HAL
open science

Deterministic, Predictable and Light-Weight Multithreading Using PRET-C

Sidharta Andalam, Partha Roop, Alain Girault

To cite this version:

Sidharta Andalam, Partha Roop, Alain Girault. Deterministic, Predictable and Light-Weight Multithreading Using PRET-C. Design Automation and Test in Europe Conference, DATE'10, Mar 2010, Dresden, Germany. hal-00765020

HAL Id: hal-00765020

https://inria.hal.science/hal-00765020

Submitted on 14 Dec 2012

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.

Deterministic, predictable and light-weight multithreading using PRET-C

Sidharta Andalam and Partha S Roop
Department of Electrical and Computer Engineering
University of Auckland, New Zealand
Email: {sand080, p.roop}@aucklanduni.ac.nz

Alain Girault
INRIA Rhone-Alpes
Grenoble, France
Email: Alain.Girault@inria.fr

***Abstract*—We present a new language called Precision Timed 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, which is guaranteed to be thread safe via the proposed semantics. Mapping of logical time to physical time is achieved by a Worst Case Reaction Time (WCRT) analyser. To improve throughput while maintaining predictability, a hardware accelerator specifically designed for PRET-C is added to a soft-core processor. We then demonstrate through extensive benchmarking that the proposed approach not only achieves complete predictable execution, but also improves overall throughput when compared to the 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 called SC and Protothreads, as well as the well-known synchronous language Esterel.**

I. Introduction

Embedded applications are reactive and 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 [11]: "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 synchronisation mechanisms provided by the RTOS. Predictability is sacrificed since concurrency is emulated through RTOS scheduling that is inherently non deterministic. 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 complete context such as the process control block.

A move away from this direction is the concept of light weight multithreading in C—a language of choice for embedded systems. Two prominent examples in this category are the recent SC [16] proposal and an earlier C-library called Protothreads [7]. SC is designed mainly for encoding SyncCharts [3] in C directly. This is achieved by having a single *tick* function that manages the state transition between threads using *computed goto* statements. The main focus of SC has been to achieve reduced code size in comparison to Esterel [5] based implementations of SyncCharts. Protothreads [7] is a light-weight C library for the programming of concurrent state-machines. The main objective is to produce minimal memory foot-print for embedded applications. Both languages rely on C macros to generate C code. Similar to PRET-C, there are a few synchronous extensions to C [6], [10]. ReactiveC [6] is the closest to PRET-C. However, none of these languages are designed for predictable execution and thread-safe shared memory communication, unlike PRET-C.

Our contributions are the following: (1) We present a new light-weight, concurrent language called PRET-C, for the predictable programming of PRET architectures. PRET-C offers a very simple mechanism for achieving thread-safe shared memory communication between light-weight C-threads through its synchronus semantics, not available in earlier light-weight threading libraries for C. (2) We offer a hardware accelerator, called ARPRET, for PRET-C execution over soft-core processors, so that predictable execution can be achieved without sacrificing throughput. (3) We demonstrate, through extensive benchmarking, that ARPRET excels in comparison to the pure software implementation of PRET-C. Software implementation of PRET-C significantly outperforms SC, Protothreads, and Esterel [5] in the average and worst case execution time, while generating consistently more compact code.

The organisation of this paper is as follows. In Section II, we present the PRET-C language through a producer-consumer example along with its semantics and the intermediate format. In Section III we present the ARPRET architecture and how PRET-C programs are executed. The experimental results are presented in Section IV, and our conclusions are presented in Section V.

II. PRET-C overview

PRET-C extends C using the five constructs shown in Table I. In order to guarantee a predictable execution, we impose some restrictions to C, such as the lack of dynamic memory allocation and recursion. Also, all loops must have at least one `EOT`. Our five C extensions are implemented as C-macros, all contained in a header file, named `pretc.h` .

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.

Statement	Meaning
`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 I
PRET-C EXTENSIONS TO C.

They are declared with the `ReactiveInput` statement. The outputs emitted to the environment are declared with the `ReactiveOutput` statement.

The `PAR(T1,...,Tn)` statement spawns `n` threads that are executed in lock step. Threads in PRET-C are always scheduled based on a fixed static order. This is determined based on the order in which threads are spawned using the `PAR` construct. For example, a `PAR(T1,T2)` statement assigns to `T1` a higher priority than to `T2`. 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.

The `EOT` statement marks the end of a tick. When used within several parallel threads, it implements a *synchronisation 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`.

The `abort P when pre C` construct preempts its body `P` immediately when the condition `C` is true. In case of a strong abort, the preemption happens at the beginning of an instant, while the weak abort (indicated by the `weak` keyword) 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 `C` (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 `pre` ensures that preemptions are always performed based on the steady state values of variables from the previous tick.

