CASL: A Rapid-Prototyping Language for Modern Micro-architectures

Edward K. Walters II [a,*] J. Eliot B. Moss [a] Trek Palmer [a]
Timothy Richards [a] Charles C. Weems [a]

[a] *University of Massachusetts Amherst, Department of Computer Science, 140 Governors Drive, Amherst, MA 01003-9264 U.S.A.*

Abstract

We introduce CASL, the CoGenT Architecture Specification Language, a mixed behavioral-structure architecture description language designed to facilitate fast-prototyping and tool generation for computer architectures with deep pipelines and complicated timing. We motivate a number of CASL features using examples drawn from modeling the IBM Cell Broadband Engine, including implicit connection of components, pipeline support, dynamic information contexts, contention, and timing annotations. We also describe a number of different applications for CASL, including generating timing simulators and instruction schedulers.

Key words: Simulation, Architecture, ADL, Pipeline

1 Introduction

Mainstream computer architecture has now whole-heartedly embraced multiple instruction streams for the masses. Multi-core processors and related strategies lead to much more complex systems—which implies the need for more complex and capable simulators to explore and evaluate designs. Further, the

* Corresponding author.
Email addresses: ekw@cs.umass.edu (Edward K. Walters II), moss@cs.umass.edu (J. Eliot B. Moss), trekp@cs.umass.edu (Trek Palmer), richards@cs.umass.edu (Timothy Richards), weems@cs.umass.edu (Charles C. Weems).

speed of evolution of architecture requires much more rapid development of simulators and matching tools (such as compiler back-ends). The field needs a framework that can model these complex systems and yet produces tools that are efficient, and produces those tools automatically.

One of the most effective ways to describe and generate an architectural simulator is to use of an architectural description language (ADL), a domain-specific language designed to describe different facets of a target architecture such as behavior, timing, and structure. Ideally, an ADL allows one to describe the architecture using abstractions and primitives that make the descriptions more intuitive and compact. It also removes the need to create tools using general-purpose languages such as C++, which is more prone to inaccuracies and errors because of its lower level of abstraction.

Current ADLs fall into three categories [1]: behavioral, structural, and mixed. *Behavioral* ADLs describe an architecture from the point of view of the Instruction Set Architecture. They specify the format and semantics of the instructions, and are typically used to generate functional simulators. *Structural* ADLs provide a detailed description of a micro-architecture, possibly down to the gate and signal level. They usually contain some form of graph-based description of the structure, and applications include detailed timing simulation and synthesis of hardware. *Mixed* ADLs are a relatively new category; they attempt to strike a balance between descriptive detail and abstraction. They describe both structural elements such as pipelines and register files, and some elements of behavior. Mixed ADLs are usually applied to architectural validation and timing simulation.

Our primary contribution here is a detailed description of the design and novel features of the new mixed ADL CASL, the CoGenT Architecture Specification Language. CASL aims to provide intuitive and powerful means to describe current and future architectures at a component level of abstraction, without resorting to a general-purpose language to implement novel components.

CASL includes the following:

- A *component-oriented behavior and structure* language, with primitives and control structures de-

signed specifically to capture common micro-architectural idioms. CASL is abstract enough to express architectural concepts more succinctly than C or C++, but flexible enough to describe complicated architectural components in detail.

- The concept of *strands*, dynamic execution contexts that traverse the structural model, e.g., instructions traversing the pipeline, or requests to distributed memory. Strands accomplish two important tasks: they provide a simple and intuitive repository for control information, and they allow the description author to indicate which parts of the simulator must be active at any point in time, minimizing the amount of extraneous execution.
- A rich *timing* description facility, including the ability to describe contention, pipelining, and complex timing without being forced to resort to explicit signals as in hardware description languages such as VHDL [2] or Verilog [3]. CASL also allows the description writer to intermingle timing descriptions with component behavioral descriptions.

We proceed as follows: Section 2 provides an overview of the CoGenT project and the role of CASL within it; Section 3 describes IBM's Cell Broadband Engine, which we use for our examples; Section 4 motivates some of CASL's structural features using the CBE top level; Section 5 shows how CASL describes pipelines using the CBE SPE as an example; Section 6 describes current and potential applications for CASL; Section 7 describes related work; and Section 8 concludes.

2 Overview of the CoGenT Project

CoGenT stands for Co-Generation of Tools, particularly compilers and simulators. Compiler and simulator tools for systems research are difficult to develop and coordinate since each tool is complex in its own right, and both are dependent on aspects of the target architecture. The CoGenT project (see Figure 1) addresses this problem by providing two sets of components: multiple coordinated specifications that describe the target instruction set architecture and micro-architecture; and a set of tools that pro-

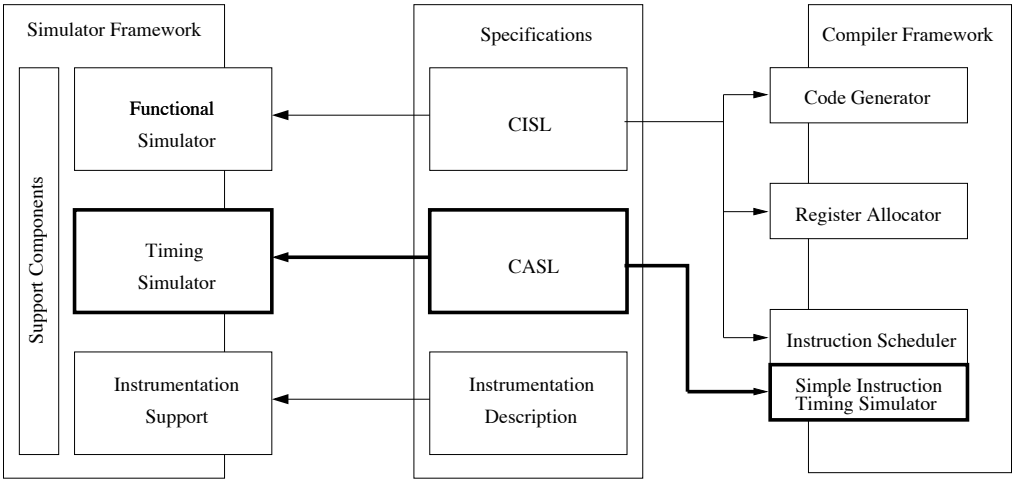

Fig. 1. The CoGenT Project

cesses these specifications and produces compiler and simulator components such as code generators, instruction schedulers, and simulators at varying degrees of detail. Generating a compiler and simulator from the same set of descriptions ensures that the two will always be consistent, and since automatic tool generation is fast, CoGenT allows designers to explore more design space in less time.

Prior systems tend to use either a simple language, making descriptions unwieldy, or an ADL augmented with a complete general purpose language, rendering analysis difficult. We address this issue by providing multiple coordinated ADLs. Currently CoGenT has two ADLs, though we may add more in the future:

- CISL, the CoGenT Instruction Specification Language [4,5], describes instruction set format and behavioral semantics of machine instructions, and
- CASL, the CoGenT Architecture Specification Language, describes micro-architectural structure, behavior, and timing.

