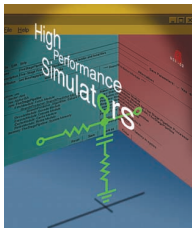
Asim: A Performance Model Framework



To cope with the complexity of modern high-end microprocessors and systems as well as the intricacy of the software that models these machines, the authors developed a modular and reusable performance model framework.

Joel Emer
Pritpal Ahuja
Eric Borch
Artur Klauser
Chi-Keung Luk
Srilatha Manne
Shubhendu S. Mukherjee
Harish Patil
Steven Wallace
Intel

Nathan Binkert
University of Michigan, Ann Arbor

Roger Espasa
Toni Juan
Universitat Politècnica de Catalunya

The Compaq (formerly Digital) processor design teams that created the VAX and Alpha processors have a long history of developing models to predict the performance of proposed processor designs. Such predictions have become both more daunting and more critical as processor and system complexity has increased.

Microprocessors now consist of several hundred million transistors. The interaction between the hardware structures built from these millions of transistors is hard to predict through simple simulation or analytical models. Having such a large number of transistors has also made modern microprocessors extremely complex, substantially extending the design cycle of new microarchitectures. Furthermore, because a design's architectural aspects are increasingly affected by signal propagation delays, modeling timing delays systematically has become more important. In this context, performance models play a crucial role because developers use them to study design alternatives and predict the performance of processors and systems long before actually building them.

These factors have resulted in a substantial increase in the complexity of the performance models themselves. In the past, the longevity and usefulness of a model depended mostly on the skills and discipline of the model writers. Unfortunately, at Compaq, our models were still becoming extremely complex and unmanageable because we lacked a structured way to develop them. To cope with these complexities, in late 1998 we began developing Asim to allow model writers to faithfully represent the detailed timing of complex modern machines and effectively manage the large software projects needed to model such machines.

Asim addresses these needs by providing a framework for creating many models, instead of being a single performance model. More specifically, Asim achieves these goals through modularity and reusability. Modularity helps break down the performance-modeling problem into individual pieces that can be modeled separately, while reusability allows using a software component repeatedly in different contexts. Reusability increases productivity and confidence in the robustness of the software component itself. Asim provides a set of tools that can effectively manage these software components to help model writers deal with a large software base's complexity.

BASIC COMPONENTS

In Asim, the basic software component, or *module,* will usually represent a physical component of a design, such as a cache, or capture a hardware algorithm's operation, such as the cache's replacement policy. A particular model will be represented as a user-selected hierarchy of modules.

Each Asim module provides a well-defined interface that lets developers reuse modules in different contexts or replace them with other modules that implement a different algorithm for the same function. Asim specifies module interfaces in terms of a set of method calls or ports. Method call interfaces provide communciation between a module and sub-

