Automatic parallelization of Nested Loop Programs (NLPs) is an attractive method to create embedded real-time stream processing applications for multi-core systems. However, the description and parallelization of applications with a time dependent functional behavior has not been considered in NLPs. In such a description, semantic information about time dependent behavior must be made available for the compiler, such that an optimized time independent implementation can be generated automatically.

This article introduces language constructs with temporal semantics to NLPs. Using these language constructs, time dependent applications can be specified and a corresponding data-driven implementation can be generated for use on a multi-core system. Despite that these time-aware language constructs can be data-dependent, the application remains functionally deterministic. Pipelining is exploited to increase the throughput of an application. The media access control (MAC) protocol of an IEEE 802.11p WLAN transceiver is used to illustrate the relevance and applicability of the introduced concepts.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features—Control structures

General Terms: Languages, Design

Additional Key Words and Phrases: Automatic Parallelization, Multi-core, Real-time, Time-aware

ACM Reference Format:

1. INTRODUCTION

Stream processing applications, of which (SDR) applications are an example, are typically executed on embedded multi-core systems. Such an application is often described as a task graph in which the tasks communicate via (FIFO) buffers. The throughput requirement of these real-time applications can usually be met by means of a pipelined execution of the corresponding task graph on a (MPSoC).

Various approaches have been proposed to reduce the programming effort of implementing stream processing applications on multi-core systems. A promising approach is to take a sequential description of an application as input and automatically parallelize the application into a task graph. Next, a suitable mapping of the application onto the multi-core system is computed such that the throughput requirement...
is met [Nadezhkin et al. 2010; Nikolov et al. 2008; Stefanov et al. 2004; Geuns et al. 2011a]. The sequential input description is often called a (NLP) to emphasize that the application is described as a number of nested loops in which statically allocated variables are accessed. The sequential nature of the language means that it is close to the style of programming that is typically used to describe radios, for instance, as a Matlab or C program.

An important shortcoming of existing (NLP) descriptions is that there are no language constructs available to describe time-aware functional behavior of the stream processing applications. If an application has time-aware functional behavior, the result of the computations performed in the application depends on the current time. Constructs for describing time-aware functional behavior are for example statements delaying the execution of a program for a fixed amount of time and statements giving a timeout in case an event does not arrive on time. These constructs are needed to include a description of the communication protocol layer of (SDR) applications into the (NLP) description of such an application. A fragment of such a description is shown in Figure 1. First a channel is sensed for a given amount of time. If the channel is idle a timeout occurs, the return code is −1, and data can be sent over the ether. If the channel is not idle the return code is one and the application must wait for a given amount of time. Afterwards this process is repeated until the channel is idle.

An important shortcoming in existing methods which allow time-aware statements to be described is that the behavior of these statements is schedule dependent. Schedule dependent behavior is highly undesirable because changes in for example processor load can have an unpredictable effect on the functional behavior of the application.

Fig. 1. Fragment of an SDR application including MAC protocol handling.
In this article we introduce a notion of time to (NLP) such that applications with time-aware functional behavior can be specified. Specification of this type of behavior can be done by means of language constructs, such as delayed and timeout statements. In contrast to most existing approaches a local notion of time is used internally and the wall time is only needed at the boundaries of the system. This means that the system internally remains functionally deterministic and independent of the schedule of the tasks. Despite the time-aware functional behavior, subsequent input events can be processed before previous output events are produced as a result of a pipelined execution.

The remainder of the article is organized as follows. Section 2 describes approaches related to the specification of time-aware application software. Section 3 describes how current sequential languages employ time and the issues with it. Next, Section 4 explains the current automatic parallelization approach and Section 5 details the new language constructs introduced in this article. The time it takes for a sample to arrive at the sink is discussed in Section 6. The real-time analysis method is discussed in Section 7. Section 8 discusses a case-study using an IEEE 802.11p WLAN transceiver application and finally Section 9 presents the conclusion.

2. RELATED WORK

Statements describing a delay in execution, such as the delay statement from Ada [Taft and Duff 1997], delay the execution of the entire program for a given amount of time. A disadvantage is that a delay statement in Ada acts like a barrier and reduces parallelism. Also no upper bound on the waiting time is defined, only a lower bound meaning the application will never continue before the requested amount of time has passed but there is no guarantee that the program continues before another predefined time. Using our approach both a lower and an upper bound can be defined and pipeline parallelism is exploiting.