We focus here on the design and features of the CASL language. We now offer a motivating example.

3 Modeling the Cell Processor Using CASL

We motivate design decisions we made for CASL using the Cell Processor as an example. Specifying concurrency and contention in modern architectures such as the Cell is a challenging problem, and we will show how we address it in CASL.

The Cell Broadband Engine (CBE) [6,7] is a chip multiprocessor designed by IBM, Toshiba, and Sony for advanced multimedia applications such as the Playstation 3. The CBE uses heterogeneous elements to provide high performance. Some of the processing occurs on a general purpose (PowerPC implementation) element, and some occurs on its SIMD elements. Because the CBE is both heterogeneous and distributed, its timing behavior is more complex than that of a conventional monolithic processor and can be difficult to describe and model. In the following sections we present some aspects of the CBE that challenge the simulator writer, and then show how specific CASL features address these challenges. We begin by discussing how to model a platform "in the large" by looking at the CBE as a whole, and then look at modeling specific aspects of one of the SIMD elements.

4 Platform-level CBE Structure

As seen in Figure 2 (adapted from [7] and [8]), the large-scale structure of the CBE contains two types of processing elements connected by a bus that also provides external connections. The two types of processing elements are the Power Processing Element (PPE), a general-purpose CPU that implements the PowerPC ISA, and the Synergistic Processing Element (SPE), a special-purpose vector unit. A typical CBE contains one PPE and eight SPEs.

A description of the CBE at this level of detail focuses on the *structure* of the platform. The specification must provide details on the *connections* between the elements, which we henceforth call *components*, and it must provide details on the *communication* between the components so that generated tools can derive accurate metrics such as timing. We discuss these in turn, providing examples of

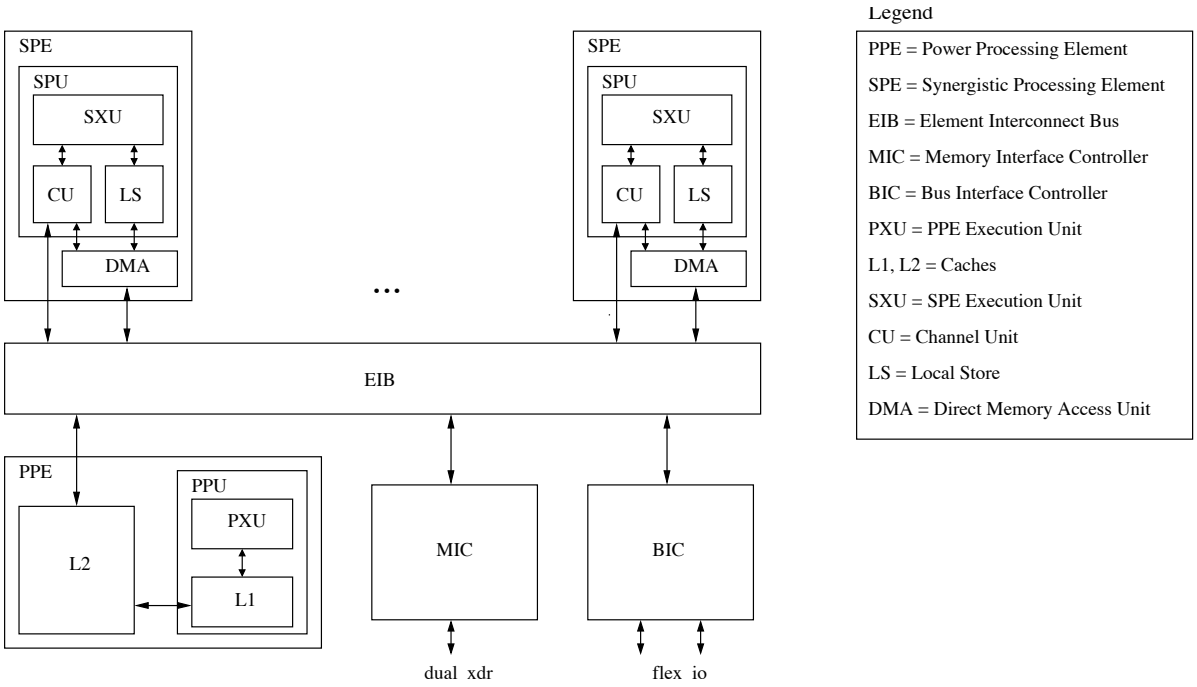

Fig. 2. The Cell Multiprocessor

corresponding CASL syntax where appropriate.

4.1 Modeling the Cell Platform with CASL

We consider in turn how to model components, connections, and communication.

4.1.1 Modeling Components

Describing how components relate to each other structurally is one of the primary functions of a structural or mixed ADL. The standard technique used by structural ADLs is a *netlist*, a graph that describes each type of component, its configuration and instances, and how it connects to other components. Each component presents a set of connection *ports*. Explicit connections between ports comprise the edges of the netlist graph.

CASL supports a similar, though more abstract, view of structure by providing the component as its primary language entity. Components can contain an *interface* section, which describes the information visible to other components such as port properties, and an *implementation* section, which describes

```
component CellProcessor {
 interface:
   connections(
    inout dual_xdr_t   dual_xdr,
    inout flex_io_t[2] flex_io);
   action clock;
   ...
```

(a) Cell Interface

```
implementation:
  SPE[8] spes;
  PPE    ppe;
  EIB    eib;
  MIC    mic;
  BIC    bic;
  ...
  action clock {
    ...
    index i in [0 .. 8);
    all i {
      spes[i].poll();
    }
  }
}
```

(b) Cell Implementation

Fig. 3. CASL Cell Component Code

internal information such as storage and behavior.

An example of a top-level CASL declaration of the CBE appears in Figure 3. The interface describes the connections (ports) and the actions visible to external components. In the CBE CASL interface in Figure 3(a), there are two connections: `dual_xdr`, which is a bidirectional (`inout`) connection that transfers a value of (a yet undefined) type `dual_xdr_t`; and `flex_io`, which is also bidirectional and transfers two values of type `flex_io_t`. The only action visible to other components is the action `clock`.

Each CASL interface may contain connection information, type, constant, and action declarations, and parameters. Components can be parameterized by types, values, or component interfaces. Each parameter may be constrained in terms of the values it may assume.

The implementation section of a component has scope similar to functions or data defined using the `protected` keyword in Java or C++, i.e., the actions, type declarations, and storage elements are visible only to the component itself and components that inherit from it. In Figure 3(b), the only elements in the implementation section are the declarations of the components contained within the CBE. Each of these components represents an instance of a component type declared elsewhere. The implementation also contains the behavioral code for actions declared in the interface.