0018-9162/02/$17.00 © 2002 IEEE

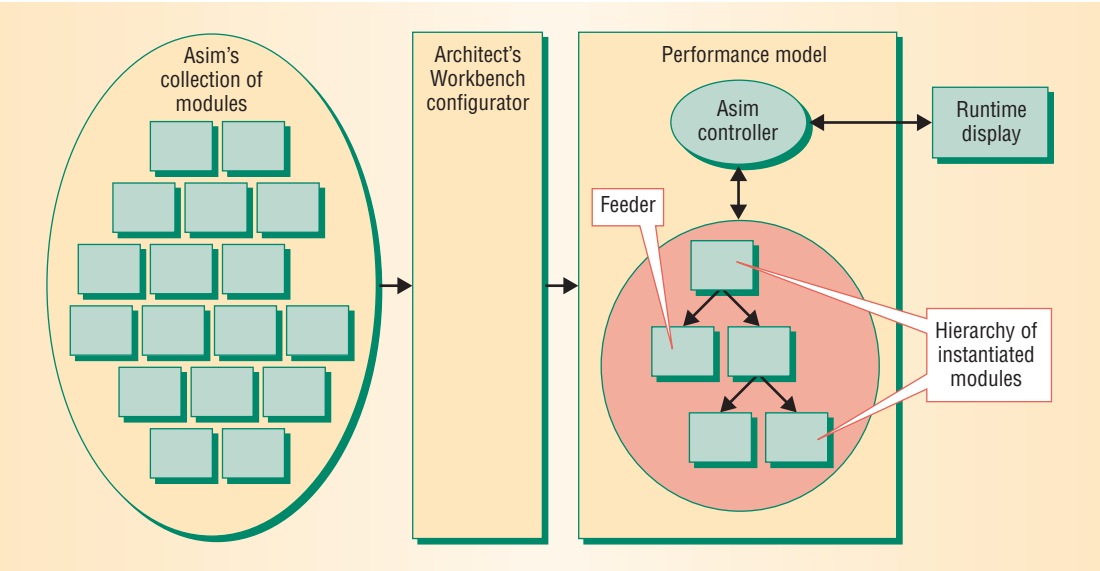


Figure 1. Asim framework. The configurator instantiates modules to create a performance model, which it builds using a standard makefile and C++ compiler. The model consists of a hierarchy of instantiated modules and an Asim controller that controls the module hierarchy's execution.

modules embedded in it. Ports provide a communication abstraction that explicitly allows a model to represent the communication channels and the timing characteristics between modules.

Asim also allows using modules that represent nonhardware-based infrastructure. Interfaces to these nonhardware modules also use the method-call interface. One such nonhardware module, the *feeder*, provides the inputs necessary to drive the performance models. In particular, *instruction feeders* supply instructions to the performance models.

Asim programmers develop models independent of feeders. In this approach, the instruction feeders manage the functional aspects of a program's execution while the performance models manage timing predictions, including the timing of incorrect program paths that arise due to improper speculation.

Figure 1 shows an overview of Asim. Asim also provides a collection of models and benchmarks not shown in the figure. Developers can use the Architect's Workbench to configure a performance model by selecting a specific set of modules, browse existing models and benchmarks, and run a performance model with a specific benchmark. Asim also provides a runtime cycle display, which lets developers visualize the activity in processor pipelines on a cycle-by-cycle basis.

Asim easily supports both timing and stand-alone performance models. Timing models—complete processor or system models—predict the time it takes to run part of a program or all of it. In contrast, with stand-alone models, we can efficiently study the behavior of individual hardware components in isolation. For example, we would use a stand-alone model to study a branch predictor's accuracy. However, to understand a branch predictor's timing behavior, we would immerse the same module in a complete processor model.

Using this framework, we developed several performance models, including models for uniprocessors, vector processors, and chip multiprocessors as well as for multithreaded, fault-tolerant, power-efficient processors and tightly coupled network architectures. We have also used Asim to study organizations for several processor subcomponents, including line and branch predictors and memory components. In our experience, Asim is an invaluable framework for modeling, maintaining, extending, and managing processor and system architecture models.

MODULES

Effective processor and system performance modeling requires a performance-modeling environment that copes with the complexity of modern machines and the software that models them. A performance-modeling environment must provide a language to express hardware structures, libraries to provide functionality common to all models, and a tool set to build these hardware structures into models. Most modern languages can express the hardware descriptions that performance models use. Asim, for example, is primarily written in C++. Although it is possible to augment such languages to help model writers, we focus instead on helping a model writer express and expose reusable alternate designs.

Key to achieving these goals is providing the user with a convenient means to select alternatives and use well-defined interfaces to specify how to communicate with other software components. Earlier performance models made it difficult to provide alternative designs for individual hardware components, such as branch predictors, or to reuse a specific component's code in different models. For example, conditional compilation was popular for providing model alternatives at compile time. This method, usually invoked with the *#ifdef* and *#endif* statements processed by the C preprocessor, selects a code variant at compile time, which can lead to

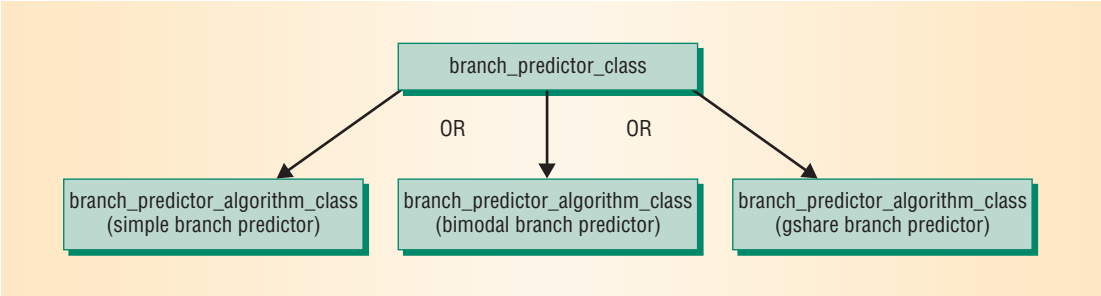


Figure 2. Asim branch predictor hierarchy. The branch_predictor_class selects either the simple, bimodal, or gshare module as its branch predictor algorithm.

more efficient execution. Unfortunately, because conditional compilation lacks clearly specified code boundaries, it does not lead to a modular design nor does it naturally specify a well-defined interface for providing or extracting a specific component for reuse.

