Fine-Grained Timing Constraints for Reactive Systems in ANSI C

Gero Dittmann
IBM Research
Zurich Research Laboratory
8803 Rüschlikon, Switzerland
ged@zurich.ibm.com

Andreas Herkersdorf
Institute for Integrated Systems
Munich University of Technology
80290 München, Germany
a.herkersdorf@ei.tum.de

Abstract

The pervasive language for the specification of embedded software today is C. Current approaches to timing specification in C require a special coding style or provide only a coarse-grained timing resolution. Reactive systems, such as process control or network processing, however, require cycle-accurate timing of individual I/O operations. Moreover, the timing constraints in these systems often depend on system inputs at runtime. Today, these dependencies cannot be modeled abstractly, which prevents a synthesis system from automatically choosing an optimum implementation strategy.

In this paper we present novel methods to specify fine-grained timing constraints in C, to abstractly specify constraints that depend on runtime data, and to represent these constraints in a new timing layer for intermediate representation. Our methods rely on pragma annotations, which makes them ANSI-C-compliant and enables the reuse of existing code.

1 Introduction

Many reactive systems are characterized by fine-grained timing constraints. If an input register is not read in the right cycle it may be overwritten with a new value, and if an output is not generated on time, it may not have the intended effect. Moreover, reactive systems often have timing constraints that depend on input data at run-time, e.g., to process a stream of network-packet data depending on a header-length field [1].

For the automated design of hardware that meets these constraints, a designer must annotate the constraints in the specification. Common approaches to timing specification in behavioral high-level languages (HLLs) provide only coarse-grained resolutions and data-dependent constraints cannot be expressed with current language constructs. This calls for a novel system of annotations. In order to facilitate the reuse of existing HLL code in the design process it is desirable that the constraint annotations do not require a redesign of the code. For the application analysis, the code with the annotations must then be transformed to a graph representation known as *intermediate representation (IR)* in compilers. The IR must represent all timing constructs of the HLL.

After a survey of related work in Section 2, we introduce a new method to express fine-grained timing constraints in ANSI C in a standard-compliant way in Section 3. The method includes a novel construct for data-dependent waiting. Section 4 presents our new IR layer of timing constructs, including a data-dependent wait, and Section 5 shows how to implement and schedule the wait operation. Section 6 concludes the paper.

2 Related Work

The fundamentals of the classification, specification and verification of timing constraints have been studied in [2]. Most methods found in the literature express minimum and maximum timing constraints as proposed there.

Interestingly, classical hardware description languages (HDLs), such as VHDL [3] or SystemC [4], only have a basic notion of time to specify strict simulation timing. They do not provide constructs to specify minimum, maximum, or range constraints that allow optimizations for synthesis [5].

Various HLLs include constructs to specify timing constraints, e.g., an annotated version of Esterel [6] and Real-Time for Java [7]. While ANSI C [8] does not provide any means to express timing information, there have been attempts to use C derivatives as HDLs, e.g., C^x [9] and HardwareC [10]. The programming style of these derivatives, however, significantly differs from ANSI C, e.g., in constructs to model parallel processes. Therefore, these languages require a fundamental rewrite of existing applications. Moreover, the derivatives introduce extensions that are not standard-compliant. Hence, the code can no longer be processed by common ANSI C tools.

3 Precise Timing Constraints in ANSI C

Most existing software for embedded systems is available in C only—tested and well understood. But C does not provide means to express any kind of timing information. Re-implementation of all applications in another language for synthesis is not a viable option. Therefore, we need an extension to C that allows the designer to supplement existing software with timing constraints without requiring a major rewrite of the code. In this section we describe a method to integrate timing information into C code in an ANSI-C-compliant way.

A timing constraint is defined by the following set of information [2]:

- The two points in the code, in particular two instructions, between which the constraint applies.
- The minimum time that must elapse between the execution of the two instructions.
- The maximum time that must not be exceeded between the execution of the two instructions.

We found that the time to pass between two operations can also depend on input data, i.e., the value in an input register may encode a time that must pass between two events.

The unit in which the time is given can be seconds or clock cycles of the system to be designed. Time values in seconds will have to be rounded to a multiple of the cycle time of the system once this has been determined. Time values in clock cycles require that the cycle time of the system be determined already when specifying the constraints. In the following we assume all timing constraints to be positive numbers, specified in the same direction as the control/data flow.

3.1 Fixed Timing Constraints between Operations

To mark the points in the code that are hooks for constraints, we use standard C labels. As their only purpose in C is to mark jump targets, they do not alter the behavior of the code. A complete *labeled statement* in C comprises an identifier with a colon followed by the statement it marks. We define reserved "`START`" and "`END`" labels as hooks for timing constraints relative to the start or the end of a program.