4.1.2 Modeling Connections

The preceding example does not contain any specific port-to-port connections, but this is acceptable because CASL allows connections between components to be defined in two ways: either the components can be defined as peers and have their ports explicitly connected using assignment operators, or the structural relationship between the components can be used to infer the connections. The former method is standard in most ADLs, but CASL attempts to make the behavioral code in the components look more like conventional C++ or Java code instead of message-passing syntax. CASL accomplishes this by using method calling syntax for action triggers (i.e., messages from one component to another).

For example, in Figure 3(b), the `clock` action of the Cell component polls all eight SPE units in parallel using a CASL `all` statement (parallel iteration). Each of the `poll` action triggers appears as a method call to the individual SPE that is the target of the message. This programming idiom is natural and intuitive, while remaining flexible enough to handle arbitrary synchronous communication.

With this method of specifying messages, a connection exists between two components if one contains another as a member. Components that are members of other components can be declared in three different ways: as an instance of another component type (essentially a component variable), as a component parameter, i.e., as an external reference passed in to the component, and as an inner component. Each has its particular uses: component variables imply that the instance (but not the definition) is fully contained within the enclosing component, component parameters imply that a component reference is shared between two or more components, and inner components have both their instances and definitions contained within the parent. It is convenient to use inner components when creating a component that will never be used outside of the context of its parent component.

We show the form of all three in Figure 4. Figure 4(a) shows a component variable of component type `A` inside the definition of component `B`—every component of type `B` will contain a specific private instance of `A`, but other components can also instantiate variables of this type. Figure 4(b) shows an

```
component B {
  ...
 implementation:
  A a;
}
```

(a) Component Variable

```
component A { ... }
component B {
 interface:
  component param
    A a_param;
}
```

(b) Component Parameter

```
component B {
 ...
 implementation:
  component A { ... }
  A a_inner;
}
```

(c) Inner Component

Fig. 4. CASL Structural Component Relationships

example of parameterization. This instance of type `A` can be used by other components. Figure 4(c) shows an example of an inner declaration, where only class `B` can instantiate components of type `A`.

4.1.3 Modeling Communication

CASL provides two different ways to describe communication between components: explicitly passing messages through connections, and invoking actions of member components (similar to functions in a conventional programming language). The connections of the member component used during an action invocation are presented as parameters in the function call, and a return value is accepted in the case of synchronous calls. While most ambiguities in identifying connections can be resolved using the types and declaration order of the connections, one can label function parameters explicitly with the pertinent connection name.

5 More Detail: The Cell SPE

The SPE is a dual-issue SIMD processing element within the Cell Processor. While it is not a super-scalar processor, it does possess enough architectural features to make it an excellent example for research evaluation of an ADL.

The SPE (see Figure 5) contains two main storage elements: the local store (LS), a 256 KB single-ported pipelined SRAM, and a 128-bit $\times$ 128 register file. The SPE also contains a DMA unit for communicating with the EIB; a series of read/write latches (R/W) for communicating with the DMA and the issue logic; the issue logic itself (Issue), consisting of an Instruction Line Buffer (ILB) and the

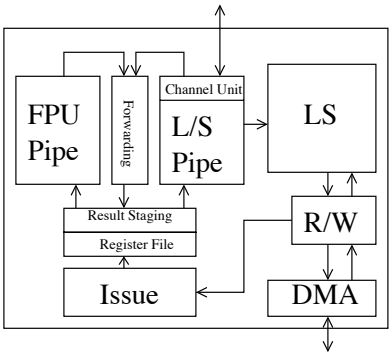

Fig. 5. Detail of the Cell SPE

issue control unit; the even pipeline, consisting of floating- and fixed-point computation units; and the odd pipeline, consisting of branch control, load/store, permuting, and channel control unit. There is also an extensive forwarding and staging system for communicating to other SPEs using channel mailboxes.

5.1 Describing the SPE

There is a significant amount of *pipelining* in the SPE: there are five execution pipelines (Fixed-arithmetic, Fixed-Shift, Byte, Permute, and Floating Point [8]), a Branch pipeline, an Issue pipeline, and the Local Store pipeline. The pipelines also bring up the issue of *control*, which is vital to any timing simulation.

Another issue is *contention*: the SPE has storage elements that may be accessed by more than one component in a single clock cycle. For example, the SPE register file has six read ports and two write ports, and the local store has contention with the DMA unit, the instruction fetcher, and the load/store execution unit on a cycle-by-cycle basis. The description of the SPE needs to address this in a concise and comprehensible manner.

The final issue is that of *timing*: some components may have actions that take multiple clock cycles. We believe it is advantageous to allow the architect to annotate these cases with CASL timing statements. In Section 5.2.5 we provide an example using the fetch stage of a conventional pipeline.

5.2 Using CASL to Model the SPE

We consider in turn support for describing: pipelining, dynamic information flows (strands), contention, and timing.

5.2.1 Pipeline Support

The above structural features, in conjunction with simple behavioral elements (e.g., logical operations), suffice to implement almost any architecture. But we aim for the designer to work at a higher level of abstraction for certain kinds of architectures. Therefore, we introduced language and library elements to support pipelines, one of the most common large-scale structures in modern architectures. CASL pipeline support is based on three types of elements: a custom set of *components* that ease describing pipeline connectivity, capacity, and functionality; a novel language construct called a *strand* that describes the control and routing information of a related group of information traversing the pipeline; and *timing annotations* that describe the degrees of parallelism within the pipeline.

The CASL components that support pipelining are *stages*, *buffers*, and *connectors*. These differ from conventional CASL components in the following ways: they can reference other components in their connection lists, they can process strands, and they must implement certain pre-defined interfaces.

A *stage* component contains the behavioral specification for one or more stages of a real pipeline. For example, the fetch stage for a simple MIPS-style pipeline would contain the behavioral code for fetching the instruction words from i-cache. Although stages typically represent a single cycle of work, the amount of time each stage takes is arbitrary. When a CASL stage contains the code for more than one stage of a pipeline, this requires two forms of additional support from the architect: explicit timing must be attached to the stage as execution will likely require multiple clock cycles to complete; and the buffers associated with the stage must be adjusted to hold the correct number of strands. For example, compressing a three stage pipeline to a one stage composite pipe requires that each strand have three-

cycle latency, and the buffers must deal with three strands at once, even if throughput is only a single strand per cycle. The only pre-defined action a stage needs to implement is the `visit()` action, which performs the stage's work.

Buffer components are designed to track the location of strands as they move from stage to stage. Every CASL strand, at any point in time, is held in exactly one buffer. The simplest implementation of a buffer is a latch, which can hold one strand. However, since CASL components can contain arbitrary code within action implementations, it is straightforward to create buffers that possess more involved behavior, such as the issue logic of the SPE. Buffers must provide actions for querying their available capacity and sending or receiving strands from other pipeline elements.

