The final version of this work can be found at: http://dx.doi.org/10.1109/ISPASS.2011.5762735

A Single-Specification Principle for Functional-to-Timing Simulator Interface Design

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

***Abstract*—Microarchitectural simulators are often partitioned into separate, but interacting, functional and timing simulators. These simulators interact through some interface whose level of detail depends upon the needs of the timing simulator. The level of detail supported by the interface profoundly affects the speed of the functional simulator, therefore, it is desirable to provide only the detail that is actually required. However, as the microarchitectural design space is explored, these needs may change, requiring corresponding time-consuming and error-prone changes to the interface. Thus simulator developers are tempted to include extra detail in the interface "just in case" it is needed later, trading off simulator speed for development time.**

We show that this tradeoff is unnecessary if a single-specification design principle is practiced: write the simulator *once* with an extremely detailed interface and then derive less-detailed interfaces from this detailed simulator. We further show that the use of an Architectural Description Language (ADL) with constructs for interface specification makes it possible to synthesize simulators with less-detailed interfaces from a highly-detailed specification with only a few lines of code and minimal effort. The speed of the resulting low-detail simulators is up to 14.4 times the speed of high-detail simulators.

I. Introduction

Microprocessor architects use simulators to evaluate new ideas, explore the design space, and validate the behavior of new microprocessors. One important class of simulators is *microarchitectural simulators* – simulators which model the microarchitecture of the microprocessor and provide cycle-accurate or near-cycle-accurate predictions of microprocessor performance. These simulators model both the timing and functionality of the microprocessor.

Microarchitectural simulators are complex pieces of software and the time required to develop them can be significant, taking up a large portion of the time available for architectural exploration [1]. They are also growing more complex over time; the advent of multicore processors implies that the simulators must be able to accurately model multiple threads of execution and operating system effects [2]. Furthermore, microarchitectural simulators are not static; they must change to reflect the points in the design space which the architects are considering. As a result, techniques which reduce simulator complexity in order to accelerate the development and modification of microarchitectural simulators are necessary; such techniques permit architects to spend less time developing simulators and more time exploring the design space.

One technique to reduce microarchitectural simulator complexity is to decouple the functional behavior of a microprocessor from the timing behavior. A decoupled microarchitectural simulator contains two separate but interacting simulators: a functional simulator which models just the functional or architecturally-visible behavior of the microprocessor and a timing simulator which models just the timing of the microprocessor [3]. This separation of functional behavior from timing behavior is advantageous because:

- The complexity of debugging the simulator is reduced. For example, it is far simpler to debug functionality errors in a purely functional simulator than in a single integrated simulator where their effects may not become apparent until millions of cycles after they occur.
- Functional validation is faster as a functional simulator is much faster than a full microarchitectural simulator.
- Techniques for accelerating functional simulators such as binary translation [4], direct execution [5], and compiled-code simulation [6] are more easily employed in a separate functional simulator.
- Stand-alone functional simulators are useful for early software development for the target microprocessor.

Timing and functional simulators must interact through some interface; this 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 which is needed 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. This change in interface requirements may occur independently of any change or lack of change in the functional behavior; even if the ISA does

not change, the timing simulator may still require interface changes.

Furthermore, a single timing simulator may require multiple levels of information or control. For example, timing simulators which support sampling [7] perform detailed simulation for only small portions of the total simulation run and "fast-forward" through the rest of the time, performing only functional simulation. During fast-forwarding, the timing simulator needs very little information from and exerts little control on the functional simulator.

The requirement to provide multiple, potentially evolving interfaces with different levels of detail leads to increased functional simulator design complexity in three ways. First, multiple interfaces can lead to code duplication. Second, a change in the level of detail may affect large portions of the simulator code. Finally, the multiple interfaces must be individually validated. This complexity may lead simulator developers to simply provide one interface at the maximum level of detail.

Unfortunately, the performance of the functional simulator depends intimately upon the level of detail of the interface. Highly detailed interfaces incur a great deal of runtime overhead to compute and store detailed functional information and to control timing of instruction behavior. For example, [8] reports a 10x performance difference in the popular Simics [9] simulator when instrumentation callbacks which yield more detailed information are enabled. This difference in performance can be very important, depending upon the microarchitecture simulator organization. For example, when sampling is used, functional simulation can be the bottleneck for simulator speed [7]. In general, if the interface is more detailed than the timing simulator needs, the microarchitectural simulator is slower than it needs to be and degrades microarchitects' ability to explore the design space.

Thus there is a tension between the need to simplify microarchitectural simulator design and the need for fast microarchitectural simulators. When the level of detail in the interface between the timing and the functional simulator has been tailored to include only the required information and control, the simulator is faster. However, both tailoring the interface and creating multiple interfaces increase the complexity of creating the functional simulator and increase the time spent writing simulators instead of exploring the design space.

The goal of this work is to obviate the choice between functional simulator speed and development time by permitting highly efficient design of timing-to-functional simulator interfaces of varying levels of detail. The key idea is to save simulator development effort by preventing the problems previously mentioned: code duplication, widespread changes to the simulator code, and validation of multiple interfaces. We do so by practicing a fundamental design principle:

Single-specification principle
Specify all the details of instructions once *and derive the desired lower levels of detail in the interface from that specification.*

We demonstrate that synthesized functional simulators with interfaces tailored precisely to the requirements of timing simulators have performance benefits of up to 14.4x. The amount of time and effort required to achieve these benefits is trivial; we demonstrate that this 14.4x performance benefit can be obtained by expending only minutes of development time writing about a dozen lines of code.