A number of programming languages exist with support for time-aware behavior but unlike NLPs, these languages are not sequential. For example, the synchronous languages gain acceptance for the specification of safety-critical control systems [Berry 2003]. In synchronous languages an abstract clock defines when statements are executed [Halbwachs et al. 1991; Berry and Gonthier 1988]. Typically, synchronous programs require systems with a global clock, although a number of approaches have been developed to transform synchronous programs into parallel (GALS) implementations [Girault 2005; Berry and Sentovich 2001; Gamatié and Gonnord 2011]. However, according to Girault [2005] such a transformation is not possible for every synchronous program, it can be complex and computationally intensive. Also, the resulting implementation is not transparent because the GALS implementation has no one-to-one relation with the input specification, making debugging more complex. Our approach preserves the control structure of an input NLP program in the corresponding output task graph and thus the input NLP program has a one-to-one relation with the corresponding task graph. In our NLP input specification synchronized periodic clocks are only required at the border of the system. Internally the system executes data driven. The throughput and latency constraints are verified by making use of the worst-case response times of the tasks that are executed on processors with run-time task schedulers. Furthermore, our dataflow analysis approach has a low computational complexity at the cost of potentially over dimensioned buffer capacities [Wiggers et al. 2007]. Another important property of our approach is that every valid input program results in a deadlock free task graph [Geuns et al. 2011a].

PTIDES is another approach for the modeling and implementation of real-time systems [Zhao et al. 2007]. PTIDES is based on the Discret-Event (DE) Model of ComputationModel of Computation (MoC) where time-stamped events arrive at the inputs
```c
while(1){
    start = time();
    f();
    diff = time() − start;
    if(diff < 10) g();
    else h();
}
```

Fig. 2. Traditional method of using time.

and the value of time-stamps in the model is increased by delay actors. An important innovation is that event queues can be distributed, which makes the PTIDES approach suitable for distributed systems. As is the case in our approach, the wall time is only observed and relevant at the boundaries of the system [Derler et al. 2008]. In case that measured execution times are optimistic instead of Worst-case execution times worst-case execution times (WCETs), slack makes it more likely that events are processed in time. Both the PTIDES approach as our approach allow for slack to be exploited [Zou et al. 2009; Bijlsma et al. 2008]. For PTIDES it has not been shown how to determine queue/buffer sizes given a throughput constraint. In our approach a Cyclo-Static Dataflow (CSDF) model is used to compute buffer capacities [Wiggers et al. 2007].

In the Precission timed machine (PRET) approach time-aware instructions, such as a timeout instruction, are supported in the instruction set of a processor [Bui et al. 2011]. With these instructions time-aware functional behavior can be specified at the processor clock cycle granularity. Our time-aware statements are implemented in software that is executed on standard-of-the-shelf processors. With these statements functional behavior can be specified at the sample period granularity.

3. DESIRED TIME RELATED PROPERTIES

In existing sequential programming languages it is already possible to specify time-aware functional behavior. Figure 2 shows an example of how a traditional sequential application relying on the wall time, can be made. The application measures the execution time of the function $f$ and based on that time, either $g$ or $h$ are executed. Since the time between the two $time$ function calls is dependent on when these function calls are scheduled, functional non-determinism is introduced.

A second problem is that pipelining is hampered by such a time construction. The $time$ function is a function interacting with the environment by fetching the current wall time. Functions that potentially interact with the environment can not be reordered since their side-effects are unknown. Therefore, $f$ must execute after the first $time$ call and before the second call. The functions $g$ and $h$ must again execute after the second $time$ call and before the $time$ call in the next iteration of the loop. Consequently the application from Figure 2 must be executed sequentially and can not benefit from pipelining.

Omphale [Bijlsma et al. 2008; Geuns et al. 2011b] is a tool for the automatic parallelization of real-time stream processing applications, such as SDR applications. Given a program in the Omphale Input Language (OIL), a functionally deterministic task graph is extracted that, for the same input trace, computes the same output trace as the OIL program. This parallel task graph can be executed in a pipelined fashion. The OIL language is a single assignment language without pointers which simplifies dependency analysis and parallelization. After parallelization a task graph is obtained that resembles a Kahn process network (KPN) but this network has bounded FIFO buffers. The OIL language has the expressivity of a finite state machine because memory can only be statically allocated and recursion is not supported. OIL programs can contain non-affine if-conditions, non-affine array index expressions, and while loops.
A throughput and latency constraint is specified in an OIL program by making use of periodic sources and sinks [Geuns et al. 2011b]. Whether the throughput constraints are satisfied after the mapping of the application onto the multiprocessor hardware, is verified using an automatically generated CSDF model. The CSDF model is derived from automatically inserted synchronization statements. These statements are executed unconditionally [Geuns et al. 2011b].

The use of periodic sources and sinks enables the compiler to translate an program with time-aware statements into a program without time-aware statements. This translation schema is discussed in the next section for the OIL language.

Besides sources and sinks, we also included time-aware statements with well defined semantics in the programming language. The inclusion of the time-aware statements in the programming language enables optimizations that remove copy overhead and reduce the memory requirements. Unlike a library based approach containing black-box functions, an OIL program with time-aware statements can be pipelined and nondeterminism cannot be introduced.

4. NLPs AND AUTOMATIC PARALLELIZATION

The OIL programming language is an NLP based language supporting conditional statements such as if-statements and while-loops. The Omphale tool generates from a sequential OIL program a parallel task graph which can be executed on an embedded multi-core system. The tasks in the generated task graph are executed data-driven and are scheduled at run-time by the scheduler. This means that tasks execute whenever there is sufficient input data available and sufficient output space. Run-time scheduling is used to prevent that a processor is idle when a task finishes early.