Connection components provide detailed routing and connection behavior between buffers and stages. Connections query the capacity of buffers and push or pull strands through the stage servicing the buffer. Connections are the only specialized pipeline elements that deal with many-to-one or one-to-many connectivity. For example, the issue stage in most multi-issue processors will involve routing from a single issue stage to multiple execute stages or reservation stations. In this case a one-to-many connector customized to the issue constraints of the target architecture would be appropriate. Similarly, a many-to-one connector would be appropriate at the other end of the pipeline, e.g., a completion unit.

Building a pipeline in CASL involves integrating these three types of components into a structure that corresponds closely to the physical pipeline. We have found that the most useful functionality arises when these elements are used in a buffer-stage-connector chain through the pipeline. Connectors serve as the drivers of the pipelined system: they pull strands out of buffers, drive them through stages, and dispatch them to the buffers in the next stage.

Figure 6 shows an example pipeline using this pattern for a simple DLX pipeline [9]. Note that we have omitted forwarding and storage (e.g., register) elements for the sake of clarity. In this example, we have five stages (IF, ID, EX, MEM, and WB), a buffer attached to the beginning of the latter four

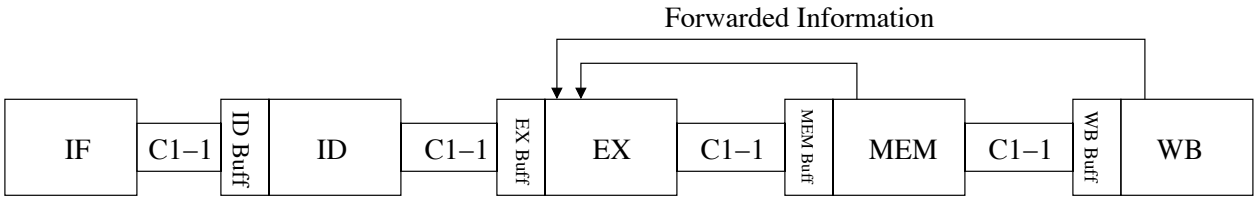

Fig. 6. DLX Pipeline Structure in CASL

stages, and a series of one-to-one (C1-1) connectors joining the stages and buffers. Information can be forwarded from the MEM and WB stages to the EX stage. Each stage, except for IF, is preceded by a corresponding buffer that stores a current or pending strand. The issue stage is a special case because it *generates* strands, which do not exist in a buffer until they are sent to the next stage.

While the pipeline can be described in a purely structural way (connecting up the ports), CASL supports a specialized notation to describe pipelines succinctly. The pipeline notation is adapted from CASL timing annotations (described below), so we provide a small bit of background here. CASL has three different operations for describing concurrency of terms: sequential operation: `a ; b` (for "`a` runs before `b`"); parallel operation: `a || b` (for "`a` runs in parallel with `b`"); and pipelined operation: `b1:s1 -{c1-1}-> b2:s2`, which indicates that connector `c1-1` joins stage `s1` with pre-attached buffer `b1` to stage `s2` with pre-attached buffer `b2` to implement pipelining. Using these three sets of operators, in conjunction with connectors that possess varying degrees of connectivity, we can describe arbitrary pipelines.

The CASL code for describing the DLX pipeline appears in Figure 7(a). As we see in the next section, CASL allows one to attach timing annotations to actions. This example declares the five stages, four buffers, and four connectors that comprise this pipeline, and joins them together using our pipeline notation. Here the only code the DLX components need to execute is the `clock` action, which models a clock tick by telling the connector `C1-1` between IF and ID to pull (i.e., generate) a strand from IF and start the execution process. The rest of the execution is driven by the strands produced by IF. While we have used generic buffers and connections for this example, most examples require the use of custom

```
component DLX {
 implementation:
  Fetch IF;
  Decode ID;
  Execute EX;
  Mem MEM;
  WriteBack WB;
  OneToOneConnector C1, C2, C3, C4;
  Buffers B_ID, B_EX, B_MEM, B_WB;

  action clock
   << IF -{C1}-> B_ID:ID -{C2}->
        B_EX:EX -{C3}-> B_MEM:MEM ->
        B_WB:WB >>
  { C1.pull(); }
}
```

(a) DLX Pipeline

```
component SPE {
 implementation:
  ...
 action clock
  << ILB:ILB_Rd -{C1_1}->
       B:Fetch -{C1_1}->
       B:Decode -{C1_1}->
       B:Dep -{C1_1}->
       B:Issue -{C1_1}->
       B:Route -{C1_1}->
       B:RF1 -{C1_1}->
       B:RF2 -{C1_n}->
       ((B:PERM1 -{C1_1}->
         B:PERM2 -{C1_1}->
         B:PERM3) ||
        (B:FX1 -{C1_1}->
         B:FX2) || ...)
        -{Cn_1} -> B:FW >>
  {  ILB_Rd.clock();  }
}
```

(b) Subset of Cell SPE Pipeline

Fig. 7. Pipelines in CASL

buffer or connector behaviors to implement the semantics of the particular target pipeline. Forwarding, on the other hand, is not included in the pipeline, since forwarded elements do not represents separate strands of information. CASL implements forwarding though inter-component messages and contention primitives (see Section 5.2.4).

CASL can also accommodate multiple issue by employing multiple buffer visits, dynamic issue using connection routing, pipelines with loops (a component reference can be used an arbitrary number of times inside a pipeline), and arbitrary issue conditions such as alignment constraints, specific issue slots for specific instruction types, and complicated windowing schemes.

Returning to the SPE in Figure 8, we show a version of the Cell SPE pipeline (adapted from a number of sources [6,8,10–13]). The structure of the pipeline is partitioned to correspond roughly to the functional units of Figure 5. Figure 7(b) show the CASL implementation of the subset of SPE pipeline that is shaded in Figure 8. The names of the stages are identical to the diagram, and we refer to buffers and connectors using generic names. Interesting features of the SPE pipeline description include the use of connectors with multi-way connectivity (`C1_n` and `Cn_1`), and parallel branches of the pipeline.