Contributions

The primary contributions of this work are:

- The single-specification principle for functional-to-timing simulator interface design.
- A description of how this principle can be practiced in general-purpose languages used for simulator development.
- A description of how Architectural Description Languages (ADLs) – domain-specific languages which provide architects with constructs to specify instruction behavior – can be extended to greatly simplify practice of the single-specification principle. No previous ADL has provided a complete set of constructs which support the single-specification principle.
- A case study with measurements of both design time and simulator speed when an ADL supporting the single-specification principle is used.

By adopting the single-specification principle and ADLs which are extended to support it, architects will be able to quickly tailor functional simulator interfaces to timing simulators, thus reducing functional simulator development time, increasing functional simulator speed, and increasing the processor design space which they explore.

The rest of this paper is organized as follows. Section II discusses common microarchitectural simulator organizations and their information and control requirements. Section III introduces related work on ADLs. Section IV introduces the single-specification principle and provides options for practicing this principle. Section V presents a case study using an ADL to practice the single-specification principle which demonstrates that interfaces with varying levels of detail and significant differences in performance can be designed with little user effort.

II. Decoupled Microarchitectural Simulator Organizations

A taxonomy of microarchitectural simulator organizations was first introduced in [3] and has since been extended. The organizations are illustrated in Figure 1. The choice of simulator organization is typically based upon performance, accuracy, and development and modification time considerations. By facilitating the design of multiple functional-to-microarchitectural simulator interfaces, we hope to reduce the importance of development and modification time in this decision.

It is helpful to classify each organization in terms of the level of detail required of the interface. We refer to the amount of information present in an interface as its *informational*

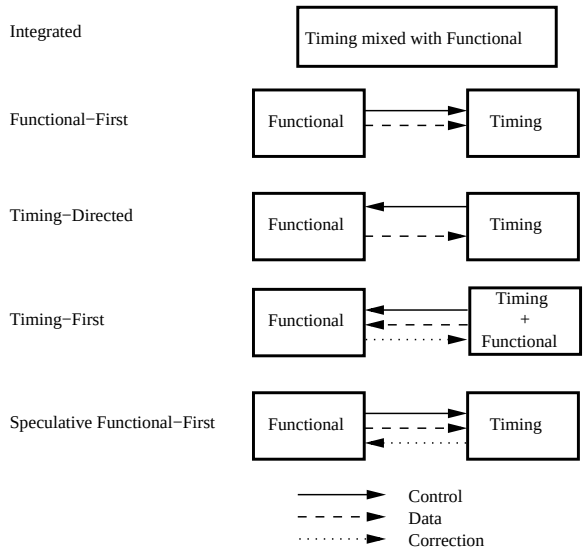

Fig. 1. Simulator organizations (modified from [3])

detail. A high informational detail interface provides more information about instruction execution to the timing simulator than a low informational detail interface. We refer to the amount of control present in the interface as its *semantic detail*. A high semantic detail interface provides the timing simulator with more control over *when* the functionality is performed than a low semantic detail interface does.

We now discuss each of the organizations.

A. Integrated simulators

The integrated organization uses only a single simulator which intermingles the functional and timing aspects of the microprocessor, and thus does not have a separate functional simulator nor an interface. These simulators compute the functional behavior of the instructions by directly modeling the datapaths of the microprocessor. As there is no separate functional simulator, the functionality must be revalidated when timing changes.

B. Functional-first simulators

The functional-first organization places the functional simulator in charge. The functional simulator executes instructions and produces a stream of information about their execution which is then consumed by the timing simulator. Trace-driven simulators are well-known [10] and use the functional-first organization. Other examples include Simplescalar [11][1] and Zesto [12].

The advantages of this organization are ease of implementation and a one-way, highly-decoupled data flow which could simplify parallelization of the simulator. The instruction stream could even be written to storage and then fed to the timing simulator or multiple timing simulators in parallel. Furthermore, fast functional simulator implementations which execute multiple instructions in a group, such as happens in binary translation, direct execution, or compiled-code simulation are easily supported.

The primary disadvantage of this organization is difficulty involved in modeling speculative execution and timing-dependent behavior such as interactions between simulated threads. A classic example of such an interaction is a spin lock; the functional behavior (i.e., which thread acquires the lock) depends upon the ordering of memory accesses, which is timing-dependent. Such interactions cannot be modeled well, particularly for relaxed memory consistency models [13] because there is no total order of memory operations in the simulated system, though many simulators just ignore the potential inaccuracies. Another disadvantage is that the timing model can be completely wrong or imply unimplementable hardware (e.g. perfect predictors) without manifesting any functional failures.

Control speculation can be supported by adding a capability to the functional simulator to undo or roll back the effects of an instruction's execution and to redirect fetch. Simplescalar [11] supports such an interface.

Interfaces for functional-first simulators usually require low semantic detail – a single functional simulator call per instruction (or even per basic block or function) – and moderate informational detail: often just decoded operand identifiers (e.g. register numbers), branch resolution information, and effective addresses. As the timing model changes, the required informational detail may change by adding or subtracting intermediate values and/or operands.

C. Timing-directed simulators

The timing-directed organization places the timing simulator in control. As instructions flow through the microarchitecture, the timing simulator asks the functional simulator to execute particular elements of each instruction's behavior (e.g. fetch, decode, operand fetch, writeback) and then uses execution information returned by the functional simulator. Examples include Asim [14] and HAsim [15]; Simics [9] also provides a timing-directed interface in addition to a functional-first interface.

The primary advantage of this organization is that speculation and thread interactions are handled very naturally in the timing simulator. The functional simulator does not need to support speculation if the timing simulator manipulates operand values directly to perform bypassing. Similarly, if the functional simulator's memory writes can be controlled by the timing simulator, many memory consistency models can be functionally simulated. Any memory model can be simulated if the timing simulator manipulates memory values. Furthermore, errors in the timing simulator involving values are likely to be manifest as functional failures; while such failures are often difficult to debug, they do allow validation of the timing simulator.

