Faster Function Blocks for Precision Timed Industrial Automation

Hammond Pearce, Partha Roop, Morteza Biglari-Abhari
Department of Electrical and Computer Engineering
University of Auckland, New Zealand
Email: hpea485@aucklanduni.ac.nz,
{p.roop, m.abhari}@auckland.ac.nz

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark, Denmark
Email: masca@dtu.dk

***Abstract*—In industrial automation, safety-critical control systems need robust timing guarantees in addition to functional correctness. Unfortunately, devices that are typically used in this domain, such as Programmable Logic Controllers, often feature architectures that are not amenable to static timing analysis, for instance relying on general purpose microprocessors or embedded operating systems. As a result, designers often rely on timing values gained from simple measurement of running applications, an approach that only provides very weak guarantees at best. The synchronous approach for IEC 61499 Function Blocks, in contrast, has been demonstrated to be time predictable when run on appropriate hardware, such as simple microprocessors. However, simple microprocessors are often not fast or powerful enough for modern automation requirements. In this paper, we examine how the performance of synchronous IEC 61499 can be improved through the usage of the multi-core T-CREST architecture, data scratchpads, and an optimised compiler. Overall, our improvements resulted in 60% shorter worst-case execution times.**

I. Introduction

There are many examples of automation systems in industry, such as complex conveyor belt networks like those found in airport baggage handling systems, robotic arms moving around large parts in manufacturing roles, and controllers that monitor and react to power systems inside smart grids.

Figure 1 shows an example of an industrial process: a lumber processing system. This would take wooden logs and cut them into wooden planks. As an example of safety requirements, the metal sawblades could be required to detect contact with human skin, such as with the SawStop system, which stops rotation within 5 milliseconds of skin contact [1]. This is an example of a safety-critical system, where the controller must not only ensure *functional* correctness, but also must react within strict *timing* deadlines. Kuo et al. call these Precision Timed Industrial Automation (PTIA) systems [2].

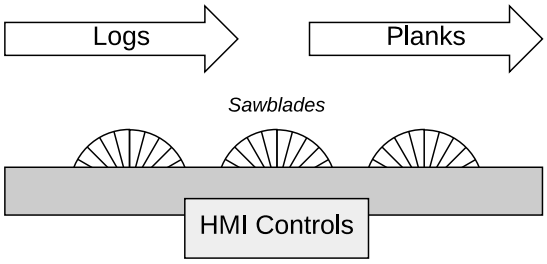


Fig. 1: Sawmill System Diagram

Currently, PTIA systems are implemented with modern Programmable Logic Controllers (PLCs). These are typically implemented with general purpose processors designed by ARM, Intel, and Freescale, and/or have complex runtimes or utilize Real-Time Operating Systems (RTOSs) [3], [4]. Additionally, they are programmed in domain specific languages such as Ladder Logic or Structured Text, which abstract away the complexities of their implementations. The latest language for these devices is IEC 61499 Function Blocks [5], [6].

Unfortunately, while approaches using these devices can still feature verifiable functional correctness, the underlying architectural decisions that increase performance over the old-fashioned bare-metal or analog approaches come with a cost — devices such as these will always struggle with static timing analysis and verification. As a result, designers will often rely on simple measurement based approaches to Worst Case Execution Time (WCET) analysis [7]–[10].

Precision Timed (PRET) machines have been proposed as an alternative to general purpose processors for the implementation of PLCs [2], [11], and the synchronous approach for IEC 61499 has consistently been demonstrated as a time-predictable way of implementing the control programs required on these platforms [10], [11]. However, until now, time-predictable implementations of the synchronous approach for IEC 61499 have been restricted in performance due to two main issues. Firstly, there are large synchronisation overheads involved when running code under synchronous semantics [12]. Secondly, as time-predictability is also affected by underlying hardware, prior implementations have been presented only on lower-performance but time-predictable architectures.

Hence, this paper presents the following contributions:

- A new open-source compiler (called goFB) targeting Function Blocks, which reimplements synchronous semantics for IEC 61499, and a comparison against the previous compiler FBC [13].
- A demonstration of time-predictable multi-core execution of Function Blocks on the performant T-CREST [14] architecture. T-CREST features a network of Patmos cores, each with their own write-through caches, scratchpad memories, and Argo Network-on-Chip (NoC) connections.