Expressing alternate designs

In Asim, a module represents a software component that developers can use in different contexts to either encapsulate the behavior of a hardware structure or represent a software component that needs to be modularized. A user explicitly selects such modules when he or she creates a model. Internally, however, we represent such modules as C++ classes.

For example, in Figure 2, the branch_predictor_class module captures the high-level description of a generic branch predictor. In C++ style, developers will derive the branch predictor class from a more generic asim_module_class. Using the selected C++ class gives the developers the flexibility to instantiate the module as needed. Thus, for example, a module representing a CPU might be instantiated multiple times to create a multiprocessor. Asim requires modules to have well-defined interfaces so that they can either be reused in multiple contexts or replaced by other modules that encode a different algorithm for the same function. Thus, the gshare algorithm can easily replace the simple or bimodal branch predictor algorithms shown in Figure 2.

To satisfy the different ways developers use modules, Asim provides two interface styles: ports and method-call. In the *port* interface style, modules that represent hardware boxes use ports to communicate with each other over cycle boundaries. In the *method-call* interface style, two modules define a set of functions that they use to communicate with each other. Typically, submodules embedded in parent modules with only intracycle communication and non-hardware-based components use such an interface.

For example, developers can implement a branch predictor by coding the appropriate module type. As Figure 2 shows, the module branch_predictor_class can implement a branch predictor, perhaps by interfacing to a branch_predictor_algorithm_ class module. Any compatible branch predictor in Asim must conform to this interface style. In this case, the interface consists of three methods:

- *GetPrediction*, which obtains a branch prediction from a predictor;
- *UpdateBranchPredictor*, which updates the branch predictor after the branch resolves; and
- *HandleMispredictedBranch*, which handles mispredicted branches.

Thus, developers can implement multiple branch predictors, such as *simple*, *bimodal*, or *gshare* by implementing these three methods for each branch predictor.

Managing design choices

Managing all the design choices expressed by an Asim programmer can be daunting. A processor or a complete system can have hundreds of modules, and each module can have several design alternatives. A performance modeling environment must provide a framework to easily manage and expose these choices. Dependence among modules further complicates management of these design choices. For example, to the branch_predictor_class shown in Figure 2, the developer must define an associated branch predictor algorithm.

Asim manages these dependences using two mechanisms. First, each module has an Asim module type, which defines what capabilities it provides and implicitly defines its interface. It also specifies which modules and types it requires. Thus, the branch predictor shown in Figure 3 requires a BranchPredictor_Algorithm. Each of the simple, bimodal, and gshare predictors, in turn, provides a BranchPredictor_Algorithm. Thus, the requires-provides interface lets Asim correctly pick dependent modules.

Second, Asim maintains the information about each module in a separate .awb file. If developers define the gshare branch predictor in two files—gshare_branch_predictor.h and gshare_branch_predictor.cpp—this information, the Asim module type, and any necessary dependence information will typically be in ghsare_branch_predictor.awb. The Asim framework processes these .awb files and exposes the dependences among modules. In the .awb file, each module also can define one or more corresponding attributes. Attributes help find mod-

```
/*************************************************************************
* Awb definitions
*
* %AWB_START
*
* %name FiveStagePipeline: BranchPredictor
* %desc Branch Predictor for FiveStagePipeline
* %attributes FiveStagePipeline
* %provides BranchPredictor
* %requires BranchPredictor_Algorithm
* %private branchpredictor.h
* %param BP_PROBE_LATENCY 1 "branch prediction latency in cycles"
*
* %AWB_END
*************************************************************************/
```

Figure 3. Sample .awb file associated with the branch predictor module. %AWB_START and %AWB_END demarcate the start and end of AWB commands, while %public (not included in this figure) and %private show the files associated with this module. All commands except %name and %desc can be specified multiple times.

ules associated with a particular complete design or indicate compatible collections of modules.