[1]While Simplescalar actually integrates the functional simulator into the code of the timing simulator, functionality is specified independently of the timing simulator (through a definition file) and thus Simplescalar conceptually has a decoupled organization. Reference [3] classifies Simplescalar as functional-first because complete instruction execution happens at a single point in the timing model.

The primary disadvantages are increased complexity in the functional simulator, potentially tight coupling between the functional and timing simulators, and the simulation speed effects of having multiple requests through the interface per instruction.

Timing-directed simulators typically require very high semantic detail, allowing the microarchitectural model to fetch and write back individual operands at different times. Furthermore, high informational detail is also needed, generally including all the information used in functional-first simulators plus individual operand values. As the timing model changes, the required informational and semantic detail is highly likely to change. Timing-directed simulators often require an additional interface for "fast-forward" during sampling; the additional interface has low semantic detail (perhaps one call to execute N instructions) and provides little, if any, information about instruction execution.

D. Timing-first simulators

The timing-first organization places the timing simulator in control, but allows it to be corrected by the functional simulator. In this organization, the timing simulator performs functional behavior which is then checked by the functional simulator; when there is a mismatch, the timing simulator's pipeline is flushed and its architectural state is reloaded from the functional simulator. Thus this organization is actually an integrated microarchitectural simulator with a functional simulator used as a checker. Examples include TFsim [3].

The primary advantage of this organization is that the timing simulator need not be totally functionally correct – corner cases and rare instructions can be ignored and bugs can be tolerated – which may reduce the simulator's development time relative to a normal integrated simulator. Furthermore, the functional simulator does not need to support speculation. The checking by a functional simulator improves debuggability of the timing simulator by providing nearly-immediate notification when an error occurs.

The primary disadvantage is that an integrated microarchitectural simulator must still be built, though it may be simpler to develop and debug than a normal integrated simulator. Furthermore, thread interactions are not correctly modeled because the functional model's memory order does not match the timing model's memory order; however, by recording the number of mismatches, it may be possible to argue that for a given simulation run with few mismatches, the discrepancy in interactions is insignificant.

The interface between the functional and timing simulators has low, one-call-per instruction semantic detail. The informational detail depends upon how the timing-first simulator is implemented. In TFsim, no per-instruction information is needed; the timing model directly queries architectural state in the functional model after requesting the execution of one instruction. However, a different timing-first simulator implementation could perform the checking based on operand values rather than state. The interface should not need to change frequently as the timing simulator changes. However, the timing simulator's functional behavior should be revalidated when the timing changes due to the integrated nature of the timing simulator. Feedback on the number of mismatches might be used to trigger revalidation only when the functional behavior of the timing model is frequently wrong.

E. Speculative functional-first simulators

The speculative functional-first organization [16] allows the functional simulator to run independently of the timing simulator, providing a stream of execution information as in the functional-first organization. The difference, however, is that all execution, not just that after a branch, is considered speculative, and when the timing simulator detects that the functional simulator's execution has differed in *any* way from the timing simulator's, e.g. by using a memory order leading to different load values, it can command the functional simulator to undo its previous behavior and continue down another path or with different memory results. Examples include UTFast [17] and FastSim [18].

The advantages of this organization are the natural parallelism of the functional and timing simulator due to the nearly one-way data flow[2] and the ability to correctly model simulated thread interactions. Furthermore, the functional simulator optimizations used in functional-first simulators can be used (e.g., UTFast uses binary translation).

The disadvantages are that the timing model needs to implement some elements of functionality – though not as many as the timing-first organization – and that the functional model must support speculation as well as a means to override the results of memory operations when recovering from misspeculation.

The interface between the functional and timing simulators has low semantic detail, with one call per instruction or basic block. The informational detail is moderate and similar to that of functional-first simulators, with the addition of memory load values or other timing-dependent values and control flow information. As the timing model changes, the required informational detail may change by adding or subtracting intermediate values and/or operands.

III. Architecture description languages and interfaces

An Architecture Description Language (ADL) is a domain-specific language which provides architects with constructs to specify instruction formats and behavior. They 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. Many ADLs have previously been proposed and only a handful can be be mentioned here; in this discussion we focus particularly on the interfaces provided to timing simulators.

Languages such as nML [19] and ISDL [20] mix descriptions of the microarchitecture with the instruction semantics

[2]This parallelism is exploited in [17] to implement the timing simulator in reconfigurable hardware.

and have no concept of decoupled functional and microarchitectural simulators.

ISP [21], LISAS [22], and SimGen [23] and an extension to EXPRESSION [24] were designed for functional simulator synthesis, providing constructs to specify instruction set decoding and semantics. However, these languages provide only a single-call-per-instruction level of semantic detail with no information.

Other languages such as MIMOLA [25], ADL [26], and RADL [27] allow descriptions of microarchitecture, but do not provide constructs identifying instruction semantics, thus behaving more like hardware description languages.

LISA [28], UPFAST [29], and ArchC [30] allow specification of instruction semantics at a high level of detail, with instructions broken up into operations or steps. These steps are then mapped onto a pipeline description to generate an integrated simulator. While they are not designed for decoupled simulator development, their separate specification of semantics could make them good candidates for extensions to support the single-specification principle.

The Facile language [31] does not support high levels of detail in the specification, but does support multiple interfaces to timing simulators: a fast interface which uses memoized microarchitectural information to perform simulation and a slow interface which performs fully detailed simulation.

D'Errico and Qin [32] describe a simple ADL which allows generation of both interpreted and compiled simulators from the same description but which does not support multiple levels of detail.

