In proceedings of 12-th IEEE International Symposium on Distributed Simulation and Real Time Applications(IEEE DS-RT 2008), Vancouver, British Columbia, Canada, October 2008

An Automated Mapping of Timed Functional Specification to A Precision Timed Architecture

Shanna-Shaye Forbes, Hiren D. Patel and Edward A. Lee
Dept. of Electrical Engineering and Computer Sciences
545 Cory Hall, University of California, Berkeley
Berkeley, CA 94720.
{sssf,hiren,eal}@eecs.berkeley.edu

Hugo A. Andrade
National Instruments Corporation
1936 University Avenue, Suite 320
Berkeley, CA 94704.
hugo.andrade@ni.com

Abstract

Most common real-time embedded programming languages provide a means to specify functionality; however, they have few constructs to specify precise timing constraints. LabVIEW is one example of a graphical programming language that supports timing specifications in the form of timed-loops. In this work, we present a plug-in for LabVIEW Embedded that maps the LabVIEW G graphical programming language and its timing specifications to the PREcision Timed machine (PRET), an architecture that exposes timing instructions in its instruction set architecture. We demonstrate the use of the plug-in with a simple producer/consumer example that uses timing to enforce synchronization.

1 Introduction

In real-time embedded systems, the ability to specify timing requirements is just as important as specifying functionality. Most embedded programming languages provide mechanisms for implementing functionality, but very few allow specifying timing requirements. This is in part because languages such as C abstract away the notion of time at the software level. For real-time embedded systems however, the ability to specify timing must be native to the programming language so that both time and function become integral parts of the design. The objective of this work is to take LabVIEW and its notion of timing and map it to a timing-aware embedded processor architecture called the precision timed (PRET) machine [3, 6].

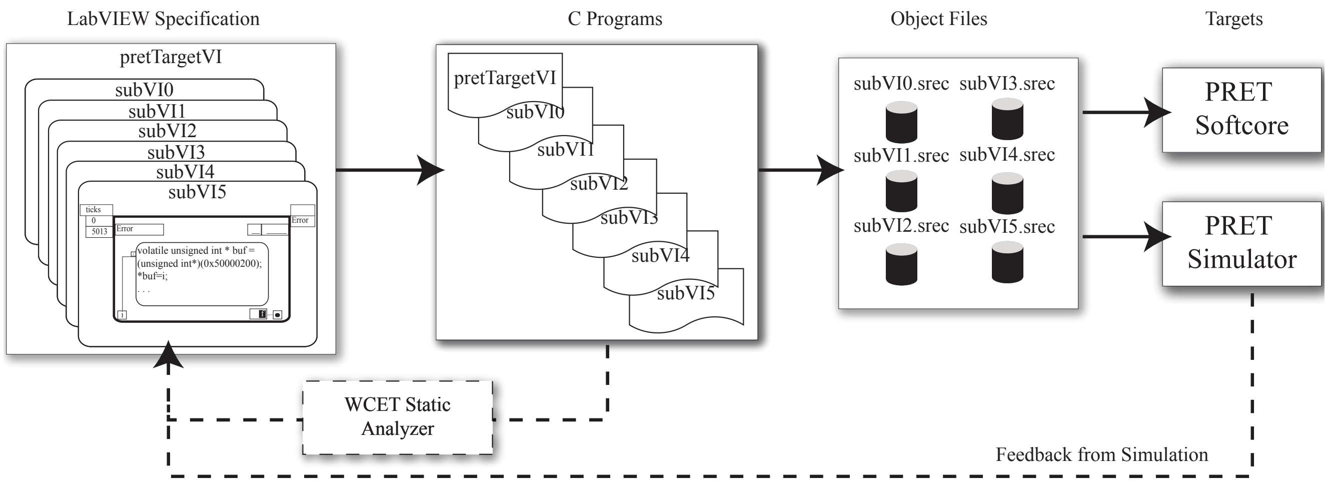
In this work-in progress, we present a plug-in for LabVIEW Microprocessor SDK [8] which specifies timing requirements at the software level and maps to the PRET architecture. Currently, our plug-in supports a timed-loop construct which ensures that a set of functions meet a lower-bound timing requirement. We show our proof of concept with a simple resource-sharing example programmed in LabVIEW's G graphical language, and synthesized to run on the PRET cycle-accurate simulator.