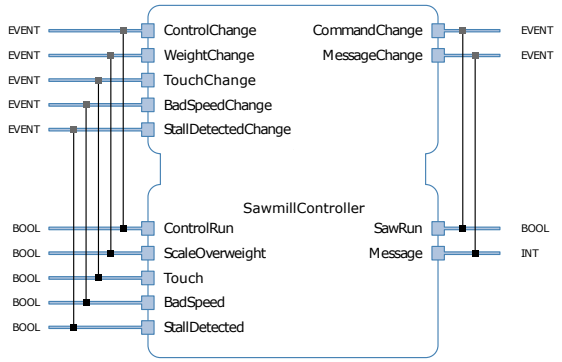


Fig. 2: Sawmill IEC 61499 FB Interface

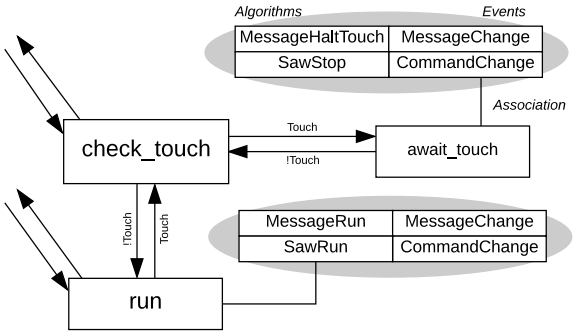


Fig. 3: (Partial) Sawmill IEC 61499 ECC

II. IEC 61499 Function Blocks (FBs)

This section introduces IEC 61499 FBs [5], the latest domain-specific language for automation systems, through the worked example of a safe sawmilling system. Firstly, the application level view in Figure 1 shows the relationshop between the various components of this system, whereby wooden logs are introduced to a number of blades, which then perform the cutting to output planks. For the purposes of this example, the sawmill has a number of safety features, including capacitive detection of skin on blade, overfull sawdust containers, and unexpected lumber velocity change detection. Any of these events occuring should cause the sawblades to stop rotating.

This system can be represented by FBs, one block of which (the controller) is depicted in Figure 2.

As can be seen, components of the system are represented directly in the IEC 61499 network via a model-driven engineering approach. Memories and behaviours are encapsulated directly into FBs, which are then composed into networks, communicating with one another via event-data interfaces, allowing for complex functionality to be realised while preserving re-usability of each individual component.

There are three main types of FBs in the standard, namely:

- *Basic Function Blocks (BFBs)*, which specify an Execution Control Chart (ECC), associated data-manipulating algorithms, and some amount of data storage. An example ECC depicting part of the sawmill controller is presented in Figure 3.
- *Service Interface Function Blocks (SIFBs)*, which are implementation-specific blocks that communicate directly with underlying hardware (such as reading/writing LEDs or switches).
- *Composite Function Blocks (CFBs)*, which provide a method of containing networks of BFBs, SIFBs, and other CFBs inside another block.

The standard also allows for networks of FBs to be contained within independent units of software known as *resources*, with resources contained within *devices*, and with networks of devices known as *systems*.

FBs communicate with one another via a defined event-data interface. This is represented by the ports and linked lines in Figure 2. Data lines only update when their associated event triggers.

To fully implement the sawmill system, 28 FBs are required if running on a normal single-core single-device architecture. If a multi-core implementation is used, additional function blocks are required to implement the communication between the cores/devices.

A. Differing Model of Computations (MoCs) for IEC 61499

For some time, IEC 61499 lacked a rigorous MoC, allowing for implementations to specify the precise behaviour of FB networks themselves. The most common MoCs became the *Sequential* acMoC [15], which executes FBs one at a time in the order that they receive events, and favoured by the FORTE [16] development environment; the *Cyclic* MoC [17], which executes FBs in a round-robin fashion according to an assigned priority, and favoured by the ISaGRAF [18] environment; and the *Synchronous* MoC, which cyclically executes FBs in logical ticks according to synchronous semantics.

III. Background and Related Work

In principle, it is simple to achieve system implementations that are amenable to timing analysis, simply by avoiding those architectural and software features that complicate the process. For instance, architectures can be utilized that are single-core, avoid caches, and don't feature speculative or out-of-order pipelines. However, while it was possible in the past to do this with industrial automation, modern requirements are starting to prevent approaches such as this from being practical. Intensive computation and sub-millisecond response times can now be required in physical process lines - such as image recognition from a camera to control high-volume conveyors managing product sorting [19].

In order to implement designs that can meet these burgeoning requirements, then, we must have performant architectures. The PRET philosophy proposes that with some effort, architectures can be designed that are both predictable and high-performance [8], [9].

However, problems can also arise at the software and application layers. Seemingly innocuous programming techniques such as interrupt-driven control flows, unbounded loop execution, function pointers, and firmware system calls can greatly

complicate timing analysis [7]. Compounding this issue, common RTOSs, which are used extensively in research and industry as aids in implementation of embedded and cyber-physical systems, extensively utilise difficult to analyse software [7].

Unfortunately, industry-standard PLCs are now seeing designs which suffer from these problems. They now employ higher-performance general purpose microcontrollers and microprocessors, and their implementations feature both RTOSs and complex runtimes that have not been demonstrated to be timing predictable [2].

A. Simplifying Timing in Hardware

The PRET philosophy [8] recognises the need for performant, predictable architectures. It proposes the usage hardware with simple, repeatable timing properties, such as scratchpads instead of caches, and thread-interleaved pipelines.

