Better Languages for More Effective Designing

Roger D. Chamberlain
Joseph M. Lancaster

Dept. of Computer Science and Engineering
Washington University in St. Louis

Better Languages for More Effective Designing

Roger D. Chamberlain and Joseph M. Lancaster
Department of Computer Science and Engineering
Washington University, St. Louis, Missouri, USA

Abstract— *Developer productivity is strongly influenced by the language(s) used during the design process. The abstraction level of the language as well as the opportunities for casual errors due to non-intuitive language features can both have a dramatic impact on developers. In this paper, we explore language design in three contexts: course-grained expression of parallelism, register transfer level hardware description languages, and specialized languages for particular purposes. Coordination languages for streaming computations are espoused for the first, properties of a true synthesis language are described for the second, and we introduce the TimeTrial language as a concrete example of the third.*

Keywords: Coordination Languages, Hardware Description Languages, TimeTrial, Performance

1. Introduction

New languages face stiff obstacles to widespread adoption. This is in part due to the fact that there is a learning curve associated with any new language. The potential productivity benefits of a new language are often lost, however, when users persist in using an older language that is inappropriate for the task at hand (the older language being one that they happen to already know).

The language used for a job should reflect the abstraction model that is most beneficial for the user while enabling the user to accomplish the job. We assert that a better match between the language user's responsibilities and the capabilities of the language used (and its associated compiler and runtime environment) cannot help but lead to better human productivity. The evolution from C to C++ is a classic example of this, with the explicit support of object oriented programming constructs in C++ that are not present in C.

This paper will discuss language properties, new language features, and new language purposes for a range of tasks, including expression of high level parallelism in an application, design of a digital system at the register transfer level, and querying the performance of an executing application.

Our initial focus is the use of coordination languages to express course-grained parallelism. The streaming data paradigm is applicable to a large class of applications, and there are many benefits that accrue if the compiler and runtime system are made cognizant of the structure of the application. The concept of slanty design [1] is invoked to encourage application developers to expose sufficient information about their application so that the compiler can reason about it reliably and effectively.

The second focus is register transfer level digital system design. The dominant languages used for this purpose started out as simulation languages, and the lack of a language focused on synthesis rather than simulation has been costly in terms of lost designer productivity. We articulate several features that a good synthesis language should possess. Some of these ideas have been previously introduced [2].

The third focus is on new languages for specific purposes. There are a host of circumstances in which it is beneficial to allow developers to interact with a system in an application-specific manner. In some cases, this interaction goes well beyond a simple user interface specification, and a specialized language is warranted. Here, we introduce the TimeTrial language, designed to enable a designer to understand the performance of a streaming application. The purpose and capabilities of the language are introduced, and a formal specification of the language is provided.

In Sections 2 and 4, we describe specific languages that either already exist (the coordination language X) or that we are introducing here (the performance language used in TimeTrial). In Section 3, however, we are not describing or proposing a specific language. Rather, we describe properties that we feel should be present for any hardware description language that is at the register transfer level.

2. Coordination Languages

A coordination language is distinct from a traditional programming language in the sense that one does not express the entirety of the application within the coordination language. A coordination language is used to express the interactions between computing elements, but not the internal computations within those computing elements.

Lee [3] has promoted the use of coordination languages as an improvement over traditional, shared memory programming by pointing out that explicit reasoning about correctness is more straightforward. Thread-based synchronization is prone to errors (such as race conditions or missing locks) that can be very difficult to find and fix. Gelernter and Carriero [4] discuss the importance of separating the concerns of computation and communication in the context of Linda. However, their discussion is in no way specific to Linda itself. A common use of coordination languages is for computational workflow management [5]. As we will see below, coordination languages are well suited for use with streaming data computations.

Examples of coordination languages include StreamC and its associated KernelC [6], X and the Auto-Pipe development environment [7], [8], topology specifications in SCF [9], and Linda [10]. Both StreamC and X are languages that express the application in terms of streaming data semantics. Here, computational kernels are interconnected via explicit communication paths. The computational kernels for StreamC are expressed in KernelC, and X supports computational kernels in C/C++ for deployment on chip multiprocessors and in VHDL or Verilog for deployment on reconfigurable logic. StreamIt [11] is another language that supports the separate descriptions of streaming application topology and kernel computations. Both, however, are described within StreamIt.

In contrast with the streaming paradigm, SCF uses a message passing paradigm, supporting both synchronous and asynchronous message delivery, and Linda's coordination mechanism is a global tuple space, in which productions (compute kernels) both remove tuples from the space and insert other tuples into the space.

Returning to the streaming paradigm, a simple example application is shown in Figure 1. Here, a sensor of some form provides a stream of data that is filtered, accumulated, and written to a file.

