic and hence unpredictable. We propose simple extensions to C to ensure predictable execution on general purpose processors (GPPs) without the need for any RTOS. The syntax and semantics of the language should be simple enough for easy adoption by C programmers. Finally, the language design should keep in mind that GPPs are essentially unpredictable, and it should facilitate easy processor customization for predictability.

We introduce a set of simple synchronous extensions to C. The proposed language is called Precision Timed C (PRET-C, Section 2). We also present a

PRET architecture called ARPRET (Auckland Reactive PRET) by customizing an existing soft-core GPP (Section 3).

Among the synchronous extensions to C, ReactiveC [3] is closest to our approach and our PAR construct is similar to the `par` in ReactiveC. In contrast, we allow multiple readers of a shared variable to read different values of this variable during an instant. ReactiveC requires a stricter notion of data-coherency, where all readers must read the same value of data. Inspired by Esterel, ECL [6] is a more recent synchronous C extension with the parallel construct that is exactly like Esterel. The main means of communication between threads in ECL is through signals. Compared to PRET-C all these languages are more complex and were not designed with predictability in mind.

## 2 The PRET-C language

The overall design philosophy of PRET-C and the associated architecture ARPRET may be summarized using the following three simple concepts: ① Concurrency is logical but execution is sequential. This ensures both synchronous execution and thread-safe shared memory communication. ② Time is logical and the mapping of logical time to physical time is achieved by the compiler and the WCRT analyzer [9]. ③ ARPRET achieves PRET by simple customizations of GPPs. The extensions to C are minimal and are implemented through C-macros.

PRET-C (Precision Timed C) is a *synchronous* extension of the C language similar in spirit to ECL and ReactiveC. Unlike the earlier synchronous C extensions, it is based on a *minimal* set of extensions and is specially designed for *predictable* execution on ARPRET. It extends C using the five constructs below:

- `ReactiveInput I`, declares `I` as a reactive input coming from the environment, sampled at the beginning of each tick.
- `ReactiveOutput O`, declares `O` as a reactive output emitted to the environment at the end of each tick.
- `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` was true in the previous instant. Parallel threads communicate through shared variables and reactive outputs. Mutual exclusion is achieved by ensuring that, in every instant, all threads are executed in a fixed order by the scheduler. When more than one thread acts as a writer for the same variable, the execution of the program still remains deterministic. This is ensured by the semantics of PRET-C: once a thread starts its execution, it cannot be interrupted; the next thread is scheduled only when the previous thread reaches its `EOT`. Thus, even 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`. It also allows to compute precisely the WCRT of a program by computing the required

execution time of all the computations scheduled between any two successive EOTs.

The `abort P when pre C` construct preempts its body `P` immediately when the condition `C` is true (like `immediate abort` in Esterel). Like 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). This ensures that preemptions are always taken based on the steady state values and guarantees functional and temporal determinism.

Finally, to guarantee a predictable execution, we impose the following restrictions on PRET-C: ① Pointers and dynamic memory allocation are disallowed to prevent unpredictability of memory allocation. ② All loops must have at least one EOT in their body. This ensures that thread compositions are deadlock free. This restriction could be relaxed for the loops that can be statically proven to be *finite*. ③ All function calls have to be non-recursive to ensure predictable execution of functions. ④ Jumps via `goto` statements are not allowed to cross logical instants (i.e., EOTs).

```

1 #include <pretc.h>
2 #define N 1000
3 ReactiveInput (int, reset, 0);
4 ReactiveInput (float, sensor,
5     0.0);
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 }

```

**Fig. 1.** A Producer Consumer in PRET-C

We present in Figure 1 a producer-consumer example adapted from [12] 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 in 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.

The execution of this code on any general purpose processor (GPP) with an RTOS to emulate concurrency will lead to race conditions. It is the responsibility of the programmer to ensure that critical sections are properly implemented using OS primitives such as semaphores. This will happen because of non-exclusive accesses to the shared variable `cnt`. However, due to the semantics of PRET-C, the execution will be always deterministic. When `cnt=cnt+1` and `cnt=cnt-1` happen during the same tick, due to the higher priority of `sampler` over `display`, `cnt` will be first incremented by 1, and once `sampler` reaches its EOT, the scheduler will select `display` which will then decrement `cnt` by 1. Thus, the value of `cnt` will be consistent without needing mutual exclusion.

### 3 ARPRET architecture

This section presents the hardware extension to the Microblaze [13] in order to achieve better throughput for the worst case execution. The ARPRET platform consists of a Microblaze soft-core processor that is connected to a hardware extension, called, the Predictable Functional Unit (PFU) using the fast simplex link (FSL) [13]. More details of this architecture are provided in [1].

Microblaze acts as the master by initiating thread creation, termination, and suspension. The PFU stores the context of each thread in the thread table and monitors the progress of threads as they execute on Microblaze.

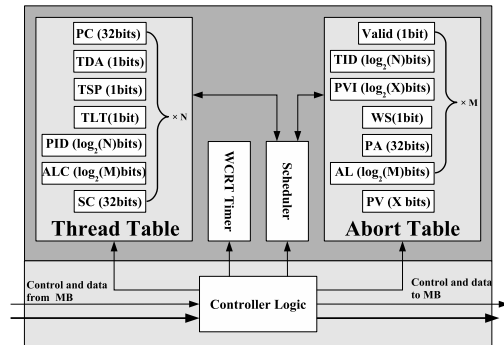


Fig. 2: Predictable Functional Unit (PFU).