The SimpleScalar toolkit [11] defines instruction behavior through a collection of C macro calls and definitions which could be seen as a limited form of ADL. The definition file can be included multiple times with different macro definitions to provide multiple levels of informational detail. However the limitations of the C macro syntax make it difficult to provide multiple levels of semantic detail.

COTson [33] uses the SimNow functional simulator, which provides multiple levels of detail through just-in-time direct execution. The timing simulator registers callbacks on functional simulator events such as instruction execution or memory accesses. The functional simulator includes the callback notification when translating target code into its code cache. The supported levels of detail are limited by the set of callbacks provided.

IV. The single-specification principle

Developing a functional simulator is conceptually straightforward – simply implement the ISA manual. However, in practice, interfacing a functional simulator to timing simulators can be rather complex. As was seen in the previous section, there are many different levels of informational and semantic detail possible in the interface.

One way to deal with this variety of interface possibilities would be to implement each interface individually, essentially implementing different functional simulators for each timing simulator. The problems with this approach are large amounts of code duplication leading to both increased development time and chance of error and the need to completely validate each interface individually. Furthermore, if changes to the timing simulator force changes to the interface, the changes must be propagated throughout the associated functional simulator and are not likely to be local changes.

Another way to deal with the variety of interface possibilities would be to implement a single interface which is a superset of all the other interfaces. Such an interface would report all conceivable functional information about an instruction – e.g., operand values, operand decoding information, branch resolutions, effective addresses, and intermediate results. The interface would provide highly detailed control of instruction behavior – e.g., allowing individual control of the time when each source operand is read and destination operand is written. For a given simulator organization, any extra informational detail would always be present and simply be ignored by the timing simulator. Extra semantic detail would be handled by "wrapping" the highly-detailed interface calls into a single lower-detailed call – e.g., a function which fetches all operands by calling the interface calls which fetch each individual operand. Unfortunately, this one-size-fits-all approach loses all potential for performance improvement through tailoring the interface to the needs of the timing simulator. The functional simulator provided with the Liberty Simulation Environment [1] uses this one-size-fits-all approach to provide two levels of semantic detail.

A better way to deal with the variety of interface possibilities is to develop a single superset interface as in the previous paragraph, but structure the functional simulator's code so that lower-detailed interfaces can be easily derived from the superset interface. This structuring of the code should ensure that the performance gains of lower-detailed interfaces are preserved. We call such an approach the *single specification* approach and can state it as a design principle:

> **Single-specification principle**
> *Specify all the details of instructions* once *and derive the desired lower levels of detail in the interface from that specification.*

The simulator design complexity problems which multiple, evolving interfaces with different levels of detail introduce are solved by practice of the single-specification principle. There is no code duplication because instruction semantics are implemented only once in the code. When practiced as described in the rest of this section, changes to the level of detail result in localized changes to the code. Validation of multiple interfaces is simplified because changes to the interfaces are localized and do not get mixed with instruction semantics.

We now describe two means of practicing the single-specification principle. The first manually organizes the code of a simulator written in a general-purpose programming language such as C or C++. The second employs Architectural Description Languages (ADLs) and synthesizes functional simulators.

```
struct dynamic_instr {
  ...
  uint64_t src_operand1, src_operand2, src_operand3,
           dest_operand;
  uint64_t effective_addr;
  ...
}

...

void compute_effective_addr(dynamic_instr &DI) {
  if (is_a_load(DI) || is_a_store(DI))
    DI.effective_addr = src_operand1 + src_operand2;
}

void do_load(dynamic_instr &DI) {
  if (is_a_load(DI))
    DI.dest_operand = perform_load(DI.effective_addr);
}

...
```

Fig. 2. A portion of a timing-to-functional simulator interface

A. Manual implementation

A single specification of instructions can be achieved through proper structuring of the simulator code. To illustrate this structuring, first we must explain what a functional-to-timing simulator interface looks like. Figure 2 shows a portion of a typical interface. The interface contains a number of function calls which cause the functional simulator to execute some portion of an instruction's semantics, e.g. computing an effective address or loading from memory. Each of these calls is passed a pointer to a data structure representing a dynamic instruction instance. This data structure has fields which give information about the dynamic instruction, e.g. its operand values and intermediate values. Examples of intermediate values might include effective addresses, calculated branch targets, or branch directions. They may also include any additional information which might be useful to a timing simulator; for example, the shifter output for a processor implementing the ARM instruction set. The fields of the dynamic instruction structure define the informational level of detail of the interface and are filled in by the calls which implement semantics. [3]

The function calls provided in the interface define the level of semantic detail of the interface. For maximum flexibility when connecting to timing simulators, any step of executing an instruction which could take place at different times in different microarchitectures or whose result could be timing-dependent should be a separate function call. For example, effective address calculation, address translation, and memory accesses should be separate high-detail calls. Similarly, each individual operand read and write should be a separate call.

Once the set of high-detail calls are created, adding interface function calls with lower semantic detail is straightforward; simply write functions which call the higher-detail functions. For example, Figure 3 shows a function which provides the complete semantics of one instruction in one call in a hypothetical simulator; it does so by calling all the individual high-detail functions. If the compiler inlines these calls, then no speed penalty is paid for using the highly-detailed functions to avoid duplicating code.

[3]Note that this particular style of interface is presented only as an example; it is not intended to illustrate a "best" interface. In particular, it may be thought desirable in some cases to pass fields or pointers to fields directly as parameters of the functions or to place information in return values. Indeed, the "best" interface is the one tailored for the individual needs of the timing simulator which must use it. The point of this work is to enable architects to easily create such interfaces.

```
void do_in_one(dynamic_instr &DI) {
  translate_pc(DI);
  fetch_instruction(DI);
  decode_instruction(DI);
  read_src_operand1(DI);
  read_src_operand2(DI);
  read_src_operand3(DI);
  compute_effective_address(DI);
  evaluate_alu(DI);
  do_load(DI);
  writeback_dest1(DI);
  do_store(DI);
}
```

Fig. 3. Interface function executing a single instruction per call

Unfortunately, this approach does not directly allow the creation of lower levels of informational detail. The problem is that this approach requires that all information which is produced in one call and used in another call be carried somehow between the calls – in this example interface it is carried in the dynamic instruction structure – whether or not the timing simulator actually needs it.