Since their proposal, a large number of PRET architectures have been developed and released, including ARPRET [20], a hardware extended MicroBlaze architecture featuring both PRET and Reactive principles; PTARM [21], a multi-threaded design featuring the ARM ISA, a thread-interleaved pipeline, and exposed memory hierarchy; and FlexPRET [22], another multi-threaded design allowing for hardware managed RTOS-like mixed-criticality systems which require some real-time guarantees. All of these architectures have demonstrated varying levels of timing predictability. However, while many of them do feature multiple hardware threads (typically due to their thread-interleaved pipelines) they universally have single-core designs, limiting their scalability and performance.

B. The T-CREST Platform

Similar to the PRET design philosophy is the T-CREST project [14], which provides the time-predictable multi-core architecture for embedded systems used in this paper. The project focuses on making the worst-case fast, and on simple analysis rather than repeatable timings. T-CREST is a multicore processor which follows the Globally Asynchronous Locally Synchronous (GALS) paradigm, where the individual processor cores are connected by two time-predictable Network on Chips (NoCs): (1) a message passing NoC called Argo [23] for communication between individual core Scratchpad Memories (SPMs) and (2) a memory NoC for communication between the cores and a memory controller for shared, external main memory.

Each processor core is a Patmos processor [24], which is a RISC pipeline optimized for low WCET bounds. Patmos contains a statically scheduled dual-issue pipeline. The Patmos processor contains scratchpad memories for time-predictable and low latency access to local data and instructions. The access to main memory is backed up by three different caches: (1) a method cache caches the instructions of full methods [25], (2) a stack cache caches data allocated on the stack [26], and (3) a data cache for the other data.

To support larger programs and data structures T-CREST uses external, shared memory. For the access to shared memory T-CREST provides two solutions: (1) the Bluetree memory tree with prefetching [27] and (2) the TDM based memory arbiter [28]. The Bluetree memory tree is optimized for the Predator memory controller [29], while the TDM arbiter is a general purpose burst based TDM arbiter.

T-CREST is supported by an adaption of the LLVM compiler that targets the Patmos instruction set and optimizes for the WCET [30]. The T-CREST project also includes the open-source, academic WCET tool called Platin [31].

C. Timing IEC 61499 Software

While IEC 61499 has plenty of literature examining the functional verification of networks under the different MoCs, such as in [32] where Dubinin et al. validate IEC 61499 networks on the basis of transition systems using the cyclic MoC, and in [33], where Yoong et al. show formal verification of IEC61499 FBs using the synchronous MoC, demonstrating reactivity and causality under any composition.

However, there is a lack of concrete examples and methodologies for *timing* verification for IEC 61499 networks in the literature. Often, proposed solutions to the timing issues focus on network-level reasoning rather than on concrete implementations. One example of this is in [34], which considers the timing composability of Function Blocks through the specification of event and internal trigger times, and validates the entire network based on these values. Another example of this is in [35], where new timing semantics for IEC 61499 were presented, adapted from *Real-time for the Masses*, which is a pre-existing set of experimental languages and tools for embedded software. However, these approaches do not provide a methodology to actually compute the execution times for the underlying IEC 61499 code on real architectures, instead focusing on reasoning on the network once those values have been obtained.

The main issue when considering concrete timing of existing IEC 61499 implementations, such as with Forte [16] or ISaGRAF [18], is that they rely on complex underlying software which have not had worst-case timing analysis performed [10].

However, the synchronous approach for IEC 61499 does not have this limitation. Synchronous programming, commonly associated with languages such as Esterel [36] and SCADE [37] involves breaking up sections of code into logical ticks, with values crossing tick boundaries only being updated at the end of each tick. For a program to be valid, it must meet the so-called *synchrony hypothesis*, which states that the delay between any environmental inputs must be greater than the amount of time the processor will take to execute any reaction, i.e., the Worst Case Reaction Time (WCRT) must be shorter than the minimum arrival time between inputs.

The synchronous approach has consistently been demonstrated to have predictable timing for IEC 61499 on PRET architectures. An example of this is in [10], where a Timed Control Flow Graph (TCFG) was constructed and WCRT derived for IEC 61499 FBs running on a simple MicroBlaze processor. Likewise, a second example is in [11], where a similar methodology was used to derive WCRTs

for IEC 61499 FBs shared amongst the multiple hardware threads on the FlexPRET architecture.

There are two main enablers for static timing analysis with the synchronous MoC. Firstly, no complex runtime or RTOS is required for the execution of the IEC 61499 code.Secondly, as the execution of the different sections of code is broken up into the concept of logical ticks, definitive points are provided whereby execution boundaries can be computed and compared from. This simplifies often-complex control flow analysis [33], [38].

Unfortunately, synchronous approaches to execution of code bring other limitations. Typically, average-case performance is worse due to the overheads in synchronising the global tick [12]. In addition, the WCRT must be knowable to ensure that programs meet the aformentioned synchrony hypothesis [39].

Some work has been undertaken to try and re-examine the underlying synchronous semantics to try and reduce this overhead - for instance in [12], GALS semantics are applied to IEC 61499 for the purposes of distributing the execution across multiple devices and cores. However, the issue again arises that no concrete timing analysis was presented for this approach.