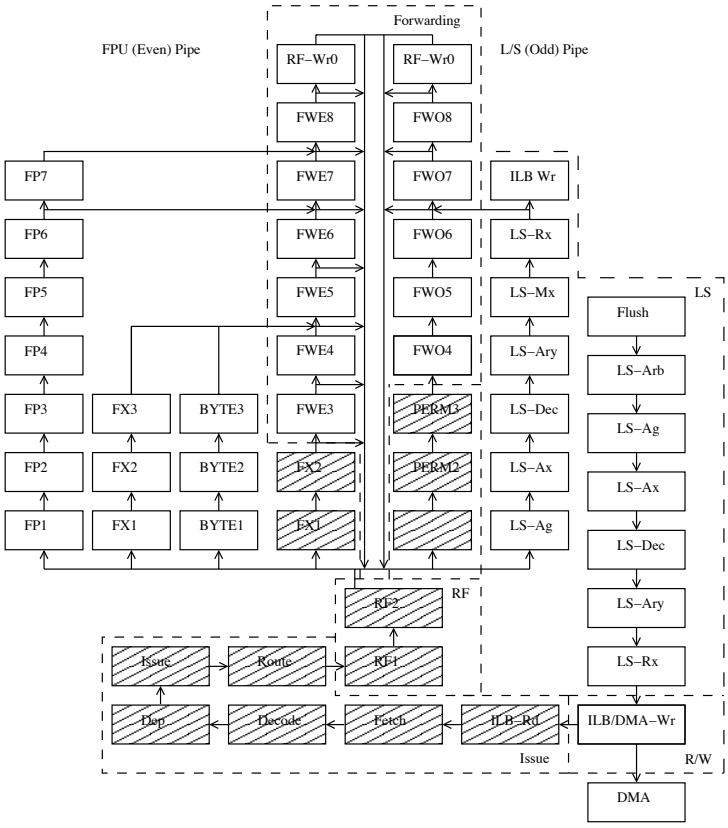

Fig. 8. Cell SPE Pipeline

5.2.2 Strands—Motivation

An issue that arises immediately when examining the pipeline in Figure 8 is how to represent the instructions and other dynamic elements that flow through the pipeline. Correct handling of instructions requires two elements: *control* for the instructions (routing through the pipeline network) and *decoding*, which places information such as register indices and computed values under the control of the instruction that produced them. A *strand* is a structure we use to hold this information.

Further examination of the pipeline shows that five strands may be useful: read/write requests from the Local Store, fetching into the ILB from the local store, decoding and issuing from the ILB, issued instructions in the pipeline and forwarding units, and requests from the DMA unit. We show these as separately labeled regions of the pipeline in Figure 9, each of which contains what we call the *info-set* of the region.

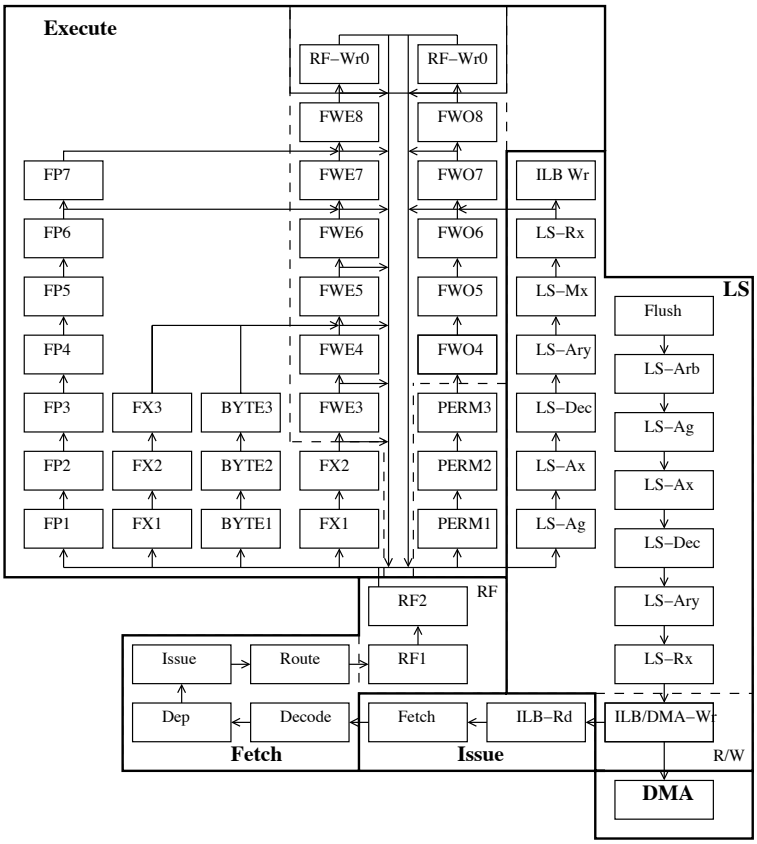

Fig. 9. Cell SPE Pipeline - Dynamic Regions

For example, the local store info-set services requests from either the DMA unit, the ILB, or the L/S pipe, providing the result of the request at some later time. Therefore, the local store info-set would contain information such as the identity of the requester, the address, and any fetched data. The ILB info-set enables the stage to request from the local store and how to handle prefetch requests, and so on for each potential strand.

A feature that makes it easier to reason about and model these structures is describing not only the structure of the pipeline, but also the characteristics of the information passing through it. This balance between structural behavior and reactive (i.e., dynamically triggered) behavior was an important consideration in the design of CASL.

A strand explicitly describes an info-set processed by an architecture, such as instructions, memory requests, or independent and/or asynchronous actions in the system. Unlike the static structural

elements of a CASL description, strands can be created dynamically, destroyed, and transferred from component to component in CASL behavioral code. Strands are similar to Java classes in that they contain data and operations that can be accessed by other elements (in this case other CASL components), but they are used under more restricted circumstances. For example, strands can be in one "location" only (buffer, storage element, etc.) at a time—copying out the strand means that the previous value is invalid. In addition, all strand data and operations are public by default, since their primary aim is to interact with other static components.

The main advantage of strands is that they allow one to address, using one construct, two major concerns of modeling the dynamic timing of an architecture: we encapsulate control information, exploited by the structural elements of the description to route and process the elements; and each strand models a particular resource request when determining if structural hazards exist. This gives CASL the best of both worlds: we can specify static structural elements as black boxes in a component graph, and we can then separately describe the dynamic elements that traverse this graph, assisting the structure in routing and processing the elements.

Looking again at Figure 9, we see potential for strands in each of the control regions. Local store strands are created once the arbitration stage (LS-Arb) is resolved, with one strand per memory request. The issue strand is a single strand whose purpose is loading the ILB. The fetch strands represent instructions pulled from the ILB that are decoded and have their read arguments fetched, and so on. We will see later that it is possible to use a single type of strand for both fetch and execute regions by utilizing dynamic sub-classing.

5.2.3 Strands—Properties and Examples

Figure 10(a) shows an example strand definition. This particular strand holds three read register indices and one write index (as in the Cell ISA), along with a raw representation of an instruction word. This instruction word is used in Figure 10(b) by the decoder to populate the register indices for use further

```
strand SPEIssueInst {
  reg_index_t read_reg1;
  reg_index_t read_reg2;
  reg_index_t read_reg3;
  reg_index_t write_reg;
  word_t inst_word;
  ...
  action path() {
   ...
  }
}
```

(a) Strand Definition

```
action issue {
  ...
  word_t raw_inst = ILB.fetch(PC);
  SPEIssueInst inst
    = new SPEIssueInst(raw_inst);
  // fills in regs
  decoder.decode(inst);
  ...
}
```

(b) Strand Creation

```
action commit {
  ...
  SPEIssueInst inst
   = CommitBuffer.next();
  delete(inst);
  ...
}
```

(c) Strand Termination

Fig. 10. CASL Strand Operations

down the pipeline. Note that we omit the decoder logic, since decoders are not typically written using CASL—they are generated from CISL (our ISA language) descriptions. We idealize the issue logic for the sake of brevity. Figure 10(c) shows termination of a strand using the `delete` keyword.