The solution is to not use the dynamic instruction structure at the highest level of semantic detail. Instead, individual fields which *could* be in the structure should be passed by reference to the highest-detailed functions. The lower-detail function assigns either local variables or fields of the dynamic instruction structure to these reference parameters. The result looks like Figure 4. In this example, the effective address and opcode are not reported to the timing simulator, but all other information is. If the compiler, after inlining, is able to recognize that the local variables can be register-allocated, then no speed penalty is paid for using highly-detailed functions with highly-detailed information. Furthermore, the computation of information which is not actually needed semantically and not part of the interface becomes dead code which can be optimized away.

This approach has two limitations. First, while it is easy to understand, it is tedious to implement; all of the highest-detail functions must use reference parameters and all of the lower-detail (whether informational or semantic) functions must get the parameters right. Second, the approach is dependent upon the quality of the compiler, particularly upon the quality of the inliner and the compiler's ability to perform analysis on pointers to local variables.

B. ADL-based implementation

The limitations of the manual approach suggest the use of Architecture Description Languages (ADLs) to specify

```
...
void compute_effective_addr(opcode_t &opcode,
                            uint64_t &src_operand1,
                            uint64_t &src_operand2,
                            uint64_t &effective_addr) {
  if (is_a_load(opcode) || is_a_store(opcode))
    effective_addr = src_operand1 + src_operand2;
}

void do_load(opcode &opcode,
             uint64_t &effective_addr,
             uint64_t &dest_operand) {
  if (is_a_load(opcode))
    dest_operand = perform_load(effective_addr);
}

...

void do_in_one(dynamic_instr &DI) {
  uint64_t effective_addr; // local for effective address
  opcode_t opcode;         // local for opcode

  translate_pc(DI.pc, DI.phys_pc);
  fetch_instruction(DI.phys_pc, DI.instr_bits);
  decode_instruction(DI.instr_bits, opcode);
  read_src_operand1(opcode, DI.instr_bits,
                    DI.src_operand1);
  read_src_operand2(opcode, DI.instr_bits,
                    DI.src_operand2);
  read_src_operand3(opcode, DI.instr_bits,
                    DI.src_operand3);
  compute_effective_address(opcode,
                            DI.src_operand1,
                            DI.src_operand2,
                            effective_addr);
  evaluate_alu(opcode,
               DI.src_operand1, DI.src_operand2,
               DI.dest_operand);
  do_load(opcode, effective_addr, DI.dest_operand);
  writeback_dest1(opcode, DI.dest_operand);
  do_store(opcode, effective_addr, DI.src_operand3);
}
```

Fig. 4. Interface function with less informational detail

and synthesize the functional simulator.[4] We recommend the following ADL-based methodology:

1) **Create a high-detail specification of the instruction set.** Creating such a specification involves two tasks. First, declare all the possible operands and intermediate values. Second, describe the instruction set at an extremely high level of semantic detail, corresponding to the highest-detail function calls of the previous approach.
2) **Describe a timing simulator interface.** Because this interface will be used for initial debugging of the instruction set, it is convenient to use a low level of semantic detail and a high level of informational detail. One call per instruction is convenient for debugging because the resulting simulator is reasonably fast and has all the semantics of an instruction in one function. Including all intermediate values and operands in the information provided by the interface is also helpful for debugging, as values in the dynamic instruction structure do not get optimized away by compilers.
3) **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.

   We have found that errors at this level of detail are usually errors in instruction semantics, instruction encoding, or emulation of operating system calls. We envision that extensive validation of the high-detail specification will be needed only once for an instruction set, though it may be performed again if the instruction set changes.
4) **As additional interfaces are needed, describe the new interfaces, re-synthesize and revalidate.** The revalidation need not be as extensive as the original validation because the semantics are already known to be good and have not changed; the only mistakes which can be made are in the interface specifications.

   Nearly all errors at this stage occur because some intermediate value or operand that needs to be visible is hidden in the interface or because a step of instruction execution was left out. These errors are manifest early in the validation process; indeed, it is usually impossible to simulate more than a few hundred instructions before the simulation goes wrong when these errors are present.
5) **Repeat step 4 as necessary.** We emphasize that it is neither necessary nor desirable to specify or validate all interfaces *a priori* when designing a microarchitectural model. Because the best interface depends upon the timing simulator's requirements, and those requirements will change in unpredictable ways as the model is refined or additional points in the design space are explored, the interface definitions should be created, modified, and validated as they are needed.

To support such a design flow, an ADL must have constructs which allow the instruction set to be defined with high levels of semantic and informational detail. It must also allow interfaces to be specified. Finally, given the prevalence of speculation in functional-to-timing simulator interfaces, it may also be useful to provide a means to specify how to speculate.

We now discuss each ADL requirement in turn, illustrating them using an ADL named LIS which we have developed specifically to implement the single-specification principle. The simulators which are generated from this LIS description are specialized for use with the Liberty Simulation Environment (LSE) [1]. Because the implementation details of LIS are neither the focus nor a contribution of this paper, we do not provide detailed explanations of the LIS constructs. We wish to emphasize that *equivalent constructs could (and should) be added to other ADLs.*

1) High-detail instruction semantics: The first ADL requirement is a means to break the semantics of an instruction into smaller steps. This requirement has been met by several past ADLs, e.g. LISA through its `OPERATION` construct and UPFAST through its Time-Annotated Procedures (TAPs).