In the task graph circular buffers are inserted for the inter-task communication. These buffers enable a pipelined execution of the tasks in the task graph. Only data dependencies restrict the order in which tasks are executed. For each variable in the sequential program, a buffer is created in the parallel task graph. These buffers can contain multiple instances of the corresponding variable. Each time a variable is written, it is stored in a new location. After a value is no longer needed, it is discarded and the buffer location is freed.

The buffers must have a sufficient capacity such that the throughput requirement can be met. The capacity of the buffers has a direct influence on the amount of pipelining. To determine the capacity of buffers, dataflow models and analysis can be used [Wiggers et al. 2007]. The throughput requirements are a consequence of communication with the environment. In OIL this communication is always periodic.

In OIL an application can communicate with the environment via ports [Geuns et al. 2011b]. A port can either be a source, when data is read from the environment, or a sink, when data is written to the environment. It is highly desirable that data delivered by a source is processed by the application and no data is processed multiple times. Therefore, every statement reading from a source should execute at the same rate as that source. Next to the buffers which ensure that no samples are lost or delivered multiple times, this ensures that all samples are processed. A similar case holds for a sink, all processed samples must become visible to the environment, but only once. Because now the rate of processing samples is the same as delivering samples and samples are processed in-order per node, the number of samples processed can be used in a task as a reference for the internal time. This can even be done when an application is executed in a pipelined fashion.

4.1. Semantics of Sources and Sinks

The time-aware statements as described in this article are based on sources and sinks that access the environment strictly periodically and which communicate with the
application via shared buffers. Periodically a new value is either stored in or retrieved from the shared buffer connected to a source or sink. Access to the shared buffers from the application is defined via Single Assignment Sections (SAS) [Geuns et al. 2011a]. Without the definition of SAS, the semantics of OIL programs can not be understood.

An SAS is a code fragment in which each variable may only be written once at runtime, called single assignment, but may be read any number of times. The length of an SAS is defined statically by the control structures in the input specification. The first SAS is defined from the start of the input specification until the first end of a while-loop is encountered. After the end of that loop, a new SAS begins and it ends again at the next ending of a while-loop or at the end of the input specification, whichever is encountered first. This process repeats itself until the end of the specification is reached. A SAS can be executed multiple times, depending on the loop condition. The length of an SAS is defined per variable, if a variable is not written in a while-loop, the SAS for this variable does not end at this loop, but at the next while-loop. This allows for efficient synchronization when variables are only read in loops.

At the end of an SAS, the value assigned to the variable corresponding to that SAS can no longer be accessed. Making a value available for the next SAS is possible using an assignment to a variable marked with the ’ operator. An example of such an assignment can be found at line 5 in Figure 3. Note that this operator must also comply with the single assignment rules. This means that if there is an assignment in an SAS to the next SAS, there may not be an assignment in the next SAS to the current SAS.

An example of an NLP with SAS is shown in Figure 3, where the SAS are visualized using a colored bar for each different SAS. The variable $x$ is written at lines one, five and seven. Therefore, the first SAS for $x$ ends at the first while-loop end, thus at line six. The next SAS for $x$ ends at the end of the next while-loop end, thus at line 8, which is also the end of the NLP. The variable $y$ is only written at line four and not in the inner while-loop. Because an SAS only ends at a loop in which that variable is written, the first SAS of $y$ does not end at line 6 but at line 8.

The sources and sinks of an application are specified in parallel with the rest of the application. However, for the view of the programmer it is defined via SASs when a source produces samples and when a sink consumes samples. The SASs of a source or sink variable always ends at the while-loop in which the source or sink is accessed, independent of whether it is a read or write operation. In a SAS defined by a sink, this sink may be written only once. In a SAS defined by a source, the source may be read any number of times, but the same value is always read.

In the view of the programmer, a source executes at the start of an SAS and a sink executes at the end of an SAS. Since the input format is a sequential specification and the accesses to sources and sinks are defined via SASs, a sequential specification can again be defined for a program including sources and sinks. A sequential program is by definition deadlock free, thus every program containing sources and sinks is also deadlock free.
5. SPECIFYING TIME-AWARE FUNCTIONAL BEHAVIOR

In this section our approach is introduced in which time-aware functional behavior can be specified in a sequential program. The starting point is the OIL programming language. To this language new language constructs are added to specify time-aware behavior. Language constructs are used because they provide more information to the compiler than a library based approach. For example, this allows the compiler to replace two time-aware statements with one statement which reduces memory requirements.

Because of the sequential nature of the input specification, the execution rate of the time-aware statements must equal the rate of the periodic source or sink it accesses, such that all samples read are processed and all samples processed are visible to the environment. Therefore, the implementation of these constructs can be done by translating the time-aware statements to a time-unaware code fragment. Since the period of the periodic ports is known at compile time and the rate of any task is known relative to a source or sink, the period of the sources and sinks can be used to derive a local notion of time in a task. In the Omphale approach, synchronization statements are inserted unconditionally, meaning that a task synchronizes for every sample produced by a source or consumed by a sink. Therefore, the number of synchronization calls can be used to derive the time that has passed since the start of the execution of the task graph.