To convey minimum and maximum time between labels we need to pass values to the compiler. ANSI C provides `#pragma` statements to pass more information to a compiler than has been defined in the standard. The compiler designer can freely define the syntax of what follows the `#pragma` token. If a compiler can parse this syntax it can use the extra information for the compilation process. If a compiler does not understand a `#pragma` it encounters, the standard requires it to ignore the statement. Accordingly, `#pragma`-annotated code can still be processed by any ANSI C compiler.

We define a `#pragma` syntax to express timing constraints. The first statements specify a minimum or maximum time:

```
#pragma mintime <src_label> <dest_label> = <time>
#pragma maxtime <src_label> <dest_label> = <time>
```

In our `#pragma` syntax, `src_label` and `dest_label` are the names of the C labels between which the constraints apply. The amount of time is given by `time`. To conveniently specify both minimum and maximum time at the same time, we define:

```
#pragma time <src_label> <dst_label>
                         = <mintime> <maxtime>
#pragma time <src_label> <dst_label> = <time>
```

The second statement sets the minimum and maximum time to the same value. Both statements are shorthand for combinations of the two `#pragma` statements defined before.

The combination of C labels and `#pragma` statements enables a programmer to specify timing constraints between two C statements. A statement in C, however, can comprise multiple basic operations. Therefore, we need to improve the resolution of the labeling.

The actual time-critical part of an application is the communication with the environment. In embedded systems, this often has the form of read and write accesses to I/O registers. In order to attach timing constraints to these accesses, we introduce another `#pragma` statement to declare that a particular variable name represents an I/O register:[1]

```
#pragma io <procedure>::<variable>
```

The name of the variable is given by `variable`, and `procedure` gives the name of the procedure in which the variable is declared. Within a C statement identified by a label, a timing constraint will now be attached to that basic operation which accesses an I/O variable, as identified by a `#pragma io`. We have thus devised a method to provide a coarse-grained HLL with fine-grained timing constraints.

The programmers must ensure that only one I/O variable is accessed in such a C statement. They can achieve this by splitting statements with more than one I/O variable and introducing new variables for intermediate results. For instance, if `in` and `out` are I/O variables, the statement

```
out = in + 5;
```

with two I/O variables can be split into

```
temp = in + 5;
out = temp;
```

resulting in two statements with only one I/O variable each, as required.

With a resolution of a single IR operation we can now specify where exactly the timing constraints apply. For $mintime = maxtime = 0$, the identified operations must be scheduled in the same cycle. For $mintime = maxtime = 1$, the second operation has to be scheduled in the cycle following the first operation. With $mintime = 0$ and $maxtime = 1$, the operations are scheduled either in the same cycle or one cycle apart. Timing constraints with larger values are interpreted in the same fashion.

Note that the timing-critical action is only the access to I/O variables, i.e., I/O registers. In contrast, algorithmic operations are not directly observable from the outside of the embedded system and their timing is therefore only relevant where they feed I/O operations that have a timing constraint. An algorithmic operation derives a latest possible execution time from such an I/O operation if

- the I/O operation has a direct data dependency on the algorithmic operation, or
- the result of the algorithmic operation is needed to compute an execution condition of the I/O operation.

Hence, timing constraints do not necessarily include any algorithmic operations on the values of these registers. To meet the timing constraint of a read access to an I/O register it is sufficient to save the register value to an internal register before the I/O register is overwritten by the environment. This fact can be exploited in operation scheduling.

3.2 Data-Dependent Wait

In control-dominated applications timing constraints are often data-dependent, i.e., a system input determines the time required between two events. Data-dependent delay operations have been proposed for high-level synthesis to model communication with the environment or conditional blocks whose total execution time depends on a runtime condition because one branch takes longer than the other [13]. We extend this concept by an explicit wait operation. This operation

[1] It might be more elegant to use the `register` keyword in C to declare I/O variables. However, we use the SUIF2/machSUIF compiler framework [11, 12] for our implementation and the only available C front-end for SUIF2 does not transform `register` statements correctly.

has one operand that is computed at runtime and specifies a time to wait in clock cycles.

An example scenario requiring a data-dependent wait is the task of finding the beginning of a TCP packet header after a variable-length IP header in a network processor [1]. The length is encoded in a header field and its value corresponds to the number of input words to bide before the TCP header appears at the network interface.

To express such a dependency between input data and timing, we introduce another `#pragma` statement:

```
#pragma wait <src_label> <dest_label>
                  <variable> <min_val>
```

Here, `variable` is the variable that determines the number of cycles to wait. The labels `src_label` and `dest_label` mark the points in the code between which the wait time must elapse. The minimum value of `variable` is provided by `min_val`. This value provides the freedom to schedule the beginning of the wait anytime between *immediately* and `min_val`. We will use this freedom in Section 5.

