Policies of System Level Pipeline Modeling

Ed Harcourt[1]

Department of Mathematics, Computer Science, and Statistics
St. Lawrence University
Canton, NY USA

Abstract

Pipelining is a well understood and often used implementation technique for increasing the performance of a hardware system. We develop several SystemC/C++ modeling techniques that allow us to quickly model, simulate, and evaluate pipelines. We employ a small domain specific language (DSL) based on resource usage patterns that automates the drudgery of boilerplate code needed to configure connectivity in simulation models. The DSL is embedded directly in the host modeling language SystemC/C++. Additionally we develop several techniques for parameterizing a pipeline's behavior based on *policies* of function, communication, and timing (performance modeling).

Keywords: pipeline, system level design, discrete-event simulation, generic programming, hardware modeling, policies, SystemC

1 Introduction

Pipelining is a well understood and often used implementation technique for increasing the performance of hardware [17,15]. Since we have a taxonomy of pipeline designs we can (and should) develop system-level techniques that allow us to quickly model, simulate, and evaluate various configurations.

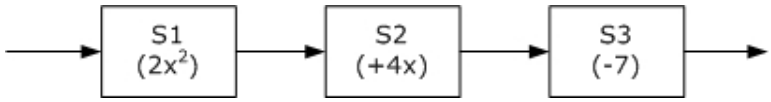
In this paper we describe several modeling techniques inspired by research in the generic and generative programming community [3,6]. We use SystemC [22,9] as our simulation framework because of its support for system level modeling and simulation and because it is embedded in C++, a language with support for generic, polymorphic, object oriented programming. Furthermore C++ is suitable for constructing domain specific languages (DSLs) [2,4].

In system modeling simulation performance usually improves when we move to more abstract models [9]. In software development it is often the opposite; abstract models suffer an *abstraction penalty* [25]. As a modeler abstracts, performance may, at some point, begin to degrade. One goal is to ameliorate the abstraction penalty by using compile time generic modeling techniques as opposed to run-time techniques (*e.g.* virtual methods).

[1] Email: edharcourt@stlawu.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Pipelines are composed of *stages* that compute outputs at regular intervals based on inputs. We'll start with a somewhat contrived example of an application specific three stage linear pipeline that computes the function $2x^2 + 4x - 7$.



Stage S_1 computes $2x^2$ which feeds S_2 adding $4x$, which finally feeds S_3 to subtract 7. A register-transfer level (RTL) implementation requires multiplexors, latches, and clock inputs on each resource considerably cluttering up the design and model. Requiring the user to model these artifacts is not helpful and hinders design exploration. In our library one simply declares the stages, the function each stage computes, and the route a *transaction* follows through the pipeline.

```
Resource<F1> s1; Resource<F2> s2; Resource<F3> s3;
Pipeline p = s1 >> s2 >> s3;
```

Pipeline stages are declared to be resources that are parameterized on a small class that implements the computational aspect of the stage. The class `Resource` is a *proxy class* for a highly configurable `Stage` class developed in section 3. The expression above specifies that a transaction enters the pipeline at S_1, proceeds to S_2 then exits the pipeline after S_3. A modeler can quickly explore a new pipeline where a stage is repeated (feedback), replicated, or skipped (bypass). For example in a floating-point multiplication pipeline the adder might be reused consecutively ten times. In our language this is specified as `Adder*10`.

A key component of our modeling framework are techniques for separating orthogonal behaviors of the pipeline into *policies* [4]. The DSL allows us to give a concise configuration of the pipeline, automatically generate mundane boilerplate code used to connect modules, insert channels, and generate pipeline control code. We handle pipelines with arbitrarily complex routing including feedback and feed-forward (bypass) paths, multi-function, and static or dynamic pipelines. This generality arises because the DSL is embedded in the host modeling language (SystemC/C++). This also eliminates the need to write separate language specific processing phases (*e.g.,* lexical analysis, parsing).

We don't claim that the code described here can be used unmodified to model every kind of pipeline imaginable; that's one of the main reasons we've chosen to embed this in a general purpose modeling language. We're motivated by the way a software design pattern [7] describes a particular problem that appears over and over again along with example code of how to solve the problem (rather than code that works in every context). What we do claim is that the techniques we describe solve problems that repeatedly appear in pipeline modeling and that the example code can be reused and modified to suit a particular modeler's needs. Moreover, the pipeline specifications are compact and efficient allowing a designer to quickly explore design alternatives.