Equation (1) now defines the arrival time of the \( j \) 'th sample at a source or sink. In the equation, \( wt \) is the wall time, \( f \) is the frequency of the source or sink and \( \xi \) is the start time of the task graph. Every task processes the samples from its input streams in order. Therefore, every task can derive using Equation (1) what the corresponding arrival time at the source was or when the sample will arrive at a sink.

\[
wt = j f + \xi. \tag{1}
\]

Because of the direct coupling of the time-aware statements with the periodic ports, the time parameter has to be a multiple of the period of the ports. However, this is not a restriction because the only coupling with the environment is via sources and sinks. A new sample from a sink becomes visible to the environment periodically. Thus if the time parameter was not a multiple of the period, the environment would not see the output of the statements until the next periodic execution of the sink.

Since the implementation of the introduced constructs counts the samples produced by the sources or consumed by the sinks, the wall time is not needed at the introduced statements. The OIL language is functionally deterministic, i.e. when the same input trace is delivered multiple times to an application, the output trace is also always the same. The generated task graph is also functionally deterministic since it resembles a KPN. Since the time-aware language extensions introduced in this article can be translated to time-unaware code, the OIL language including time-aware statements remains functionally deterministic.

5.1. The Delayed Statement

The delayed statement is a statement used for delaying the input value for a certain amount of time. After the delay time has passed, the delayed value is returned as an output. The delayed statement can operate on sequences of values, meaning that the statement can be called multiple times before the first output value is returned. Because of this feature, the delayed statement can be executed in a pipelined fashion.
The syntax of the delayed statement is given below. Here the value of \( x \) of \( T \) time ago is assigned to the variable \( y \). Executing this statement for the first \( T \) time results in reading from a negative time. Since no values are stored in this statement before the first execution at time zero, the default value \( d \) is returned. If the delay time \( T \) is not a constant, a maximum delay time \( TM \) should be given. If \( T \) is constant, this is naturally also the maximum delay possible and the \( TM \) parameter can be omitted and takes a default value of \( T \).

\[
y = x \texttt{ delayed } T \texttt{ init } d \texttt{ max } TM;
\]

A delayed statement can be translated into an equivalent OIL code fragment without time. The notion of local time using sample counting from Equation (1) can be used for this translation. Since only samples have to be returned for the last \( TM \) time, an array \( b \) is defined which contains these samples. The total number of samples that need to be stored for the last \( TM \) time is \( TM \times f \), with \( f \) the frequency of the source or sink. Consider now an index \( p \) in this buffer which represents the location of the oldest value in the buffer. This index \( p \) circulates through buffer \( b \) such that \( p \) is also the next place being written. Therefore, in a buffer \( b \) element \( b[i] \) contains the sample corresponding to wall time \( \beta(i) \), with \( \beta \) defined as in Equation (2).

\[
\forall 0 \leq i < TM \times f \quad \beta(i) = \begin{cases} \frac{i}{f} + k \times TM & \text{if } 0 \leq i < p \\ \frac{i}{f} + (k-1) \times TM & \text{if } p \leq i < TM \times f \end{cases}
\]

Equation (1) is now rewritten to Equation (3). For convenience it is assumed that the task graph starts at time zero, ie. \( \xi = 0 \).

\[
wt = \frac{p}{f} + k \times TM
\]

A delayed statement must return the sample from time \( wt - T_d \), with \( wt \) the current time and \( T_d \) the time difference derived from the delayed statement. For the first case of Equation (2) the following expression can be derived using Equation (3).

\[
wt - T_d = \frac{p}{f} + k \times TM - T_d \\
= \frac{p - T \times f}{f} + k \times TM \\
= \beta(p - T \times f) \quad \text{if } 0 \leq p - T \times f < p \\
= \beta(p - T \times f) \quad \text{if } T \times f \leq p
\]

From the second part of Equation (2) the following expression can be derived.

\[
wt - T_d = \frac{p}{f} + k \times TM - T_d \\
= \frac{p}{f} + (k-1) \times TM + TM - T_d \\
= \frac{TM \times f - (T \times f - p)}{f} + (k-1) \times TM \\
= \beta(TM \times f - (T \times f - p)) \quad \text{if } p < T \times f
\]

Figure 4(a) shows these cases as an implementation in OIL. The first case of the first if-statement shows the case when the delay is zero, thus \( T = 0 \). This value is not
if (T == 0) y = x;
else if ((T + f <= p) y = b[p - T * f];
else y = b[(T * f) - (T * f - p)];

for (0 ≤ i < (T * f)) {
    if (i == p) b'[i] = x;
    else b'[i] = b[i];
}
p' = (p + 1) % (T * f);

(a) Implementation in OIL

(b) Example signal trace, with f = 1

Fig. 4. delayed statement implementation and example.

yet in the buffer since a value written to b' is only available in the next iteration of the while-loop around the delayed statement. The other two cases of this if-statement contains the result of the above two derivations.

