

© 2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

The final version of this work can be found at: <http://dx.doi.org/10.1109/SAMOS.2011.6045458>

# ADL-Based Specification of Implementation Styles for Functional Simulators

David A. Penry and Kurtis D. Cahill  
Department of Electrical and Computer Engineering  
Brigham Young University  
Provo, UT, USA  
dpenry@ee.byu.edu

**Abstract**—Functional simulators find widespread use as subsystems within microarchitectural simulators. The speed of functional simulators is strongly influenced by the implementation style of the functional simulator, e.g. interpreted vs. binary-translated simulation. Speed is also strongly influenced by the level of detail of the interface the functional simulator presents to the rest of the timing simulator. This level of detail may change during design space exploration, requiring corresponding changes to the interface and the simulator. However, for many implementation styles, changing the interface is difficult. As a result, architects may choose either implementation styles which are more malleable or interfaces with more detail than is necessary. In either case, simulation speed is traded for simulator design time.

We show that this tradeoff is unnecessary if an orthogonal-specification design principle is practiced: specify *how* a simulator is to be implemented separately from *what* it is implementing and then synthesize a simulator from the combined specifications. We show that the use of an Architectural Description Language (ADL) with constructs for implementation style specification makes it possible to synthesize interfaces with different implementation styles with reasonable effort.

## I. INTRODUCTION

Microprocessor and SoC architects use simulators to evaluate new ideas, explore the design space, and validate the behavior of new microprocessors and embedded systems. One important class of simulators is *functional simulators* – simulators which model the instruction-set behavior of a microprocessor. These simulators are useful for early software development, but also find widespread use as subsystems within microarchitectural simulators which provide near-cycle-accurate predictions of performance.

Ideally, architects would like to explore substantial portions of the design space. This can only be possible if development time for simulators is short and simulation speed is high enough to allow simulation of large samples of code. Functional simulation speed can easily become a bottleneck, particularly when the microarchitectural performance prediction model is simple or uses sampling techniques [1].<sup>1</sup>

The simulation speed of a functional simulator is closely tied to its *implementation style*. We define implementation

style as “the means by which the simulator emulates target instructions.” Researchers have introduced many implementation styles; familiar examples include interpretive simulation, compiled simulation, and binary translation. Architects must choose which implementation style to apply in their simulator, making tradeoffs between simulator development time, simulation speed, and simulator capabilities.

When a functional simulator is combined with a timing simulator to form a microarchitectural simulator, the simulation speed of the functional simulator is also closely tied to the interface which it presents to the timing simulator. The interface defines the means of control that the timing and functional simulator offer each other and the information which is communicated between them. This interface is affected by the needs of the timing simulator and the overall organization of the microarchitectural simulator. Some microarchitectural simulator organizations will require more or less detailed information and more or less detailed control than others. For example, one timing simulator may wish to control the time at which operands are read and written, but another may not need this control. The level of detail may evolve over time as the microarchitectural design space is explored. For example, early in the design process, timing simulators may abstract many details of the timing and require little information from the functional simulator. As the design becomes better-specified, the timing simulators become more detailed and may then require more information. Furthermore, a single timing simulator may require multiple levels of information or control – e.g., one for detailed simulation and one for fast-forwarding.

The requirement to provide multiple, potentially evolving interfaces increases the difficulty with which an implementation style can be applied. In higher-performing implementation styles such as binary translation, the code which must be generated will be different for each interface, leading to much more implementation complexity. As an example, [2] reports the difficulties involved in changing QEMU [3], a binary-translating simulator, to support speculation and provide more instruction information in its interface.

This complexity may lead simulator developers to either choose easier – and often slower – implementation styles or to avoid interface changes which would otherwise be beneficial. In either case, simulation speed is traded for development time.

This work was supported by National Science Foundation grant CCF-1017004.

<sup>1</sup>When sampling is used, most of the microarchitectural simulator’s execution time may be spent in fast-forward mode between samples and functional simulation speed may be more influential than timing simulation speed.

The goal of this work is to obviate the choice between functional simulator speed and development time. We do so by practicing a fundamental design principle:

### **Orthogonal-Specification Principle**

*Specify how a simulator is to be implemented separately from what it is implementing and synthesize a simulator from the combination of the specifications.*

### *Contributions*

The primary contributions of this work are:

- The Orthogonal-Specification Principle for functional simulators.
- A description of how Architectural Description Languages (ADLs) – domain-specific languages which provide architects with constructs to specify instruction behavior – can be extended to enable orthogonal specification of implementation styles. No previous ADL has provided a set of constructs which support orthogonal specification.
- A case study using such an ADL demonstrating the potential of orthogonal specification.

By adopting the Orthogonal-Specification Principle and ADLs which are extended to support it, architects will be able to easily specify implementation styles and apply them to create simulators with multiple, evolving interfaces – thus improving both simulator speed and development time and allowing exploration of a greater portion of the design space, leading to improved designs.

## II. BACKGROUND

### *A. Implementation Styles*

Researchers have proposed many implementation styles for functional simulators. The most notable are listed below.

**Interpretive** simulation is the simplest of the implementation styles. The simulator fetches, decodes, and executes each dynamic instruction. The heart of the execution code is usually a switch statement based on the result of decoding or an indirect call to an instruction-specific execution function (called an *instruction function*). For example, an ADD instruction has a corresponding ADD function which performs its behavior. This style is able to easily support self-modifying code. However, these simulators tend to be slow.