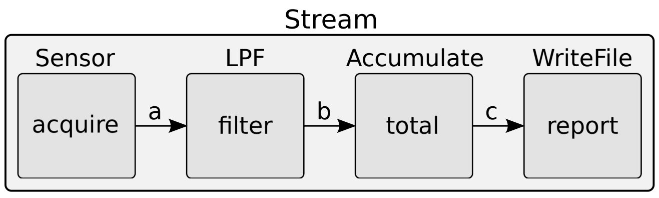

Fig. 1: Example streaming application.

The X language specification for the example application is given in Figure 2. In the terminology of X, the computational kernels are called *blocks*. The data into and out of each block is specified in X, but the functionality of the block itself is not. There can be one or more *implementations* of each block, each expressed in a different language, targeting a distinct type of computational resources. For example, a C/C++ implementation would target a processor core while a VHDL implementation would target reconfigurable logic. Note that SCF also supports this notion of implementations for blocks (called tasks in SCF) authored in different languages for distinct targets.

At the lowest level within X, the typed input and output ports of each block are specified. Blocks can then be composed into higher level blocks, which form the complete application. Also specified (but not shown in the figures) are: (1) the implementations that are available for each lowest level block, (2) the set of computation and communications resources used for application deployment,

```
block Sensor {
    output unsigned16 y; // port declaration
}
block LPF {
    input unsigned16 x;  // port declarations
    output unsigned16 y;
}
block Accumulate {
    input unsigned16 x;  // port declarations
    output unsigned32 y;
}
block WriteFile {
    input unsigned32 x;  // port declaration
    config string filename;
}
```

(a) `Sensor`, low-pass filter (`LPF`), `Accumulate`, and `WriteFile` block definitions. On all blocks, the input port is named `x` and the output port is named `y`.

```
block Stream {
    Sensor     acquire; // block declarations
    LPF        filter;
    Accumulate total;
    WriteFile  report (filename="out.txt");

    a: acquire -> filter; // topology
    b: filter -> total;
    c: total -> report;
}
```

(b) Application topology for `Stream`. As part of topology specification, edges are given labels `a`, `b`, and `c`.

Fig. 2: X language specification for example application `Stream`. Note that block definitions only specify I/O and configuration, not function.

and (3) the desired mapping of blocks to resources. It is this separation between the application topology specification in a coordination language and the block function specification in other languages, point (1) above, that defines what it means to be a coordination language.

The benefits of separating the overall application into a coordination of lower-level components include the following:

- Re-use of code blocks. With a clear distinction between the language that specifies what individual blocks do and how they are composed, there is a greater tendency for the application developer to think first in terms of using blocks that already exist, rather than building new blocks. To be effective, this does require the existence of a rich block library, but once that library exists, the language separation will promote its use.
- Data movement is handled by the system, not the application developer. There are two benefits that accrue from the system handling the data movement. First, the application developer need not actually author the code required for data movement (e.g., DMA engines and the like). With less code to write, the application

developer's time is saved. Second, there are fewer errors introduced since the application developer is working at a higher abstraction level.

- The application decomposition is known to the system. This facilitates the system being capable of parallel execution while being cognizant of the required data dependencies. It also facilitates the system specifying the mapping of compute kernels to computational resources.
- The application is less error-prone. Memory races don't exist. Locks are unnecessary. Reasoning about the correctness of a pipelined application is very similar to reasoning about a sequential application.

3. Hardware Description Language

Current hardware description languages, such as VHDL and Verilog, were originally developed as simulation modeling languages, not synthesis languages. It was common practice to express a design in one of these languages, use the simulator to debug the design, and then manually re-express the design using a lower-level tool (such as a schematic capture tool) for physical implementation. The need for re-expression was motivated by the fact that synthesis technology was not up to the task of implementing the physical system directly from the original HDL description (which often existed at the behavioral level rather than the gate level needed for implementation).

The benefits of avoiding re-entering a design are self-evident, and as synthesis tools matured, a natural choice for the synthesis language is the language already in use for simulation. Designs are already expressed in the language, and engineers performing the design function are already familiar with the language. As a result, the dominant simulation languages (VHDL and Verilog) quickly became the dominant synthesis languages.

We have paid a substantial price for this choice, however, due explicitly to the fact that these languages were originally designed as simulation languages, not synthesis languages. Their semantics were specified with simulation as the primary target, and that has serious implications for their efficient use as synthesis languages. First and foremost among the implications is the fact that state is inferred rather than explicitly declared, leading to innumerable errors in design descriptions on the part of both novice and experienced developers.

How many of us has inadvertently implied state in a portion of the design description that we intended to be purely combinational? How many of us has inadvertently described a latch instead of the flip-flop we intended to describe? How about missing signals on a sensitivity list? How about confusing the meaning of a blocking vs. non-blocking assignment statement? Need we go on? With a properly designed synthesis language, rather than a borrowed simulation language, all of the above errors can be either entirely eliminated, checked during compilation, or at least significantly diminished in frequency.