Within LIS, detailed instruction semantics are specified via an `action` construct which associates a snippet of C++ code with an instruction and names the snippet with an *action name*.

[4]The situation is analogous to using an object-oriented programming style in C vs. C++; you can do it in C, but the language constructs in C++ simplify the task.

The following LIS code shows a simple action for the effective address computation from Figure 4.

```
action load @compute_effective_addr = {
  effective_addr = src_operand1 + src_operand2;
}
```

2) High-detail instruction information: The second ADL requirement is a means to specify intermediate values and operands which can be communicated through the interface. These intermediate values and operands define the highest level of informational detail supported by the ISA description. This requirement has been met in past ADLs; e.g. LISA through its `GROUP` and `PIPELINE_REGISTER` constructs, UPFAST through its `controldata` construct, and Facile through its `val` construct.

In LIS, intermediate values are called *fields*. The field definition construct gives the name of the field and its type. The following LIS code defines the effective address field in Figure 2.

```
field effective_addr uint64_t;
```

In LIS, operands are defined using an `operand` construct. This construct adds an operand to an instruction. The following code defines the operands used in Figure 2. The functional call in the construct refers to another LIS construct called an *accessor* (not shown) which describes how operands should be decoded, read from architectural state, and written to architectural state. The arguments are bitfields from the instruction encoding. The operand names are defined in a separate `operandname` construct which provides action names for operand decode, fetch, and write.

```
operand load src_operand1 R(Rsrc1);
operand load src_operand2 R(Rsrc2);
operand load dest_operand R(Rdest1);

operandname src 1  (decode_instruction, read_src_operand1)
            = src_operand1;
operandname src 2  (decode_instruction, read_src_operand2)
            = src_operand2;
operandname dest 1 (decode_instruction, writeback_dest1)
            = dest_operand1;
```

3) Interfaces: The third ADL requirement is a means to specify interfaces. Each interface specification must define the levels of both informational and semantic detail for the interface. No previous ADL has provided constructs which allow the specification of multiple interfaces with different levels of detail. This lack of support exists because previous ADLs which allowed high-detail specifications of semantics have generated integrated simulators.

In LIS, interfaces are defined using the `buildset` construct. We do not enforce a pre-determined set of buildset definitions or levels of detail. Informational detail is controlled via a `visibility` construct which lists fields and operands to make visible. Semantic detail is controlled via an `entrypoint` construct which lists the actions which are to be called within the entrypoint. The following LIS code defines a buildset corresponding to Figure 4.

```
buildset example ALL {
  visibility hide effective_addr, opcode;

  entrypoint void do_in_one() =
    translate_pc, fetch_instruction, decode_instruction
          { opcode }
    read_src_operand1, read_src_operand2, read_src_operand3,
    compute_effective_address, evaluate_alu,
    do_load, writeback_dest1, do_store;
}
```

4) Speculation: Support for speculation is an optional ADL requirement, but can be very useful for creating functional simulators supporting the functional-first and speculative functional-first organizations. No previous ADL has documented support for speculation, though it is not hard to imagine that speculative simulators could be generated from many ADLs, particularly those which directly declare destination operands.

In LIS, speculation is handled with a very simple approach: the instruction information structure carries enough information to roll back the architectural effects of each instruction. Methods to store and use this information are part of the operand accessors; default methods using the read and write accessors are generated if specialized ones are not given. Speculation can be enabled on a per-buildset basis. An interesting area for future work is ADL constructs which support the description of speculation, allowing easier use of approaches such as that of [34] or [35].

V. Evaluation

We evaluate the effectiveness of the single-specification principle by specifying functional simulators with a variety of interfaces with differing levels of detail. We provide insight into the difficulty of doing so by comparing the amount of time and lines of code required to specify an instruction set and to specify a new interface. We measure the speed of the resulting simulators and analyze the cost of semantic and informational detail.

A. Instruction sets

We used three instruction sets for the evaluation: Alpha, ARM v5, and PowerPC. Only user mode instructions were included; the ARM description also did not include floating point instructions. Operating system calls were emulated.

Several LIS description files were prepared for each instruction set. One file defines the instructions themselves. Another file describes how the functional simulator calls an operating system emulator. These calls are made by overriding the semantics of the instruction conventionally used to enter into the operating system. A handful of other files describe the buildsets to be generated. Table I shows the sizes of the instruction set descriptions and approximate number of instructions in each instruction set. The overall size of the specification is affected by many language elements beyond those described in this paper, such as sharing of behavior within instruction classes.

The descriptions were created by an experienced simulator developer already familiar with the instruction sets, LIS, and

TABLE I
INSTRUCTION SET CHARACTERISTICS

	ISA		
	Alpha	ARM	PowerPC
Lines of LIS code (excl. comments and blank lines)			
ISA description	1656	2047	3805
OS/simulator support	317	225	182
Binary translator support	1104	1107	1105
Buildsets	308	308	327
Lines per experimental buildset	13	13	14
Approximate number of instructions	200	40	240
Development/debug time (man-hours)	abt. 40	abt. 40	abt. 60

functional simulation. The man-hours required – 40 to 60 per instruction set – is a reasonable amount of effort to obtain a working user-mode functional simulator. Less-skilled developers will require longer – development of a SPARC V9 LIS description required 286 man-hours using students unfamiliar with either SPARC or functional simulation.

The lines per buildset statistic in Table I represents the amount of work necessary to add a new interface; about a dozen lines of code per interface on average. These few lines can be created in mere minutes; as the buildset description in Section IV-B3 demonstrated, all that is needed is a list of the information to be hidden or shown and a list of the semantic steps to go into each interface call.