**Static compiled** simulation [4], [5] translates each static instruction in a target program into high-level (e.g., C) or assembly-level simulation code through a pre-processing step and compiles said code to form the simulator. An ADL compiler would actually synthesize the compiler which translates target programs into simulators, not the simulator itself. Simulators implemented using this style can be very fast, as all fetch and decode behavior is performed at simulator compile time, instruction fields can be taken as constants, and inter-instruction optimizations are often possible. However, such simulators are unable to support self-modifying code, and large target programs can cause host compilers to take excessive

time or memory if care is not taken when generating the code [6], [7], [8].

**Dynamic compiled** simulation [9], [10] performs the same kind of translation as static compiled simulation, but at run time instead of compile time. The simulator generates code in a high-level language, asks an external compiler to create a shared object from that code, and then loads the shared object. Self-modifying code is supported by invalidating the compiled code when an instruction changes. The overhead of calling an external compiler is quite high, though the resulting code is fast for the same reasons it is fast in static compiled simulation.

**Binary-translated** simulation [11], [12], [13], [14] is similar to dynamic compiled simulation, however a dynamic code generator in the simulator is used instead of an external compiler. In general, binary translation is seen as a very high-performance method of simulation with the same optimization possibilities that exist in static compiled simulation. There is usually a tradeoff to be made between the quality of the generated code and the speed of code generation. Many high-speed binary translation schemes attempt to only translate code which is sufficiently “hot” in order to avoid code generation costs for infrequently executed code.

**Just-in-Time cache compiled (JIT-CCS)** simulation [15], [16] adds caching to interpretive simulation. As a static instruction executes for the first time, the simulator caches the results of decoding – the instruction function and pointers to the operands. The simulator reuses this decoding when it executes the instruction in the future. Thus decode happens only once per static instruction. Self-modifying code is supported by invalidating the cache when an instruction changes.

**Instruction-set compiled (ISCS)** simulation [17] uses a pre-processor which initializes the state of a JIT-CCS-like cache for a target binary. This pre-processor also specializes the instruction functions for the instructions actually present in the program by selecting some bits from the instruction encoding and making those bits a constant in the specialized instruction. (For example, the ARM instruction set benefits from specialized instructions whose predicate is always true.) An ADL compiler would synthesize the pre-processor, which is then run to produce simulators. Self-modifying code is supported by invalidating the cache when an instruction changes and falling back to interpretive simulation for that instruction. Large target programs can cause host compilers to take excessive time or memory.

**Hybrid compiled** simulation [18] reduces the compile-time overhead of ISCS simulation by removing the initialization of the decoded-instruction cache. The pre-processor generates the same kinds of specialized instruction functions, but also generates a specialized decoder which can select the specialized execution functions when appropriate and is used at runtime to fill the cache. An ADL compiler would synthesize the pre-processor used to create the decoder and specialized instruction functions. Self-modifying code is supported as in ISCS, but the specialized decoder and the specialized instruction functions are used as the fall-back mechanism.

TABLE I: Microarchitectural simulator organizations and required levels of detail

Organization	Informational detail	Semantic detail
Functional-First	Moderate: register numbers, branch resolution, effective addresses	Low: single call per instruction or basic block
Timing-Directed	High: register numbers, branch resolution, effective addresses, operand values	High: individual operand read/write, steps of execution
Timing-First	varies	Low: single call per instruction
Speculative Functional-First	Moderate: register numbers, branch resolution, effective addresses	Low: single call per instruction or basic block

### B. Functional simulator interfaces

Functional simulators are often integrated into microarchitectural simulators. In such simulators, the timing simulator makes calls to the functional simulator through some interface; these calls perform instruction semantics and return information about instruction execution. We refer to the amount of information present in an interface as its *informational detail*. We refer to the amount of control present in the interface as its *semantic detail*. Differing levels of detail may affect the number of functions, function signatures, and data structures comprising the interface.

The design of the interface and its level of informational and semantic detail are affected by the needs of the timing simulator and the overall organization of the microarchitectural simulator. A taxonomy of microarchitectural simulator organizations was first introduced in [19] and has since been extended [20]. Table I lists the various organizations as well as the typical levels of informational and semantic detail required to support each organization. Multiple interfaces may be required; e.g., microarchitectural simulators which use sampling may require a “fast-forwarding” interface which simply executes  $N$  instructions and reports no information; this interface is provided in addition to the “detailed” interface.

### C. ADLs

An Architecture Description Language (ADL) is a domain-specific language which provides architects with constructs to specify instruction formats and behavior. ADLs are commonly used to provide descriptions of an instruction set architecture which can be synthesized into a functional simulator or used to retarget a compiler. Most ADL compilers synthesize simulators using a single implementation style; interpretive simulation is the most common style. A few ADL compilers have been extended to synthesize multiple styles. Many ADLs have previously been proposed; we mention only those which have some relevance to implementation styles.

D’Errico and Qin [21] describe a simple ADL which allows synthesis of both interpretive and dynamic compiled simulators from the same language description. Specific language constructs for describing styles are not provided, though they describe some instruction constructs which make it easier to perform the synthesis of both styles.

