AWB: The Asim Architect's Workbench

Joel Emer[†‡], Carl Beckmann[‡], Michael Pellauer[†]

[‡]Intel Corporation
VSSAD Group
77 Reed Rd
Hudson, MA 01749
{joel.emer,carl.beckmann}@intel.com

[†]Massachusetts Institute of Technology
Computer Science and A.I. Laboratory (CSAIL)
32 Vassar St
Cambridge, MA 02139
{emer,pellauer}@csail.mit.edu

Abstract

Performance models are typically characterized on three metrics: model speed, model development time and model fidelity. While in many cases these metrics can simply be traded one for the other, adding modularity and reuse to a modeling infrastructure can enhance both model development time and model fidelity without significant deleterious impact on model speed. This paper describes the Asim Architect's Workbench, AWB, which aims to be a practical plug-n-play mechanism that can improve development time by reusing portions of existing models, and can easily allow fidelity trade-offs. Furthermore, AWB can enhance that aspect of fidelity related to correctness through the reuse of more highly debugged modules and through the ability to plug in and test a module in multiple environments.

AWB and the Asim infrastructure [ASIM] have been in use for almost 10 years at DEC, Compaq and Intel. A large number of models have been developed and over 20 distinct repositories are in use. This infrastructure has repeatedly demonstrated opportunities for module reuse at the fine grain, e.g., branch-predictor algorithms as well as at the coarse grain where entire CPU pipelines or memory hierarchies are reused. This reuse has often resulted in greatly enhanced model development time for high-fidelity models.

1.0 Introduction

The Asim Architect's Workbench (AWB) is a structured framework for the development of performance models. It aims to improve the performance modeling process, especially in the early exploratory stage, by supporting modularity and code reuse. This support is provided at two levels: First, Asim/AWB supports a representation of a model as a hierarchical tree of modules, where each module can be replaced with alternative implementations that satisfy the interface requirements of the module. In fact, these replacements allow complete control of the structure of the tree for a particular model. This allows a wide variety of different models to be constructed out a common pool of modules. At a second level of modularity, AWB allows these modules to be obtained from an arbitrary set of independently-maintained source-code repositories.

Traditionally, performance models are characterized on three metrics: model speed, development time and fidelity. Each of these characteristics plays an important role in the utility of a model. Clearly the number of instructions that can be modeled in a given amount of time is critical to the utility of a model. While the speed of the model is probably the most prominent and most often characterized metric, the others can be equally important.

When considering how to model a new design, model development time is frequently the foremost consideration in the modeler's mind. Thus, if model development time is especially long there will be a disincentive to create new models, and we will end up only studying minor variations of those designs that have already been modeled. This could have a stagnating effect on computer architecture research.

Model fidelity can also be an important model characteristic. A model that is absolutely faithful to the ultimate design will, of course, give very accurate estimates, while an easier to develop but more abstract model will naturally give less accurate results.

Another aspect of fidelity is the presence of errors in the model. Unfortunately, it is often difficult to detect bugs directly in a performance model because functional verification will not expose timing bugs. Commonly used techniques to check for errors in a model include checking timing against estimated expectations, detailed manual examination of the timing information generated by the model, and comparison against an RTL representation of the design (when it exists).

It is often noted that one can trade off these metrics against one another. Thus, a lower-fidelity model will run

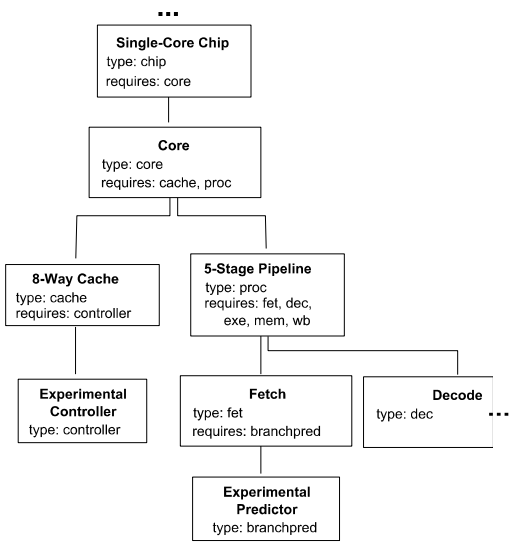

Figure 1: An example Asim awb model tree built from selecting and combining existing modules. Note that the provides/requires hierarchy is separate from the communication hierarchy, so that for example, the caches could communicate with the memory via ports.

faster and have shorter development time. Conversely, a high-fidelity model will take longer to develop and its increased level of detail will result in longer run times. Recent efforts to move performance modeling from software to FPGA-based hardware are designed to address model speed, but because the model writing effort is now a hardware design effort model development time will likely suffer [RAMP, FAST, HASIM].

