Predictable Parallel Programming using PRET-C

REPORT,
UNIVERSITY OF AUCKLAND
FACULTY OF ENGINEERING,
NO. 679

August 19, 2010

Eugene Yip and Partha S Roop and Morteza Biglari-Abhari

ISSN 1178-3680
Abstract

Embedded system designers are always faced with the challenge of achieving a good price-performance trade-off. Recently, the advent of affordable multicore processors has offered designers the choice of high-performance architectures. However, efficient and predictable programming of multicore systems still poses considerable challenges.

This paper offers a novel solution to this problem by answering two nagging questions. Firstly we address the question of efficient parallel programming using C, a language of choice in the embedded domain. Secondly, we address the issue of timing predictability, a significant concern in most embedded systems. In doing so, we propose a novel technique of parallel programming by extending C with macros and a synchronous execution semantics. We also demonstrate using extensive experimentation that the proposed approach is both efficient, scalable, while also enabling predictable execution. To our knowledge, this is the first approach for predictable parallel programming using synchronous C.
1 Introduction

C is the language of choice for programming embedded systems. C follows the inherently sequential model of computation and is an ideal language for programming small-scale embedded devices (which are either 8-bit or 16-bit microcontrollers). Recently, embedded applications have exploded in complexity [1] and functionality demanding the use of high performance processors. The advent of affordable embedded multicore processors [2] offers interesting alternatives to designers to balance the price-performance trade-off. However, programming of such multicore to enhance performance has been a challenge [3] with some studies showing that multicore execution actually reduces throughput without careful tuning [2]. An orthogonal challenge is the issue of achieving guaranteed worst case performance that does not sacrifice throughput. Such predictability is essential for most embedded systems that are safety-critical.

This paper addresses these two key questions by developing new execution semantics for C, called Precision Timed C (PRET-C), for predictable parallel programming of embedded multicore systems. PRET-C extends C with a set of simple macros to enable parallel execution and also to facilitate static timing analysis.

Concurrent programs may either use the shared memory model or the message passing model. For the shared memory model, there is the well known OpenMP library [4] for parallel programming. OpenMP allows both task oriented parallelism and loop parallelism. Another alternative for the shared memory based multi-threading is POSIX threads (or Pthreads) [5]. For the message passing model the MPI library [6] is the most well known one. Two key problems of all these approaches are the lack of determinism and also the lack of understandability. Ensuring correctness of applications (beyond some trivial ones) is the programmer’s responsibility and is a very hard task. More importantly, the issue of achieving good throughput while also maintaining predictability has not been looked at.

Timing analysis of concurrent programs running on multicore exist such as [7] which models concurrent programs as scenarios using Message Sequence Charts. Another presents a multicore architecture [8] designed to simplify timing analysis of concurrent programs where exclusive shared memory access is required.

Concurrent programming for embedded systems is not new. Synchronous languages [9] offer deterministic concurrency that can be implemented without the need for any operating system. Concurrency is logical and is compiled away to produce sequential code. Multicore execution of synchronous programs is difficult due to the complexity of distributing synchronous programs in the presence of causality issues [10]. Hence, the traditional approach [11] is to first generate a sequential equivalent of the concurrent program and then distribute this program such that the control parts are replicated on the nodes of a distributed system while the data parts are distributed for parallel execution. This approach is not suitable, however, for multicores as achieving good throughput is not feasible. Very recently, [12] is early work on compiling programs, of synchronous languages, to multi-threaded C-programs utilizing OpenMP. However, debugging a synchronous program may require the debugging of its generated C-program.

Considering the widespread use of C in embedded systems, we present the PRET-C language extensions to enable predictable parallel programming on embedded multicore. PRET-C extends C with a set of simple macros to enable parallel programming, simple management of critical sections, programming of logical time, and predictable preemptions without using any interrupts. Users can invoke standard C functions and reuse existing libraries. We then use the GCC compiler to generate code amenable to
both single core and multicore execution.

A key characteristic of PRET-C is its synchronous execution semantics. This guarantees that PRET-C programs execute both deterministically and predictably. However, unlike conventional synchronous programs, PRET-C programs are always causal by construction [9] thus being amenable to distributed execution on multicores.