IV. The goFB compiler

The current IEC 61499 Function Blocks compiler that targets synchronous semantics is called FBC, and was first released in [13]. This section will first describe synchronous semantics for IEC 61499, and then how the compilers run internally. Both goFB and FBC input IEC 61499 XML format [40] files, and output C code for further compilation.

A. The Synchronous Model for Function Blocks

The synchronous MoC for IEC 61499 Function Blocks sees FBs within a network as concurrently executing modules of a synchronous system. This is convenient as industrial control software is often described as a collection of concurrently-running processes.

In the model, the notion of the logical tick is mapped directly to the period of a *scan cycle*. During each tick, the following steps happen:

1) Input signals captured.
2) Each FB within the network is invoked.
3) Output signals emitted.

Hence, all function blocks in a given network are conceptually viewed as running concurrently in lock-step with one another, performing atomic computations in each tick.

Across a resource, the atomic operations within a tick consist of the evaluation of all ECC transitions in all current ECC states, and the corresponding computation of the action(s) in all destination states, should individual transitions be taken. The lifetime of events is strictly defined to persist for the duration of the tick in which it occurred. Thus, no more than one ECC transition can occur within a tick in each ECC within a network. Note that this differs to the official IEC 61499 standard [5], whereby the run of an ECC should consume one input event, and continue until no more valid transitions can occur.

Since all inputs are read at the start of the tick, and the tick itself is conceptually instantaneous, simultanous events are possible in this model, as in other cyclic-scan models [33]. This is again in contrast to the official standard, which states that only one event may be present in any given instant in an IEC 61499 network [5].

However, if the standard is followed, and multiple events can be emitted in a given ECC state, and event-connection loops exist in the network, the official event-driven approach will either result in the purposeful loss of events, or the need for unbounded queues to store events. Both of these are undesirable in safety-critical systems.

Even with the synchronous model, however, compositions of function blocks involving event connection loops may still be problematic, with event feedback loops possibly resulting in non-causal cycles, where the distinction between input and output events are blurred [33]. In synchronous systems, non-causal cycles will manifest themselves as either deadlocks or as starvations of modules not part of the cycle. As a result of this, all FB communication in the synchronous MoC is delayed by one cycle, to the next tick.

This pipelining of the send and receive operations in each function block guarantees that their parallel composition will always be acyclic [33]. In addition, since all communications within a function block network are delayed by one cycle, the function blocks can be arbitrarily scheduled, while still ensuring an overall deterministic behaviour.

B. Compiling Function Blocks (FBs)

Both FBC and goFB represent compiled function blocks with a C structure and several associated functions. To derive these functions, the compilers traverse the FB network, via a depth-first search through all layers, then perform a bottom up compilation of each block in the network.

1) Compiling Basic Function Blocks (BFBs): For both compilers, the main task in compiling a BFB is the translation of the internal ECC and associated data-manipulating algorithms into a C `run` function. Internally, the `run` function must perform the following steps:

(i) In the current state, check possible exit transitions in order of priority.
(ii) If a transition can be taken, enter destination state.
(iii) Invoke associated algorithms, then emit associated events/data.

This is managed by converting the ECC to a C `switch` statement. In FBC there is a single `switch` for both transitions and algorithms, inside a double-entering `for` loop. This has the unintended consequence of causing most static timing analysis tools to believe that the worst-case path is through the long algorithms twice (even though this path is not possible).

goFB's procedure for BFB generation is presented in Listing 1. As can be seen, the loop is unwound, and there are two `switch` statements generated instead.

2) Compiling Composite Function Blocks (CFBs): When compiling CFBs, both compilers convert the internal list of components into a netlist, embedding instances of the

Listing 1: goFB's BFB Generation

```
procedure GenerateBFB(fb)
  S := set of all states in fb;
  IE := set of all input events in fb;
  OE := set of all output events in fb;
  ID := set of all input data in fb;
  OD := set of all output data in fb;
  T := 1 if fb took a transition this tick;
  clear all output events in OE;
  foreach s ∈ S do
    write new case for s in switch-statement;
    foreach transition condition, t, of s do
      if t leads to next state n ∈ S then
        generate code to test for t;
        assign next state to n;
        assign T = 1;
      end
    end
  end
  write check for T containing:
    foreach s ∈ S do
      write new case for s in switch-statement;
      foreach action, a, of s do
        if a has algorithm, alg then
          generate call to function[alg];
        end
        if a has oe ∈ OE then
          set oe;
        end
      end
    end
  end
end procedure
```