Here, we describe some of the properties that a synthesis language should have. We make no attempt to describe a complete language, nor are we doing anything but hinting about syntactic conventions, our focus is matching the level of abstraction of the language with that of the designer. In the spirit of focusing on the most frequently used aspects first, the discussion will start by constraining the discussion to the description of synchronous finite-state machines (applied to both control and data path), which are by far the bulk of what we implement as hardware designers. While as designers we might spend a large fraction of our time transforming applications to expose available parallelism, or exploring optimization opportunities within designs, by the time the rubber meets the road and we are actually implementing the system, the majority of the effort is finite-state machine design.

Figure 3 illustrates a fairly classic finite-state machine (FSM) as a Mealy machine, with input `x`, state `y`, next state `nxt_y`, and output `z`. It is being clocked by the signal `clk_a`. CL represents the combinational logic (which is memoryless), and FF represents the flip-flop (which retains state `y`). A description of FF in SystemVerilog (an extension of Verilog that has several useful features) is provided in Figure 4. Here we learn that the FF supports a synchronous reset (called `reset`) as well as a load enable signal (called `enable`).

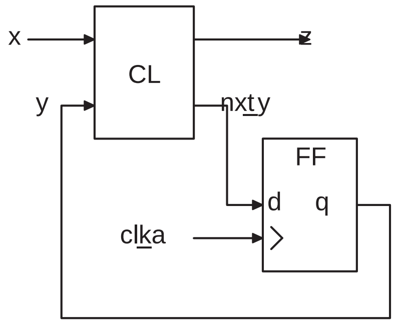

Fig. 3: Simple finite state machine.

```
always_ff @(posedge a_clk iff
            (enable==1 or reset==1)) begin
    if (reset) begin
        y <= 0;
    end
    else begin
        y <= nxt_y;
    end
end
```

Fig. 4: Declaring state in SystemVerilog.

Note that it takes 8 statements to describe a fairly simple, commonly used flip-flop, and there are many typographical errors that can result in an erroneous design that the compiler cannot check because they are legal language constructs. We are forced to describe the specific functional properties of our flip-flip, rather than simply say to the compiler, "Give me a flip-flop." State is inferred, rather than declared.

We contend that state should be a first class object in any synthesis language. As such, when a designer wishes to instantiate state (in the form of a register) he or she should explicitly declare it. Consider the example data path of Figure 5. The block is an instantiation, in reconfigurable logic, of the Auto-Pipe `Accumulate` block of Figures 1 and 2. Figure 5 shows the data path (minus the control logic), and Figure 6 illustrates a candidate form for declaring all of the signals necessary for the block.

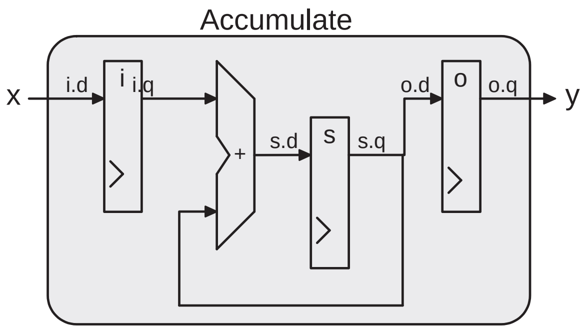

Fig. 5: Auto-Pipe `Accumulate` block.

In the state description, there need to be ways to explicitly describe properties of:

- the data register – its name, whether it is implemented as a flip-flop or a latch, its bit-width, and its type (e.g., signed or unsigned);
- the clock signal – its name, whether it is edge-triggered or level-sensitive, and its signal polarity (rising vs. falling or active high vs. active low);
- the reset signal – its name, whether it is synchronous or asynchronous, its signal polarity, and whether or not it is present;
- the reset value (the value loaded into the data register when reset is triggered); and
- the enable signal – its name, signal polarity, and whether or not it is present.

When not explicitly specified, it is important that these values have well defined defaults.

As alluded to in Figure 5, the declaration of state implies the existence of a pair of signals, one that holds the current value of the register and the other indicating the future value that will be stored on the next clock. There are many conventions for naming these signals, for example, in Figure 3 we used `y` for the current value of the register and `nxt_y` for the future value. In what follows, we will use the convention of appending a "`.q`" to the name of the state for the current value and appending a "`.d`" to the name of the state for the future value. In this way, the register `s` has associated with it signals `s.q` and `s.d`.

The semantic meaning of the state declarations in Figure 6 is expressed in the SystemVerilog code of Figure 7. Note that there is nothing special about the use of SystemVerilog here, it is simply being used to unambiguously describe what is intended by the given state declarations. Its use does, however, illustrate a related point. The design of a new, more appropriate synthesis language does not in any way require that the entire design flow be re-engineered. It is quite reasonable to target any synthesizable subset of one of the commonly used languages as the back-end of a compiler for the synthesis language.

```
always_ff @(posedge clk iff
            (enable==1 or reset==1)) begin
    if (reset) begin
        i.q = 0;
        s.q = 0;
        o.q = 0;
    end
    else begin
        i.q = i.d;
        s.q = s.d;
        o.q = o.d;
    end
end
```

Fig. 7: Compiled SystemVerilog state.