1.1 Related Work

Excellent overviews of pipelining, hardware implementation techniques, and taxonomies are described in [17,15]. The compiler research community has developed high-level notations for pipelines to generate instruction schedulers [5,20]. Our notation is inspired by that of [20]. The work in [1,13,14] describes notations for specifying pipelines for downstream tools. Mishra and Dutt [18] describe how to validate a pipeline specification written in the architectural description language [10]. Petri Nets [21] and Process Algebra [11,12] have been used to model and simulate pipelines.

We view our research as building on this work in two fundamentally different ways. The first is how we separate pipeline behaviors into orthogonal policy classes, the second is how these policies are then configured into a working pipeline with a DSL that is itself embedded in the general purpose system simulation language SystemC. As [25] points out external DSLs not embedded in a general purpose language "tend to have short life-spans due to limited support and portability, suffer from a lack of tools (particularly debuggers), and it is usually impossible to use two DSLs in the same source file."

1.2 SystemC: Very Briefly

SystemC is a discrete event modeling and simulation language for designing hardware/software systems [22,19]. SystemC *modules* have *ports* connected through *channels*. SystemC has predefined channels for hardware like wires (`sc_signal`) and higher-level channels such as blocking FIFOs. Users can also define their own channel types. A SystemC module is a class that inherits `sc_module`. Modules can contain threads (`SC_THREAD`) or methods that fire on event changes (`SC_METHOD`). SystemC also has a large library of hardware data types including bit vectors and fixed-point types.

2 System Level Pipeline Specification

Our pipeline specification framework consists of a small DSL to specify pipeline structure, and generic models of *transactions*, *stages*, and *transaction routers*. These components are configured by the user with compile time parameters. These techniques are inspired by software engineering research in meta-programming [2], generative and generic programming [6,3], design patterns[7], and some advanced C++ programming techniques [4,24,23].

2.1 Pipeline Specification DSL

A pipeline expression defines the route a transaction follows through a pipeline. In a static pipeline this route is fixed. In a multi-function static pipeline there may be two or more different transaction types each with a different route. The language provides three binary operators, $>>, +, *$, defined on pipeline resources. Figure 1 shows a small grammar for our DSL.

Before a stage name can appear in an expression it must be declared as a

(1) Pipe ::= Term >> Pipe | Term
(2) Term ::= Stage | Stage * int | Stage + Stage
(3) Stage ::= id

Fig. 1. The pipeline DSL grammar

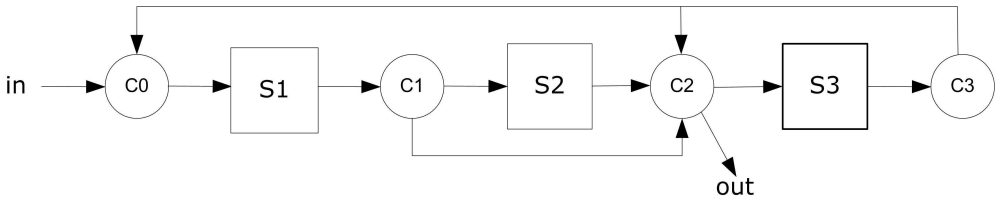


Fig. 2. Module hierarchy generated by the pipeline expression equation 4.

`Resource<F>` where `F` is a user defined *functor class*. A functor class encapsulates a function that takes a transaction as an argument and returns a transaction.

In a pipeline expression S_1 `>>` S_2 ... `>>` S_n, we call S_1 the pipeline *entry* and S_n the *exit*. The expression S_1 `>>` S_2 indicates that the output of stage S_1 is fed into stage S_2. The expression S_1 `*` `n` indicates that a stage should be repeated `n` times. This operator represents feedback, not replication of a stage. The `*` operator is shorthand for repeated sequencing; S_1 `*` `3` is shorthand for S_1 `>>` S_1 `>>` S_1. Reusing a stage name in an expression means the stage is also reused and that a transaction is fed back to the stage. For example, the expression S_1 `>>` S_2 `>>` S_3 `>>` S_1 means that after a transaction exits S_3 it goes back to be operated on by S_1 and then exits the pipeline. The expression S_1 `+` S_2 means that two stages are used in the same cycle and that a transaction is sent to both stages. The expression S_1 `>>` S_2 `+` S_3 `>>` S_4 means S_1 sends the transaction to both S_2 and S_3, then S_2 and S_3 each then send their transaction to S_4. S_4 needs to know how to handle receiving two transactions simultaneously.

Resource declarations and pipeline expressions are valid C++ and not an external language; reminiscent of expression templates [24]. To parse these expressions we overload the `>>`, `+`, and `*` operators and build an abstract syntax tree which we process to generate SystemC code for connectivity and control. Pipeline expressions are really compact representations of *reservation tables* [17].