After the if-statement from Figure 4(a) that determines the return value, the new input value is stored in the array. Because every variable value expires at the end of a SAS, all old values must be copied. Finally, the next to oldest value is now the oldest value, which is the value which will be overwritten the next execution of this delayed statement.

In the parallelized task graph, the implementation of the delayed statement is aggregated in one task to prevent a potentially large synchronization overhead. Another advantage of using language constructs and a compiler which generates an implementation for the delayed statement is that the output can be optimized for the output language. If the output language is a language which limits the lifetimes of variables, such as OIL, the whole array b has to be copied every execution of the delayed statement. This is because the lifetime of the variable ends at the end of an SAS. Since a delayed statement must be used inside a while-loop, there is always an SAS end before the same delayed statement is executed again. However, when the output language of the compiler is C, there is no such notion of SASs which limit the lifetime of variables. Therefore, the old array b does not need to be copied such that it can be used in the next execution of the delayed statement. Only the oldest value in the array must be overwritten by the new value.

During every execution of the delayed statement, the value at index p in b is returned and the current value of x is stored in this location. Essentially, this means that the array b is used as a circular buffer. In y, the input value from T time ago is stored. Retrieving the value from the past is done by subtracting T * f from the current index p. Next the index p is incremented by one location such that the next access to the buffer returns the correct value.

An example of an execution of a delayed statement is given in Figure 4(b). An input variable x is delayed for two time units giving the variable y as a result. The value of the variable represent data items at that time. The first two values stored in y are the default values, in this case 0. From time t = 2 and further, the value from x at time t - 2 is stored in y.

When multiple delayed statements are used in an NLP, there might be room for optimizations. Not only optimizations on the generated code are then possible, but also across time-aware statements. This is an additional advantage of using language constructs instead of a library based approach when implementing these time-aware statements.

Consider the case where two delayed statements are used sequentially. When no sink reads from the intermediate result, this result is not visible to the environment.
Therefore, instead of delaying twice a single delayed statement can be used which delays the input variable the sum of the two delay times. Grouping sequential delayed statements improves performance since only a single value has to be delayed. Memory requirements remain the same because the sum of the individual time parameters is used, thus resulting in one buffer with a size of the sum of the two individual buffers.

Another optimization can be applied when two delayed statements are used in an if-statement, as is shown in Figure 5(a). In the first branch of the if-statement, the input stream is delayed for 8 time units while in the second branch a delay of only 5 time units is applied. On this program re-timing can be applied such that the delay is partially done before the if-statement. The result of this re-timing is shown in Figure 5(b). When this optimization is applied, the memory usage is reduced since the total memory requirement is reduced from a buffer for 13 time units before re-timing is applied to a buffer for only eight time units.

These optimizations are not possible when a library based approach was chosen to implement time-aware behavior in a programming language. The compiler is in a library based approach not aware of the semantics of the parameters passed to the time-aware statements and can therefore not exploit any relation between these parameters.

Because the rate of execution of the delayed statement is imposed by the connected periodic ports and the wall time is not used, determinism is guaranteed. The local notion of time implemented by sample counting ensures correct functional behavior over time.

5.2. The Timeout Statement

Applications often have to wait for data to become available from the environment. However, since the environment might not respond in time or not at all, an application must be able to resume its execution when no data arrives in time. When data does arrive, execution must resume immediately such that this data is processed as fast as possible.

The timeout statement implements this behavior. The statement waits until a condition evaluates to true before it returns true itself. If the condition remains false for a given amount of time, a timeout occurs. The syntax of the timeout statement is given below. Similar to the delayed statement, this statement must be called periodically in order to operate correctly. As a consequence also this timeout statement can be executed using a pipelined fashion. The application containing the timeout statement is not stalled until either data arrives or a timeout occurs, other tasks can execute in the time that an applications waits for a timeout statement to finish. The return value is either 1, −1 or 0 meaning respectively that the condition evaluated to true, a timeout occurred or neither is true yet.

Also for this statement there must be a periodic source delivering samples or a periodic sink collecting samples. Thus the task created from this statement must be
Sequential Specification of Time-Aware Stream Processing Applications

```c
if (e){
    y = 1;
    i' = 0;
} else if (i < T*f){
    y = 0;
    i' = i + 1;
} else{
    y = -1;
    i' = 0;
}
```

(a) Implementation in OIL

![Example signal trace, with \( f = 1 \)](image)

(b) Example signal trace, with \( f = 1 \)

The `timeout` statement cannot be implemented using a `delayed` statement because if the condition is delayed and becomes true, it cannot be reset to false for successive calls to the `timeout` statement. A requirement for the `timeout` statement is that it periodically writes values to \( y \). If this was not the case, the time \( T \) cannot be larger than the period of any of the sources or sinks used without violating the real-time constraints.