AWB, the Asim Architect's Workbench is a component of the Asim infrastructure [ASIM] that is designed to aid in two of the three metrics: model development time and model fidelity without detracting from the third metric: model performance. This aid is provided by providing a plug-n-play style modularity for the model.

Borrowing existing code is a time honored technique for accelerating software development. Structured modularity, such as provided by AWB and in slightly different ways by other modeling infrastructures such as Liberty [LIBERTY] and M5 [M5], can improve model development time by allowing for the easy re-use of existing code in just those parts of the model where the sharing is applicable.

Modularity can also help in making fidelity trade-offs by allowing easy replacement of high-fidelity components with lower-fidelity components. Thus, one can easily make trade-offs between model speed and fidelity. Such modularity can facilitate the use of mixed-fidelity modeling where a particular component can be studied in detail in an environment of lower-fidelity components. Examples of such approaches are the mixing of behavioral and circuit-level components in RTL simulation and the use of multiple fidelities of processors in the FastMP project [FASTMP].

Modularity can also help improve that aspect of fidelity associated with the likely correctness of the model. As noted above, proving a performance model correct is not a particularly exact science, so having modules that are used repeatedly in multiple designs allows one to take advantage of the verification scrutiny that the module has undergone previously. Furthermore, clear interfaces and the ability to do plug-n-play of a particular module can be used to facilitate the examination of a module in a isolated test harness as well as in the full context of the design. Also, using shared, as opposed to copied, versions of the code also helps fidelity by automatic propagation of bug fixes in a module to all users of that module. Finally, although the effect is a bit more subtle, the fact that a user of a modular infrastructure is encouraged to design interfaces intended for modular replacement can result in better structured and hence more reliable code.

Asim/AWB implements a modeling infrastructure that supports the notion of modularity. Modularity in AWB is manifest in two ways. First, all models are represented as a hierarchy of modules. The user constructs a model by selecting the modules that form this hierarchy in a plug-n-play fashion. At each node of the model there may be one or more modules that can be plugged into that spot.

Second, the modules are selected from a pool, which may come from multiple source-code repositories. Using modules from multiple repositories facilitates full and partial sharing, i.e., through the use of access-restricted repositories. Thus, the ability to use code from multiple repositories forms another type of modularity.

2.0 Models and Modules

The fundamental abstraction provided by Asim/AWB is the notion that a *model* is represented as a tree-structured hierarchy of nodes, called *modules*. Each of these modules has a type, which is referred to as the *awb-type* of the module.

The current convention calls for the root of the tree to be a module of *awb-type* **model**. Furthermore, each module can specify which submodules it requires, i.e., the *awb-type* of its children. This recursive specification of modules and their children modules down to the leaf modules of the tree comprise the entire model as shown in Figure 1.

The *awb-type* can be thought of as an assertion that a particular module provides a particular interface that can

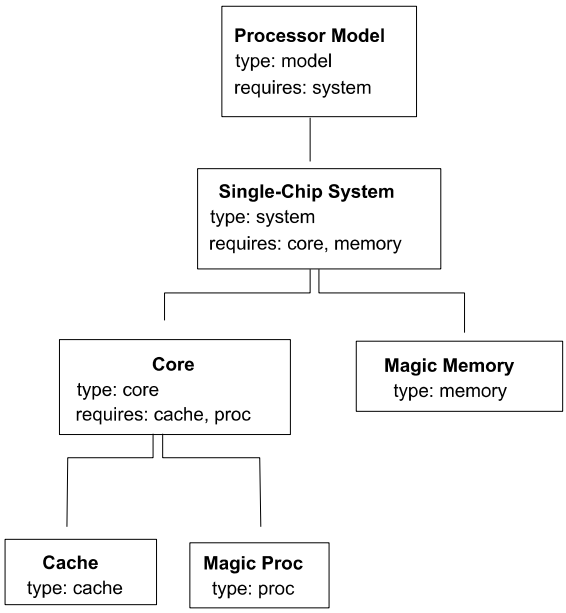

Figure 2: An alternative model. This setup could represent an experiment studying a cache in the context of a more abstract processor and memory subsystem.

be used compatibly by the other modules in the system. Note that the *awb-type* does not provide a formal specification of the interface supported by the module, but rather provides a handle to describe that a particular module claims to provide an implementation compatible with that interface.

The fact that the *awb-type* of a module implies the interface supported by the module and is the unit of plug-n-play means that more effort is often made to design the interface to be more modular in the long run. Although this means more upfront work before the coding can begin, we believe this can be a good thing because it results in a better structured and more general design.