Note that the `variable` name must be valid in the scope of the `dest_label` so that its value is accessible to the wait operation. Moreover, the operation at `dest_label` is not executed in the same cycle when the wait triggers, but will be scheduled in the *next* cycle. Therefore, `min_val` must not be less than 1 to allow for this one cycle delay.

4 Intermediate Representation for Timing

In order to capture the fine-grained timing constraints in the compiler IR, we introduce a timing layer for control/data flow graphs (CDFGs) that can be compared to output transition graphs for FSMs [14]. Graph edges in the timing layer are annotated with the minimum and maximum time between nodes. Scheduled nodes are annotated with the determined time step.

The nodes in the timing layer between which timing constraints exist are start and end nodes, I/O nodes, and wait nodes. I/O nodes represent communication with the environment of the ASIP, i.e., read or write accesses to variables that have been declared as I/O registers in the C code by a `#pragma io`. I/O nodes act as operands to operation nodes in the data-flow layer.

To provide a representation of the wait statements introduced in Section 3.2, our timing layer features a new type of operation node that connects the data-flow layer with the timing layer. We call this node a *wait node*. It has one data-flow edge as an input whose value determines the delay the node represents, given in clock cycles. Furthermore, to be meaningful, a wait node must have at least one incoming and one outgoing timing edge because it provides a delay between two other nodes. Finally, a wait node also represents a control construct in that it blocks the control flow until its timer triggers. Therefore, just like a branch instruction, a wait node ends a basic block, and a control edge connects it to the next basic block.

5 Implementing a Data-Dependent Wait

The wait node does not translate directly into a primitive processor instruction. Instead, it is transformed into one of two possible implementation types as the example in Figure 1 demonstrates. The wait node, depicted on the left-hand side, can be implemented

1. entirely in software, as shown on the right-hand side of Figure 1, by moving a start value `data_in` into a register `wait`, decrementing this register in appropriate intervals with an explicit subtraction, and branching when the register reaches zero, or
2. partially in hardware, as shown in the middle column of Figure 1, by providing a counter register that is decremented implicitly by a constant value—typically by one—and compared with zero in each clock cycle. The counter is set by writing a value into the counter register. Then the processor is stalled by a special control instruction, which we call *wait-for-counter (WFC)*, until the counter reaches zero. The WFC instruction stops the execution of all instructions at the end of the cycle in which it has been issued. When the counter triggers, the processor resumes execution in the following cycle.

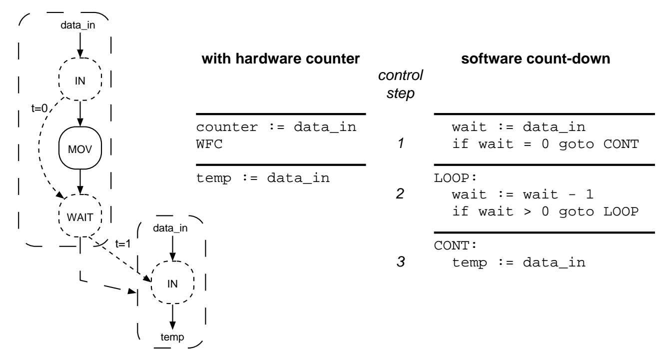

Figure 1. Wait-node implementation.

The software implementation requires more instructions in the application code for the repeated subtractions and tests for zero. Each additional instruction complicates the instruction-scheduling process. Note also that updating the wait variable and testing it for zero have to be scheduled in the same cycle.

The hardware solution, on the other hand, relies on additional infrastructure. Moreover, a counter can be used for only one wait node at a time. This, however, is not a severe constraint as a wait stalls the entire processor for an unbounded time, and hence we cannot have two waits in parallel.

Using the hardware counter in the application code requires two instructions: a *move* to set the start value of the counter, and the WFC instruction. Writing the start value to the counter, however, needs to be scheduled exactly in the cycle required by the timing edges that lead to the wait node. Otherwise, the counter would not go off at the intended point in time.

For both implementations we can achieve the freedom to schedule the counter start earlier or later by introducing another *add* or *subtract* node, respectively, to adjust the start value accordingly. This additional node may be arithmetically merged with other nodes in the delay computation by appropriate optimization methods.

The adjustment value depends on the final scheduling of the instruction that starts the counter. Hence, the value can

only be determined after the final instruction scheduling and might then even be zero. The scheduler needs to be aware of operations that implement wait nodes in the application so that it can decide whether it should introduce an adjustment node.