Another interesting feature of PRET-C is that it is designed to facilitate static timing analysis using an abstraction of the program. The proposed abstraction enables tight timing analysis using model checking [13]. Model checking based WCET analysis is, in general, not competitive to the standard approach based on integer linear programming (ILP) combined with abstract interpretation (AI) [14, 15]. However, we demonstrate that by using synchronous semantics combined with an abstraction of the program, it is possible to do very tight and scalable timing analysis of C programs using model checking. This is unlike earlier approaches to timing analysis of synchronous C programs that were primarily based on either the max-plus approach [16, 17] or the ILP approach [18]. Both these approaches failed to deliver tighter analysis and were not designed for the static timing analysis of parallel programs, which is the focus of the current paper.

The main contributions of the paper are: (1) We present an approach for programming multicore systems efficiently by extending C using a set of macros and executing the resultant code synchronously. The proposed approach is amenable to distributed execution on multicores since the semantics of the synchronous program is causal by construction [9]. (2) The parallel programming model is simple and intuitive and avoids complex OS-based synchronization mechanisms. Our programs are thread-safe by construction, thus removing the programming burden [19] of multicore systems. (3) We offer an interesting approach for static timing analysis of parallel programs using model checking. We demonstrate that the proposed approach, being based on an abstraction, offers scalable and tight analysis. (4) We have extensively benchmarked the proposed approach to demonstrate that predictable parallel programming is feasible without sacrificing throughput.

The organization of the paper is as follows. In Section 2 we present the overview of the PRET-C language extensions and also motivate the programming style using a running example. In Section 2.6 we present the intermediate format for PRET-C that is used for static timing analysis. In Section 3 we present the overview of the proposed timing analysis. Section 4 describes our experimental setup and Section 5 presents our results followed by conclusions in Section 6.

2 PRET-C Overview

2.1 Background

PRET-C brings synchronous execution semantics to the C-language. Esterel [20] is a well known synchronous language used in industry for writing safety-critical systems. The C-language is popular among embedded developers so C-extended synchronous languages include ReactiveC [21] and ECL [22]. The advantage with PRET-C is its minimal set of extensions designed to simplify the creation of multi-threaded programs and its distribution over multicores while providing time-predictable execution.

Through the use of a synchronizing barrier, synchronous programs evolve in discrete time steps called ticks. A tick starts from one synchronizing barrier till the next and its tick length is determined by the execution time required to complete the tick. By
Table 1: Summary of PRET-C extensions to C.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactiveInput I</td>
<td>Declares variable I as an input from the environment.</td>
</tr>
<tr>
<td>ReactiveOutput O</td>
<td>Declares variable O as an output emitted to the environment.</td>
</tr>
<tr>
<td>EOT</td>
<td>Synchronizing barrier marking the end of a tick.</td>
</tr>
<tr>
<td>PAR($T_0, \ldots, T_n$)</td>
<td>Executes n+1 threads in parallel on available cores. Execution priority only exist for critical sections where $T_i$ has higher priority over $T_{i+1}$.</td>
</tr>
<tr>
<td>[weak] abort P when C</td>
<td>Terminate body P when C was true at the start of a global tick.</td>
</tr>
<tr>
<td>CS_START</td>
<td>Start of a critical section.</td>
</tr>
<tr>
<td>CS_END</td>
<td>End of a critical section.</td>
</tr>
<tr>
<td>CS_NONE</td>
<td>Aliases the critical section status of this thread to its predecessor.</td>
</tr>
</tbody>
</table>

forcing tick lengths to be identical, the program’s evolution becomes *time-predictable*. For this, the longest tick length needs to be determined.

Execution of each tick must be atomic so synchronous programs react to the environment at *tick boundaries*. Thus, a program’s reaction time is its current tick length while the *worst-case reaction time* (WCRT) is its longest tick length. The synchrony hypothesis [9] requires that a program’s WCRT be shorter than the inter-arrival time of environmental events.

Synchronous languages naturally describe parallel processes. To maintain synchrony, parallel processes synchronize with each other before proceeding to the next tick. In PRET-C each parallel process is a *thread*. We distinguish a tick of a thread as a *local tick* and a period for all threads to complete a local tick as a *global tick*. Inter-thread communication in PRET-C is achieved through shared variables. To maintain data consistency, regions of shared variable access called *critical sections* need to be protected.