Listing 2: goFB's CFB Generation

```
procedure generateCFBPullEvents(fb)
  B := set of all fbs in the network of fb;
  EO := set of event output ports on fb;
  foreach b ∈ B do
    if b.type = CFB then
      generate call to this function on b;
    end
  end
  foreach eo ∈ EO do
    I := eo.connectionSet();
    generate code to assign eo = or of all i∈I;
  end
end

procedure generateCFBPushEvents(fb)
  B := set of all fbs in the network of fb;
  foreach b ∈ B do
    bEI := b.EventInputs;
    foreach bei ∈ bEI do
      I := bei.connectionSet();
      generate code to assign bei = or of all i∈I;
    end;
  end
  foreach b ∈ B do
    generate call to this function on b;
  end
end

procedure generateCFBPullData(fb)
  B := set of all fbs in the network of fb;
  D := set of data output ports on fb;
  foreach b ∈ B do
    if b.type = CFB then
      generate call to this function on b;
    end
  end
  foreach d ∈ D do
    I := d.connectionSet();
    generate code to assign d = I;
  end
end

procedure generateCFBPushData(fb)
  B := set of all fbs in the network of fb;
  foreach b ∈ B do
    if b.type = BFB then
      bEI := b.EventInputs;
      bDI := b.DataInputs;
      foreach bei ∈ bEI associated with bdi ∈ bDI do
        bdi′ := bdi.connectionSet();
        update bdi with bdi′ when bei occurs;
      end
    else
      foreach bdi ∈ bDI do
        bdi′ := bdi.connectionSet();
        update bdi with bdi′;
      end
    end
  end
end

procedure generateCFBRun(fb)
  B := set of all fbs in the network of fb;
  foreach b ∈ B do
    call run function on b;
  end
end

procedure GenerateTopRun(f)
  f := topmost CFB;
  generate call to CFBPullEvents(f);
  generate call to CFBPushEvents(f);
  generate call to CFBPullData(f);
  generate call to CFBPushData(f);
  generate call to CFBRun(f);
end procedure
```

component blocks inside a structure in C, and listing the members of the event-data interface.

FBC breaks CFBs into two functions, `init` and `run`. `init` ensures correct initialisation of all components in the network, and `run` recursively calls the `run` functions of those embedded blocks as well as blindly copying events and data around the network.

This is because in FBC compile code, there are two copies of each data variable inside each BFB instantiation, an internal, and an external. The external variable is updated every cycle by the connected FBs, and inside the BFB, whether it needs to be or not, and the data is only copied to the internal variable if the associated event is active.

This blind copy is a source of much inefficiency in FBC, especially in architectures with write-through caches (such as T-CREST). Every IO signal in the entire network is propogated, even if it hasn't changed, and even if it is going to be ignored at the destination.

In goFB, this behaviour is optimised away by removing the extra copy of variables. Instead of a blind copy, the parent block itself examines the events associated with data lines (i.e. a memory read), and only performs the costly write command if necessary. To achieve this, synchronisation is separated of the `run` function and into a number of other `sync` functions, similiar to how FBC's code was broken up in [11].

During each tick, communication is resolved via several depth-first traversals of the IEC 61499 Function Block networks, first for events, and then for data. Once this is completed, each FB is invoked in turn, just like with FBC. This communication can be seen in Listing 2[1]. As can be seen, in each tick, firstly all events are resolved by recursively calling PullEvents and then PushEvents, which will copy events between all function blocks. Then, data that can be copied, is copied, by recursively calling PullData and PushData. There must be two functions as data must be able to propagate from the most embedded FBs to the least embedded and vice versa between each tick. Events and Data must be handled separately, as all events must be resolved before any associated data copies can occur.

[1] `connectionSet()` returns the set of connections to a given port.

C. Multi-core Execution on T-CREST with FBC and goFB

Both FBC and goFB produce code amenable to multi-core execution, due to their support of the conceptual IEC 61499 *resource*. Resources are meant to be self-contained networks

of FBs, with no direct inputs nor outputs, except via those provided through internal SIFBs (such as to hardware or networking). However, until now, this has never been undertaken on a time-predictable multi-core platform.

For this paper, both goFB and FBC's outputted C code was given a new top-level file to distribute the resources amongst the different cores of T-CREST, rather than sequentially executing them on a single core.

Because T-CREST has bounded timing on all parts of its execution semantics, including within its predictable memory hierarchy and arbitration, the entire system as a whole can be considered timing predictable if the running software is also time predictable.

In addition, as T-CREST's NoC Argo is also time predictable, timing bounds for applications that use it (such as the Sawmill) can still be obtained.

V. Results

goFB was initially benchmarked against FBC using the T-CREST and Patmos platforms running at 50MHz with the worked sawmill example from Section II. Six benchmarks were conducted, and WCRT from both static analysis (using the static analysis tool platin [31]) and measurement (using the built-in cycle-accurate timers) were gathered and compared. Each processor cycle takes 20 nanoseconds. When compiling to the four-core T-CREST architecture, it was necessary to change the top-level C file, as described in Section IV-C.

The six benchmarks are:

1) (g-T4-S) goFB compile FBs to four-core T-CREST, using SPM.
2) (g-T4-M) goFB compile FBs to four-core T-CREST, using main memory.
3) (g-P1-S) goFB compile FBs to single-core Patmos, using SPM.
4) (g-P1-M) goFB compile FBs to single-core Patmos, using main memory.
5) (F-T4-M) FBC compile FBs to four-core T-CREST, using main memory.
6) (F-P1-M) FBC compile FBs to single-core Patmos, using main memory.