Note, that it is important to make the distinction between a module and its *awb-type*. There can be multiple alternative module implementations that all correspond to the same *awb-type.* These alternative implementations can correspond to different fidelity implementations of a particular function or to alternative algorithms as shown in Figure 2.

Thus, in order to create a model (with a root of *awb-type* **model)** one must first select as the root of the tree a specific implementation of a module that represents itself to be an implementation of the *awb-type* **model.** That specific implementation will then include a specification of the particular *awb-types* of its children modules. Since each module implementation can make such a specification of its children, the entire structure of the tree can be different for each model.

```
/*************************************************
*
* %AWB_START
*
* %name Simple Branch Predictor
* %desc Branch Predictor for Five Stage Pipeline
* %attributes fivestagepipeline
* %provides BranchPredictor
* %requires BranchPredictor_Algorithm
* %private branchpredictor.h
* %param BP_LATENCY 1 "bp latency in cycles"
*
* %AWB_END
*************************************************/
```

Figure 3: Awb example

Even at the model root we find opportunities for alternatives, for example we have one model root for building software models and another for FPGA-based models [HASIM].

More concretely, in AWB the specification of a module's implementation is contained in a file called an awb file, which uses .awb as its suffix. An awb file typically contains the following specifications for the module:

1. `%name` – a unique name for this module
2. `%description` – a textual description of what this module implements
3. `%provides` – the *awb-type* of the module that this implementation provides
4. `%requires` – the *awb-types* of the children modules required by this module.
5. `%public` or `%private` – the names of the files that actually implement this module, e.g., the source-code files for the module.

Additional directives allow for the specification of parameters for the module, attributes that can be used to aid in the creation of a model as well a means of specifying the use of a library instead of source code as the implementation of the module. An example .awb file is shown in Figure 3.

In general, the source code for a module will be expressed in some programming language. Throughout this paper we will assume that language is C++, although Asim/AWB is largely agnostic to the programming

language being used and we are actively developing models in both C++ for normal software models and the high-level synthesis language Bluespec [BLUESPEC] for FPGA-based simulation.

In order for a particular implementation of a module to be useful, each alternative module for a particular *awb-type* will generally implement a C++ class with the same well-known name. This well-known class name can then be used by the parent module to declare and use instances of the child module's C++ types, irrespective of which child module implementation was selected for use in a specific model. Furthermore, as will be described in more detail later, the Asim/AWB infrastructure also creates a header file named after the *awb-type* of the module so that the parent module has a well-known filename to use in its #include directives.

Thus the Asim/AWB infrastructure provides a means of substituting module variants that is an alternative to C++ inheritance mechanism. Modeling languages, such as SystemC, based on object-oriented languages could model variants to be specified by using a base class to represent the common interface, and derived classes to implement the model variants. But this in itself does not make it easy to compose models from the module variants without "infecting" some portion of the model code with knowledge of the specific subclasses, or worse yet, requiring *all* module variants to be compiled into the code. The Asim/AWB mechanism separates the model code from the module selection, avoiding these problems, while also not requiring any particular language for the model code, thus making it equally applicable to composing FPGA-based hardware models and C++ based software models.

A frequently-used alternative to allowing full alternative source-code implementations of a module is to use C preprocessor directives to select among alternative implementations within a single source-code implementation. While we believe that such techniques are appropriate for relatively small and well-defined alternatives, it is not nearly as flexible or as structured as the Asim/AWB approach. For example, it does not facilitate independent implementations of the same *awb-type*. In fact, the initial implementation of Asim and AWB was, in part, a reaction to a large and ill-structured performance model that had so many #define variables that almost nobody knew what they meant or which combinations were valid. In many places the resulting code was practically unreadable. For example, there was one place in the code where 3 levels of #ifdef's surrounded a single brace ({}). Furthermore, many tasks that should have been relatively straightforward, such as changing the branch predictor algorithm, were scoped out as 3 person-month efforts.