When a thread completes an EOT macro on Microblaze, it sends appropriate data to the Thread Control Block through the FSL. In response to this, the PFU sets the local tick bit for this thread to 1, and then invokes the scheduler. The scheduler then selects the next highest priority thread for execution by retrieving its program counter from the thread table and sends it to Microblaze. Microblaze blocks whenever it awaits for the next program counter value from the PFU. The PFU can be configured for two different execution strategies, either with a variable execution time for performance, or with a constant execution time for predictability. In the latter case, Microblaze awaits for the worst case tick length to expire before starting the next tick. This constant tick length is decided by static WCRT analysis of the PRET-C program.

### 4 Benchmarks and results

As PRET-C is a new language, there are no existing benchmarks. We created three sample programs called *Smokers*, *Robot Sonar*, and *Producer Consumer*. The *Robot Sonar* example was developed considering the application domain for

PRET-C. These three PRET-C programs were also programmed in Esterel. To preserve behavioral equivalence, we replaced all non-immediate preemptions in the benchmarks in Esterel with their immediate counterparts. Since we are comparing with Esterel, we also programmed three examples from the Estbench [4] suite in PRET-C, keeping their behavior identical (*ABRO*, *Channel Protocol*, and *Reactor Control*).

Example	Columbia CEC			Esterel V5A			Esterel V5N			PRET-C	
	B	A	W	B	A	W	B	A	W	WCRT <sub>max</sub>	WCRT
ABRO	60	78	109	83	185	266	367	403	438	89	89
Channel Protocol	137	232	313	86	271	432	967	1099	1149	174	152
Reactor Control	89	112	144	144	189	311	633	677	714	121	118
Producer Consumer	275	408	417	397	524	596	608	742	763	118	99
Smokers	33	552	1063	80	723	1256	743	1186	1603	531	449
Robot Sonar	275	408	417	397	524	596	608	742	763	419	346
Average	145	298	410	198	402	576	654	808	905	242	208

**Table 1.** Quantitative Comparison with Esterel.

Criteria	Berkeley-Columbia solution	Auckland solution
Platform	Tailored processor	Customized GPP
Notion of time	Physical ( <i>dead</i> instruction)	Logical (EOT and WCRT analysis)
Mutual exclusion	Through time interleaved access	By construction
Memory hierarchy	Scratchpad memories	No
Prototyping	SystemC model	Implemented on FPGA
Input language	Concurrent C (without formal semantics)	Synchronous extension to C (with formal semantics)
Preemption support	No	<b>abort when pre</b> statement

**Table 2.** Comparison between the Auckland and Berkeley-Columbia PRET approaches.

The benchmarks currently are small, hence, both Esterel and PRET-C code fit on the on-chip program memory. Thus, the behavior of caches are not taken into account. For Esterel, we generated code using a range of compilers, and we compared the code size and execution time with that of PRET-C (with random test vectors over one million reactions). The result of these experiments is presented in Table 1. B stands for best case execution time, W for worst case, and A for average case. For PRET-C, we first determined the WCRT with the static analysis method of [9] (WCRT sub-column). We also computed the maximum WCRT value of a program by adding the maximum tick lengths of all participating threads (WCRT<sub>max</sub> sub-column). For programs with a large number of threads and good computation to communication ratio such as the Robot Sonar example, the static analysis method of [9] produces tighter results because it considers both data dependency and state-based contextual information to remove redundant execution paths.

The WCRT value of PRET-C is systematically better than the worst measured execution time obtained with the three Esterel compilers. In most cases, it is also better than the average measured execution time obtained with the three

Esterel compilers. This is because PRET-C is executed very efficiently using ARPRET where the concurrency and preemption are implemented in hardware. Also, PRET-C doesn't have any compilation overhead (only macro-expansion) to generate C code. Finally, Table 2 provides a qualitative comparison between the Berkeley-Columbia solution and the solution presented in this paper.

## 5 Conclusions and future work

PRET-C is a new language for programming PRET machines: it extends the C language with synchronous 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 also designed a new PRET machine, called ARPRET, by customizing the MicroBlaze soft-core processor. The benchmark results show that the proposed approach achieves predictable execution without sacrificing throughput. In the future, we shall extend ARPRET with scratchpad memory to execute large embedded applications predictably and efficiently. We shall also develop tools that can highlight the timing behavior of code to the programmer statically. This shall guide the programmer to optimize the code depending on the application requirements.

## References

1. 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, <http://hal.inria.fr/inria-00391621/fr/>, 2009.
2. M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science*, 203(4):65–79, June 2008. SLA++P'07.
3. F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, Apr. 1991.
4. S. Edwards. Estbench Esterel benchmark suite. <http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz>.
5. S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of DAC*, pages 264–265, June 2007.
6. L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of DAC*, New Orleans, USA, June 1999.
7. B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of CASES*, 2008.
8. P. Roop, Z. Salcic, and M. Dayaratne. Towards direct execution of Esterel programs on reactive processors. In *Proceedings of EMSOFT*, 2004.
9. P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis of synchronous C programs. In *Proceedings of CASES*, Grenoble, France, Oct. 2009.
10. Z. Salcic, P. Roop, M. Biglari-Abhari, and A. Bigdeli. Reflex: A processor core for reactive embedded applications. In *Proceedings of FPL*, 2002.
11. M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009. Article ID 758480.
12. F. Vahid and T. Givargis. *Embedded System Design*. John Wiley and Sons, 2002.
13. Xilinx. *MicroBlaze Processor Reference Guide*, 2008.