### 2.2 Language extensions

As shown in Table 1, there are eight PRET-C extensions which are implemented as C-macros and all reside in a header file called `PRET-C.h` that is `#include`d at the start of every PRET-C program. A C-compiler supporting computed `gotos` is sufficient for compiling PRET-C programs.

A PRET-C program executes as a series of ticks where the end of each tick is marked by `EOT`, the synchronizing barrier. For time-predictability, `EOT` ensures the global tick lengths are equal to the computed WCRT.

Like Esterel’s valued signals [20], the environment is sampled into `ReactiveInput` variables and outputs are emitted to the environment through `ReactiveOutput` variables, at each global tick boundary. Reacting to the environment is through `abort P when C` where the body `P` is preempted when condition `C`, sampled at global tick boundaries,
is true. A strong preemption (abort) will not let the body execute while a weak preemption (weak abort) will let the body execute one last time.

\( \text{PAR}(T_0, \ldots, T_n) \) allows a parent thread to spawn child threads \( T_0 \) to \( T_n \) and execute them in parallel. While child threads are alive, the execution of the parent thread is suspended. \( \text{PAR} \) is explained further in Section 2.5.

\( \text{CS}_\text{START} \) and \( \text{CS}_\text{END} \) define the start and end, respectively, of a critical section. Critical sections are executed in fixed order where thread \( T_i \)'s critical section must execute before thread \( T_{i+1} \)'s. Thus, even when threads execute out-of-order, deterministic read-modify-write of shared variables is ensured. These macros implement the necessary access protocol to ensure determinism and removes the burden from the programmer. \( \text{CS}_\text{NONE} \) defines a local tick with no critical section. When thread \( T_i \) executes \( \text{CS}_\text{NONE} \), it tells thread \( T_{i+1} \) to wait on the critical section thread \( T_i \) would have waited on. This ensures thread \( T_{i+1} \) will not wait for thread \( T_i \) to exit its non-existent critical section.

2.3 Language Restrictions

The C-language contains features that make determinism and predictability difficult to guarantee in any program and should be avoided unless the user can statically analyze them. These features include dynamic memory allocation, pointer reassignments, gotos with statically unknown destinations, and recursive function calls.

Restrictions also exist for the PRET-C extensions. To avoid the possibility of infinite loops, each loop must have at least one \( \text{EOT} \). This restriction is relaxed if the loop is statically bounded. Due to their fixed order, critical sections cannot be split over multiple local ticks and each local tick may only have one critical section. Due to the access protocol, control cannot jump in and out of critical sections.

2.4 A ChemicalAnalyzer Example

We present ChemicalAnalyzer to show how a multi-threaded PRET-C program is constructed and how it is distributed. The program monitors the composition of a chemical solution being processed at a factory using five sensors each collecting pH, oxygen and water readings. Monitoring is required to prevent chemical hazards from developing. The program alerts the user when each chemical’s thresholds are breached and at the user’s command the factory immediately shuts down. To achieve these goals, the program must react in a timely and predictable manner.

Listing 1 captures ChemicalAnalyzer as a concise PRET-C program. Reactive inputs are defined for the five sensors and user command (lines 5 and 11). Reactive outputs are defined for the pH, oxygen and water analysis results (line 12). Before analysis, readings from the five sensors need to be collated into pH, oxygen and water readings. Thread \( T_0 \) does this and writes the collated readings into \( \text{solution} \) (line 27). Using \( \text{solution} \), threads \( T_1, T_2 \) and \( T_3 \) each take a chemical (lines 36, 46 and 56) and emit to the environment whether the readings are above their specified threshold (lines 38, 48 and 58). Since \( \text{solution} \) is read from and written to by multiple threads, it must be accessed in a critical section.