2 Background

2.1 Precision Timed Machine (PRET)

PRET is a real-time embedded processor architecture that has instructions to control timing in its instruction set architecture (ISA). We select the PRET architecture because it implements a subset of the SPARC ISA [12] and has instruction set extensions that provide control over timing. PRET upholds the philosophy of making temporal characteristics as predictable as functionality and is built for straightforward worst-case execution time analysis [3, 6].

PRET has a fine-grained multi-threaded architecture with scratchpad memories and time-triggered access to shared main memory. It provides precise timing control to software via a *deadline* instruction that allows the programmer to set and access cycle-accurate timers. The *deadline* instruction sets a deadline register to the specified value if the current value of the register is zero. It then continues to decrement at the main system clock while the program code continues to execute. On another occurrence of a deadline instruction, it blocks the hardware thread whenever the specified deadline register being written to is not yet zero. If the deadline register reaches zero before the corresponding deadline instruction is executed, then we have missed a real-time deadline. A PRET design could take one of several actions. It could ignore the event, in which case the timing guarantee is only a minimum execution time. Or it could raise an exception and execute user-provided code to handle the exception. We assume for now the former. In essence, the *deadline* instruction amounts to setting a lower bound *deadline* on the execution time of a segment of pro-



```
volatile unsigned int * buf =
(unsigned int*)(0x50000200);
*buf=i;
. . .
```

Figure 1. Plug-in Usage

gram code. At the moment, the PRET architecture only has a cycle-accurate simulator [6]. This simulator accepts programs written in C with additional *deadline* instructions. To demonstrate our plug-in, we use this simulator as our target.

Examples of other processors that feature predictable timing are JOP [10], SPEAR [2], KEP [5], and REMIC [9]. JOP makes a case for the use of Java in the development of embedded real-time systems and enables accurate worst-case execution time bounds but provides no means of controlling execution time. SPEAR is designed to meet the timing needs of real-time applications and executes code with small temporal jitter, but it does not allow conditional branches. KEP and REMIC provide precision timing but are limited to the synchronous reactive programming language Esterel [1].

2.2 LabVIEW Microprocessor SDK

LabVIEW Microprocessor SDK is the embedded version of National Instruments' graphical programming interface in the G programming language. G is an actor-oriented [7] programming language with a structured data-flow model of computation and has a C software synthesis back-end to automatically generate code. Note that LabVIEW is just one of the many possible frameworks. For example, Ptolemy II [11] also uses actor-oriented programming and it has support for C code generation from several of its models of computation. An interesting approach we plan to investigate is using the Giotto [4] programming model in Ptolemy II to target the PRET architecture. A large contributing factor for selecting LabVIEW is its industry-strength C software synthesis back-end.

LabVIEW enables domain experts (e.g. chemists, physicist, and automotive engineers) to program embedded devices with a graphical programming environment. An interesting language feature of LabVIEW is the notion of timed-loops. Timed-loops enable the designer to specify the period at which a set of functions are to be executed. The semantics of timed-loops are: it executes one or more sub diagrams, or frames, sequentially in each iteration of the loop at the specified period. The timed-loop can be used when one wants to develop VIs with multi-rate timing capabilities, precise timing, feedback on loop execution, timing characteristics that change dynamically, or several levels of execution priority. Figure 2 shows a sampling of different options available for timed-loops. The configuration tab allows you to specify the initial controls (i.e. frequency of the timed-loop). The run time status tab provides information on the progress of the loop. The run time configuration tab allows you to modify the initial configuration settings during execution and the final status tab provides information on the last execution of the timed loop.

The C code synthesized by the LabVIEW Microprocessor SDK is combined with target specific operating system APIs to determine the time elapsed and to schedule the thread for execution. Since the timed-loops rely on an operating system, the timing predictability cannot be guaranteed, beyond the jitter characteristic of a given operating system on a given processor (with appropriate external hardware timer support). Today, it is difficult to predict such jitter statically, so the timed-loop provides a dynamic run-time exception mechanism in which the end-user can specify in G what action to take, should the actual period exceed the specified value. It should be noted that there are some targets supported today by LabVIEW, e.g. FPGAs, where the static analysis of the content of a timed-loop can be performed. PRET promises to also bring this flexibility, ease and speed of compilation of a processor with the static analyzability of FPGAs, which makes it particularly interesting as a LabVIEW target.