Leupers, Elste, and Landwehr [22] describe the generation of both interpretive and static compiled simulators from a single description; however in this case the description is of a

microarchitecture and the instruction set is inferred from the description.

## III. THE ORTHOGONAL-SPECIFICATION PRINCIPLE

Developing a functional simulator is conceptually straightforward – simply implement the ISA manual. However, implementing a simulator with multiple interfaces can be rather complex. Each interface requires its own implementation, and the implementation must take into account details of the interface such as data structure definitions and function signatures. The most obvious way to do this is to implement different functional simulators for each interface. Obviously such an approach takes significant time, leads to large amounts of code duplication, and requires extensive validation.

The creation of multiple interfaces can be greatly simplified by specifying a single highly-detailed interface *once* and then synthesizing lower-detailed interfaces from that specification as needed. This concept was introduced in [23] as the Single-Specification Principle and was shown to lead to very short development times – mere minutes to specify a new interface – as well as easier validation and improved simulation speed.

An underlying assumption of [23] was that an implementation of any derived lower-detail interface could be provided without significant user intervention. For implementation styles which have been built into a tool chain, this is clearly possible. However, such tool chains limit architects to using only the styles built into the tool chain; they cannot use newly developed implementation styles and cannot tune an implementation style for their needs.

A better approach is to allow architects to describe styles of implementation. We call such an approach the *orthogonal specification* approach and can state it as an additional design principle to be practiced in conjunction with the Single-Specification Principle:

### Orthogonal-Specification Principle

*Specify how a simulator is to be implemented separately from what it is implementing and synthesize a simulator from the combination of the specifications.*

The simulator design complexity problems which multiple interfaces and multiple implementation styles present are solved by the joint practice of the Orthogonal-Specification and Single-Specification Principles. The architect only specifies a particular implementation style once and then is able to implement multiple interfaces with that style. The architect

does not need to change instruction specifications or interface specifications to support different implementation styles.

#### A. Applying the Orthogonal-Specification Principle

Orthogonal specification can be seen as a form of aspect-oriented programming [24]; implementation styles and interfaces are aspects of the system while instructions are the objects. Aspect-oriented programming is made much easier with tool and/or language support. Therefore, we propose that constructs for specifying and synthesizing implementation styles be added to ADLs. The resulting methodology for simulator design, extended from that of [23], looks like:

- 1) **Specify the instruction set at a high level of detail.**
- 2) **Describe a timing simulator interface.** This interface is used for initial debugging of the instruction set description. We recommend an interface with a low level of semantic detail and a high level of informational detail.
- 3) **Describe the interpretive implementation style.** This style is the easiest to understand and debug. Our tool chain includes interpretive simulation as a built-in style so the user need not specify it.
- 4) **Synthesize and validate the resulting simulator.** The validation should be done by running a large number of programs whose output can be tested; ideally an ISA validation suite would be used.
- 5) **Describe and validate additional interfaces and/or styles.** Interfaces are validated using previously validated styles while styles are validated using previously validated interfaces. These validations need not be as extensive as the original validation; the instruction semantics are already known to be good and the only mistakes which can be made are in interface or implementation style specifications.
- 6) **Repeat step 5 as necessary.** We emphasize that it is neither necessary nor desirable to specify or validate all interfaces or implementation styles *a priori* when designing a microarchitectural model.

#### B. ADL support for orthogonal specification

To support such a design flow, an ADL must have constructs which allow the specification of implementation styles. These specifications must be able to:

- 1) specify the form in which code is to be generated when the simulator is synthesized.
- 2) query the interface and instruction definitions.
- 3) extend the interface and instruction definitions.
- 4) support the per-static-instruction code specialization used in many implementation styles.

We now discuss each ADL requirement in turn, illustrating each using an ADL named LIS which we have extended to support orthogonal specification.<sup>2</sup> The simulators which are generated from this LIS description are specialized for use with the Liberty Simulation Environment (LSE) [25]. Because

<sup>2</sup>LIS was developed specifically to implement the Single-Specification Principle and is partially described in [23].

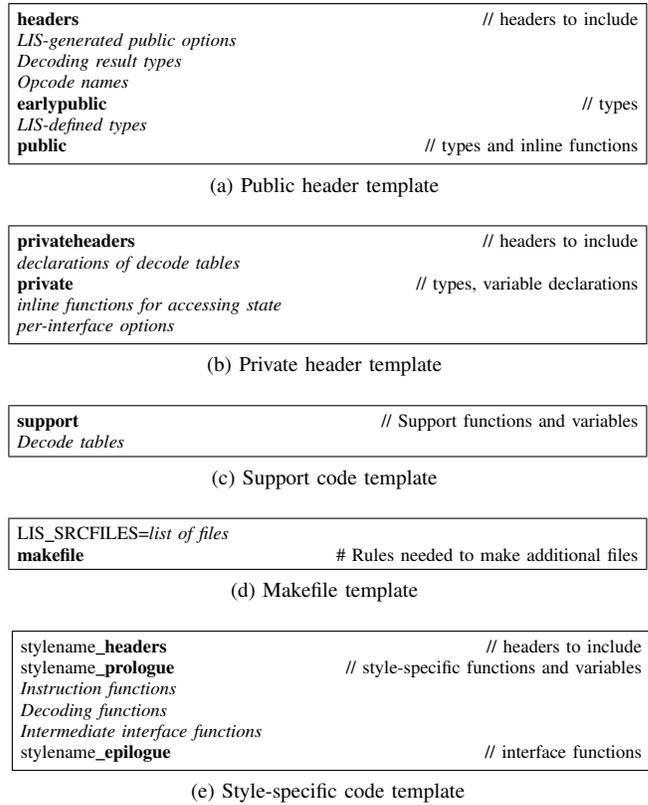


Fig. 1: Templates for generated C++ files

the implementation details of LIS are neither the focus nor a contribution of this paper, we do not provide detailed explanations of all LIS constructs. We wish to emphasize that *equivalent constructs could (and should) be added to other ADLs.*