The key to the utility of the Asim/AWB plug-n-play functionality is that it is actually possible to have alternative implementations of an *awb-type,* i.e., that at a practical level the module's interface is sufficiently clear that alternative implementations are possible. In practice we have found that most module interfaces tended to fall into four different categories. Thus while there is currently no formal categories for modules we refer to modules as being of these four kinds:

> *Plumbing modules:* These are the modules that form the topological structure of the design. While the model structure is strictly hierarchical, it would be insufficient to restrict communication to be strictly between parent and child modules. Thus in Asim we use another abstraction, referred to as a *port* [ASIM, APORTS] that can be used to convey information from one module to another module in the model. These ports can be thought of as pipelines that transfer information between modules in a fixed number of model cycles. Because of these plumbing-like ports that connect together modules, those modules that are dominated by these port connections are called plumbing modules.[1]
>
> *Algorithm modules:* These are the modules that provide alternative algorithms for a module in the model. Their interfaces are specified completely by the methods of the C++ class in the module implementation and they are typically used exclusively by their parent modules. Such modules are invariably children of a plumbing module.
>
> *Message modules:* These modules correspond to the structure of a message that is passed from module to module. A variety of modules might include the declaration of this kind of module and use them to define the datatype for the information that is passed through a port from one module to another.
>
> *Library modules:* This type of module is used for utility classes that are included and used in a variety of locations throughout the model. These modules are typically not structural and do not correspond to any particular part of the design, but rather are used for modeling infrastructure.

1 An important aspect of our ports is that when two modules use ports to communicate their parent modules do not include code to facilitate that communication. This enhances modularity since parent modules do not need to be changed when the communication requirements of their children change.

Each of these kinds of modules forms an opportunity for plug-n-play style modularity. Algorithm modules probably are the easiest form of modularity to use. Generally, an algorithm module that implements some policy will have a standard set of method calls to exercise that policy. Thus, for example, a branch predictor will typically be a class with methods for predicting, updating and aborting a prediction. There are often opportunities to have alternative implementations of modules like the branch predictor module by simply providing multiple implementations of that class.

For plumbing-style modules there are fewer opportunities for replacing a single module with an alternative implementation. However, frequently an entire sub-tree consisting of many plumbing-style modules comprise a higher-level architectural interface. Thus, replacement of that entire sub-tree can often form a useful place for alternative implementations. Examples of this include the interface between a processor and its cache hierarchy, the interface between a cache hierarchy and the memory subsystem or the interface between the front-end and the back-end of a processor pipeline.

For message-style modules, while there seem to be fewer opportunities for alternatives within a model, this kind of module is often quite amenable to being shared across multiple models. Thus, for example, the module that implements the representation of an x86 instruction can be used in many different models.

Finally, a major plug-n-play opportunity for library-style modules is the functional model portion of a split timing/functional simulator [ASIM]. Thus, we can plug in alternative functional models ranging from a simple trace-driven implementation to a full-system functional model. In addition they are many opportunities for sharing library-style modules across multiple models, thus saving implementation effort. Examples, in current use are the plug-in power and reliability modeling libraries.

Over the almost 10 year life of the Asim/AWB infrastructure we have repeatedly found opportunities for module reuse of algorithm modules, e.g., branch predictor algorithms. We have also found opportunities for reuse of plumbing modules in the form of entire units or parts of units such as CPU pipelines or memory hierarchies. Message modules have found reuse both across models and within models where (lower-fidelity) models have been able to be built that are ISA agnostic and the module specifying the instruction object can be substituted to switch ISAs. Finally, a variety of library modules have been implemented that are used across many models. Overall, this reuse has often resulted in greatly enhanced model development time for producing high-fidelity models.

3.0 Abstractions and Tools

The support for modularity in Asim/AWB makes its use a little different than a traditional software package, where one simply unpacks a tarball containing the source code and types `./configure; make`. Because there is no single model that is statically specified, there can be no single Makefile that will create the desired model. Furthermore, Asim/AWB supports the notion of obtaining parts of models from different source-code repositories, so there is not even a single tarball that corresponds to all possible models. Thus, Asim/AWB introduces the notion of a *workspace* for working on Asim models as well as some other abstractions.

The principal Asim abstraction are listed in the table below, some of which are described by a specification in a file with the given extension:

Abstraction	Description	Extension
Workspace	A directory subtree for working with Asim/AWB	N/A
Model	A hierarchical collection of modules representing a design	.apm
Module	The unit of plug-n-play in a model	.awb
Benchmark	A specification of benchmark characteristics and set up	.cfg

Asim/AWB provides a set of programs to manipulate *workspaces* and the other abstractions it supports. The most basic Asim/AWB command is `asim-shell`. `Asim-shell` is a command-line tool for creating an AWB environment and for manipulating that environment. For example, it is used for obtaining and managing code from a source-code repository and for configuration and running of models. To provide flexibility in its use, `asim-shell` accepts commands directly on the command line or inside a shell like environment that supports command completion and command editing and history. Basic information on the command line use of `asim-shell` can be obtained by typing:

```
% asim-shell –help
```

Information on the specific commands available can be obtained with the following:

```
% asim-shell help commands
```

Many of the operations that `asim-shell` provides for models and benchmarks can also be accessed through the GUI tool `awb`. The `awb` program allows a user to browse, edit and build models, as well as to browse and run benchmarks. Together these tools provide a command line-based and a GUI-based interface to manipulate the various abstractions provided by Asim/AWB.