Moreover, the scheduling freedom of the counter start depends on the minimum possible start value of the counter. This value determines the time after which the counter must be tested for zero for the first time. It is the latest possible start time of the counter—even with adjustments. The designer must specify the minimum value of the wait input, using the `min_val` of our wait-pragma. The larger this value, the larger the scheduling freedom.

Figure 2 shows an example of a wait node with three possible schedules. The `min_val`, after subtraction of 1 for the transition to the operation following the wait, allows an adjustment of up to 3. The adjustment value is subtracted from the operand of the addition node. In the first implementation with no adjustments, both instructions must be executed in the same cycle to start the counter correctly. Using adjustments this situation can be relaxed. With an adjustment value of 2, there is even an idle slot, marked by the `nop`, that can be filled with a productive instruction.

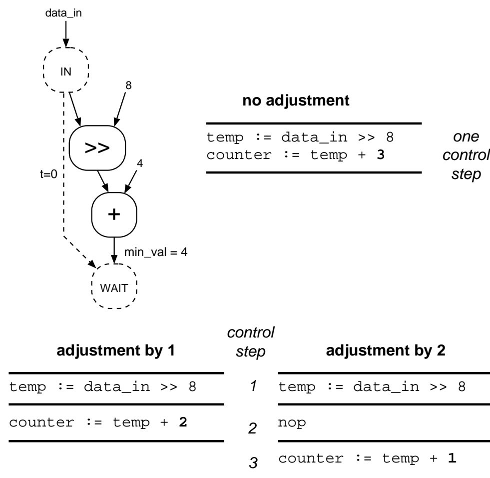

Figure 2. Wait-node adjustment for scheduling.

For a software implementation of a wait node, an enumeration of the possible delay values offers another optimization opportunity. Gaps between the values correspond to scheduling slots in which the register used for the countdown does not have to be decremented or tested for zero. To compensate the gaps, the counter merely has to be decremented by a higher value later.

In conclusion, the wait node offers the programmer an additional abstract expression, and enables the instruction scheduler to select an optimum implementation strategy for the expression.

6 Conclusions and Future Work

In this paper we have presented a novel method to specify timing constraints in C with a resolution of a single register access. This resolution is needed for the I/O specification of reactive systems, but so far has not been delivered by C annotations. Furthermore, we defined a new wait construct to express timing constraints that depend on runtime information. This type of constraint is also important for reactive systems. Our extensions are compliant with the ANSI-C standard, and allow the reuse of existing code. We proposed a timing layer to capture the timing constructs in a CDFG-based IR. Finally, we described a number of implementation strategies for the wait construct. These strategies enable new optimizations in code synthesis and scheduling on the critical path of real-time applications.

We have implemented a compiler front-end that transforms the pragma annotations to the timing layer. The next step will be to integrate the timing constraints and wait-node strategies into the scheduling-decision process and to quantify the optimization potential that the timing constructs provide.

References

[1] Gero Dittmann. Programmable finite state machines for high-speed communication components. Master's thesis, Darmstadt University of Technology, `http://www.zurich.ibm.com/~ged/`, 2000.

[2] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[3] IEEE, New York, NY. *Standard VHDL Language Reference Manual*, 2002.

[4] The Open SystemC Initiative (OSCI). *Functional Specification for SystemC 2.0*, version 2.0-q edition, April 2002.

[5] Petru Eles, Krzysztof Kuchcinski, Zebo Peng, and Alexa Doboli. Timing constraint specification and synthesis in behavioral VHDL. In *Proceedings of EURO-DAC/EURO-VHDL 95*, pages 452–457, Brighton, UK, September 1995.

[6] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venter, Daniel Weil, and Sergio Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In *Proceedings of Computer Aided Verification 2001*, pages 391–395, July 2001.

[7] Real-Time for Java. `http://www.rtj.org/`.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[9] R. Ernst and Th. Benner. Communication, constraints and user directives in COSYMA. Technical Report CY-94-2, Technical University of Braunschweig, Institute of Computer Engineering, Germany, June 1994.

[10] David Ku and Giovanni De Micheli. HardwareC - a language for hardware design. Technical Report CSL-TR-90-419, Stanford University, April 1990.

[11] The Stanford SUIF Compiler Group. `http://suif.stanford.edu/`.

[12] Michael D. Smith's Research Group on Compilation and Computer Architecture. `http://www.eecs.harvard.edu/~hube/software/`.

[13] David C. Ku and Giovanni De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer, Norwell, MA, USA, 1992.

[14] John A. Nestor and Vili Tamas. Exploiting scheduling freedom in controller synthesis. In *Proceedings of the Sixth International Workshop on High-Level Synthesis*, pages 74–86, November 1992.