1) *Code generation specifications:* An implementation style must be able to specify how to generate the code of the simulator (or for styles such as static compiled simulation, the simulator compiler). This code may include datatype definitions, support functions for the style, instruction functions which perform semantics, and functions which themselves may generate code. Some simulator code needs to be generated only once, but other simulator code may need to be generated once per interface which is implemented using this style.

The LIS compiler takes a LIS specification as input and outputs several C++ files – a public header file used by clients of the functional simulator, a private header file, a support code file, a makefile, and source files for each implementation style. These files are generated from templates; the user can view the process of writing a LIS description as a process of filling in sections of the template. Figure 1 provides an outline of the templates for each generated file. Italicized portions of the outline are generated by the LIS compiler. Bolded portions of the outline are supplied by the user and are known as *codesections*. Each codesection has a name indicated in bold; their intended uses are given as a comment following the name.

LIS provides two constructs for filling code sections. These constructs are not found in other ADLs. The `codesection` construct is used for code which is to be inserted *once* into the generated simulator. The `generator` construct is used for code which is to be inserted once per interface function implemented using the style. The user may define additional, non-standard code sections and indicate where they are to be inserted.

The following LIS code illustrates these concepts. It comes from the ISCS specification and defines a data structure which is used to cache pointers to the instruction functions for each interface function. The `generator` statement on lines 1-3 inserts its body into the user-defined `ISCS_STRUCTDEF` code section once for each interface function. The `codesection` statement on lines 5-9 inserts its body, which includes the `ISCS_STRUCTDEF` code section, once into the private header file. The odd-looking macros prefixed by `%%` will be explained later.

---

```

1  generator iscs_structdef {
2    void (%%NAME())(REGPTRS_ptr_t, %%PARMS());
3  }
4
5  codesection private {
6    struct LIS_etableC_t {
7      LIS_CODESECTION(iscs_structdef)
8    };
9  }

```

---

Note that not all of the code of the simulator needs to be in the LIS specification; the specification only requires the code which could be influenced by the interface or instruction set. The generated code can (and usually does) call support libraries. Such libraries increase code reuse and can greatly reduce simulator design time. For example, the binary translated style we have defined calls LLVM libraries [26] to perform code generation.

Multiple styles can be generated in one specification by grouping `codesection` and `generator` statements within named `style` constructs.

2) *Querying interfaces and instructions*: Orthogonal specification requires that the implementation style be independent of the interfaces. In other words, the implementation style may not make assumptions about the signatures and datatypes of interfaces. However, it does need to know about them in order to implement them. Similarly, the implementation style should not make assumptions about the semantics of individual instructions, but does need to know what they are.

LIS provides knowledge of interfaces and instructions to the implementation style through macros which are placed within code sections. These macros expand into function signatures, function names, data type definitions, and instruction semantics; they provide the link between the very general code placement constructs of the previous subsection and the specific details of implementing a simulator. The macros and their functionality are listed in Table II. Detailed explanations are not given, as they are highly specific to the form in which LIS specifies instructions and interfaces; the purpose of the table is to give an idea of the kinds of macros that

TABLE II: LIS code generation macros

Name	Purpose
AFTER()	Post-decoding instruction semantics
AFTERARGS()	The arguments to an instruction function
AFTERPARMS()	The parameters of an instruction function
AFTERPTR()	Looks up an instruction function
AFTERREC()	Looks up all instruction functions for an instruction
ARGS()	The arguments to an interface function
BEFORE()	Semantics which come before decoding
BUILDSET()	The name of an interface
COMMA()	Inserts commas in parameter lists if needed
DECLS()	Declares instruction fields and operands
ENTRYTEXT()	Looks up the text of an instruction function
ENTRY()	A name of an internal function used to implement an interface function
IFAFTER()	Inserts its arguments only if the interface function contains post-decoding semantics
IFNAFTER()	Inserts its arguments only if the interface function does not contain post-decoding semantics
NAME()	The name of an interface function
PARMS()	The parameters of an interface function
RTYPE	The return type of an interface function
STYLE()	The name of the implementation style
TEXT()	Quoted text of its argument
TOKEN()	A variable holding the results of decoding
TOKENSWITCH()	A switch statement with cases corresponding to instructions and their semantics

are needed. While we feel (and demonstrate) that this list of macros provides the information needed to support a very wide variety of implementation styles, from binary translation to interpreted to static-compiled, it is possible that additional macros will need to be added as new styles are invented.