A. Producer Consumer example

We present in Figure 1 a producer-consumer example adapted from [15] to motivate PRET-C. 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 it 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

```
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 }
```

Fig. 1. A Producer Consumer in PRET-C

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 last local tick, the index `i` of `buffer` and the total number of data `cnt` in `buffer` 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 is 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, the `i` index of the buffer is incremented (line 26) and the total number of data `cnt` in `buffer` is decremented (line 27). During its last local tick, it sends the data read from `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). When a strong preemption happens, the two threads are aborted and the `cnt` variable is reset (line 38). The `main` thread pauses for an instant before flushing the buffer and restarting the two threads again.

When `cnt=cnt+1` and `cnt=cnt-1` are executed in the same tick, due to the higher priority of the `sampler` thread over the `display` thread, `cnt` will be first incremented

by 1, and once `sampler` reaches its `EOT`, the scheduler will select the `display` thread which will then decrement `cnt` by 1. Thus, the value of `cnt` will be consistent without the need for enforcing mutual exclusion between the `sampler` and `display` threads. Emulating this concurrency on RTOS would lead to race conditions, and it would be the programmer's responsibility to enforce mutual exclusion.

B. Mapping logical time to physical time

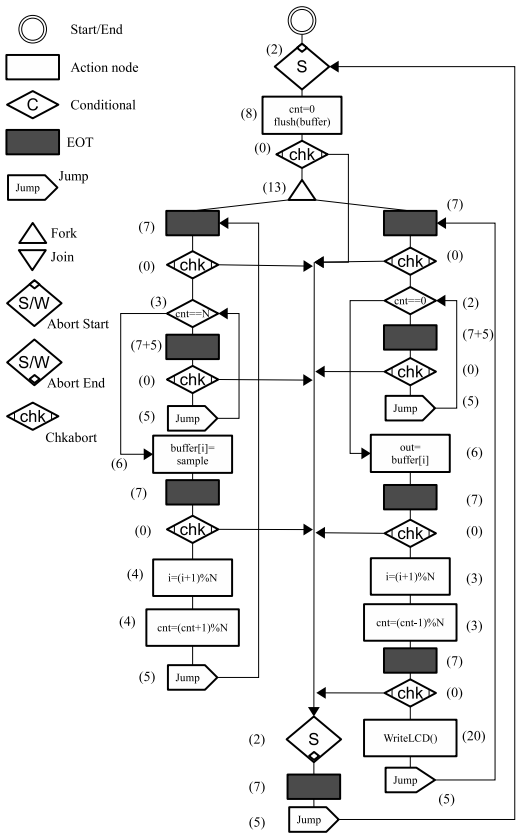

Fig. 2. TCCFG of the Producer-Consumer

Static timing analysis of a synchronous program is equivalent to determining the worst case tick length of this program, termed as the worst case reaction time (WCRT) [13]. For timing analysis of PRET-C, we first generate an intermediate format, called Timed Concurrent Control Flow Graph (TCCFG). It encodes the explicit control-flow of the threads as well as the forking and joining information of the threads. The TCCFG corresponding to our example of Figure 1 is shown in Figure 2.

A PRET-C program is first converted into ARPRET assembly code, and then the TCCFG is extracted from analysing the assembly code. All threads in TCCFG are modelled as concurrent processes in a model checker that supports operations on bounded integers [1]. Then, by evaluating a CTL query, the value of the tight tick length can be determined. More details are presented in [14].

III. ARPRET architecture

This section presents a hardware accelerator to Microblaze [17] in order to achieve better throughput for the worst case execution. The hardware accelerator performs PRET-C specific thread scheduling and preemption. 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) [17]. More details of this architecture are provided in [2].

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. When a given thread completes an `EOT` macro, it sends appropriate control information 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 sending it to Microblaze.

IV. Benchmarks and results

In this section, we first compare the execution of PRET-C on ARPRET with the pure software execution on MicroBlaze to assess the efficacy of the proposed hardware acceleration. We then compare the software execution of PRET-C with Protothreads, SC, and Esterel. Comparison is done over both execution time and memory usage of a set of benchmark programs with a high degree of concurrency and preemption. Some of the benchmarks are adaptations of programs from the Estbench [8] suite.

To preserve behavioural equivalence between Protothreads and PRET-C, we made Protothreads synchronous by using the *yield* construct that is similar to EOTs, and also by forcing tick synchronisation to facilitate a synchronous execution like 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 their immediate counterparts.