However, the `timeout` statement can be implemented in OIL as is shown in Figure 6(a). It is required that a source is read or a sink is written in the surrounding loop. The frequency \( f \) from this port is used to determine the timeout period in the number of samples. As was the case for the `delayed` statement, time needed for the timeout parameter can be determined by counting samples which are delivered at frequency \( f \). Since the timeout statement must be called periodically by the user and always returns a value, the timeout value can be larger than the period of the ports. Despite that the timeout value is larger than what the throughput constraint allows for, the throughput of the application is not disturbed.

The implementation shows the three possible return values of the `timeout` statement. When the expression evaluates to true, a 1 is returned and the timeout counter \( i \) is reset. The second branch implements the state in which the expression evaluates to false, but the waiting time has not yet expired, thus returning 0. Finally, in the last case the timeout expired, therefore \(-1\) is returned and the timeout counter \( i \) is reset again.

Since also the implementation of the `timeout` statement is generated, optimizations can be performed by a compiler. Similar to the `delayed` statement, the implementation can be clustered in one task and generated specifically for the targeted architecture. Generating a clustered task can avoid synchronization overhead between the time-unaware implementation and generating the implementation for a specific platform directly can reduce the execution time even more.

The result of an example program containing a `timeout` statement is shown in Figure 6(b). The `timeout` statement is assumed to be executed every time unit. At time \( t = 0 \), the function \( f(R) \) evaluates to true. Therefore, the variable \( y \) contains a 1 at time \( t = 0 \). However, during the next four time units, from time 1 to time 5, \( f(R) \) remains false and therefore a timeout is triggered at \( t = 5 \) at which \(-1\) is stored in \( y \).
5.3. Conditional Statements

The time-aware statements introduced in the previous sections can also be specified inside conditional statements. An example of such a combination is shown in Figure 7(a). The result of this program is that, dependent on the condition of the if statement, y is assigned either the previous value of the function f() or the next to last value produced by the function g().

This example shows the advantage of using the previous values of x over assigning to the future values of y. Verifying single assignment is not always possible when assigning to future values of y, because statements with different delay values can write to a y in the same loop iteration. Because of the unknown condition in the if statement, this cannot always be verified automatically.

An issue with conditional statements in combination with a delayed statement is that the code transformation schema presented in Section 5.1 places the generated code at the same place as the delayed statement. Therefore, if the delayed statement is placed inside a branch of the if statement, also its implementation is placed there. To guarantee that an output value can always be read from the internal buffer, the required input value must always be written to the internal buffer. In case the delayed statement is executed conditionally, this does not have to be the case. To solve this issue, the program must be transformed to make the filling of the internal buffer unconditional. Writing the result to the output variable must remain conditional to preserve the functional correctness of the program.

An example of this transformation is shown in Figure 7. In Figure 7(a) the variable y is written using two delayed statements with different delays. Therefore, this code is transformed to make the delayed statements unconditional. This is shown in Figure 7(b), where the variables t1 and t2 are introduced to temporarily store the results. The variable y is assigned conditionally, as was required by the original program.

If the parameter describing the delay amount is a variable assigned conditionally, the delayed statement cannot be moved completely before the if statement as there would be a write-before-read violation. The solution is in this case to split the implementation of the delayed statement into two parts. One part is filling the internal buffer using the for loop. The second part is conditionally reading a value from the past from this internal buffer.

Another issue with the delayed statement arises if the variable read by the delayed statement is written conditionally. The semantics of the program are then undefined since the previous values required by the delayed statement might not have been written. Therefore, a variable read by a delayed statement must be written in all branches of a conditional statement.
c = e && (state == 0);
t = c delayed T init 0;
switch(state){
  case 0:
    if(e){
      state' = 1;
      v' = x;
    }
    y = x;
  }
  case 1:
    if(t){
      state' = 0;
    }
    y = v;
    v' = v;
}

y = e ? x hold T;

5.4. Complex Behaviors

Next to combining time-aware statements with conditional statements as presented above, the delayed statement can also be combined with the specification of a state machine to create more complex behaviors.

An example of such a more complex behavior could be a state machine that keeps returning the same value for a given amount of time after an expression evaluates to true. Suppose this behavior is represented by the hold statement, which is shown below. Informally, when the condition e of the hold statement evaluates to true the hold statement returns the value of x at that time for the next T time. If the condition is false outside of this time period, the input value x is returned.

\[ y = e \ ? \ x \ \text{hold} \ T; \]

An example implementation of the hold statement is given in Figure 8(a). In the implementation the expression e determines when x is sampled and stored. Initially, state is zero. When the expression e becomes true, the state machine switches to the second state and keeps returning the same value. The end of this period is indicated by t, which is the expression c delayed for T time. Therefore, when t is true, normal operation, where y = x, resumes. The delayed stream t is derived from c, which is a composition of e and the proposition state == 0 such that any true expression of e when the holding period is in effect does not affect the result.

Again, as with the other time-aware statements, the implementation of the hold statement can be clustered in one task to keep the overhead as low as possible.

An example execution trace of the hold statement is shown in Figure 8(b). When the condition e becomes true at time \( t = 1 \), at times 1 to 4, y takes the value of x at time 1, which is 1 in the example. When e is false from time \( t = 5 \) and later, y takes the same value as x.