The following example illustrates how these macros are used; this example comes from the interpretive simulation specification and defines how to generate an interface function. The `INTERP_EPILOGUE` code section is placed at the end of the interpretive style's generated source file. A `generator` statement is used so that an interface function is generated for each function in each interface implemented using the interpretive style. Lines 2-6 define the form of interface functions. The `%%NAME()` and `%%PARMS()` macros provide the correct function name and signature. The `%%DECLS()` macro on line 3 defines all intermediate and operand value variables that the instruction semantics within this function may need. The `%%BEFORE()` macro (line 4) expands to the semantics (code) shared by all instructions, e.g. fetch and decode. The `%%AFTERPTR()` macro (line 5) expands to a lookup of a table holding pointers to instruction functions. The lookup is indexed by the results of decoding. The `%%ARGS()` macro provides the correct arguments to the instruction function.

---

```

1  generator interp_epilogue {
2    void %%NAME()(%%PARMS()) {
3      %%DECLS()
4      %%BEFORE()
5      %%AFTERPTR()(%%ARGS());
6    }
7  }

```

---

3) *Extending interfaces and instructions*: Some implementation styles need to perform calculations based upon the results of decoding. For example, the ISCS and hybrid com-

piled simulators need to calculate which bits of the instruction encoding should be specialized. While it is possible to write a function to perform this decoding in the LIS file (possibly in the SUPPORT codesection), it is much more convenient to define this additional per-instruction calculation as auxiliary instruction semantics.

LIS uses *open* constructs – constructs which can be modified after their initial declaration. Thus the implementation style can add semantics to entire classes of instructions without requiring changes to the original instruction specification. The following example from the PowerPC ISCS implementation adds a particular bit of a branch instruction’s encoding to the specialized bit computation:<sup>3</sup>

---

```
1  action b +decloc = { specbits |= AA << 1; }
```

---

Open constructs also allow LIS implementation styles to add intermediate values which can be used for instruction execution in conjunction with new semantics. The following example from the PowerPC JIT-CCS specification declares a new intermediate value `regptrsP` which holds cached pointers to operands. It then overrides the semantics of fetching the first operand of an instruction from the register file; instead of indexing into the register file using the operand specifier in the instruction, the cached pointer is dereferenced.

---

```
1  field regptrsP CACHE_REGPTRS_ptrSP_t;
2
3  action r2_nOE_rC_l @fetchOp1Step = {
4      src1 = *(regR_t *)regptrsP->regPtr1;
5  }
```

---

Note that the constructs which were used here are not new; their equivalents are present in many ADLs. The point we wish to emphasize is that the constructs need to be open, as closed constructs would require users to modify the original instruction specification.

4) *Specialization support*: Implementation styles such as static compilation, ISCS, and binary translation generate specialized code for each static instruction. This specialization uses the values of particular bits of the instruction encodings (e.g. the condition predicate in the ARM instruction set) and/or the instruction address.

It is convenient to generate the specialized functions in a “nested” form which calls normal instruction functions. The specialized function first sets the values to be specialized to constants. It then calls the normal instruction function, which has its parameter list extended to include the specialized values. At compile time, the call is then inlined and the constant specialized values propagated.

LIS provides three new constructs which support specialization. The `entry` construct allows the signature of instruction functions to include parameters which are not part of the signature of the interface function. It also allows the instruction functions to have a different return type from that of the

<sup>3</sup>The `action` construct is used to assign semantics to an instruction or class of instructions. The first argument is the instruction name while the second argument is the name of the *step*; interfaces may choose which steps to include in interface functions.

corresponding interface function. The `specialize` construct declares which intermediate values, operands, or fields of the instruction encoding have been specialized. These specialized values are then added as parameters to the instruction functions. Typically the specialized function sets the values of these fields to constants. The `exclude` construct is a simple declaration which removes semantics from the BEFORE and AFTER macros. This construct is useful for removing semantics such as decoding which are implied by the specialization and do not need to be done at run time.

The following example from the PowerPC ISCS implementation style shows how these constructs work together. On lines 1-4, the `entry` construct adds a parameter to all instruction functions; this parameter points to the cached operand pointers. It also defines the form of the instruction functions. Lines 6-7 define the instruction encoding fields which are to be specialized. Line 9 excludes all the fetch and decode semantics from the instruction functions. Finally, lines 11-27 are the code which actually generates specialized functions. The LIS compiler places this code into the generated pre-processor function which processes instructions. That function, at pre-processing time, outputs a function which assigns constant values to a number of instruction encoding fields and then performs the instruction’s semantics.

---

```
1  entry (void) (CACHE_REGPTRS_ptrSP_t regptrsP) {
2      %%BEFORE();
3      %%AFTERPTR()(%%AFTERARGS());
4  }
5
6  specialize AA, BI, BO, CRM, d, FM, IMM, MB, ME, NB, OE,
7      Rc, Rc2, SH, SIMM, UIMM, L, DS;
8
9  exclude 0: findOpcodeStep;
10
11 generator ISCS_generators {
12     cstream
13     << "void LIS_" << templateno << '_'
14     << %%TEXT(%%NAME())
15     << "(CACHE_REGPTRS_ptrSP_t regptrsP, "
16     << %%TEXT(%%PARMS())
17     << ") {\n"
18     << %%TEXT(%%DECLS())
19     << "\tconst unsigned sbits = 0x" << std::hex
20     << specbits << std::dec << "U;\n"
21     << "\tconst unsigned AA = (sbits >> 1) & 0x1;\n"
22     << "\tconst unsigned BI = (sbits >> 16) & 0x1f;\n"
23     << "\tconst unsigned BO = (sbits >> 21) & 0x1f;\n"
24     ... // other specialized fields
25     << %%ENTRYTEXT(ii.decodetoken) << "\n"
26     << "}\n";
27 }
```

---

## IV. EVALUATION

We evaluate the effectiveness of the Orthogonal-Specification Principle by specifying simulators with three interfaces with different levels of detail and a variety of implementation styles.