Advantages

Modules express alternate design choices for a hardware structure by identifying a module as a member of an Asim module class that characterizes the interface to that component. The code that implements that model is isolated into separate files. Developers can reuse modules in a full-fledged processor pipeline or in a simple, stand-alone model that reads an instruction stream and makes requests to the modules. This flexible interchange of code between simple and complex models saves coding time and lets developers study the interactions between modules.

Additionally, having multiple modules that implement a single processor component facilitates experimenting with design alternatives. We have conducted several successful runs of full factorial experiments with modules. This mix-and-match capability also makes the complexity of testing a performance model painfully obvious. We typically test a module in only a small number of contexts, not in all contexts where it might appear. With Asim, the increased reuse of the module's code in different contexts, and the consequent maturity of the module's code, give us greater confidence in the accuracy of the individual modules.

The more detailed a model becomes, the slower it runs. However, performance model studies often do not need the detailed accuracy of certain hardware structures. In such cases, developers can replace a module with a simpler and less detailed module to speed up the performance model's execution.

PORTS

Current trends in semiconductor scaling indicate that wire delay will play a critical role in processor performance. Thus, time and, more specifically, delays between and in hardware structures must become first-class abstractions in the processor design representation. For this reason, and to keep the interfaces between modules cleaner and better defined, we created Asim's ports communication paradigm.

Ports have several functions in Asim. First, modules that represent hardware structures exchange information through ports. A real processor or system typically does not have any global information instantaneously accessible anywhere in the chip. Instead, a processor's state is distributed among several hardware components that might take several cycles to communicate. Components that need to access the state of other components must do so through explicit communication. Asim's ports represent communication channels that let modules representing hardware components communicate such state information with one another.

To pass information from one module to another, the sending module must write the relevant information to the port. The receiving module on the other side of the port's connection reads the information from the port. For example, in the microprocessor pipeline shown in Figure 4, the module that executes instructions can send the correct branch outcome to the branch predictor through a port that connects the execution and branch predictor modules.

Asim uses ports to model communication delays between modules. Asim models an intrinsically synchronous clocked system that clocks every module once per cycle. In terms of actual code, each module has a *Clock method*, which a scheduler calls at the simulated time during which the module must be active. Within the Clock method, a module defines all the logical activity it needs to do in that cycle such as sending information to and receiving it from other modules through ports.

Like real hardware, ports have a fixed latency and a maximum bandwidth. The information that a module sends through a port does not appear in the receiving module before the port's fixed latency elapses. Similarly, a module cannot send more

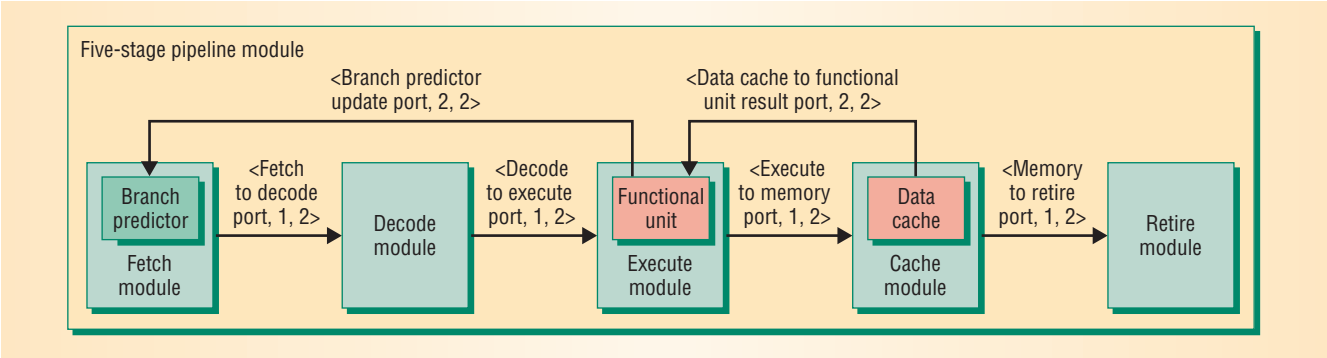


Figure 4. Sample modules and ports for a simple five-stage processor model in Asim. Ports are represented as X, Y, Z, where X is the port name, Y is the port's latency, and Z is the port's bandwidth expressed as data items sent per cycle. Thus, for example, the fetch module can send two instructions to the decode module.

information than the port's bandwidth allows. Thus, Asim can accurately model wire delays and bandwidth between modules.