A strand must implement a `path` action, which specifies the strand's routing and actions as it traverses the structural components. For example, an instruction strand traversing a pipeline would have a `path` action designating what actions it requires in each stage so as to execute correctly. An integer instruction would require execution on an integer unit, in addition to stages such as memory and writeback (as in the DLX). This means of describing a strand's requirements has the following advantages: it allows the strand to choose the actions it needs from a component as it passes through, facilitating the simulation of heterogeneous strands that traverse the same path; and it provides a natural way of choosing among multiple functional units with the same operation set (e.g., two integer pipes), since the path action denotes only the type of components the strand interacts with, not the specific instances.

In CASL, paths are implemented as an action with special behavior and syntax. Each strand has a single path implemented as a `path` action. Within the body of a path action are action calls to the component types that the strand requires, in the order they are required. We provide an example of a path function for a basic SPE instruction in Figure 11(a). This fills out the path action in the previous figure. Here, `SPEIssueInst.path` contains the pertinent actions for the each of the stages of the

```
strand SPEIssueInst {
  ...
  action path {
   Fetch.visit();
   Decode.visit();
   Dep.visit();
   Issue.visit();
   Route.visit();
   RF1.visit();
   RF2.visit();
   execute();
   Forward.visit();
   ...
  }

  abstract action execute;
}
```

(a) Paths with Abstraction

```
strand SPEFixedPointInst
  extends SPEIssueInst {
 ...
 action execute {
   FX1.visit();
   FX2.visit();
   FX3.visit();
 }
}
```

(b) Child Strand Paths

Fig. 11. CASL Strand Paths

pipeline that the strand must traverse. In this particular instance, we assume every stage has a `visit` action that implements the functionality required by `SPEIssueInst`.

Paths can also contain triggers to ("calls on") other actions in their home strand or any ancestor of this strand. Combining this with inheritance (all CASL components and strands can use inheritance and method overriding) provides means to build a strand that can choose its path at run time. CASL supports this by using a limited form of dynamic subclassing similar to roles [14].

Dynamic paths work as follows: the architect creates a superclass with the path and the actions that will be dynamically sub-classed. These actions can either be abstract, or may have some default functionality. The actions must be included in the path description to be available for subclassing. Once the strand is ready to be subclassed, it can be transformed into the subclass using casting syntax (e.g., `inst = (SPEFixedPointInst) inst;`). The dynamically subclassed strand subsequently uses the subclass's actions.

For example, Figure 11(a) has an abstract `execute` action in its path, which must be implemented by a subclass before the execute stage is reached in the pipeline. Figure 11(b) contains a subclass of `SPEIssueInst` that represents a fixed-point instruction called `SPEFixedPointInst`. The execute action contains the implementation of this subclass's path for execution (this appears to be the 3-cycle Fixed Point pipeline). Once the strand reaches the `execute` action, it will continue the path in

`execute` and return to the original (superclass) path upon completion.

Dynamic subclassing typically occurs in response to additional information inferred about the strand. For example, an instruction that has been decoded (by a CISL-derived component, in CASL's case) is an excellent candidate for dynamic subclassing, since now more information is available within the strand that can be used for specialized processing or routing. This is how one would unify the `fetch` and `execute` strands in the Cell. The superclass instruction strands generated at the beginning of `fetch` can be dynamically subclassed to specific instruction types (e.g., floating-point) after decode. This additional information, along with the modified path information, can assist the pipeline in routing the instruction to the correct execution pipe.

One final point to note about strands is how they interact with the pipeline-specific components of CASL. As we stated above, strands interact with buffers, stages, and connectors in different ways to model progression through the pipeline. Buffers are used to store the strands and determine factors such as ordering and capacity. The combination of buffers and strands is CASL's preferred method of modeling structural hazards. Connectors are responsible for movement through the system by querying buffers and allowing eligible strands to proceed to the correct destination buffer. The strands then traverse the stage using the action mentioned on the path.

The actions within a path are all calls to stage actions. When a strand is inside the code for a particular stage, it is as if the strand called the stage with an implicit context called `this`. In this way the strand data (and actions) can be referred to within the stage code without forcing the stage to identify a specific strand. All strands that deal with a specific stage must conform to the same type (or supertype), however.

5.2.4 Contention Support

Any element in a storage unit is a candidate for side-effects during a clock cycle. Most architectures ensure that these side-effects possess some form of sequential consistency. In a simulator, non-deterministic

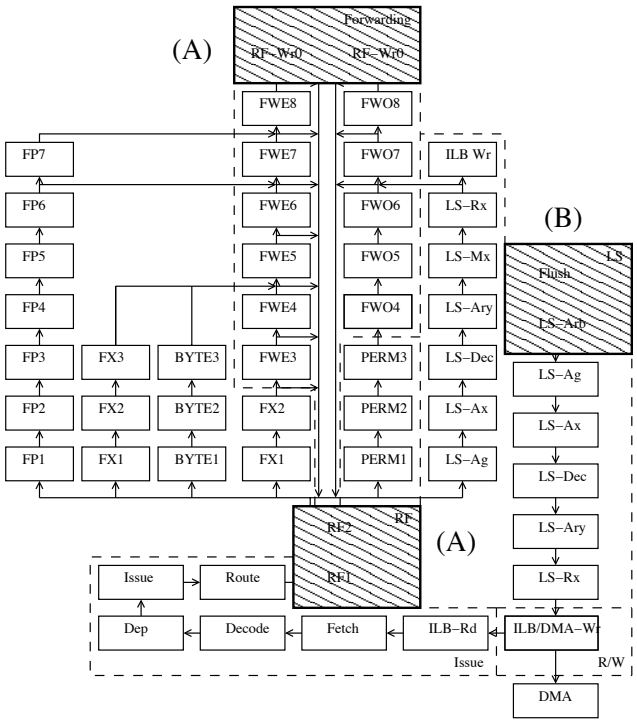

Fig. 12. Cell SPE Pipeline - Contention Regions

behavior is usually undesirable because it makes results more difficult to reproduce. While there are circumstances where conflicts and contention are desirable, we do not want it to occur when it is unwanted.

CASL addresses these concerns by providing language support for eliminating undesirable contention. We allow the architect to define an order over the actions of a particular component. When multiple actions are triggered during a particular clock cycle, the order arbitrates between the actions in a straightforward and predictable manner.