ChemicalAnalyzer begins execution by running the factory and then spawning threads \( T_0, T_1, T_2 \) and \( T_3 \) (lines 16 - 17) and suspending the parent thread. The threads execute and reach their local ticks (lines 25, 34, 44 and 54) thus, ending the global tick. The next global tick begins and thread \( T_0 \) collates the new sensor readings and threads \( T_1, T_2 \) and \( T_3 \) analyze the readings. When all threads reach their local tick
#include <PRET-C.h>

struct Sensor {
    int ph, oxygen, water;
} ReactiveInput sensors[5];

struct Readings {
    int ph[5], oxygen[5], water[5];
} solution;

ReactiveInput int operatorStop;
ReactiveOutput int phOut, oxygenOut, waterOut;

int main() {
    abort {
        runFactory();
        PAR(T0, T1, T2, T3);
    } when operatorStop;
    shutdownFactory();
    return 0;
}

thread T0() {
    while (1) {
        EOT;
        CS_START;
        collateReadings(&sensors, &solution);
        CS_END;
    }
}

thread T1() {
    int phLevels[5];
    while (1) {
        EOT;
        CS_START;
        getPh(solution, &phLevels);
        CS_END;
        phOut = analyze(phLevels, 8);
    }
}

thread T2() {
    int oxygenLevels[5];
    while (1) {
        EOT;
        CS_START;
        getOxygen(solution, &oxygenLevels);
        CS_END;
        oxygenOut = analyze(oxygenLevels, 120);
    }
}

thread T3() {
    int waterLevels[5];
    while (1) {
        EOT;
        CS_START;
        getWater(solution, &waterLevels);
        CS_END;
        waterOut = analyze(waterLevels, 309);
    }
}

Listing 1: ChemicalAnalyzer is a multi-threaded PRET-C program
the global tick ends. This repeats until `operatorStop`, the abort condition sampled at global tick boundaries, is true. The child threads are preempted and the parent thread resumes execution after the abort statement (lines 18 - 19). The factory is shutdown and the program terminates.

ChemicalAnalyzer demonstrates three advantages of PRET-C: PAR abstracts, from the source code, the operations required to distribute child threads to available cores; critical sections are naturally documented without burdening the programmer with an access protocol; and the source code is not polluted with extensions.

### 2.5 Parallel Thread Scheduling

We assume a multicore architecture of identical cores and a memory hierarchy where each core has local data and instruction memory and access to some global memory. Global memory can be hierarchical assuming the read/write latencies are statically bounded. Caches are disabled as we do not consider cache analysis. To implement the critical section access protocol, a global hardware mutex is required. The mutex should contain a bit vector that tracks the execution status of each critical section. By reading the bit vector, threads can determine if higher priority critical sections remain to be executed. This architecture is shown in Fig. 1.

To statically analyze the execution of PRET-C programs, the programmer must specify how child threads in each PAR will be partitioned over the cores. The partitions are specified in a separate file so that modifications can be easily made during development. In this way, PRET-C programs may be executed on an arbitrary number of cores.

Part of Fig. 2 shows the compilation flow for PRET-C programs. Using a preprocessor, a customized PAR macro with the necessary thread scheduling code is created for every PAR declared in the program. The PRET-C.h header file is updated to include the customized PAR macros and the program file is updated to use them. With these updated files, GCC generates the program binary for the target multicore architecture. Currently, the same program binary is downloaded to each core with a routine executed at boot up to set each core’s program counter to the correct instruction address. Fig. 3 shows each core’s memory map.

We borrow Columbia Esterel Compiler’s [23] linked list idea to track the future execution of local ticks. A linked list is created for each PAR’s partition. Each node of the linked list corresponds to a thread and stores the program counter (PC) value that its next local tick starts from. Although the linked lists dynamically created, the length of each list is known at compile time. Critical sections must execute in some fixed order so nodes are linked in the priority order as defined in the PAR. To nest a PAR, we expand, at compile time, its parent’s node to include its linked list. At runtime, nested
child threads busy wait for its parent to reach its PAR.

Using ChemicalAnalyzer as an example, Fig. 4 shows its linked lists with threads T0 and T2 partitioned to core 0 and threads T1 and T3 partitioned to core 1. During the execution of PAR, the PC value of T0 is fetched from the starting node of the linked list. Upon reaching its EOT, T0 saves into its node the PC value that its next local tick starts from. T0 then fetches the PC value of the next thread in sequence (T2 in this example) given by the linked list. Each linked list ends with a global tick.
synchronization task that sets the core’s status variable to done and busy waits for the next global tick. However, one core must execute a special synchronization task that reads all the core statuses to determine when a global tick has ended.