Asim also uses ports to model delays within a hardware structure. In Figure 4, the port delay from the data cache to the function units includes a delay of one cycle to account for the time it takes to access the data cache. Asim allows such time representations because it creates a clear separation between a hardware algorithm, independent of time, and the timing component itself.

Ports define interfaces between modules to facilitate coding of reusable modules. A well-defined interface disallows implicit side effects, which might hinder the fidelity, reusability, and portability of model code. Traditionally, performance models use method calls and global variables to pass information between hardware structures. Unfortunately, these techniques permit an unrestricted information flow that allows unexpected side effects to occur, making reuse more difficult.

Having a well-defined port interface also forces developers to explicitly specify modeling delays between modules. This feature is critical because programmers often tend to approximate time. To obtain branch predictions in Asim, a module defines two ports to the branch predictor: one that requests a prediction and one that returns it. Each port has the requisite latency. The module requesting the predictions will not even see them until they are truly available.

Asim implements ports through first-in, first-out queues. Each module declares one end of the port. The sending module declares an output port with an identifier string, and the receiving module declares the corresponding input port with the same identifier string. Asim's utility code handles the actual connection between output and input ports. The utility code runs at initialization and connects output and input ports using the identifier strings. This technique lets submodules from one module connect to submodules in other modules without the parent modules' being aware of the connection. The automatic connection also lets developers connect replicated modules and their corresponding ports automatically, which can be useful for multiprocessors that might instantiate several processors connected to a network through ports.

FEEDERS

For improved efficiency, performance models traditionally focus on modeling only those machine operations that affect performance. In particular, developers often can use instruction traces or abstract architecture emulations to avoid details of an architecture's execution. Asim supports this capability by using specially designated modules called *feeders*, which supply inputs to drive a performance model.

Because a program or group of programs eventually drives a real computer system, the key feeder for a performance model must supply instructions that make up a program. To study hardware components in isolation using stand-alone models, we might use several feeders. For example, to drive a network model, we might need a feeder that supplies network packets either from a prior trace or from an analytical model. Alternatively, to study a branch predictor in isolation, we need only a stream of branch instructions or a subset of a program's instructions. Asim recognizes the need for such feeders and, to create sets of interchangeable feeders, uses its module abstraction to specify standard feeder interfaces.

Instruction feeders

Currently, Asim supports three instruction feeders. The *static trace feeder* reads instruction traces, interprets them, and supplies them to the performance model. Either a real machine or a different performance model could generate this kind of trace. The static trace feeder's simplicity, inherent repeatability, and corresponding execution speed make it appealing. However, because the trace only has the correctly executed path, it does not supply instructions from incorrectly executed paths.

The *dynamic trace feeder* runs an instruction emulator to generate a trace on the fly and supply it to the performance model. Compared to the static trace feeder, the dynamic trace feeder saves disk space because Asim does not have to store the traces, which could be several gigabytes. Asim uses the dynamic trace feeder to read instructions from the SimOS system model,[1] which simulates full computer systems running a variety of applications, including databases. The dynamic trace feeder also has trouble supplying instructions from incorrectly executed paths.

The *Aint feeder,* the most aggressive in this set of instruction feeders,[2] supplies instructions from a program binary. The performance model directs Aint to fetch and execute instructions on its behalf, as determined by its timing model and predictors. Additionally, Aint maintains enough internal state to act as a verifier for the correct path. Aint fetches and executes any instruction under the direction of the performance model, but it refrains from committing instructions from incorrect paths. Aint can help a performance model correctly simulate a modern, dynamically scheduled, speculative microprocessor. However, because Aint does significant additional computation, it is somewhat slower than the dynamic trace feeder.

Table 1. Possible execution scenario in performance model and feeder.