B. Interface levels of detail

To measure the effects of detail on simulator speed, we described a total of twelve different interfaces having a variety of levels of semantic and informational detail; some also had speculation support. These interfaces are intended as examples of typical interfaces; we emphasize again that the best interface for a particular timing simulator is one specialized for that timing simulator.

Three levels of semantic detail were investigated:

- **Block:** Each interface call executes a basic block.
- **One:** Each interface call executes a single instruction.
- **Step:** Seven interface calls (for fetch, decode, operand fetch, evaluate, memory, writeback, and exception) together execute a single instruction.

Three levels of informational detail were investigated:

- **Min:** Only minimal information needed to control the simulator (address, instruction encoding, next PC, faults, and simulator context) is made visible in the interface.
- **Decode:** Minimal information plus decode information and effective addresses are visible. This detail may be appropriate for many functional-first simulators.
- **All:** All fields and operand values are visible.

Speculation support was added to some interfaces; the presence or absence of speculation support is labeled as **Yes** or **No**, respectively.

C. Synthesis

We generated functional simulators for each instruction set. Each simulator contained all of the interfaces described above; the interface to use was selected via command-line option. The synthesis process specialized the code for each interface by directly inserting instruction semantics into each interface function (thus removing the need for aggressive inlining in the compiler) and by declaring hidden fields as local variables rather than references to the instruction information structure. The generated simulators used binary translation techniques based upon the LLVM compiler framework [36] to accelerate simulation speed.

D. Validation

The descriptions were validated by testing the generated simulators using suites of benchmark programs. The CPU 2000 integer benchmarks were used for all the instruction sets; the CPU 2000 floating point benchmarks were additionally used for Alpha and PowerPC. MediaBench was additionally used for ARM and PowerPC.

During the development and debugging of the descriptions, an interface with low semantic detail and high informational detail without speculation was used as suggested in section IV-B; this interface would be labeled as **One/All/No**. This interface was easy to debug because all of the instruction information is stored and each instruction has a single function. After all benchmarks worked for this interface, the other interfaces were written and debugged in a few moments with a short benchmark. We observed that the typical specification error in an interface is leaving out some instruction step or leaving a field hidden which must be communicated between steps. Such errors tend to manifest themselves rapidly as unexpected behavior in *any* benchmark.

We then validated all the interfaces by running all the benchmarks, calling the interfaces on a rotating basis; each dynamic instruction or basic block used a different interface than the previous one. This procedure ensured the validity of all of the interfaces without requiring a complete validation run per interface. No additional errors were found during the interface validation runs.

E. Results

Table II shows the simulation speed in instructions per second for each interface. The reported speed is the geometric mean of the speed measured over the first 4 billion instructions of six of the twelve SPEC CPU2000int benchmarks. The simulators were run on a system equipped with two 2 GHz dual-core Opteron 2212s and 4 GByte of memory and were compiled using gcc 4.1.2 with flags `-g -O2`.

Table III shows the cost of detail in terms of the number of host instructions per simulated instruction. These numbers include the amortized cost of binary translation. The baseline is the **One/Min/No** case; all other costs are incremental (i.e. they should be added to the base cost to determine the actual cost of a level of detail).

Two trends are seen in these results. First, as either the informational or the semantic detail becomes higher, the speed of simulation decreases as expected. Second, when speculation support is included, the speed of simulation decreases, again

TABLE II
SIMULATION SPEED (MIPS)

Detail			Instruction Set		
Semantic	Informational	Spec.	Alpha	ARM	PowerPC
Block	Min	No	37.8	26.8	19.3
Block	Decode	No	11.4	12.3	8.10
Block	Decode	Yes	9.85	10.0	6.90
Block	All	No	9.26	7.60	7.08
Block	All	Yes	8.29	6.60	6.25
One	Min	No	16.0	14.00	11.2
One	Decode	No	8.50	9.22	6.45
One	Decode	Yes	7.66	7.94	5.78
One	All	No	7.47	6.19	5.61
One	All	Yes	6.92	5.53	5.15
Step	All	No	2.79	2.54	2.34
Step	All	Yes	2.62	2.35	2.20

TABLE III
COSTS OF DETAIL (HOST INSTRUCTIONS)

	Alpha	ARM	PowerPC
Base cost for instruction	103.98	134.95	143.61
Incremental cost of decode information	46.17	53.77	63.10
Incremental cost of full information	150.51	268.48	221.5
Incremental cost of block-call	-52.28	-49.73	-49.87
Incremental cost of multiple calls	237.7	222.7	213.1
Incremental cost of speculation	14.75	32.66	27.32

as expected. The lowest-detailed interface is up to 14.4 times faster than the highest-detailed interface. These results are roughly in agreement with those of [8], which reports a 10x difference in Simics.

Differences in semantic detail provide the largest performance differences. These performance differences are driven primarily by the scope of optimizations which the binary translator can perform. At the **block** level of detail, optimizations can be performed across several simulated instructions. For example, if a simulated register value is generated in one simulated instruction and used in a later instruction, the binary translator may register-allocate the value. Such an optimization is not possible at higher levels of semantic detail. The **step** level of detail naturally has the smallest scope for optimizations.

Differences in informational detail are less important, but still lead to approximately a 4x difference in performance. This performance difference is primarily due to the amount of work to be done per instruction – there are many additional stores which must be performed to record information – but may also be due in part to poorer optimization in the binary translator when there are so many stores.

Speculation support is the least important element of performance, though it can affect it by 20%.

The base cost of instruction execution is measured using the **One/Min/No** interface. This cost is quite high, despite the use of binary translation.[5] The binary translator we use often generates very good code, but is somewhat slow. Better tuning of the translator, especially the choice of hot code to translate, would reduce this base cost. Note that as the base cost goes down, the relative importance of both semantic and informational detail will increase.