What took 3 statements to declare (one statement per register), is expressed in 12 statements of SystemVerilog. The lines of code reduction, however, is not the major benefit. One major benefit of declaring state is that it eliminates the temptation to mix state definition and combinational logic definition in one place, often with disastrous results. Some state is being inferred, and some combinational logic is being inferred. It is extremely easy to get it all wrong, and it can be extremely difficult to find out what is wrong in order to fix it. Recall that these errors are in the context of a simulation language. The simulation looks fine, and the synthesis tool can only guess what the designer really intends.

There have been some attempts to address the basic issue discussed above. In SystemVerilog, the `always_ff` construct makes it explicit to the synthesis tool that state is what is intended. All this means, however, is that designers who use a subset of the language get helped. The principle of slanty design says that it shouldn't just be possible to do things the right way, it should be encouraged to do things the right way, and it should be discouraged (or outright banned) to do things the wrong way. Language constructs that have been shown over and over again to lead to subtle, hard to find and fix errors should go.

Another major benefit of the direct declaration of state, rather than its inference from a description of its properties,

```
state [16] i clocksig=clk resetval=0[sync] resetsig=reset enablesig=enable;
state [32] s clocksig=clk resetval=0[sync] resetsig=reset enablesig=enable;
state [32] o clocksig=clk resetval=0[sync] resetsig=reset enablesig=enable;
comb  [16] x;          // input data
comb  [32] y;          // output data
comb       avail_x;    // control input (data available on x)
comb       avail_y;    // control output (data available on y)
comb       wait_x;     // control output (wait upstream)
comb       wait_y;     // control input (wait downstream)
comb       enable;     // flip-flop enable
```

Fig. 6: Declaring state and combinational signals. The state declarations explicitly specify the bit width of the data, the reset value (the value the data takes on when reset), synchronous reset, the reset signal, the enable signal, and the clock signal. Other parameters are set to their defaults (e.g., edge-triggered flip-flop, unsigned data type). The combinational signals have specified bit widths and default data types.

is that the requirement to repeatedly describe (rather than specify) the properties is susceptible to subtle errors that are legal in the language (and therefore cannot be checked at compile time) but are not at all what the designer intended. With a simple misplacement of the nested parentheses, the synchronous reset of Figure 7 could end up half-way between synchronous and asynchronous, e.g.,

```
always_ff @(posedge clk iff
            (enable==1) or reset==1)
```

While the above construct might look like an asynchronous reset specification, the reader can be assured it isn't. Designers should simply be stating "synchronous" or "asynchronous" without having to re-engineer each time what those words mean.

Given the state declarations, what the `Accumulate` block of Figure 5 actually does is specified by the combinational logic. In this example, we want to make the data path connections that are illustrated in Figure 5 as well as specify the control function (which is not illustrated in the figure). The semantics of an Auto-Pipe block are such that data is received on input port `x` if and only if the `avail_x` signal from upstream is active at the same time the `wait_x` signal generated by the block is inactive. This same signaling convention is used on the output port `y`, with the distinction that the roles are reversed; the block is the upstream party (generating `avail_y`) and the Auto-Pipe runtime environment is the downstream party.

The current mechanisms for specifying combinational logic are (for the most part) quite reasonable and should be continued. One relatively straightforward improvement that can be made is the unambiguous use of register names in the combinational logic specification and the compiler's ability to automatically discern the correct intent. Figure 8 illustrates an appropriate specification of the combinational logic for the `Accumulate` block. The references to state on the left-hand side of an assignment operator are unambiguously referencing the next state information (the `.d` signal) while any state references on the right-hand side of an assignment operator (or in the conditional of an `if` statement) are unambiguously referencing the current state information (the `.q` signal).

```
comb_logic begin
    i = x;
    s = i + s;
    o = s;
    y = o;
    enable = avail_x && !wait_y;
    wait_x = wait_y;
    avail_y = enable;
end
```

Fig. 8: Description of combinational logic.

Synthesizeable Verilog code that could be generated by the compiler would therefore look like Figure 9. The only real transformation that has taken place to this point is the insertion of the appropriate signal names in place of the originally indicated register names.

```
always @(x,i.q,s.q,o.q,avail_x,wait_y) begin
    i.d = x;
    s.d = i.q + s.q;
    o.d = s.q;
    y   = o.q;
    enable = avail_x && !wait_y;
    wait_x = wait_y;
    avail_y = enable;
end
```

Fig. 9: Compiled Verilog combinational logic.