In our plug-in we implement timed-loops with the *deadline* instruction to provide a minimum time deadline for the timed-loop. We currently take no action if the period specified by the timed-loop is exceeded.

3 Automated mapping of LabVIEW Microprocessor SDK to PRET

We created a PRET target whose usage flow is shown in Figure 1. Thus far, we have successfully implemented the flow depicted with solid arrows. The dotted paths are areas that need further exploration.

The PRET target requires that concurrent tasks are specified in one of the six subVIs included in a Virtual Instrument shown in Figure 1. These subVIs use existing actors from G in addition to the timed-loops to specify the intended real-time embedded application. We only allow six subVIs because the PRET core only supports six hardware threads. Virtualization of these threads is also a subject for further study. Using LabVIEW's C software synthesis back-end, we can generate C code for each of the subVIs. The target then post-processes the generated C files by separating the generated code into six independently running C files and removing operating system code. Then, we replace the timing specification segments of the code with a method invocation that implements the instructions for setting deadlines.

Note that we take a designer's period specification either in engineering units such as milliseconds or in cycles for the target PRET architecture. Converting engineering units to cycle count is based on the processor speed and the clock base specified in the timed-loop. Each subVI is then mapped to one of the available hardware threads. The flow then compiles the C files with the SPARC GCC tool chain and runs the post-processed program on PRET's cycle-accurate simulator. The idea with the dotted paths from the simulator and the C programs is to provide feedback to the designer regarding feasibility of the deadlines specified in the G. This is currently not a part of the plug-in and remains a direction to pursue in the future. In essence the PRET target:

- Generates C code using the LabVIEW Code Generator
- Converts the LabVIEW timed loop specifications to the *deadline* instruction
- Maps each of the 6 subVIs to a PRET hardware thread
- Compiles annotated LabVIEW generated code with the SPARC's GCC tool chain
- Executes the LabVIEW specified program in the PRET simulator

A difficulty with the current approach shown in Figure 1 is the need to post-process the generated C code. Ideally, we want to traverse the LabVIEW internal model database and generate the appropriate C code. We are currently working with National Instruments to get access to the models.

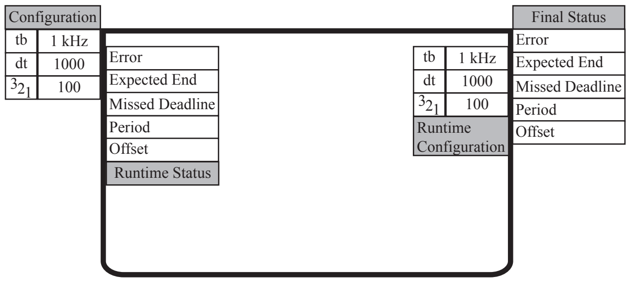


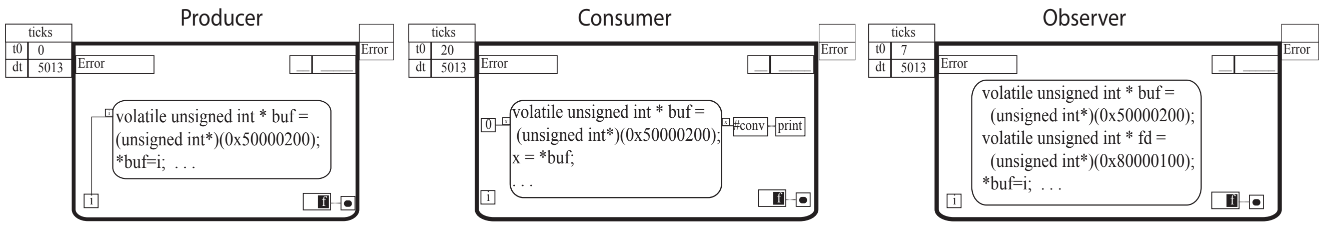
Figure 2. Timed-loop. Reproduced with permission [8].