The causes of the differences in performance among the instruction sets are not completely clear, but appear to stem from differences in the quality of the code generated by the binary translator. Investigation of the generated code shows that the binary translator does a much better job of removing redundant loads for the Alpha instruction set. We believe that small variations in the coding style used in the instruction semantics cause the alias analysis of translated Alpha instructions to be more precise than that of ARM and PowerPC instructions. Further work to make binary translation more robust is needed. It may also be of benefit to explore *not* using binary translation when high levels of detail are required.

[5]The base cost is still better than interpreted simulation; similar measurements using an interpreted rather than binary-translated style of execution give a base cost of 205.5 host instructions for the Alpha instruction set.

VI. CONCLUSIONS

Decoupled microarchitecture simulator organizations reduce simulator complexity but require interfaces between functional and timing simulators. The level of detail of these interfaces is determined by the needs of the timing simulator but strongly affects the speed of the functional simulator, thus it is desirable to include only the detail which is needed.

In this paper we have demonstrated that functional-to-timing simulator interfaces with levels of detail tailored to the needs of a timing simulator can be created by practicing the single-specification principle: specify all the details of instructions once and derive the desired lower levels of detail in the interface from that specification. We have further demonstrated that the use of an ADL which provides constructs to support this principle can reduce the time to develop a new interface to mere minutes. These few minutes of effort can result in performance differences of up to 14.4x.

By adopting the single-specification principle and ADLs which practice it, architects will be able to quickly tailor functional simulator interfaces to timing simulators, thus reducing functional simulator development time, increasing functional simulator speed, and increasing the processor design space which they explore.

VII. 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; the SPARC instruction set is capable of full-system simulation.

We thank the anonymous reviewers for their helpful comments. This work was supported by National Science Foundation grant CCF-1017004.

REFERENCES

[1] 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.

[2] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese, "Impact of chip-level integration on performance of OLTP workloads," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000.

[3] 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.

[4] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *Proceedings of the 1988 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1988, pp. 4 – 11.

[5] M. Durbhakula, V. S. Pai, and S. Adve, "Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors," in *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999, pp. 23–32.

[6] S. Pees, V. Živojnović, A. Ropers, and H. Meyr, "Fast simulation of the TI TMS 320C54x DSP," in *Proceedings of the International Conference on Signal Processing Application and Technology (IC-SPAT)*, Sep. 1997.

[7] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–97.

[8] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '08)*, 2008, pp. 77–86.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[10] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys*, vol. 29, no. 2, pp. 128–170, 1997.

[11] D. Burger and T. M. Austin, "The SimpleScalar tool set version 2.0," Department of Computer Science, University of Wisconsin-Madison, Tech. Rep. 97-1342, June 1997.

[12] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 53–64.

[13] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th International Symposium on Computer Architecture*, June 1990, pp. 15–26.

[14] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: A performance model framework," *IEEE Computer*, vol. 0018-9162, pp. 68–76, February 2002.

[15] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2008, pp. 1–10.

[16] 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.

[17] 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.

[18] E. C. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using memoization," in *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998, pp. 283–294.

[19] A. Fauth, J. V. Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proceedings of the Conference on Design, Automation and Test in Europe*, March 1995, pp. 503–507.

[20] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proceedings of the 34th ACM/IEEE Design Automation Conference*, June 1997, pp. 299–302.

[21] M. R. Barbacci, "Instruction set processor specifications for simulation, evaluation, and synthesis," in *Proceedings of the 16th ACM/IEEE Design Automation Conference*, 1979, pp. 64–72.

[22] T. A. Cook and E. Harcourt, "A functional specification language for instruction set architectures," in *Proceedings of the 1994 IEEE Computer Society International Conference on Computer Languages*, 1994, pp. 11–19.

[23] F. Larsson, P. Magnusson, and B. Werner, "SimGen: Development of efficient instruction set simulators," Swedish Institute of Computer Science, Tech. Rep. R97:03, November 1997.

[24] M. Reshadi, N. Dutt, and P. Mishra, "A retargetable framework for instruction-set architecture simulation," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 431–452, May 2006.

[25] G. Zimmerman, "The MIMOLA design system: A computer aided processor design method," in *Proceedings of the 16th Annual Design Automation Conference*, 1979, pp. 53–58.

[26] E. S. T. Fernandez, "Microarchitecture modeling through ADL," in *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, 1988, pp. 100–104.

[27] C. Siska, "A processor description language supporting retargetable multi-pipeline DSP program development tools," in *Proceedings of the 11th International Symposium on System Synthesis*, Dec. 1998, pp. 31–36.

[28] S. Pees, A. Hoffman, and H. Meyr, "Retargetable compiled simulation of embedded processors using a machine description language," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 4, pp. 815–845, October 2000.

[29] S. Önder and R. Gupta, "Automatic generation of microarchitecture simulators," in *Proceedings of the 1998 International Conference on Computer Languages*, 1998, pp. 80–89.

[30] *The ArchC Architecture Description Language Reference Manual*, Computer Systems Laboratory (LSC) – Institute of Computing, University of Campinas, 2004. [Online]. Available: http://www.archc.org

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

[32] 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.

[33] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTson: Infrastructure for full system simulation," *Operating Systems Review*, vol. 43, no. 1, January 2009.

[34] H. Wang, S. Manor, D. LaFollette, N. Nesher, K. jei King, P. Wang, S. Levy, S. Satt, G. Carmeli, A. Kapur, I. Schoinas, E. Rubinstein, and R. Bhatt, "Inferno: a functional simulation infrastructure for modeling microarchitectural data speculations," in *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, 2003, pp. 11–21.

[35] 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.

[36] 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.