An important distinction, however, needs to be made between what should be allowed combinational logic specifications in a synthesis language and what is currently allowed in languages such as Verilog and VHDL. Note that an explicit sensitivity list was provided in the generated code of Figure 9. While there are mechanisms to avoid needing to provide this list in existing languages (e.g., use of the `@(*)` notation in Verilog), they are not required, and the use of an explicit sensitivity list is allowed.

We are all familiar with the erroneous results that follow from a sensitivity list that is missing a needed signal. The semantics of the language specify that the signal "retains its previous value," an action that is perfectly reasonable for a simulator, but implies state for a synthesizer. Here again is an example of the pre-existing simulation semantics getting in the way of efficient design activity. A well designed language should not let us easily specify something that is almost always the wrong thing to do. A well designed language makes it hard to do things that are almost always wrong. A synthesis language should allow a designer to declare a signal to be combinational, and then not allow a specification of that signal's value that isn't combinational.

Another difference should be highlighted between the needs of a synthesis language and those of a simulation language. The whole distinction between blocking vs. non-blocking assignments in Verilog or between signals and variables in VHDL is all unneeded, unwanted, and frankly harmful in a synthesis language. A clear semantic meaning should be consistently applied to any assignment. In this example, we clearly intend the combinational value of `avail_y` to be the same as `enable`. Requiring the user to distinguish between blocking vs. non-blocking assignments and therefore putting `enable` in the sensitivity list (which would be required if the alternative form of the assignment operator had been used in Figure 9) is simply wrong. We, as designers, should insist upon a true synthesis language that clearly expresses the logic we intend.

The example of Figure 9 doesn't really require the use of an `always` construct at all. The same effect could have been achieved through the use of Verilog's `assign` construct. The use of `always`, though, does beg the question of, "what do we do if the value of a combinational signal is left unspecified?" It is common practice on the part of many designers to insert a default value for all of their next state signals (the `.d` signals in our terminology) of the current state. I.e., if register `s` exists, a default assignment via the statement `s.d = s.q;` is included early in the `always` block. With the proposed mechanisms for the compiler to be well aware of the designer's intent as to what symbols refer to registers vs. combinational logic, it is straightforward for the above default assignments to be implied.

It is reasonable to consider default values for other (non register related) signals as well, but one should probably proceed here with more caution. In this case, it might be wiser to enable the designer to declare a default combinational value, but flag as an error any combinational specifications that rely on a default value that wasn't explicitly provided.

Independent of the restrictions placed on specifying combinational logic, some attention also needs to be given to the poor choice of operator overloading characteristics that currently exist in VHDL. While this is not truly a synthesis vs. simulation language issue, it is worthy of mention. In any reasonable software language, the `+` operator is appropriately overloaded to reflect the data types of the two values to be added. In our example, the `+` operation should not be determined by what package is listed at the beginning of the file (a fact that is true of VHDL specifications). It should, in fact, be determined by the type of the registers `i` and `s`. If either of them is signed, the `+` should be signed, and if both are unsigned, the `+` should be unsigned. Appropriate type conversions should be available to override the default semantics when necessary.

The above is focused on finite state machine implementation, but it is worth mentioning the need for testbench specification. It is our opinion that testbenches should be specified using a traditional imperative language (e.g., C/C++), with appropriate mechanisms for signal monitoring and assignment. While one could use a simulation language for testbench specification, and this might be a reasonable approach, both VHDL and Verilog have significant limitations with respect to their use as languages for specifying testbenches. For example, the file I/O capability of VHDL is quite constrained, and data-driven testbench design can be a robust, flexible approach to providing test vectors to a design under test.

There are a whole set of topics that should be included in hardware description languages, but are currently left to be individually specified to the design tools. These topics include timing constraints, clock generators, and the like. When designing an asynchronous system, or even an asynchronous interface within a synchronous system, it is often these issues that determine (or undermine) proper functioning of the system. While the low-level specifics of these topics might very well have aspects that are platform dependent (e.g., chip pinout), a common language for describing the designers desires would be a dramatic improvement over the current state of affairs. We encourage research in this direction, but do not make any specific proposals in this paper.

There are several other approaches to specifying digital designs that are not explicitly at the register transfer level. In JHDL [12] (as in VHDL and Verilog) one can describe designs at the gate level, but that is generally considered to be a lower level specification and therefore not as designer efficient. Bluespec SystemVerilog (BSV) [13], which is based on Haskell, allows the specification of hardware via a functional language. There are several projects that translate C to gates (e.g., ROCCC [14], SPARK [15]) which are aimed at a higher level of abstraction than the register transfer level. Our focus here is not to compete with any of these efforts, but instead to improve the situation when the designer is working specifically at the register transfer level.

4. TimeTrial: A Language to Measure and Validate Performance

As applications increase in complexity, designing and tuning for reliable performance can be a time-consuming and frustrating task. This is especially true for FPGAs, where there is limited visibility into the internal state of the design. Now consider an application that utilizes both multi-core CPUs and FPGAs. Where is the bottleneck, on the CPU portion or the FPGA or both? Does it depend on the input data, and does it change over time?