Fig. 5 illustrates the scheduling with an execution trace of ChemicalAnalyzer with the parent thread executing on core 0. Local ticks are drawn as vertical lines with critical sections as rectangles and busy waiting as dots. The end of a local tick is a short horizontal line and the end of a global tick is horizontal line across all cores.

This type of multithreaded scheduling is coarse-grained with EOTs acting as synchronizing barriers and places to switch to the next thread. During compilation, thread parallelism that is expressed by the programmer is retained and embraced by the scheduler, not discarded.

**Theorem 1** All valid PRET-C programs are reactive and deterministic.

**Proof sketch:** By extending the semantics of PAR [24] to execute critical sections in fixed order and non-critical sections in partial order, the proof is given by structural induction.

### 2.6 Intermediate Format

To extract the desired timing information from a PRET-C program, we extract an intermediate format called Timed Concurrent Control Flow Graph (TCCFG). The goal is to perform cycle-accurate timing analysis of the program running on the target multicore architecture. Thus, the program is compiled into an assembly program and the TCCFG is derived by analyzing the control flow of the assembly. As an example, Fig. 6 shows ChemicalAnalyzer’s TCCFG.

TCCFG decomposes an assembly program into different nodes as listed in the key of Fig. 6. The nodes *abort end* and *abort check* are used purely to show the scope of an abort in the TCCFG and therefore have zero execution time. Directed arrows are drawn between nodes to show the control flow. A computation node may be composed of other TCCFGs. With a timing model of the target multicore architecture, (safe) cycle-accurate estimates can be derived for the instruction latency of each node.

### 3 WCRT Analysis using Model Checking

Time-predictable execution requires all global tick lengths to be equal to the program’s WCRT. Since EOTs act as state-barriers, it is easy to extract a set of finite state machines, where each machine represents a concurrent thread. The transition between the states will update an integer value that is equal to sum of the instruction latencies for all the nodes between these EOTs.
The synchronous composition of such automaton represents an abstraction of the program and is ideal for static analysis using model checking. This is unlike earlier Integer Linear Programming formulation for timing analysis of concurrent programs [18] where the program’s concurrency is first compiled away. We believe that the proposed analysis opens the way for the timing analysis of both concurrent [25] and parallel programs, where the concurrency and parallelism could be preserved during model checking. This will also enable the model checker to perform aggressive state-space optimization [13] that is now part of every model checker.

We will employ the UPPAAL [26] model checker by mapping TCCFG directly into a set of timed automata (without clocks). As reference, Fig. 2 shows the timing analysis flow thus far. Fig. 7 shows ChemicalAnalyzer’s TCCFG mapped into UPPAAL and should be referred to below.
Figure 6: TCCFG of ChemicalAnalyzer with threads colored by the core they have been partitioned to. Beside each node, in brackets, is their instruction latency.

3.1 Mapping

Threads. Each parent and child thread is modeled as a (clockless) timed automaton. A PAR is modeled as a state in the parent thread with incoming transitions setting the parExecuting boolean to true. Each child thread begins in an initial state with all
outgoing transitions guarded by `parExecuting`. When `parExecuting` becomes true, the outgoing transitions are enabled and the child threads proceed through their automaton. When a child thread terminates, it returns back to its initial state and wait again for the guard to enable its outgoing transitions. Meanwhile, the parent thread is guarded from transitioning out of its `PAR` state until all its child threads have terminated.

**Ticks.** Each `EOT` (synchronizing barrier) is modeled as a state in the timed automaton. Following an `EOT` may be computations and/or a critical section before another `EOT` is reached. This is modeled by appending a state for reaching the start of a critical section (`CS_START`) and a state for the end of a critical section (`CS_END`). Following this is a synchronizing state (`synchronized`), reachable when all child threads are at their `EOT`s, to model a global tick. When model checking, each child thread gets a
Synchronized progression. We introduce another timed automaton (barrier synchronizer) with two transitions to track the progress of all threads. One transition sets globalTick to false if all threads have reached their EOT to signify a global tick has ended. The other transition sets globalTick to true if all threads have reached their synchronized state to signify a global tick has begun. globalTick is used to guard outgoing transitions from EOT and synchronized states.