Consider the pipeline expression below.

(4) `S1 >> S2 >> S3 >> S1 >> S3*2 >> S1 >> S2`

Our library generates the SystemC module hierarchy depicted in figure 2. The circles are *transaction routers* that are automatically inserted into the module hierarchy.

3 Abstracting Stages and Transactions

In addition to the DSL, generic representations of pipeline stages, transactions, and transaction routers are key components of the framework. A pipeline is composed of one or more interconnected *stages* that communicate *transactions*. A stage consumes a transaction, operates on it, and sends it on to a subsequent stage through a router. A transaction contains user specified *data* the stage operates on and *control*

```
1  class Transaction {
2  public:
3     void advance() { curr++; }
4     const double orig;
5     double data;
6  private:
7     static Route route;
8     Route::iterator curr;
9  };
```

Fig. 3. Naive implementation of a transaction for pipeline in section 1.

information derived from the pipeline expression. A transaction keeps track of where it is in the pipeline. A transaction router examines where the transaction currently is, where it has to go, and uses a lookup table to forward it through the proper port. Stages have exactly one input and one output, though they may carry complex types. Routers are multi-ported and handle multiple inputs and outputs of a stage.

Rather than using low-level digital or RTL modeling constructs (*e.g.,* `SC_METHODS` and `sc_signals`) we use a *transaction level model* (TLM) [9]. Pipeline resources are thread processes that communicate arbitrarily complex transactions through channels (single place FIFOs) much like a dataflow simulation [9]. We use this framework only for explication, our classes are not wedded to using threads and FIFOs but are parameterized on precisely these design choices. By switching policies FIFOs can be replaced with other SystemC channels such as Verilog/VHDL like signals (SystemC's `sc_signal` type) and thread processes with method processes — useful in *communication refinement* as a modeler migrates their design to an implementation.

We'll begin developing generic classes using our simple pipeline from the introduction (section 1) as an example. We'll first develop naive implementations of transaction and stage classes and use these as a basis for our policy based classes. A pipeline stage needs the original value of x and the output from the previous stage — information we'll keep in a transaction. A transaction also keeps track of its current location in the pipeline. Figure 3 shows an initial version of a transaction. Lines 4-5 specify the data and lines 7-8 specify control information. The data member `route` represents the path a transaction follows through the pipeline and is static because all transactions in a uni-function pipeline share the same route. For a multi-function pipeline we would have different transaction classes for each pipeline function. The member `curr` is not static as it represents where a particular transaction is within the pipeline.

The first stage (naive version) of the pipeline computes $2x^2$ (figure 4). Lines 3-4 show the stage's port interface, line 11 the function. Line 12 models a one cycle delay, line 13 advances the transaction to the next stage, and line 14 writes the modified transaction to the output. This is all well and good but we'd like to be able to abstract a stage so that it is as reusable as possible. `Stage` bundles many design choices into one class and doesn't give a modeler flexibility over the large number of possible design choices such as functionality, timing, and interface. We'd

```
1  class Stage : public sc_module {
2  public:
3    sc_fifo_in<Transaction *> in;
4    sc_fifo_out<Transaction *> out;
5
6    Stage() { SC_THREAD(process); }
7
8    void process() {
9      while (1) {
10       Transaction * t = in.read();
11       t->data = t->data + 2 * sqr(t->orig);
12       wait(1, SC_NS);
13       t->advance();
14       out.write(t);
15     }
16   }
17 };
```

Fig. 4. Naive implementation of Stage 1 for pipeline in section 1.

```
1  template <typename T>
2  class Transaction {
3  public:
4    void advance() { curr++; }
5    T value;
6    // ... routing code stays same
7  };
```

Fig. 5. Abstract implementation of a transaction.

like a modeler to be able to *configure* a stage to suit a variety of situations.

Enter policies, patterns, and generic programming [4,6,2]. "Policies represent configurable behavior for generic functions and types" [23]. In C++ a policy is an orthogonal unit of behavior passed as a template argument. The combination of templates and multiple inheritance gives us the mechanics to cope with combinatorial behaviors by factoring out design choices into classes. The main criticism of our `Transaction` and `Stage` classes are that they hard-code design choices making them difficult to reuse.

3.1 Transactions

The transaction class hard-codes the two data members `orig` and `data` that are particular to the pipeline; an easy fix with a template parameter (figure 5). For our example pipeline instantiating the template parameter `T` with an STL `pair<double,double>` gets us back the original transaction with two data members. A `typedef` aids readability.

```
typedef Transaction<pair<double, double> > MyTransaction;
```

During communication refinement we can instantiate `T` with a SystemC hardware data type.