We have developed a new language, the TimeTrial language (simply referred to as TimeTrial in this section), that helps developers ask these questions of their applications. The language drives a performance monitoring capability of the TimeTrial performance monitor that answers developers' questions in an automated way. TimeTrial enables a developer to both query an application's performance and assert his or her understanding of an application's performance by writing performance expressions. TimeTrial performance expressions are then compiled along side the streaming application source description (e.g., in X) to instrument the application with measurements and assertions for runtime validation. Our previous work [16] describes a runtime instrumentation system to measure performance queries on real-world datasets (called the TimeTrial performance monitor). Here, we describe a method to articulate measurements and assertions at the source level, which are used to automatically instrument a streaming application for runtime performance validation.

TimeTrial is intended to augment streaming languages with the ability to articulate performance statements about the application *at the level of block interactions*, regardless of which computations resource is targeted (e.g. FPGA vs. CPU). Understanding the performance at the block level is useful for locating bottlenecks to particular blocks, or a set of blocks, in a streaming application. By not attempting to query internal to a block, TimeTrial can stay agnostic with respect to what language is used to specify the functionality within a block. It can stay focused on the block interfaces and interactions.

To introduce TimeTrial, we begin with two example performance queries of the application from Figure 1, "What is the throughput of edge `a`?" and "How long does it take a datum to travel from the block `acquire` to the block `report`?" These queries can be stated as measurements in TimeTrial as:

```
m1: measure rate @ Stream.a

p1: path Stream.acquire.y -> Stream.report.x

m2: measure mean of latency @ p1
```

The first performance expression begins with a label, `m1` followed by the keyword `measure`. The key-

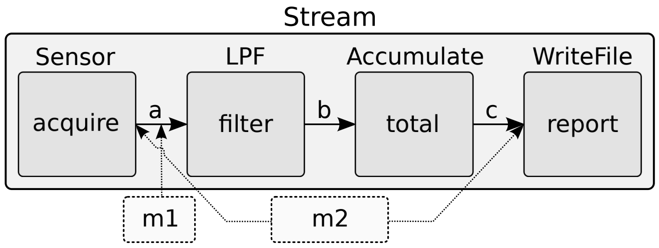

Fig. 10: Stream application instrumented with measurements `m1` and `m2`. Dotted lines and blocks represent the performance monitoring instrumentation.

word `rate` specifies the type of measurement, followed by `@` and the edge `Stream.a`, the target of the measurement. The second performance expression, `m2`, adds three additional elements. We declare a labeled path, `p1`, from one port (`Stream.acquire.y`) to another port (`Stream.report.x`) so that we can refer to it in `m2`. Next, we describe a statistical aggregation to be performed on the measurement, in this case `mean`, indicating that the mean value of the latency is to be measured. The other new feature is the use of the label `p1` inside the measurement statement. Figure 10 depicts the logical insertion of these measurements into the application `Stream`.

The target of a measurement can be described as either a single point in the application (a `tap`) or a data path through the application (a `path`). Tap and path statements declare the location(s) within the application that the performance expressions will observe. Target declarations can be used in both `measure` and `assert` expressions. For example, two tap declarations, one on a port and the other an edge, are declared as:

```
t2: tap Stream.filter.x

t3: tap Stream.b
```

Performance measurements operate on a multiplicity of events on taps and aggregate these into one or more statistical measures per execution. To provide a precise picture of the performance over time, the TimeTrial system supports the notion of a *data frame*, or frame for short. A frame is defined as a pre-sized segment of the data stream. As each segment flows across a tap on a streaming application, the performance is summarized within that frame. Large frames provide opportunities for strong aggregation while collapsing temporal effects within the data frame into a single result. Small frames provide more precise temporal information (e.g., capturing rare events). However, the likelihood of interfering with the application increases as one decreases the frame size, since the volume of performance metadata also increases. Different applications have different characteristics, so TimeTrial allows the developer to choose the frame size to suit.

Strong aggregation within a data frame is desirable to minimize the potential impact of measurement on the application's performance, both in terms of memory required to hold performance meta-data and communication overhead. TimeTrial supports a number of aggregation functions, including minimum, maximum, mean, summation, histogram, and trace of the values at a target. These aggregation functions have optional parameters that can be specified as well, such as requesting a harmonic mean rather than the default arithmetic mean or specifying the number and bin width of a histogram.