Apart from this example statement, many other combinations can be created which consist of one or more time-aware statements and a state machine.

6. SAMPLE DELAYS

In an application specified using our approach, samples produced by the source can be delayed by multiple statements before the processed results are visible at the sink. This section gives an overview of these delays.

Fig. 8. hold statement implementation and example.
source Src @ γ Hz;
sink Snk @ γ Hz;
start Snk δ ms after Src;

loop{
x = f(Src) delayed ϕ ms init 0;
Snk = g(x);
} while(1);

(a) Input program

(b) Task graph with explicit time blocks

(c) Trace with different delays outlined

Fig. 9. Overview of a time-aware system with a periodic source and sink illustrating the different sample delays.

Figure 9(a) shows a system where each 1/γ seconds a sample is produced by a source and consumed by a sink running at the same rate. The sink is started δ ms later to allow for initial processing, therefore the first sample must be available at the sink δ ms after it is injected in the system by the source. Samples produced by the source Src are stored in the FIFO buffer bSrc. Every iteration of the processing loop, a sample is taken from bSrc by the function f. After a sample is processed by f it is stored in the FIFO buffer bx until it is processed by g, which stores the resulting value in the output buffer bSnk. Periodically, the sink Snk takes a sample from this output buffer.

The task graph extracted from the program of Figure 9(a) is shown in Figure 9(b). In this system, a sample is stored in buffer bx by task tf on average every 1/γ seconds. However, since the task tf contains a delayed statement internally, an individual sample is delayed by this task for ϕ ms. An example schedule illustrating this behavior is shown in Figure 9(c). The figure visualizes an input stream and the corresponding output stream. When the sink is not yet started, no outputs are taken from the buffer bSnk and no value is produced to the environment. After the sink is started, default values are produced by the delayed statement for ϕ ms. The first output value which is derived from a source value is visible to the environment after ϕ + δ ms.

The delay ϕ is realized in task tf using the generation pattern as described in Figure 4(a). The internal circular buffer ensures that values are outputted later to the output buffer than they are read from the input buffer. In Figure 9 this internal buffer is also shown in task tf.

7. REAL-TIME ANALYSIS

This section shows how a corresponding dataflow model can be generated from an OIL specification including time-aware statements. The dataflow model can be used to determine buffer capacities for a given throughput, which is imposed by the environment. In order to generate a dataflow model, the time-aware OIL specification is first transformed to a time-unaware specification. After this specification is parallelized the dataflow model can be generated.

The generated dataflow model is a CSDF model, such that arrays with can be described [Bijlsma et al. 2008; Geuns et al. 2011b]. A CSDF model derived from an OIL specification always models the synchronization statements. These synchronization
Statements are automatically inserted by the compiler and are always executed unconditionally. This means that even if a location is not accessed, synchronization is performed on this location.

Synchronization consists of acquire and release statements. A distinction is made between read and write accesses. After an acquire statement returns a location in the circular buffer is either available for reading or writing. After a buffer location is no longer needed, it is released. Synchronization statements are placed in the generated task graph such that they are always executed unconditionally. Therefore, synchronization is always executed, but actually reading from or writing to the buffer remains conditional.

Since synchronization is performed unconditionally, a corresponding cyclo-static pattern can always be derived for each buffer from these statements, despite if-statements and while-loops in the OIL program. Each phase from the cyclo-static pattern either contains a 1 or a 0, meaning that at certain points in the code an acquire or release statement are placed. To analyze the generated CSDF model, conservative analysis algorithms with a polynomial time complexity are used [Wiggers et al. 2007].

The CSDF model is conservative with the parallel task graph, meaning the finish times of tasks in the task graph will never be later than the corresponding finish times of the actors in the dataflow model [Geilen et al. 2011]. Using the dataflow model, buffer capacities are determined which are sufficient to meet the throughput constraint imposed by the periodic sources and sinks and to guarantee a deadlock free execution. The capacity of the buffers has a direct influence on the amount of pipelining.

An important advantage of translating a time-aware statement to a time-unaware code fragment is that the generated dataflow model is independent of the time parameter of the time-aware statements. This allows the programmer to change the time parameter of the time-aware statements at run-time without the need to recalculate buffer capacities or affect system throughput. The generated dataflow model is independent of the time parameter because the time-unaware statements are clustered in a single sequential task such that the internal buffer used to store values for the delayed statement is not visible in the task graph. Therefore, also the CSDF model derived from the task graph does not model this internal buffer. An example of a variable time parameter is the delay time $T$ from the delayed statement.

Figure 10 shows the dataflow model corresponding with the task graph from Figure 9(b). All actors consume and produce one token, therefore the phases of the actors are omitted from the figure for clarity. For every task in the task graph a corresponding actor is generated and every buffer is translated into a pair of edges modeling the finite buffer. The capacity of the buffer is modeled by the number of tokens on the pair of edges.