3.2 Stages

`Stage` also hard codes design choices. FIFOs are the communication model, $2x^2$ is the function it computes, and it all takes one nanosecond. What if we want our pipeline to be untimed? Substituting zero in `wait` wont do as an untimed model is different than a zero time model. The non-terminating while loop implies we're using an `SC_THREAD` process as opposed to an `SC_METHOD` process. All of these design choices can be turned into policies and passed to the `Stage` class as template parameters. One might wonder what remains of `Stage`, our host class? "At an extreme, a host class is totally depleted of any intrinsic policy. It delegates all design choices and constraints to policies. Such a host class is a shell over a collection of policies and deals only with assembling the policies into a coherent behavior" [4].

Decomposing a stage into policies for timing, function, communication, and process yields a highly configurable class. Importantly a modeler can implement their own custom policies and does not have to use ours.

3.2.1 Function Policy

Creating a policy class for function is a straightforward application of a functor class [2].

```
1  template<typename T>
2  struct TwoSqr {
3    static inline T f(T p) {
4      p.second = p.second + 2*sqr(p.first);
5      return p;
6    }
7  };
```

`TwoSqr` declares a function `f` and is parameterized on the type of data in a transaction. Below is a modified `Stage` that uses a function policy. Function policies for other stages are analogous.

```
1  template <class Function>
2  class Stage : public sc_module,
3                public Function {
4  public:
5    // ... as before
6      t->data = f(t->data);
7    // ... as before
8  };
```

Parameterized inheritance (line 3) allows us to call `f` (line 6) in the function policy.

[2] For readability we name the function `f` as opposed to overloading the function call operator `()()`

3.2.2 Communication Policy

`Stage` hard-codes FIFOs as the communication medium. Factoring out `Stage`'s port interface into a separate class is more involved. SystemC's port classes are template classes. C++ allows us to specify a template class as a template parameter; *template template parameters* appear frequently in policy based design [4].

```
1  template<
2     typename Transaction,
3     template <typename> class InPort,
4     template <typename> class OutPort
5  >
6  struct StageInterface {
7    InPort<Transaction *> in;
8    OutPort<Transaction *> out;
9  };
```

`StageInterface` is now parameterized on the transaction (line 2) and the input and output port interfaces (lines 3-4). A `typedef` helps with readability and gets us back our example stage interface that uses FIFOs.

```
typedef
  StageInterface<MyTransaction, sc_fifo_in, sc_fifo_out> FIFOInterface;
```

3.2.3 Timing Policy

`Stage` hard-codes a performance model of a one nanosecond delay. A trivial way to generalize this is to allow the user to pass an integer through the constructor and use that as the delay. This assumes that the performance model will be a simple wait statement and nothing more complicated; not very general. Additionally we might want to support an untimed model where we would expect there to be no simulation performance penalty in calling `wait`. One way to do that is to ensure that the call to `wait` is removed by the compiler for untimed models. We define two timing policies `TimedPolicy` and `UntimedPolicy` fully aware that the modeler could design more complicated policies.

```
struct TimedPolicy {
  inline static void wait(int t) { ::wait(t, SC_NS); }
};

struct UntimedPolicy {
  inline static void wait(int t) { }
};
```

If `UntimedPolicy` policy is instantiated the call to `wait` is inlined with an empty function body, eliminating function call overhead.

3.2.4 Our Host `Stage` *Class*

Having defined several orthogonal policies figure 6 shows our generic stage class. This new stage class goes a long way in being generic and reusable. However the non-terminating `while` loop in the `process` function is not generic; it implies we're

```
1  template <class Transaction,
2            class Function,
3            class DelayModel,
4            class PortInterface>
5  class Stage : public sc_module,
6                public Function,
7                public DelayModel,
8                public PortInterface {
9  public:
10   Stage() { SC_THREAD(process); }
11
12   void process() {
13     while(1) {
14       Transaction * t = in.read();
15       t->data = f(t->data);
16       wait(1);
17       t->advance();
18       out.write(t);
19     }
20   }
21 };
```

Fig. 6. Our generic policy based pipeline stage class.

using thread processes (`SC_THREAD`) as opposed to method processes (`SC_METHOD`), or clocked threads (`SC_CTHREAD`). While we don't have space to show it here we also factor out the *process policy* into `ThreadPolicy` and `MethodPolicy` adding one more template parameter `ProcessPolicy` to the `Stage` class.