In addition to specifying how the performance data is aggregated, TimeTrial provides a number of pre-defined measurement types. We've already seen throughput (`rate`) and latency (`latency`) in `m1` and `m2`. Additional measurement types include utilization (`util`), queue or block occupancy (`occupancy`), `backpressure` from downstream, and the data value of the tap being observed (`value`). `rate` is a measure of the frequency that data flows across an edge. `util` measures the fraction of the execution that a target is active. `occupancy` keeps track of the occupancy of a queue or block over time. `latency` measures the time for data to flow from one port to another. `backpressure` measures whether or not the downstream path is free or is blocked. Finally, `value` observes the value of the data flowing through a port or across an edge. Note that `value` is distinct from the other measurement types because it is the only type that looks directly at the value of the data. Some of these measurements have only one logical value per data frame (e.g., rate, utilization), whereas others are multi-valued during the frame (e.g., occupancy, value). Hence, the aggregation performed can depend on both the type of measurement and the developer's desire for detail.

In addition to performance measurements, TimeTrial also supports performance assertions through the `assert` statement. Assert statements let a developer specify high-level performance requirements of the application, such as, "The throughput of edge `a` should always be at least 100 mega-transfers per second." Assertions are useful for designs that are nearing the final stage of development to ensure a wide variety of data sets execute within the performance specifications. Writing this example in TimeTrial looks like:

```
a1: assert m1 >= 100 Mtps
```

Thus far, we have introduced the use of two TimeTrial performance expressions, `measure` and `assert`. These expressions form the basis for TimeTrial statements. However, TimeTrial provides the ability to articulate more powerful expressions as well. Here we introduce the notion of conditionals and Boolean expressions followed by their use within the context of `measure` and `assert` expressions.

A conditional in TimeTrial is used to express a propositional logic predicate. Conditionals are specified as a measurement, a relational operator (e.g. `>`, `<=`), a value, and a units specification or another measurement. This should look familiar, since we used a conditional (albeit a simple one) when writing `a1`. The conditional is everything that follows the keyword `assert`. The units can be chosen from a list of time units (`s, ms, us, ns`) or rate units (`tps, ktps, Mtps, Gtps`), which represent data transfers per second. Unit specification is optional and appropriate default units are inferred from the measurement type (e.g., time for a latency measurement, unitless for a utilization measurement). Note that in the case of a measurement type that returns a Boolean value (e.g. backpressure), the relational operator and what follows is omitted.

Conditionals are evaluated for each instance under which the underlying measurement can hold a distinct value. As such, if a data aggregation is specified the conditional is only evaluated once per data frame. The default aggregation for a conditional is trace aggregation (i.e., the conditional is evaluated for each instance).

Complex Boolean logic expressions can be formed in TimeTrial. These Boolean expressions and are formed by combining any number of conditionals with and, or, and not (`&, |, !`) operators. The simplest Boolean expression is a single conditional statement, as in `a1`. Boolean expressions are used to qualify an `assert` statement, or are used to restrict the scope of a `measure` statement by using the `when` keyword. The `when` keyword is added to the end of a measurement statement to specify that a measurement statistics are to be collected *only* when the Boolean expression evaluates to true.

To illustrate the expressiveness of Boolean expressions, we provide two TimeTrial performance statements, a qualified measurement and an assertion, as examples.

```
acc_in:  tap Stream.total.x

acc_out: tap Stream.total.y

q_occ:   measure occupancy @ Stream.b

m3: measure rate @ acc_in when (q_occ >= 1 &
                    !backpressure @ acc_out)

a2: assert (m1 < m3) | (util @ acc_in < 0.5)
```

The tap statements declare `acc_in` (the input port on the `Accumulate` block `total`) and `acc_out` (its output port) as well as `q_occ` (the queue occupancy at the block input). `m3` measures the throughput into `total` when there is data available in its input queue and no backpressure from it output. This enables the developer to measure the achievable throughput of `total` (i.e., when it is not hindered by its environment).

The assertion `a2` is stating that the unqualified rate measure `m1` will be lower than `m3` or the utilization of `total` is fairly low (below 0.5). Expressions such as these allow for articulation of very specific performance statements that enable precise reasoning about the performance of an application in different contexts.

Finally, we articulate the formal grammar of the TimeTrial language. The language grammar is described in Extended Backus Naur Form (EBNF) [17]. To be clear, here are notes on the EBNF syntax we use. Figure 11 gives the formal syntax of TimeTrial.

- A symbol on the left-hand side of ::= is defined by its substitution on the right.
- Symbols in **boldface** are non-terminal symbols, and begin with upper-case.
- Symbols in `plainface` and those found in single-quotes (`''`) are terminal symbols.
- The pipe character (`|`) delineates substitution choices.
- Parenthesis (`()`) group a set of symbols into one logical symbol.
- Square brackets (`[]`) group a set of *optional* symbols into one logical symbol.
- An asterisk (`*`) follows a symbol that may be replicated *zero* or more times.
- A plus sign (`+`) follows a symbol that may be replicated *one* or more times.
- White space is ignored.

5. Conclusions

The language used during the design process can dramatically impact the productivity of the developer. Typically, developers are very hesitant to adopt new languages, based in large part on the learning curve required (whether that learning curve be real or perceived). We contend that in many circumstances, the productivity gains associated with using a language more suited to the task dramatically outweigh any drawbacks associated with learning the new language.