4 Example

We create the simple producer-consumer example from [6] using our plug-in with the timed-loop construct for shared memory synchronization. This is a classical mutual exclusion problem where we must deal with shared resources. PRET does not have an operating system, so synchronization must be done explicitly. The authors in [6] use *deadline* instructions to specify synchronization, so we use that convention in our plug-in. Computing the values for these deadlines is done by taking on the role of a worst-case execution time analyzer. Figure 3 shows this example implemented with three threads; `Producer`, `Consumer` and `Observer`. The producer writes an integer to the shared resource, the consumer reads an integer from the shared resource, and the observer reads an integer and writes it to a memory-mapped peripheral for output. For simplicity in execution time analysis, we do not perform any additional computation on the consumed data.

In the LabVIEW implementation of the application, we have been able to use timed-loops to precisely control the period timing of the `Producer`, `Consumer` and `Observer`. In addition to the period we have implemented a timing offset, i.e. the relative phase delay before the beginning of the timed-loops. In this case, we have specified the timing in clock ticks. Alternatively, we are able to specify the timing in engineering units, e.g. milliseconds. We automatically convert this time specification to the cycle counts required for the deadline instruction.

The PRET architecture ensures that access to shared data is atomic within each of the six hardware threads and as a



```
volatile unsigned int * buf =
(unsigned int*)(0x50000200);
*buf=i; . . .
```

```
volatile unsigned int * buf =
 (unsigned int*)(0x50000200);
x = *buf;
. . .
```

```
volatile unsigned int * buf =
  (unsigned int*)(0x50000200);
volatile unsigned int * fd =
  (unsigned int*)(0x80000100);
*buf=i; . . .
```

Figure 3. Simple Mutual Exclusion Example in LabVIEW.

result necessitates only an ordering of the accesses [6]. We determine the period and offsets for the producer-consumer example shown in Figure 3 to ensure synchronization. The `Observer` is the most computationally expensive thread taking 5013 ticks to complete so we set this as the period of each timed loop. The `Observer`'s computation time is long because it calls the `printf` function which generates a large number of additional instructions whose execution time must be included in the deadline count. We also want to ensure that the `Consumer` and `Observer` read data right after the `Producer` generates it so we set the offsets of the `Observer` and `Consumer` to 7 and 20 respectively. The offsets for the `Consumer`, `Observer`, and `Producer` differ because the setup code for each executes a different number of instructions; thus taking different amounts of execution time.

5 Conclusion and Future Work

To demonstrate the effectiveness of a programming language and direct hardware support for precise timing we will develop more time critical examples such as a VGA example in LabVIEW. We will develop the plug-in so that it works with nested timed-loops and timed sequences. This however, requires processing the specified model through an object reference model instead of post-processing the generated code. In addition, we will also work with other models of computation such as Giotto in the Ptolemy II framework, which specify timing requirements at the software level that can be mapped to the PRET architecture.

6 Acknowledgements

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

References

[1] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.

[2] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor Support for Temporal Predictability–The SPEAR Design Example. pages 169–176. 15th Euromicro Conference on Real-Time Systems, 2003.

[3] S. A. Edwards and E. A. Lee. The Case for the Precision Timed (PRET) Machine. *Proceedings of the 44th Design Automation Conference*, pages 264–265, June 2007.

[4] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. volume 91(1) of *Proceedings of the IEEE*, pages 84–99, 2003.

[5] X. Li, M. Boldt, and R. von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. pages 912–917. ACM Symposium on Applied Computing, ACM Press, New York, NY, April 2006.

[6] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable Programming on a Precision Timed Architecture. Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2008.

[7] J. Liu, J. Eker, J. Janneck, X. Liu, and E. A. Lee. Actor-oriented Control System Design: A Responsible Framework Perspective. *Control Systems Technology, IEEE Transactions on*, 12(2):250–262, March 2004.

[8] National Instruments Corporation. NI LabVIEW The Software that Powers Virtual Instrumentation. http://www.ni.com/labview.

[9] Z. Salcic, D. Hui, P. Roop, and M. Biglari-Abhari. REMIC: Design of a Reactive Embedded Microprocessor Core. pages 977–981. 2005 Conference on Asia South Pacific Design Automation (Shanghai, China, January 18 - 21, 2005)., M 2005.

[10] M. Schoeberl. A Java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, 2008.

[11] The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/.

[12] SPARC International Inc. The SPARC Architecture Manual. http://www.sparc.org/standards/V8.pdf.