### A. Instruction sets

We used LIS to describe two instruction sets: user-mode ARM v5, and user-mode 64-bit PowerPC. The ARM instruction set description is contained in 2272 lines of LIS code, while the PowerPC description uses 3987 lines of LIS code.

TABLE III: Size of style and interface descriptions

Lines of LIS code, excluding comments and blank lines	ISA	
	ARM	PowerPC
<b>Styles</b>		
Interpretive	20	19
Static compiled	233	227
Binary translation	248	249
JIT-CCS	415	443
ISCS	736	772
Hybrid compiled	530	556
New: Binary translation + JIT-CCS	618	686
New: Binary translation + JIT-CCS + ISCS	742	778
<b>Interfaces</b>		
Minimal	8	8
Decode	10	10
Steps	21	23

Operating system emulation and emulated memory are handled by C++ libraries called from the synthesized simulator. The ARM description required approximately 40 hours to develop and debug, while the PowerPC description required approximately 60 hours. Neither instruction set description required later modifications to support the implementation styles.

### B. Implementation styles

For each instruction set, we used LIS to describe six implementation styles from the literature: interpretive, static compiled, binary translation (based upon the LLVM compiler framework [26]), JIT-CCS, ISCS, and hybrid compiled. These style descriptions were developed first for ARM and then modified slightly for PowerPC. Table III shows the size of each style description. In all cases the style specification is fairly small; note that the ISCS and hybrid compiled styles share LIS code for manipulating operand pointers with JIT-CCS and this shared code (356 lines and 378 lines for ARM and PowerPC, respectively) is counted multiple times. Additional library code not written in LIS supports binary translation through LLVM.

The development time for a single style ranged from several minutes (interpretive) to several weeks (binary translation). The difference in development times stemmed primarily from the relative complexities of the implementation styles; figuring out how to harness LLVM was much more difficult than creating an interpreter. Unfortunately, exact development times cannot be given, as LIS was under development concurrently with the styles. However, developing all of the implementation styles for PowerPC required significantly less work after having done so for ARM. For example, PowerPC binary translation was begun and finished on the same day. We also found reuse of LIS code between implementation styles to be easy; the shared operand pointer manipulation code previously mentioned is a good example.

The use of an ADL with implementation style support also eased new style creation. As an example, we created two additional implementation styles which combine binary translation with JIT-CCS or ISCS. Designing and debugging these new styles required mere hours. The size of these specifications are also shown in Table III.

The first new style caches decode information for cold code until it is determined to be hot enough for binary translation. The LIS description is a straight-forward merging of the descriptions of binary translation with JIT-CCS: decode information is added to the code cache and the template for generated code adds specialization on the operand pointers.

The second new style is prompted by the additional observation that the overhead of optimizing and translating code is very high in LLVM; if the number of translations could be reduced, the simulator could be faster. The ISCS style reduces static compilation time by limiting the amount of code created to just specializations of the instruction execution functions. We merge this idea with binary translation in the second variant to only translate specializations which are then reused. We also use the JIT-CCS technique for not-yet-translated code. This LIS description is again a straight-forward merging of the first style’s description with the ISCS compilation style’s description.

### C. Interfaces

We used LIS to describe three interfaces for each simulator. The first interface (**Minimal**) is called once per dynamic instruction and provides only minimal information: whether a fault occurred and the next PC. This interface is suitable for fast-forwarding. The second interface (**Decode**) is called once per dynamic instruction and provides the minimal information plus effective addresses, branch direction and targets, instruction classification, and operand decoding information. It is also speculative: the results of instructions can be rolled back. This interface is appropriate for a speculative functional-first simulator architecture [27]. The final interface (**Steps**) is called seven times per dynamic instruction – once for each of a set of major steps of execution – and reports all the information of the **Decode** interface plus operand values. This interface would be appropriate for a timing-directed simulator architecture [19]. The size of these descriptions is also shown in Table III. We required only a few minutes to create these interfaces and did not need to modify them to support different implementation styles.

### D. Results

We measured the speed of each combination of interface and style by running the SPEC CPU2000int benchmarks using reference inputs. The simulators were run on systems equipped with two 2.8 GHz Intel Xeon X5560 processors and 24 GB of memory and were compiled using gcc 4.1.2 with flags `-g -O2`. For the static compiled, ISCS, and hybrid compiled styles, a distinct simulator was created for each benchmark. Table IV reports the average speed of each combination measured in millions of simulated instructions per host second. It also shows the standard deviation of the speeds.