Tick lengths. By tracing paths from one EOT to the next EOT, we can obtain the cost for a local tick in three parts: (1) the cost before the critical section; (2) the cost of the critical section; and (3) the cost after the critical section. In an automaton, to reach the CS_START state, (1) must be incurred so we map (1) to a transition into CS_START. Likewise, we map (2) to a transition into CS_END and we map (3) to a transition into EOT. Execution of critical sections must be serialized so threads may need to wait for their turn. This waiting time must be added to (2).

Aborts. A non-deterministic input from the environment is modeled as two non-deterministic transitions from a single state where one transition sets the presence of the input and the other sets the absence of the input. An abort is then a transition guarded by the presence of the input.

Multicore distribution. Although child threads are given turns to proceed through their local ticks in a fixed order, the cost of executing a local tick is only attributed to the core it has been allocated to. During the calculation of the global tick, each core has its own integer variable for tracking its total execution cost. Since the cost of a local tick will only affect one core, the partial ordering of threads during the execution of non-critical code is preserved.

Computing the global tick length. From initial states, each thread’s automaton is allowed to proceed through its states until it reaches an EOT. Because threads are given turns, we can determine when the higher priority critical section exits and therefore how long the next thread must wait before entering its critical section. As each transition is taken, assigned costs are attributed to their allocated core. The barrier synchronizer then sets globalTick to false and takes the global tick length as the longest execution cost of all cores. The computed global tick length is assigned to an integer variable called globalTickLength. The variables tracking each core’s execution time are then reset to zero. Lastly, the barrier synchronizer sets globalTick to true so threads may proceed through their next local tick.

3.2 WCRT as a CTL property

Using the model of the program, we determine its WCRT by finding its longest global tick length with the following Computational Tree Logic (CTL) query: $AG(synchronized \Rightarrow (globalTickLength < x))$ where $x$ is an integer bounded by $[x_{\text{min}}, x_{\text{max}}]$. If the model checker finds this query to be false, it may generate a counter example demonstrating a global tick length greater than or equal to $x$. Thus, the interval $[x_{\text{min}}, x_{\text{max}}]$ bounds the WCRT. $x_{\text{min}}$ is the program’s shortest global tick length computed by only considering the shortest local tick length of each thread. $x_{\text{max}}$ is the program’s longest global tick length computed by only considering the longest local tick length of each thread.

The complexity of model checking a single query is $O(|x| \times |M| \times |\phi|)$. $|x|$ is the number of possible valuations of globalTickLength given by $x_{\text{max}} - x_{\text{min}}$. $|M|$ is the standard model checking complexity of the CTL query. Using binary search,
the maximum number of queries required to find the WCRT is \( \log_2(x_{\text{max}} - x_{\text{min}}) \). Therefore, the overall complexity is \( O(\log_2(x_{\text{max}} - x_{\text{min}}) \times (x_{\text{max}} - x_{\text{min}}) \times |M| \times |\phi|) \).

### 4 Experimental Setup

To validate our work we wrote PRET-C programs and executed them on a dual-core Xilinx MicroBlaze system. The dual-core setup consists of identical single-issue cores with three-stage pipelines. Each core has local data and instruction memory, access to dual-ported global memory, with instruction and data cache disabled. The latency for accessing local and global memory is one cycle. Inter-core communication occurs at the hardware mutex but the cost of using it is constant for all cores.

After compiling the programs, their TCCFGs were extracted from low-level assembly and costs were attributed to each node. Each TCCFG was then mapped into UPPAAL and the WCRT was computed. We also used UPPAAL to find each program’s execution path that led to its WCRT. Using this information, each program was executed with their WCRT elicited and measured. To determine the average-case reaction time (ACRT), each program was executed for one million reactions in a free running mode, where ticks were completed as soon as a reaction was over.