Figure 12 shows a motivating example, the Cell SPE Pipeline with two sets of regions shaded ("A" and "B"). The "A" regions correspond to register accessing stages and represent register reads and writes during the pipeline. The "B" region represents the arbitration that takes place during access to the single-ported SPE local store. These two regions embody the main sources of contention in the SPE. While there may be others (e.g., the forwarding unit or the ILB), we concentrate on resolving contention in these regions because their behavior definitely requires arbitration to achieve sequential consistency.

The policy for local store access is comparatively well defined [8,12]: an arbitration occurs on

a cycle-by-cycle basis between three potential requests. DMA requests have the highest priority, and are often scheduled ahead-of-time by the DMA unit. Load/Store instruction requests from the Odd SPE pipeline have the next highest priority, and ILB fetch requests have the lowest priority, fetching only when the local store is otherwise idle. Since the local store has only a single write port, it can process only one request each cycle, and the others block until the store is ready for them.

We model this contention using strands and ordering of the actions of the local store component. Notice that each category of request from the local store is modeled by a separate type of strand, i.e., we have DMA strands, L/S execution strands from the Odd SPE pipeline, and the issue strand. Instead of attempting to arbitrate explicitly between these three requests within the local store, we have each strand (operating by proxy through a stage component) trigger access to the local store using a different action. The order of these actions determines the priority.

In Figure 13(a) we show code for the local store that implements the action ordering. We use the same syntax for pipeline descriptions: `;` denotes that the left operand has higher priority than the right, and `||` denotes that the two have equal priority. The effects of this ordering on activity concerning the local store are as follows: if all three are triggered within the same time period, then the DMA request proceeds first, followed by the L/S request, followed by the Issue request. If the effects of these requests are instantaneous (i.e., take less than one clock cycle of time), then all three will be executed in order without the passage of time. If any are not instantaneous, then the others *block* while waiting for the higher priority ones to execute.

In effect, ordering implies the existence of queues for each type of action. Requests from higher priority queues are serviced before lower priority ones. If two actions have the same priority, then they share the same queue, and the actions are serviced in receive order. Receive order is also used for multiple triggers of the same action. Other queuing policies require explicit queue definition.

It is also possible to have a component with some actions that belong to the order and some that

do not. In this case only those actions that belong to the order are subject to blocking and priority. The actions that do not belong to the ordering are serviced in the cycle they are received with no ordering guarantees.

The same priority system can be used to ensure that the register file is updated correctly. In Figure 13(b) and (c), there are two components: the component in Figure 13(b) represents a generic register file parameterized by size, type of data stored, and read and write ports; the other in Figure 13(c) represents the register file for the Cell SPE. The generic register file does not make any assumptions about the ordering of its reads and writes. The Cell SPE, on the other hand, has an explicit read and write action for each port (used by the execution pipelines), and enforces the ordering of reads before writes (although we note that the ordering of the writes is arbitrary—we were not able to find sufficient evidence to determine this priority in Takahashi et al. [10])

5.2.5 Timing Support

CASL also supports describing the length of time a particular piece of code takes to execute. While a full description of CASL's timing descriptions is beyond the scope of this paper, we offer a brief overview of the options.

We have already seen examples of two of CASL's timing utilities in the previous section: pipeline support, and contention support. CASL also provides means to describe detailed timing in close coordination with component behavioral code. There are two different ways to accomplish this: either as a summary attached to the action as a whole, or at the statement level in combination with timing operators.

Attaching a timing summary to an action is similar attaching a pipeline to an action. A timing expression for the action is attached to the action implementation (see Figure 7 for example). This expression can contain terms such as integer constants and operators (e.g., subtract and add), and action triggers that occur in the behavioral code. For example, timing for a fetch stage with an instruction

```
component LocalStore  {
 interface:
  ...
  action dma_request;
  action loadstore_request;
  action issue_request;

  dma_request;
  loadstore_request;
  issue_request;
  ...
}
```

(a) Local Store Priorities

```
component RegisterFile {
 interface:
  type  param data_t;
  value param int size;
  value param int read_ports;
  value param int write_ports;
  type reg_index_t = bit[numbits(arch_size)];

  connections(
   in  reg_index_t[read_ports]  read_indices,
   out data_t[read_ports]       out_data,
   in  reg_index_t[write_ports] write_indices,
   out data_t[write_ports]      in_data);

  action read;
  action write;

 implementation:
  Register[size] regs;
  index i in [0 .. size);
  regs[i].type_t = data_t;
  ...
}
```

(b) Generic Register File

```
component SPERegisterFile {
 interface:
  action read_port_1;
  action read_port_2;
  action read_port_3;
  ...
  action write_port_1;
  action write_port_2;

  (read_port_1 || read_port_2 || ...);
    write_port_1 ; write_port_2 ;

  ...
 implementation:
  RegisterFile rfile;
  rfile.size = 128;
  rfile.data_t = bit[128];
  rfile.read_ports = 6;
  rfile.write_ports = 2;
  ...
}
```

(c) Prioritized Register File

Fig. 13. CASL Strand Contention

cache access (implemented by triggering `ICache.fetch(addr)`) would require a timing annotation of `action fetch << value(ICache.fetch) >>` { `...` }. This behavioral code is considered to have completed once the appropriate amount of time has elapsed, although the architect must take care since (unless otherwise protected by the priority operators above) action implementations are not atomic sections unless explicitly indicated as such by the `atomic` keyword.

While this level of timing annotation serves well for most situations, we can also describe timing on a statement-by-statement level. In this case, each statement can be annotated with a timing expression using a syntax similar to the action-level annotations. Statements can also be combined using the parallel and sequence operators and grouping operators (i.e., blocks) to make parallelism more explicit.

In conclusion, CASL treats the issue of combining behavior with timing in two different ways:

either at the action level or the statement level. Action level timing is convenient because it effectively allows the author to write behavioral code without regard to detailed timing concerns. Statement level timing is more difficult to reason about, but provides the maximum amount of expressiveness for complicated behavior.

6 Applications

The CASL language has a number of important applications, including:

- *Timing Simulators*: CASL is the CoGenT Project's primary means of describing micro-architecture and timing, and as such forms the basis for creating any tools that simulate timing, whether it is a detailed timing simulator, or a less detailed simulator for use with a functional simulator in a fast-forwarding capacity.
- *Instruction Schedulers:* CASL is also the basis for using the architectural structure present in its description to generate instruction schedulers. Since our architecture description language (CASL) and our behavioral language (CISL) share many of the same operators in their basic behavioral language, it is straightforward to match ISA functionality with architectural functionality.
- *Static Timing Analysis*: Since most of a CASL description is static outside of strands, it is possible to adapt standard static timing analyses such as WCET [15] to obtain some approximation of static timing numbers for an architecture. In addition, CASL's behavioral interaction constructs operate only over finite sets (CASL is deliberately non-Turing complete in this aspect). Thus infinite execution sequences cannot exist, and one can always complete a finite timing estimate.