For the **Minimal** interface, we see overall results in general agreement with previous work: all styles are much better than interpretive and ISCS outperforms hybrid compiled which outperforms JIT-CCS. The speed of the static compilation and binary translation styles is much lower than that reported

TABLE IV: Average simulation speed in MIPS – standard deviation shown in parenthesis

Style	ISA					
	Minimal	ARM Decode	Steps	Minimal	PowerPC Decode	Steps
Interpretive	18.03 (1.75)	12.81 (0.99)	9.34 (0.77)	19.82 (1.52)	13.06 (0.58)	10.92 (0.49)
Static compiled	34.79 (13.38)	10.39 (4.51)	5.46 (1.62)	31.18 (12.65)	12.28 (4.38)	6.18 (1.84)
Binary translation	30.18 (7.13)	19.59 (5.49)	5.03 (0.72)	30.57 (6.59)	18.60 (4.48)	5.60 (0.59)
JIT-CCS	29.10 (4.59)	21.47 (3.07)	10.73 (1.03)	34.42 (3.9)	28.12 (2.83)	12.62 (0.49)
ISCS	35.97 (9.03)	17.99 (4.12)	7.29 (1.42)	37.91 (8.78)	18.39 (3.44)	8.88 (1.39)
Hybrid compiled	35.73 (5.83)	25.79 (4.47)	11.08 (1.19)	36.19 (4.59)	30.8 (4.16)	12.06 (0.55)
New: Binary translation + JIT-CCS	30.28 (7.72)	19.91 (5.96)	4.91 (0.73)	30.73 (6.27)	19.49 (4.58)	5.59 (0.68)
New: Binary translation + JIT-CCS + ISCS	30.04 (6.33)	18.30 (4.65)	5.42 (0.55)	33.41 (5.05)	20.69 (3.71)	6.30 (0.49)

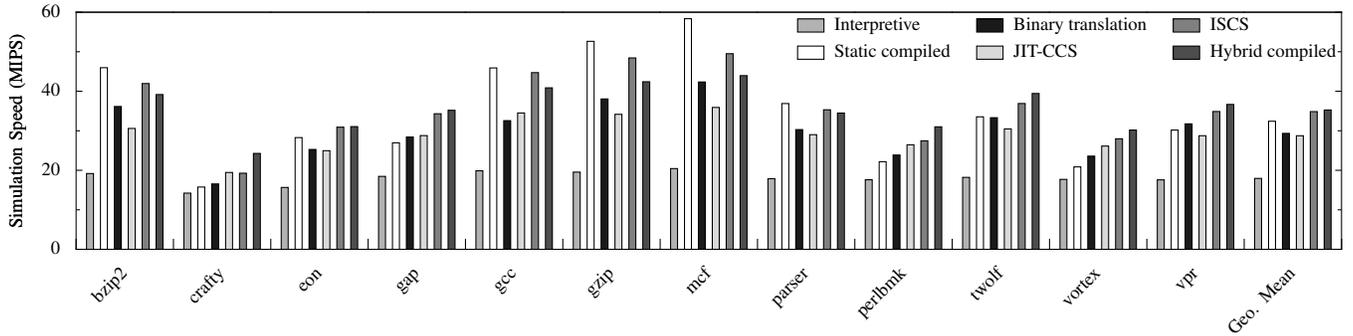


Fig. 2: Simulation speed of ARM minimal interface

previously, achieving speeds only comparable to the other implementation styles; this difference in speed stems from differences in the level of detail of the interface. In previous work, static compilation and binary translation are rarely run on an instruction-by-instruction basis; instead, one call to the simulator simulates an entire basic block, trace, or program and reports no information about the instructions. For comparison, we have generated binary translating simulators that execute one basic block per call and do not report any information<sup>4</sup>; the resulting average simulation speeds are 77.1 MIPS for ARM and 70.58 MIPS for PowerPC. These speeds are comparable to those reported in [2] for binary translation, but are still lower than some previously reported speeds because we do not use certain optimizations such as branching directly between translated functions.<sup>5</sup> The LLVM binary translator also has very high overhead and the threshold for considering code to be hot (and thus worthy of translation) has not been tuned. We do not believe that code quality is an issue; manual inspection of the statically compiled simulator binaries and dynamically generated code indicates that the code quality is high in both cases.

Figure 2 shows the individual benchmark results for the ARM simulators with the **Minimal** interface. There are rather large differences among the benchmarks; not only do the speeds vary, but the ordering of the styles changes. An interesting subject of future work is development of styles which can adapt to benchmark characteristics; the ADL features we

<sup>4</sup>These simulators required writing both a new style description and a new interface description capable of handling multiple instructions per call.

<sup>5</sup>Note also that such optimizations would not be possible at high levels of semantic detail.

have introduced will make such styles much easier to develop.

The **Decode** interface provides enough detail to be useful in a microarchitectural simulator. However, this additional detail comes at a price. First, all speeds are lower than **Minimal** due to the additional work which must be done to report decoding information. Second, the benefit of non-interpretive styles decreases because the additional work is not accelerated as much by the more sophisticated styles. Static compiled simulation is now slower than interpretive simulation and that ISCS, which also has a large static compilation component, is worse than JIT-CCS. This difference likely stems from an increase in code size as more information is required.