Cycle	Performance model			Feeder		
	PC1	PC2	PC3	PC1	PC2	PC3
1	Fetch			Decode		
2	Decode	Fetch			Decode	
3	Execute	Decode	Fetch	Calculate effective address		Decode
4	Memory cache miss		Decode			
…						
				Load value		
22	Commit	Execute		Commit	Execute	
23			Execute			Calculate effective address
24		Commit	Store		Commit	Store value
25			Commit			Commit

Separation of feeder and performance model

Asim separates architectural instruction execution in the feeder from the timing predictions in the performance model, which simplifies the models by avoiding full architectural emulation. The following code snippet helps to illustrate this functionality separation:

```
PC1: R1 ← (R2)
PC2: R3 ← R1 + R5
PC3: R3 → (R2)
```

This code shows a sequence of three instructions at addresses PC1, PC2, and PC3. The first instruction loads a value from a memory location, the second adds it, and the third stores the added value back to the same location.

Table 1 shows a possible execution scenario in the feeder and performance model, in which the performance model emulates a five-cycle pipeline that includes Fetch, Decode, Execute, Memory, and Commit functions. In contrast, the feeder only performs a fixed and smaller set of tasks to execute the program correctly.

In Table 1, the performance model controls the feeder's operation. The feeder tracks all architectural-state attributes such as register and memory values. Typically, the performance model has no notion of these values. Hence, the model obtains the effective address of the load instruction from the feeder and does not calculate it explicitly. The feeder has no notion of cache or memory hierarchy, but instead has a flat memory space. The performance model maintains the cache, but it does not track the cache data. Instead, it keeps the cache tags only for timing purposes to check cache hits or misses.

As Table 1 shows, the performance model experiences and simulates the timing of a cache miss, but during this time the feeder performs no action for the corresponding load instruction. After the miss is resolved and its timing is simulated correctly, the performance model directs the feeder to read the value returned by the miss from the feeder's memory space.

A performance model communicates with a feeder through a fixed interface that consists of a set of method calls, including calls to decode, execute, perform memory operations, kill, and commit an instruction in the feeder. The performance model can have several pipeline stages to account for the actual working of a long microprocessor pipeline, and must invoke these method calls in the correct order.

For example, if a store feeds a load, the performance model must ask the feeder to execute the store before it executes the load. Alternatively, the model must detect the store-load order violation, avoid invoking the commit method call on the load instruction, and request that the feeder kill the load instruction to clean up the feeder's internal state corresponding to these instructions.

ARCHITECT'S WORKBENCH

The Architect's Workbench is a collection of tools for using, managing, and debugging Asim's performance models and benchmarks. AWB consists of two basic features: the configuration files and the tools to manipulate these files. The configuration files specify the architectural model and the benchmark for a specific experiment. The configurator is an interactive tool that manipulates these configuration files and exposes the module alternatives to an Asim user. The configurator uses a graphical user interface, but developers can also

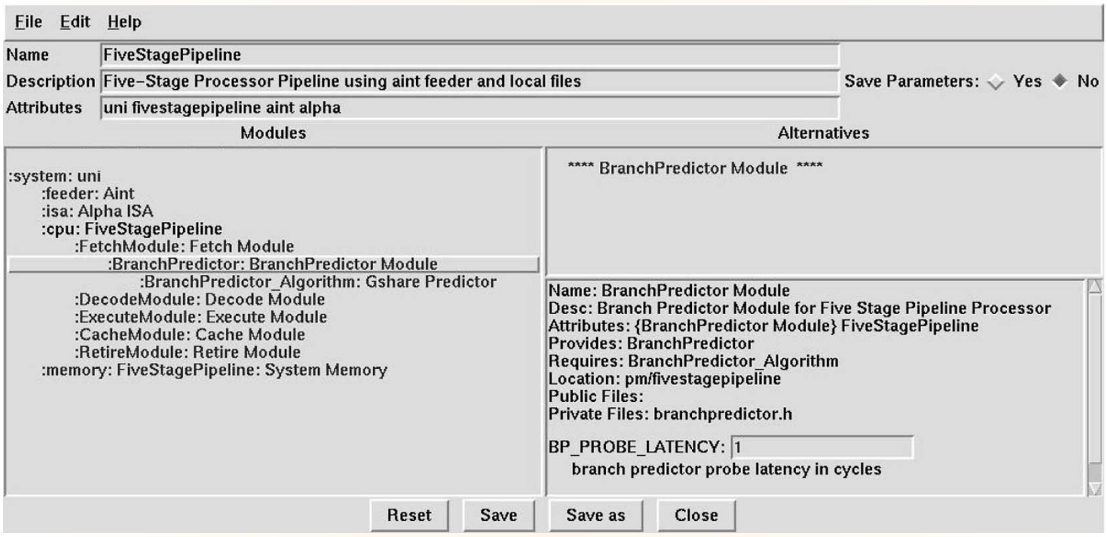


Figure 5. AWB configurator displaying the builder dialog. The builder dialog builds a complete model from scratch. The displayed module requires an instruction feeder, CPU pipeline, and memory subsystem.

employ all GUI functions through noninteractive command-line tools.