The figure shows that the buffer created for the delayed statement in the task $t_f$ is not modeled. This buffer is internal to $t_f$ and has a fixed capacity, which is not of influence on the throughput of the application. This buffer is required for the functional behavior of the application and can not be made smaller nor does making it larger speed up the application. Therefore, if the time parameter of the delayed statement is changed, the CSDF model remains the same.
Also the periodic execution of the sources and sinks is not modeled in the CSDF model. However, it is sufficient to show that a periodic schedule exists that satisfies the throughput constraints [Geilen et al. 2011].

If the task extracted from the time-unaware code fragment finishes \( \Delta \) time earlier due to run-time scheduling or a shorter execution time than the worst-case execution time, a sample may be delayed shorter than the requested time. However, this effect is compensated by the periodic sink with the same \( \Delta \) time due to the buffering of samples. All buffers, including the buffers at the sources and sinks, keep the samples in order, such that samples which are delivered too early to the sink buffer are prevented from overtaking other samples. If a later sample is produced earlier by a task, the sink waits until all previous samples are processed until it takes the later sample from the buffer.

8. APPLICATION STUDY

This section illustrates the applicability of the introduced concepts using the MAC protocol of an, for explanatory reasons simplified, IEEE 802.11p WLAN transceiver. It is shown that pipelining can still be exploited despite time-aware statements being used.

The example from Figure 11 shows three states of such a transceiver. First in the Sense state the channel is sensed to detect if it is idle. If the timeout expires, the channel was idle and sending can commence. In contrast to the normal usage of a \textit{timeout} statement, it is used here for regular operation, not as an exception detection mechanism. If the channel is not idle, thus the timeout does not expire, the transmitter waits a predefined time period in the Backoff state. The \textit{timeout} statement is used here as a blocking call. Any sensing and sending operations are blocking until the timeout period expires and the state is again set to Sense.

If the channel is idle, the sender passes data symbols to the physical layer, which submits it to the ether. After a complete packet is sent, an acknowledgement must be sent by the receiver within a predefined time period. If this acknowledgement is not received in time, the packet is resent.

When sending a packet, it is split into symbols which are sent one-by-one. A symbol first has to traverse through the MAC layer and then through the physical layer before it is put into the ether. This processing of symbols can be executed pipelined, see Figure 12 for an example execution trace. The MACL and PHYL functions are executed pipelined since the second execution of the MACL function starts before the first symbol is processed by the PHYL function.

The figure also illustrates the behavior of the \textit{timeout} statement in the transition from the Sense state to the Backoff state. In the beginning of the trace, activity is sensed in the ether, thus the transceiver must wait. After sensing restarts, no activity is sensed and sending data can start.

The parallelized task graph resulting from the example contains 19 data-driven tasks and two periodic time-triggered tasks. In total, ten circular buffers are required internally and two additional buffers connect the source and sink tasks to the data-driven tasks. Due to the large number of small tasks, clustering may be required to reduce synchronization overhead.

Translating time-aware statements to time-unaware statements causes additional memory being used after the translation. For a delayed statement, two additional variables are used in the time-unaware implementation. An array that temporarily stores all intermediate values from the maximum time back to the current time, is required for the functional behavior of the delayed statement. Without storing all intermediate values, pipelining over the delayed statement and data-dependent delays is not possible. Additionally, a counter is used to store the index of the oldest value.
In the implementation of the $timeout$ statement, a counter is required to keep track of the current expired time. Since in the example application from Figure 11 three $timeout$ statements are used, the total overhead caused by the translation of time-aware statements to time-unaware statements is only the memory size to store three counters and the time it takes to update these counters.

9. CONCLUSION

In this article we introduced an approach to include time-aware behavior into the NLP specification of an application. This time-aware behavior can be translated automatically to time-unaware behavior by using periodic sources and sinks. Semantic
information from the language constructs allows for transformations, such as a pipelined execution and code optimizations. A key property of the introduced approach that the functional result of a time-aware statement is not dependent on the schedule of the tasks.

The basic language constructs that are introduced are delayed and timeout statements, but as implementation of the hold statement shows, more complex behaviors can be created based on these statements. From these time-aware statements a time-unaware implementation is generated by the compiler. The compiler exploits the notion of time provided by the periodic execution of sources and sinks to generate this time-unaware implementation. Therefore, a local notion of time is introduced based on the counting of samples from sources or to sinks. Additionally, the generated implementation supports a pipelined execution and run-time scheduling. Only the periodic clocks of the sources and sinks in the implementation must be synchronized.

Buffers are used to communicate between the tasks in the parallelized application. Their size can be computed using existing analysis techniques on an automatically generated Cyclo-Static Dataflow (CSDF) model. This CSDF model is independent of the value of the time parameter defined in a time-aware statement. Buffer capacities, and consequently throughput, are therefore not affected by changes to the value of a time parameter.

The relevance and applicability of the presented approach is demonstrated using a simplified version of an IEEE 802.11p WLAN transceiver, including the media access control (MAC) layer which contains the time dependent processing. The application specification includes time-aware statements, which are used to describe the functional behavior of the application.

REFERENCES


Received June 2012; revised August 2012; accepted September 2012