The **Steps** interface provides a high level of detail and control to a microarchitectural simulator. For this interface, static compilation, binary translation, and ISCS perform exceedingly poorly – worse than interpretive. This poor performance is primarily due to the simulators having a much larger instruction cache working set. This bloated working set stems from more work to report operand values, lost optimization opportunities, and overhead in function prologues and epilogues.

The new implementation styles which combine binary translation with other styles show mixed results. They usually improve slightly upon binary translation, especially as the level of detail increases, but are occasionally worse. Neither presents a compelling enough argument to suggest that they be adopted. What we wish to emphasize is that the use of an ADL supporting the Orthogonal-Specification Principle enabled us to very quickly determine that these styles are not interesting after only a few hours of work.

## V. CONCLUSIONS

We have introduced the Orthogonal-Specification Principle for functional simulators and described how ADLs can be extended to enable orthogonal specification of implementation styles. No previous ADL has provided a set of constructs which support orthogonal specification. We have shown through a case study that using an ADL with these extensions enables rapid development and application of implementation styles to multiple interfaces.

As a result of this work, architects will be able to easily specify implementation styles and apply them to create simulators with multiple, evolving interfaces – thus improving both simulator speed and development time and allowing exploration of a greater portion of the design space, leading to improved designs.

## AVAILABILITY AND ACKNOWLEDGMENTS

Source code for LIS can be downloaded as part of the Liberty Simulation Environment at <http://bardd.ee.byu.edu/>. Descriptions are provided for the PowerPC and SPARC instruction sets with interpretive simulation; the SPARC instruction set is capable of full-system simulation. Descriptions for other implementation styles are available by contacting the authors.

## REFERENCES

- [1] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003, pp. 84–97.
- [2] D. Sunwoo, J. Kim, and D. Chiou, "QUICK: A flexible full-system functional model," in *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 249–258.
- [3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005, pp. 41–46.
- [4] C. Mills, S. Ahalt, and J. Fowler, "Compiled instruction set simulation," *Software – Practice and Experience*, vol. 21, no. 8, pp. 877–889, Aug 1991.
- [5] V. Živojnović, S. Pees, C. Schlager, R. Weber, and H. Meyr, "SuperSim – A new technique for simulation of programmable DSP architectures," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1995, pp. 1748–1763.
- [6] R. Amicel and F. Bodin, "Mastering startup costs in assembler-based compiled instruction-set simulation," in *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, 2002.
- [7] M. Bartholomeu, R. Azevedo, S. Rigo, and G. Araujo, "Optimizations for compiled simulation using instruction type information," in *Proceedings of the 16th Symposium on Computer Architecture and High-Performance Computing*, 2004, pp. 74–81.
- [8] M.-K. Chung and C.-M. Kyung, "Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time," in *Proceedings of the 15th IEEE Workshop on Rapid System Prototyping (RSP '04)*, 2004, pp. 38–44.
- [9] D. Jones and N. Topham, "High speed CPU simulation using LTU dynamic binary translation," in *Proceedings of the 2009 International Conference on High-Performance Embedded Architectures and Compilers*, ser. LNCS, vol. 5409, 2009, pp. 50–64.
- [10] W. Qin, J. D'Errico, and X. Zhu, "A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation," in *Proceedings of the 3rd Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 193–198.
- [11] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Proceedings of the 1994 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 128–137.
- [12] C. May, "Mimic: A fast system/370 simulator," in *Papers of the Symposium on Interpreters and Interpretive Techniques*, 1987.
- [13] W. S. Mong and J. Zhu, "DynamoSim: A trace-based dynamically compiled instruction set simulator," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, 2004, pp. 131–136.
- [14] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," in *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996, pp. 68–79.
- [15] R. Bedichek, "Some efficient hardware simulation techniques," in *Proceedings of the USENIX Winter 1990 Technical Conference*, 1990, pp. 53–63.
- [16] A. Nohl, G. Braun, A. Hoffmann, O. Schliesbusch, R. Leupers, and H. Meyr, "A universal technique for fast and flexible instruction-set architecture simulation," in *Proceedings of the 39th ACM/IEEE Design Automation Conference*, 2002, pp. 22–27.
- [17] M. Reshadi, P. Mishra, and N. Dutt, "Instruction-set compiled simulation: A technique for fast and flexible instruction set simulation," in *Proceedings of the 40th Design Automation Conference (DAC)*, 2003, pp. 758–763.
- [18] M. Reshadi and N. Dutt, "Reducing compilation time overhead in compiled simulators," in *Proceedings of the 21st International Conference on Computer Design*, 2003, pp. 151–153.
- [19] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002, pp. 108–116.
- [20] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST methodology for high-speed SoC/computer simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 295–302.
- [21] J. D'Errico and W. Qin, "Constructing portable compiled instruction-set simulators – an ADL-driven approach," in *2006 Conference on Design, Automation and Test in Europe*, 2006, pp. 112–117.
- [22] R. Leupers, J. Elste, and B. Landwehr, "Generation of interpretive and compiled instruction set simulators," in *Proceedings of the Asian-Pacific Design Automation Conference*, 1999, pp. 339–342.
- [23] D. A. Penry, "A single-specification principle for functional-to-timing simulator interface design," in *Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software*, 2011.
- [24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241, June 1997, pp. 220–242.
- [25] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2002, pp. 271–282.
- [26] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.
- [27] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Computer Architecture Letters*, vol. 8, no. 2, July 2009.