We are currently developing most of these applications, with emphasis on fast accurate simulation.

7 Related Work

While there are many structural and behavioral ADLs, it is only recently that there has been substantial research in mixed ADLs. Some current mixed ADLs are EXPRESSION [16], LISA [17], Facile [18], MADL [19], ADL [20], Pipe [21], and Liberty [22]. CASL is most closely related to Liberty and Facile.

Comparing with Liberty, CASL's structure and parameterization design is similar. However, CASL differs in the following respects: it does not treat components as strict black-boxes for inter-component analysis—its behavioral language is much more integral; it provides explicit support for pipelines and timing annotations; it handles control using strands instead of explicit control ports; and its descriptions support generating compiler-related components.

Facile provides what could be considered a precursor to strands in both its inference of dynamic simulation code from a description and its pipeline memoization techniques [23]. However, CASL's techniques apply to any dynamic element that appears in the architecture, not just pipelined instructions.

Further advantages of CASL include: it aims for better and more comprehensive tool support; it can express the details of modern architectures, such as complex pipelines and timing; and it supports separate ISA specifications (syntax and semantics) in CISL [5], keeping CASL specifications smaller and cleaner, and simplifying generation of functional simulators and compiler code generators.

8 Conclusions

We introduced here a new ADL called CASL, whose innovations include built-in support for pipelines, timing, and contention, and a new control abstraction called strands. Strands leverage features from object-oriented languages such as dynamic subclassing: these features provide a compact means of describing information flow throughout a pipeline, while at the same time enable specialization of instructions to improve simulation performance. Strands also serve as a convenient means of representing control information, and can be used in groups to model structural hazards.

We also motivated CASL's new features by looking at a modern architecture, the Cell Broadband Engine, elucidating the challenges posed in describing such a system. We paid particular attention to the Cell SPE, since it is well-documented and has interesting pipeline and contention behavior.

Future work on CASL includes completing a static analysis engine attached to the compiler front-end that takes advantage of CASL's extensive behavioral and timing features. We also intend to generate compiler and simulator components from CASL in conjunction with the other languages that comprise the CoGenT project.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant number CNS-0615074. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] W. Qin, S. Malik, Architecture Description Languages for Retargetable Compilation, CRC Press, 2002.

[2] R. Lipsett, C. F. Schaefer, C. Ussery, VHDL: Hardware Description and Design, Kluwer Academic Publishers, 1989.

[3] D. Thomas, P. Moorby, The Verilog Hardware Description Language, Kluwer Academic Publishers, 1995.

[4] J. E. B. Moss, T. Palmer, T. Richards, E. K. W. II, C. C. Weems, CMDL: A class-based machine description language for co-generation of compilers and simulators, in: Proceedings of the 2004 International Parallel and Distributed Processing Symposium Workshop on Next Generation Software, IEEE, IEEE Computer Society, Santa Fe, NM, 2004, p. 8 pp.

[5] J. E. B. Moss, T. Palmer, T. D. Richards, E. K. W. II, C. C. Weems, CISL: A class-based machine description language for co-generation of compilers and simulators, International Journal of Parallel Programming 33 (2-3) (2005) 231–246.

[6] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy, Introduction to the Cell multiprocessor, IBM J. Res. Dev. 49 (4/5) (2005) 589–604.

[7] M. Gschwind, Chip multiprocessing and the Cell Broadband Engine, in: CF '06: Proceedings of the 3rd conference on Computing Frontiers, ACM Press, New York, NY, USA, 2006, pp. 1–8.

[8] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire,

M. Peyravian, V. To, E. Iwata, The microarchitecture of the synergistic processor for a Cell processor, IEEE Journal of Solid-State Circuits 41 (1) (2006) 63–70.

[9] D. A. Patterson, J. L. Hennessy, Computer architecture: a quantitative approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[10] O. Takahashi, R. Cook, S. Cottier, S. H. Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, H. Oh, S. Onish, J. Pille, J. Silberman, The circuit design of the synergistic processor element of a Cell processor, in: ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design, IEEE Computer Society, Washington, DC, USA, 2005, pp. 111–117.

[11] N. Mäding, J. Leenstra, J. Pille, R. Sautter, S. Büttner, S. Ehrenreich, W. Haller, The vector fixed point unit of the synergistic processor element of the Cell architecture processor, in: DATE '06: Proceedings of the conference on Design, automation and test in Europe, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006, pp. 244–248.

[12] S. Dhong, O. Takahashi, M. White, T. Asano, T. Nakazato, J. S. A. K. H. Yoshihara, A 4.8GHz fully pipelined embedded SRAM in the streaming processor of a Cell processor, in: Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC, Vol. 1, 2005 IEEE International, 2005, pp. 486–612.

[13] S. Mueller, C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. M. S. Dhong, The vector floating-point unit in a synergistic processor element of a Cell processor, in: 17th IEEE Symposium on Computer Arithmetic (ARITH-17) 2005, 2005, pp. 59–67.

[14] G. Gottlob, M. Schrefl, B. Rock, Extending object-oriented systems with roles, ACM Transactions on Information Systems 14 (3) (1996) 268–296.

[15] P. Puschner, C. Koza, Calculating the maximum execution time of real-time programs., Real-Time Systems 1 (2) (1989) 159–176.

[16] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau, EXPRESSION: a language for architecture exploration through compiler/simulator retargetability, in: DATE '99: Proceedings of the conference on Design, automation and test in Europe, ACM Press, New York, NY, USA, 1999, p. 100.

[17] S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr, LISA—machine description language for cycle-accurate models of programmable DSP architectures, in: Design Automation Conference, 1999, pp. 933–938.
URL `citeseer.ist.psu.edu/pees99lisa.html`

[18] E. C. Schnarr, M. D. Hill, J. R. Larus, Facile: A language and compiler for high-performance processor simulators, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2001, pp. 321–331.

[19] W. Qin, S. Rajagopalan, S. Malik, A formal concurrency model based architecture description language for synthesis of software development tools, in: LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, ACM Press, New York, NY, USA, 2004, pp. 47–56.

[20] S. Onder, R. Gupta, Automatic generation of microarchitecture simulators, in: ICCL '98: Proceedings of the 1998 International Conference on Computer Languages, IEEE Computer Society, Washington, DC, USA, 1998, p. 80.

[21] C. W. Milner, J. Davidson, Quick piping: a fast, high-level model for describing processor pipelines, in: Proceedings of the Joint Conference on Languages Compilers and Tools for Embedded Systems (LCTES), 2002, pp. 175–184.

[22] M. Vachharajani, N. Vachharajani, D. I. August, The Liberty structural specification language: A high-level modeling language for component reuse, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2004, pp. 195–206.

[23] E. Schnarr, J. R. Larus, Fast out-of-order processor simulation using memoization, in: ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems, ACM Press, New York, NY, USA, 1998, pp. 283–294.