A. Results for the Sawmill Scenario

The results for the sawmill example are shown in Figure 4. It used 28 FBs in single-core mode, and 37 when on the four-core T-CREST platform (the increase is due to the Argo-specific FBs). The original design took 1.63 milliseconds of processor time in the worst case to detect skin contact.

As can be seen, each of the optimisations in turn resulted in a measured speed increase, although just using FBC with the multi-core architecture resulted in a statically analysed worse-case — this is due to the combination of large numbers of read-write operations in FBC causing possible contention in the Bluetree memory hierarchy.

Individually, moving from FBC to goFB sees a 30.1 % lower analysed worst-case time in the single-core Patmos, and 37.0 % lower in the multi-core design. As expected, using the SPM gives further improvements, for instance giving an additional 27.4 % speed increase for the four-core T-CREST.

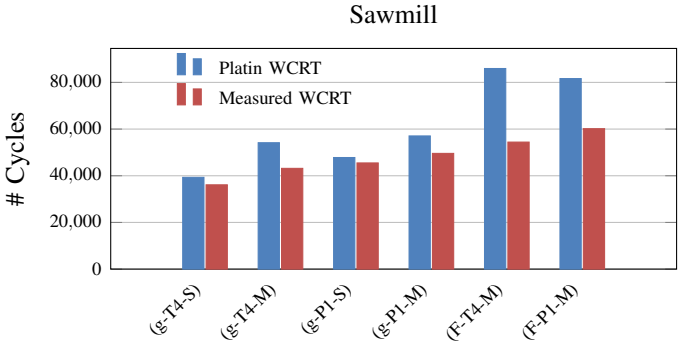


Fig. 4: Sawmill Scenario

Overall, with all optimisations taken into account (increasing cores, using goFB, and using the SPM) now takes 786 microseconds to respond in the worst case, giving a total speed increase of 51.8 % from the original

B. Results for other Scenarios

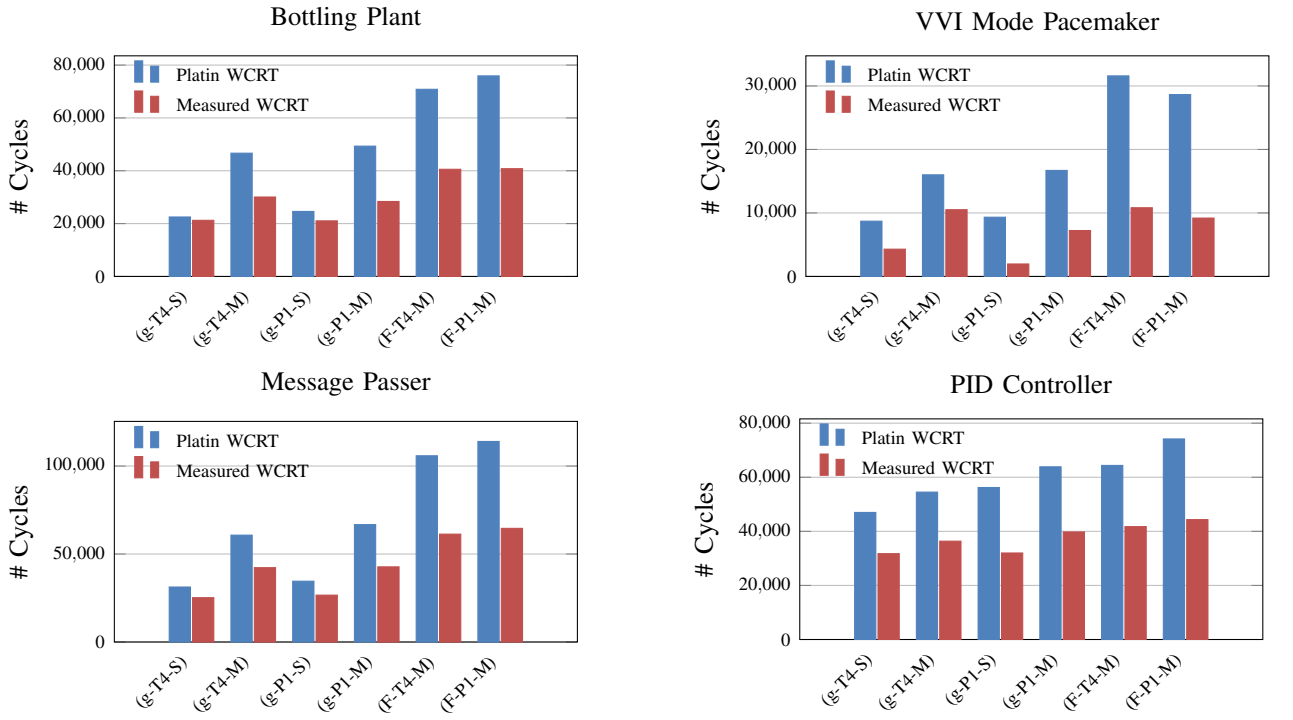
In addition to the sawmill scenario, four other IEC 61499 networks were examined using this methodology on the T-CREST platform, the results of which are presented in Figure 5a. Bottling Plant (36 FBs) and VVI Mode Pacemaker (32 FBs) were presented in [11], and represent the controller for a bottling plant and pacemaker respectively. Message Passer (88 FBs) and PID Controller (20 FBs) are two additional networks created for this paper. Message Passer is simply a chain of FBs that receive tokens from a generator and pass them on to a receiver. PID Controller implements a PID controller and simple plant.