Configuration files

To run an experiment in Asim, developers need both a performance model and a program or benchmark. Asim captures both the performance model and the benchmark configurations in structured, text-based configuration files. These configuration files let a user run the same model with different benchmarks or different models with the same benchmark, thereby allowing greater control over the experimental setup.

An Asim performance model configuration file provides a complete specification. Each file contains all the modules in a provides-requires hierarchy needed to create a complete model and the corresponding model parameters. Asim records the default parameter values in .awb files, so the configuration file need only record parameter values that differ from the defaults.

A configuration file consists of information about the benchmark and how to run it. The information about the benchmark specifies the benchmark's name, directory, feeder, and so forth. The control portion of the configuration file directs AWB's execution of the benchmark, for example to specify skip or sampling intervals. An interpreter inside the model processes the commands.

GUI configurator

AWB provides Asim's GUI, including a browser for already configured models and benchmarks and a builder to create and edit models and benchmarks. Asim uses the builder dialog to build a complete model from scratch. The builder in Figure 5 shows a preconfigured model, which Asim builds by reading the model's structure and user-specified parameters from a performance model configuration file. Alternatively, developers can create a complete model from scratch by selecting modules from the module pool. Asim determines the pool of available modules by reading their .awb files.

Model building proceeds hierarchically, starting at the top, where the model requires an Asim system module. Selecting the system node in the hierarchy causes the builder to show all modules that implement the system type. When an Asim user selects a system, the hierarchy includes the selected system module and shows the set of submodules the selected module requires. The system module in Figure 5 requires an instruction feeder, CPU pipeline, and memory subsystem. Additionally, when the user selects the specific system module, the builder displays all the module's configurable parameters. Asim can override default values for all the builder's and recursively select and set parameters for all the system's modules and submodules.

After selecting all modules and setting the appropriate parameters, the user can instruct the AWB to save the new information as a model configuration file. Asim then uses this file repeatedly to build the same model and run experiments with it. This process creates a *build tree* for compiling the specified model. In this build tree, Asim represents all files that implement a module as symbolic links to the appropriate files in the source tree. For each module in the model, the tree also synthesizes and includes a new header file, which specifies as constants the parameters in the .awb and model configuration files.

TYING IT ALL TOGETHER

The Asim controller ties together the interactive display and module hierarchy, with the controller and modeling framework running in the same address space as separate threads. Unlike the modeling framework, which runs on a cycle-by-cycle basis, the Asim controller is event driven and thus maintains an ordered event list. Events consist of commands to or from the modeling framework. Commands to the framework might be benchmark control commands from a benchmark configuration or generated interactively. When the *AwbRun* command is sent, for example, the controller puts this event on the event list. At the appropriate point, the controller invokes an event to run the modeling framework for the number of cycles or instructions specified in the AwbRun command.

The modeling framework can also send commands to the Asim controller. Currently, the modeling framework only sends commands to create events for the runtime display. This display, which resembles one developed for an Alpha microprocessor,[3] shows the activity within a processor pipeline on a cycle-by-cycle basis, a capability that has proven invaluable for debugging microprocessor pipeline and performance model operations on small, selected sections of benchmark code. Thus, for example, the performance model can use the Asim controller to send event requests to the runtime display to show how an instruction proceeds through different stages of a microprocessor pipeline.

We plan to extend Asim in several ways. Currently, instantiating multiple Asim modules to create a hybrid module is difficult. For example, we anticipate extending Asim to compose bimodal and gshare branch predictors to create a hybrid branch predictor. Additionally, we hope to parallelize Asim in ways similar to the Wisconsin Wind Tunnel II approach.[4] We also plan to mix and match detailed and approximate module models to experiment with trading execution speed for timing prediction accuracy. Finally, we plan to look at more flexible clocking schemes and extend Asim to simulate different instruction set architectures and system models. ■

Acknowledgments

We thank David Goodwin for helping to build the initial Asim framework, the many Compaq engineers and interns who developed several performance models within Asim, and Michael Adler, Geoff Lowney, and Paul Rubinfeld for providing helpful feedback on initial drafts of this article.