### 5 Results

Using the setup described in Section 4, Table 2 shows the ACRT and WCRT of six different 4-threaded PRET-C programs executed on single (S) and dual (D) cores. It also indicates the speedup in performance (\( \% \), as a percentage) for dual core compared to single core processing. Like all timing analysis methods, abstractions used to model the program and timing of hardware led to overestimation in the computed WCRT. Table 2 shows, for dual core, the computed WCRT (D) and its tightness (T\%) to the measured WCRT.

The running example, ChemicalAnalyzer, achieved speedup even though the proportion of critical sections in each local tick was significant (\( \sim 60\% \) of the instruction latency). The tightness of the computed WCRT was also close to the measured WCRT. ControlSystem showed good speedup in ACRT and WCRT even though local tick lengths were highly variable (38 to 216 clock cycles long). ProducerConsumer, through the sharing of a buffer between threads, also had high proportions of critical

Table 2: Measured and computed execution times (in clock cycles) of PRET-C programs on a dual-core system.

<table>
<thead>
<tr>
<th>PRET-C Program</th>
<th>Measured ACRT</th>
<th>Measured WCRT</th>
<th>Computed WCRT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>D</td>
<td>%</td>
</tr>
<tr>
<td>ChemicalAnalyzer</td>
<td>662</td>
<td>536</td>
<td>24</td>
</tr>
<tr>
<td>ControlSystem</td>
<td>527</td>
<td>369</td>
<td>43</td>
</tr>
<tr>
<td>ProducerConsumer</td>
<td>324</td>
<td>260</td>
<td>25</td>
</tr>
<tr>
<td>MultiSampler</td>
<td>965</td>
<td>551</td>
<td>75</td>
</tr>
<tr>
<td>TrafficLights</td>
<td>301</td>
<td>222</td>
<td>36</td>
</tr>
<tr>
<td>PipelinedProcess</td>
<td>385</td>
<td>288</td>
<td>34</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>39</td>
<td>30</td>
<td>91</td>
</tr>
</tbody>
</table>
sections (~56%) but still achieved noticeable speedup. MultiSampler showed that very high speedup can be achieved if completely independent threads were used. Maximum speedup of 100% was not achieved as a portion of the special synchronization task cannot be distributed for parallel execution. TrafficLights contained many control statements including an abort but reasonably tight WCRT was still computed. Pipelined-Process had minimal speedup because its longest global tick involved the parent thread executing for a long time before reaching a PAR.

Because we did not have 4, 8 and 16-cored systems to test scalability, we chose to compare the computed WCRTs of programs executing on such systems. To model memory hierarchy, we increased the cost of reading and writing to global memory and updated the UPPAAL models accordingly. Apart from the single core scenario, we assumed global memory was located in DDR SDRAM with a worst-case latency of 15 clock cycles ([27] Xilinx Virtex-4 DDR SDRAM controller).

Fig. 8 shows the computed WCRTs of two 16-threaded programs distributed evenly on 2, 4, 8 and 16 cores. The first program was created by expanding ChemicalAnalyzer to monitor three additional chemical solutions with ~68% critical sections in each local tick. The second program controls a basic robot that avoids obstacles by analyzing four sonar sensors, driving a set of wheels, and feedback to the user about its intended direction of travel. The proportion of critical sections was lower, at ~50%.

These results show a general trend of improved WCRTs with increasing number of cores. The exception being program one with increased WCRT for the dual core scenario but this was of no surprise given its high percentage of critical sections. For both programs, the 16 cored scenarios showed the greatest speedup. These being computed WCRTs, it is reasonable to expect better ACRTs.
6 Conclusions and Future Work

While multicores are becoming the norm of the day, they still pose considerable challenge for efficient programming. The problem is even more exacerbated in the domain of embedded systems, where there is a need to improve throughput while maintaining predictability for timing safety. This paper, tries to address this challenge by (1) proposing some syntactic sugar over restricted C to facilitate parallel programming of multicores, and (2) also developing a framework for guaranteeing predictable execution over multicores. To our knowledge, the proposed approach, for the first time, paves the way for predictable programming of multicores using C.

In the future, we will seek to overcome several limitations of the current work. We will explore caches and multi-level memory hierarchy in our timing model, and combine software multithreading with hardware multithreaded cores.

References