We have described candidate new languages (or new language features) in three areas: parallelism, digital systems design, and performance understanding. For the expression of parallelism, we advocate the use of coordination languages that explicitly disclose program structure to the compiler. For digital systems design, we contend that the current use of simulation languages should be replaced by the use of a language explicitly designed for synthesis rather than simulation. For performance understanding, we propose that the task is sufficiently unique that it warrants a language focused on its needs. It is our contention that the widespread adoption of these language ideas would have significant benefit to the design community.

Acknowledgments

This work was supported by NSF grants CNS-0751212, CNS-0905368, and CNS-0931693.

References

[1] R. Beale, "Slanty design," *Communications of the ACM*, vol. 50, no. 1, pp. 21–24, Jan. 2007.

[2] R. D. Chamberlain, J. M. Lancaster, and R. K. Cytron, "Visions for application development on hybrid computing systems," *Parallel Computing*, vol. 34, no. 4-5, pp. 201–216, May 2008.

[3] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[4] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992.

[5] R. Prieto-Diaz and J. Neighbors, "Module interconnection languages," *J. of Systems and Software*, vol. 6, no. 4, pp. 307–334, Nov. 1986.

[6] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proc. of Int'l Conf. on Parallel Architecture and Compilation Techniques*, Sept. 2006, pp. 33–42.

[7] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-pipe and the X language: A pipeline design tool and description language," in *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[8] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. F. B. Shands, and N. Singla, "Auto-Pipe: Streaming applications on architecturally diverse systems," *IEEE Computer*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

[9] V. Aggarwal, R. Garcia, G. Stitt, A. George, and H. Lam, "SCF: A device- and language-independent task coordination framework for reconfigurable, heterogeneous systems," in *Proc. of High-Performance Reconfigurable Computing Technology and Applications Workshop*, Nov. 2009.

[10] N. Carrier and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, Apr. 1989.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. of 11th Int'l Conf. on Compiler Construction*, 2002, pp. 179–196.

[12] P. Bellows and B. Hutchings, "JHDL – an HDL for reconfigurable systems," in *Proc. of Symp. on FPGAs for Custom Computing Machines*, 1998.

[13] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler, "A history of Haskell: being lazy with class," in *Proc. of 3rd ACM SIGPLAN Conf. on History of Programming Languages*, 2007.

[14] Z. Guo, W. Najjar, and B. Buyukkert, "Efficient hardware code generation for FPGAs," *ACM Trans. on Architecture and Code Optimization*, vol. 5, no. 1, pp. 6:1–6:26, May 2008.

[15] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[16] J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *Proc. of IEEE Int'l System-on-Chip Conf.*, Sept. 2009, pp. 23–28.

[17] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?" *Communications of the ACM*, vol. 20, no. 11, pp. 822–823, Nov. 1977.

PerfStmts	::=	`(`**AssertStmt** | **MeasureStmt** | **LabeledTap** | **LabeledPath**`)+`
AssertStmt	::=	`[`**Identifier**`:]` `assert` **BooleanExp**
BooleanExp	::=	`!` **BooleanExp** | **Conditional** & **BooleanExp** | **Conditional** `'|'` **BooleanExp** | **Conditional**
Conditional	::=	**CondOperand** `[`**RelOp** **CondOperand**`]`
CondOperand	::=	**MeasureType** @ **TargetType** | **Number** `[`**Unit**`]` | **Identifier**
RelOp	::=	`>` | `>=` | `<` | `<=` | `==` | `!=`
Unit	::=	`s` | `ms` | `us` | `ns` | `tps` | `ktps` | `Mtps` | `Gtps`
MeasureStmt	::=	`[`**Identifier**`:]` `measure` `[`**StatType** `of]` **MeasureType** @ **TargetType** `[when` **BooleanExp**`]`
StatType	::=	`(min` | `max` | `mean` | `sum` | `hist` | `trace)` `['('`**ParamList**`')']`
MeasureType	::=	`rate` | `util` | `occupancy` | `latency` | `backpressure` | `value`
TargetType	::=	**Tap** | **Path** | **Identifier**
LabeledTap	::=	**Identifier**`:` `tap` **Tap**
Tap	::=	**PortID** | **EdgeID**
LabeledPath	::=	**Identifier**`:` `path` **Path**
Path	::=	**PortID** `(->` **PortID**`)+`

Fig. 11: EBNF for TimeTrial. Non-terminals that are not explicitly defined in the grammar either come from the target streaming language (e.g., **PortID** and **EdgeID** are identifiers of ports and edges, respectively) or follow common usage (e.g., **Identifier**, **Number**, and **ParamList**). While not explicitly included above, parentheses are also supported in Boolean expressions for the purpose of describing operator precedence.