References

1. M. Rosenblum et al., "Complete Computer Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology*, vol. 3, no. 4, Winter 1995, pp. 34-43.
2. A. Paithankar, "AINT: A Tool for Simulation of Shared-Memory Multiprocessors," master's thesis, Univ. of Colorado, Boulder, 1996.
3. M. Reilly and J. Edmondson, "Performance Simulation of an Alpha Microprocessor," *Computer*, May 1998, pp. 50-58.
4. S.S. Mukherjee et al., "Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator," *IEEE Concurrency*, Oct.-Dec. 2000, pp. 12-20.

Joel Emer *is an Intel Fellow in VSSAD at Intel. His current research interests include multithreaded processors, processor pipeline organization, and performance modeling frameworks. He received a PhD in electrical engineering from the University of Illinois. Contact him at joel.emer@intel.com.*

Pritpal Ahuja *is a microprocessor architect at Intel. His current research interests include performance modeling, benchmarking, and memory hierarchy issues. He received an MS in computer science from Princeton University. Contact him at pritpal.ahuja@intel.com.*

Eric Borch *is a senior hardware engineer in VSSAD at Intel. His current research interests include computer architecture and performance modeling. He received an MS in computer engineering from the University of Colorado, Boulder. Contact him at eric.borch@intel.com.*

Artur Klauser *is a microprocessor architect in VSSAD at Intel. His current research interests include speculation control, on-die communication paradigms, and modular design. He received a PhD in computer science from the University of Colorado, Boulder. Contact him at artur.klauser@computer.org.*

Chi-Keung Luk *is a senior systems engineer in VSSAD at Intel. His current research interests include computer architecture and compilers. He received a PhD in computer science from the University of Toronto. Contact him at chi-keung.luk@intel.com.*

Srilatha Manne *is a senior hardware engineer in VSSAD at Intel. Her current research interests include high-performance microarchitectures, low-power processors, and performance modeling. She received a PhD in electrical engineering from the University of Colorado, Boulder. Contact her at srilatha.manne@intel.com.*

Shubhendu S. Mukherjee *is a senior hardware engineer in VSSAD at Intel. His current research interests include fault-tolerant processors, tightly coupled network architectures, and performance modeling. He received a PhD in computer science from the University of Wisconsin–Madison. Contact him at shubu.mukherjee@intel.com.*

Harish Patil *is a senior systems engineer in VSSAD at Intel. His current research interests include compilers and computer architecture. He received a PhD in computer science from the University of Wisconsin–Madison. Contact him at harish.patil@intel.com.*

Steven Wallace *is a senior computer engineer in VSSAD at Intel. His current research interests include computer architecture and dynamic optimization. He received a PhD in computer engineering from the University of California, Irvine. Contact him at steven.wallace@intel.com.*

Nathan Binkert *is a doctoral candidate at the University of Michigan, Ann Arbor. His current research interests include simultaneous multithreading, operating systems, and high-speed I/O. He received an MSE in computer science and engineering from the University of Michigan, Ann Arbor. Contact him at binkertn@umich.edu.*

Roger Espasa *is an associate professor at Universitat Politècnica de Catalunya (UPC). His current research interests include high-performance microarchitectures, high-bandwidth memory systems, and simulation and modeling infrastructure. He received a PhD in computer science from UPC. Contact him at roger@ac.upc.es.*

Toni Juan *is an associate professor at Universitat Politècnica de Catalunya (UPC). His current research interests include high-performance microarchitectures, high-fetch-bandwidth engines, and technology implications for microarchitectures. He received a PhD in computer science from UPC. Contact him at antonioj@ac.upc.es.*

IEEE COMPUTER SOCIETY

Career Service Center

- Certification
- Educational Activities
- Career Information
- Career Resources
- Student Activities
- Activities Board

computer.org

Introducing the

IEEE Computer Society

Career Service Center

Advance your career
Search for jobs
Post a resume
List a job opportunity
Post your company's profile
Link to career services

computer.org/careers/