A. 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. To enable a fair comparison with the hardware scheduler, thread

	Hardware			Software		Gain%	
Example	A	W	U	A	W	A	W
ABRO	29	58	64	36	94	19.45	38.29
Channel Protocol	57	88	90	91	122	37.36	27.86
Reactor Control	64	82	86	98	114	34.69	28.07
Producer Consumer	42	50	53	43	62	2.32	19.35
Smokers	224	409	413	328	412	31.70	0.73
Robot Sonar	73	92	96	130	175	43.85	47.43
Average						28.23	26.96

TABLE II
QUANTITATIVE COMPARISON OF HARDWARE VERSUS SOFTWARE EXECUTION OF PRET-C

	PRET-C (SW)			SC			Protothreads			Esterel			AC Gain%			WC Gain%		
Example	A	W	M	A	W	M	A	W	M	A	W	M	SC	PT	Esterel	SC	PT	Esterel
ABRO	36	94	10480	261	493	9506	53	138	10544	78	109	12340	86.21	32.07	62.82	80.93	31.88	13.76
Channel Protocol	91	122	11832	684	757	11528	139	162	11680	232	313	17628	86.07	34.53	75.43	83.88	24.69	61.02
Reactor Control	98	114	10652	444	520	10094	93	106	10668	112	144	12716	77.93	-5.37	42.86	78.08	-7.54	20.83
Producer Consumer	43	62	14520	355	422	13818	74	86	17336	408	417	17060	87.89	41.89	89.71	85.31	27.90	85.13
Smokers	328	412	10720	589	671	11054	268	520	10648	552	1063	12716	44.31	-22.38	59.43	38.60	20.76	61.24
Robot Sonar	130	175	8198	720	770	8054	194	236	8154	408	417	22388	81.94	32.99	82.11	77.27	25.85	58.03
Average													77.50	18.95	68.72	74.01	20.59	50.00

TABLE III
QUANTITATIVE COMPARISON OF THE PRET-C SOFTWARE APPROACH WITH SC, PROTOTHREADS AND ESTEREL

scheduling was done very efficiently in software using a CEC-like [9] linked-list based scheduler. We call this approach the *software* compilation approach for PRET-C. We present the results of the hardware versus software execution of PRET-C in Table II.

For execution time comparison, we used random test vectors and measured the execution time over one million reactions. The worst case (W) is the maximum of the measured values, while the average case (A) is obtained by averaging all samples. Our estimated WCRT value (U) is obtained through the static analysis approach based on model checking presented in Section II-B. Table II shows that the hardware approach is 28% more efficient for the average case, and 26% for the worst case execution when compared to the software approach.

The comparison results of PRET-C software execution with SC, Protothreads, and Esterel is presented in Table III. Code was generated on Microblaze for SC, Protothreads, Esterel, and the PRET-C software approach. We used the CEC Esterel compiler since it consistently generated the most efficient code compared to all other Esterel compilers. PRET-C yields significantly more efficient code compared to all others in both the average (A) and worst (W) cases. Finally, the memory usage (M in bytes) of PRET-C is better compared to Protothreads and Esterel, while being slightly worse than SC, by only 4%.

V. Conclusions and future work

We have presented the language PRET-C, targeting real-time embedded systems, by simple synchronous extensions to the C language. PRET-C has constructs for logical time, synchronous concurrency, and preemption. It also offers deterministic access to shared memory, such that all PRET-C programs are *causal* [4] and *thread-safe* [11] by construction. We have also designed a hardware accelerator to improve the worst case behaviour of PRET-C programs so that overall real-time implementation is achieved without sacrificing throughput [12].

We benchmarked the proposed approach by comparing an efficient software implementation of PRET-C with the hardware approach. We also compared the software approach with two other light-weight C libraries. In all cases, the proposed approach excels both in terms of worst case execution time and code size. When compared with Esterel, PRET-C achieves consistently better results. Interestingly, since the average case performance of PRET-C (software approach) is also significantly better than the average case execution of SC, Protothreads, and Esterel, it implies that PRET-C can be used not only for real-time systems but also for any systems where throughput is important. In the near future, we will investigate multicore execution of PRET-C and memory hierarchy issues.

References

[1] Uppaal tool. www.uppaal.com. last accessed on 12.3.09.

[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, Jan 2003.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, April 1991.

[7] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006.

[8] S. Edwards. Estbench Esterel benchmark suite. http://www1.cs.columbia.edu/ sedwards/software/estbench-1.0.tar.gz. last accessed on 12.3.09.

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

[10] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of Design Automation Conference (DAC)*, New Orleans, USA, June 1999.

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

[12] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *In Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, October 2008.

[13] M. Mendler, R. von Hanxleden, and C. Traulsen. WCRT algebra and interfaces for Esterel-style synchronous processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, April 2009.

[14] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, Grenoble, France, October 2009.

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

[16] R. von Hanxleden. SyncCharts in C - A Proposal for Light-Weight Deterministic Concurrency. In *ACM Embedded Software Conference (EMSOFT)*, 11-16 October 2009.

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