3.3 Putting it all together

Recall that our DSL initially uses a proxy class `Resource` for stages. To generate a complete simulation model we pass our policy classes to `Resource`.

```
Resource<Sqr<Data>, MyTransaction, TimedPolicy, FIFOInterface, Threading> s1;
```

To shorten this up a bit we can give reasonable default values to the template parameters or use an extra configuration repository class [6].

We don't have room to show the *transaction routers*. These are multi-ported modules parameterized on a transaction and use a vector indexed by the stage number to lookup the appropriate channel to write the transaction to. We have not yet decomposed these further into other policies. Pipelines that demand a more complicated resource arbitration scheme require an *arbitration policy*.

4 Conclusions and Future Work

We've presented techniques for modeling and simulating system level pipelines. A small DSL gives a compact representation of the pipeline and enables us to automatically generate tedious boilerplate and control code. Generic representations

of transactions and stages decomposed into policy classes allow us to reuse large amounts of code used to describe pipeline resources.

One aspect we haven't addressed is *when* a transaction can be initiated in the pipeline, the *issue latency*. Issue latencies are derivable from the DSL; [17] shows us how and [20] makes it fast. One area of future work are policies for gathering performance statistics as well as policies for generating implementation level models. More policies will be discovered as we model more complicated pipelines, including pipelines that use global state, such as processor instruction pipelines with instruction and data caches. In terms of abstractions used in our framework *concept models* [8] can help clarify requirements of our policies.

The pipeline DSL needs enhancing. For example, a stage replication operator S_1^3 could mean *replicate* hardware; instantiate S_1 three times as opposed to feedback (S_1*3). At the moment parentheses are meaningless but giving semantics to expressions such as ((S_1 >> S_2)*2 >> S_3)*3 by "unrolling" makes sense. Also, we could probably make pipeline descriptions even more concise by using the Boost lambda library [16] for specifying functors.

References

[1] Mark Aagaard and Miriam Leeser. A framework for specifying and designing pipelines. In *Proceedings 1993 International Conference on Computer Design*, pages 548–551. IEEE Computer Society Press, 1993.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison Wesley, 2005.

[3] ACM. *Proceedings of the Sixth International Conference on Generative Programming and Component Engineering (GPCE'07)*. ACM, October 2007.

[4] Andrei Alexandrescu. *Modern C++ Design; Generic Programming and Design Patterns Applied*. Addison Wesley, 2005.

[5] David Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, August 1991.

[6] Krzysztof Carnecki and Ulrich Eisenecker. *Generative Programming; Methods, Tools, and Applications*. Addison Wesley, 2000.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[8] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 06)*, pages 291–310. ACM Press, 2006.

[9] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer, 2002.

[10] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*. IEEE Computer Society, IEEE Computer Society, March 1999.

[11] Ed Harcourt, Jon Mauney, and Todd Cook. Formal specification and simulation of instruction-level parallelism. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*. IEEE Computer Society, 1994.

[12] Ed Harcourt, Jon Mauney, and Todd Cook. From processor timing specifications to static instruction scheduling. In Baudouin Le Charlier, editor, *Proceedings of the International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer, 1994.

[13] Jason Higgins and Mark Aagard. Simplifying the design and automating the verification of pipelines with structural hazards. *ACM Transactions on Design Automation of Electronic Systems*, 10(4):651–672, October 2005.

[14] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[15] Kai Hwang and Fayé Briggs. *Computer Architecture and Parallel Processing*. McGraw Hill, 1984.

[16] J. Jarvi and G. Powell. The Boost Lambda Library. `http://www.boost.org/doc/html/lambda.html` Date accessed 1/15/08.

[17] Peter M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corporation, 1981.

[18] Prabhat Mishra and Nikil Dutt. Modeling and validation of pipeline specifications. *ACM Transactions on Embedded Computing Systems*, 3(1):114–139, February 2004.

[19] Open SystemC Initiative. *SystemC*. www.systemc.org.

[20] Todd Proebsting and Christopher Fraser. Detecting pipeline structural hazards quickly. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 280–286. ACM, 1994.

[21] Mehrdad Reshadi and Nikil Dutt. Generic pipelined processor modeling and high perofmrnace cycle accurate simulator generation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'05)*. IEEE Computer Society, IEEE Computer Society, 2005.

[22] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*. IEEE, std 1666-2005 edition, December 2005.

[23] David Vandevoorde and Nicolai Josuttis. *C++ Templates*. Addison Wesley, 2003.

[24] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[25] Todd L. Veldhuizen. C++ templates as partial evaluation. In *PEPM'99 Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18. ACM SIGPLAN, 1999.