As can be seen, all of these networks also show similar results to the sawmill example. There is always a speed-up when moving from FBC to goFB, and usually a speed-up when moving from single-core to four-core. In addition, and as expected, there is always a speed up when the data for the networks can be located on the SPM instead of main memory. This is because the SPM has single-cycle access time, whereas main memory requests must pass through the Bluetree memory hierarchy if the data is not cached and ready to access.

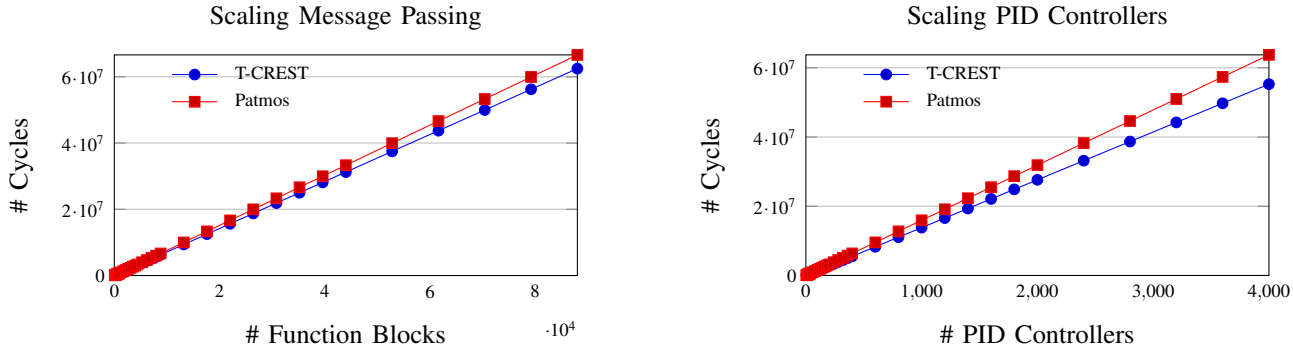
C. Scaling large networks with multiple cores

To examine the speed-up of large networks when executing on the four-core T-CREST compared to the single-core Patmos further, the PID Controller and Message Passer networks were each duplicated a varying number of times, and WCRT computed statically by Platin. The results of this are shown in Figure 5b. As can be seen, the WCRT savings (according to Platin) is almost static, for the PID Controller at around 15% and for the Message Passer at around 7% (reflecting the increased computational difficulty in the PID Controllers).

Typically, benefits from executing code on multi-core architectures present themselves when the code to be executed



(a) Other Scenarios



(b) Multi-Core Scaling

Fig. 5: Results

features low numbers of memory accesses and high numbers of computational instructions. Unfortunately, FBs typically feature relatively low computation between loads and stores of their internal data. As a result, there typically isn't a significant gain when moving to a multi-core architecture (of the five networks examined, the savings were around 10% each time).

VI. Conclusion

In this paper a new approach for compiling IEC 161499 Function Blocks to C using synchronous semantics is presented and compared with the old methodology from the FBC compiler. The output is demonstrated to be significantly faster than the old compiler's output. In addition, multi-core execution using both the old and new compiler is demonstrated on T-CREST, although the multi-core execution of IEC 61499 proved to be somewhat lacklustre due to the high memory bandwidth requirements of each core (as FBs typically have large numbers of memory operations compared to their pure computation instructions.

The optimised approach, using the four-core architecture, goFB, and SPMs, proved to have 60 % lower statically analysed Worst-Case Reaction Times (on average) than FBC running on a single-core Patmos. In the sawmill scenario, this meant savings of 844 microseconds, meaing that more of the hard real-time deadline of 5 milliseconds could be used by other physical components of the braking system.

VII. Source Access

The source code for goFB, and all examples used in this paper, are available under the MIT License at https://github.com/PRETgroup/goFB.

REFERENCES

[1] SawStop. (2018) How it works. [Online]. Available: www.sawstop.com/why-sawstop/the-technology

[2] M. M. Y. Kuo, S. Andalam, and P. S. Roop, "Precision timed industrial automation systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1024–1025.

[3] A. Zoitl, *Real-Time Execution for IEC 61499*. ISA, 2008.

[4] M. D. Schwartz, J. Mulder, J. Trent, and W. D. Atkins, "Control system devices: Architectures and supply channels overview," *Sandia Report SAND2010-5183, Sandia National Laboratories, Albuquerque, New Mexico*, 2010.

[5] *International Standard IEC 61499-1: Function blocks - Part 1: Architecture*, International Electrotechnical Commission Std., April 2013.

[6] V. Vyatkin, "The IEC 61499 standard and its semantics," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 40–48, Dec 2009.

[7] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A survey of WCET analysis of real-time operating systems," in *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, May 2009, pp. 65–72.

[8] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 264–265.

[9] I. S. Liu, "Precision timed machines," Ph.D. dissertation, University of California at Berkeley, 2012.

[10] M. Kuo, L. H. Yoong, S. Andalam, and P. Roop, "Determining the worst-case reaction time of IEC 61499 function blocks," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, July 2010, pp. 1104–1109.

[11] H. A. Pearce, M. M. Y. Kuo, P. S. Roop, and M. Biglari-Abhari, "RunSync: A predictable runtime for precision timed automation systems," in *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2016, pp. 116–123.

[12] L. H. Yoong, G. D. Shaw, P. S. Roop, and Z. Salcic, "Synthesizing globally asynchronous locally synchronous systems with IEC 61499," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1465–1477, Nov 2012.

[13] L. H. Yoong, P. Roop, and Z. Salcic, "Efficient implementation of IEC 61499 function blocks," in *Industrial Technology, 2009. ICIT 2009. IEEE International Conference on*, Feb 2009, pp. 1–6.

[14] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[15] V. Vyatkin and V. Dubinin, "Sequential axiomatic model for execution of basic function blocks in IEC 61499," in *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 2, June 2007, pp. 1183–1188.

[16] 4DIAC. (2015, Dec.) FORTE. [Online]. Available: http://www.eclipse.org/4diac/en_rte.php

[17] P. Tata and V. Vyatkin, "Proposing a novel IEC61499 runtime framework implementing the cyclic execution semantics," in *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, June 2009, pp. 416–421.

[18] I. T. I. Inc. (2015, Dec.) ISaGRAF. [Online]. Available: http://www.isagraf.com/

[19] W. Dai and V. Vyatkin, "Redesign distributed PLC control systems using IEC 61499 function blocks," *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 2, pp. 390–401, April 2012.

[20] S. Andalam, P. Roop, and A. Girault, "Predictable multithreading of embedded applications using PRET-C," in *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, July 2010, pp. 159–168.

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

[22] M. Zimmer, D. Broman, C. Shaver, and E. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, April 2014, pp. 101–110.

[23] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø, "Argo: A real-time network-on-chip architecture with an efficient GALS implementation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 479–492, 2016.

[24] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, Grenoble, France, March 2011, pp. 11–20.

[25] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for Patmos," in *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*. Reno, Nevada, USA: IEEE, June 2014, pp. 100–108.

[26] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time-predictable stack cache," in *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[27] J. Garside and N. C. Audsley, "Prefetching across a shared memory tree within a network-on-chip architecture," in *System on Chip (SoC), 2013 International Symposium on*, Oct 2013, pp. 1–4.

[28] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, "A time-predictable memory network-on-chip," in *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, Madrid, Spain, July 2014, pp. 53–62.

[29] M. D. Gomony, B. Akesson, and K. Goossens, "Architecture and optimal configuration of a real-time multi-channel memory controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 1307–1312.

[30] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard, "The T-CREST approach of compiler and WCET-analysis integration," in *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, 2013, pp. 33–40.

[31] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner, "The platin tool kit - the T-CREST approach for compiler and WCET integration," in *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[32] V. Dubinin, V. Vyatkin, and A. Shalyto, "Formal modeling and verification of IEC 61499 function blocks on the basis of transition systems," in *2016 International Siberian Conference on Control and Communications (SIBCON)*, May 2016, pp. 1–4.

[33] L. H. Yoong, P. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *Computers, IEEE Transactions on*, vol. 58, no. 12, pp. 1599–1614, Dec 2009.

[34] L. Lednicki, J. Carlson, and K. Sandström, "Model level worst-case execution time analysis for IEC 61499," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '13. New York, NY, USA: ACM, 2013, pp. 169–178.

[35] P. Lindgren, M. Lindner, A. Lindner, V. Vyatkin, D. Pereira, and L. M. Pinho, "A real-time semantics for the IEC 61499 standard," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, Sept 2015, pp. 1–6.

[36] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(92)90005-V

[37] (2016, September) SCADE. Esterel Technologies. http://esterel-technologies.com.

[38] L. H. Yoong, P. S. Roop, Z. E. Bhatti, and M. M. Y. Kuo, *Model-Driven Design Using IEC 61499: A Synchronous Approach for Embedded and Automation Systems*. Springer Publishing Company, Incorporated, 2014.

[39] E. Yip, M. M. Y. Kuo, P. S. Roop, and D. Broman, "Relaxing the synchronous approach for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 89–100.

[40] *International Standard IEC 61499-2: Function blocks - Part 2: Software Tool Requirements*, International Electrotechnical Commission Std., April 2013.