4.0 Workspaces and Packages

An Asim *workspace* is a stylized directory structure that can be used to contain all of the components of the Asim/AWB environment. This primarily consists of the individual source-code repositories that have been checked out as well providing standard (but overridable) places for the various activities that one might like to do such as create, build and run models. The models, modules and benchmarks are mostly contained within the checked-out repositories.

The raw structure of a workspace is as follows:

> `awb.config` – a configuration file describing the state of the workspace. Probably the most important configuration variable in awb.config is the SEARCHPATH which specifies the *awb-searchpath, i.e.,* set of directories containing checked-out copies of repositories. Various Asim/AWB commands manipulate the contents of awb.config automatically, so the user should not have to modify this file by hand very often.
>
> `src/` - this directory contains the checked-out copies of repositories
>
> `build/` - this directory contains the instances of models that have been created from a particular model tree.
>
> `var/` - this directory contains various book-keeping files used by the Asim/AWB tools

Creating an Asim workspace is performed with `asim-shell.` Thus, the basic command for creating a workspace is:

```
% asim-shell new workspace <dir>
% cd <dir>
```

As demonstrated above, using a particular workspace is achieved by cd'ing into the workspace. All of the Asim/AWB tools can automatically determine your current workspace if your current directory is anywhere within that workspace.

An empty workspace is not particularly useful, so one needs to check out a copy of a source repository in order to use it. At present, `asim-shell` supports SVN, CVS and Bitkeeper style repositories, and we uniformly call a checked-out copy of a repository a *package*. In order to do proper workspace management and a little bit of book-keeping, `asim-shell` supports checkout, update and commit operations on packages. For other operations on a package, such as adding or removing a file, the native repository operations should be used. Thus, the usage sequence for a set of packages might be:

```
% asim-shell checkout package hasim
    ... various questions the user must answer
    ... before the checkout is performed
```

The user can now work on the contents of the package.

To re-synchronize the checked-out package with any changes in the repository, the user can update the package with the following command:

```
% asim-shell update package hasim
```

Finally the user can commit changes back to the repository with the following command:

```
% asim-shell commit package hasim
    ... an editor session is started to allow the user
    ... to describe the changes made.
```

Again the use of `asim-shell` rather than the native repository commands is encouraged to maintain certain bookkeeping operations on the package.

The structure of a package is typically as follows:

`admin/`
- a directory containing Asim/AWB bookkeeping files

`config/`
- a directory for model and benchmark configurations

`config/bm/`
- a directory tree containing benchmark configurations

`config/pm/`
- a directory tree containing model configurations

`modules/`
- a directory tree containing the pool of modules

In order to keep track of the packages that have been checked out, Asim/AWB has the concept of the *awb-searchpath*. The *awb-searchpath* is a list of directories containing packages and are stored in the workspace's `awb.config` file. These packages could be repository checkouts in the workspace's `src/` directory or simply a reference to some other directory on the system. Such references allow for sharing of packages. A variety of `asim-shell` commands are available to display and modify the *awb-searchpath* as well as to list the packages that it references.

To make file references less location specific, many file references throughout Asim/AWB can be made relative to the root of some package. Thus as part of the file dereferencing operation, the Asim/AWB tools search in order through the elements of the *awb-searchpath* for a file matching the given filename. The result is a sort of

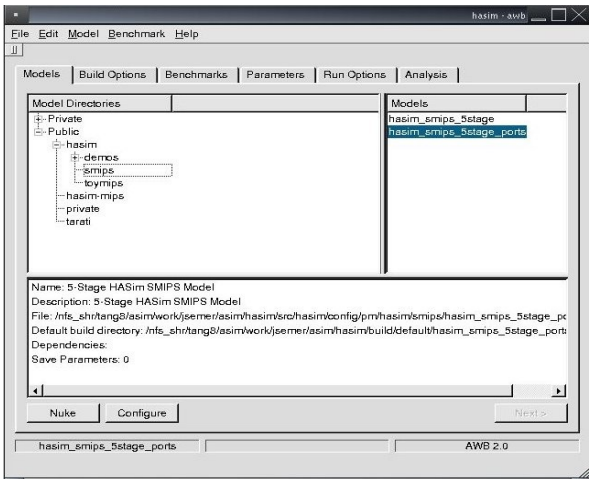

Figure 4: Awb - Model Tab

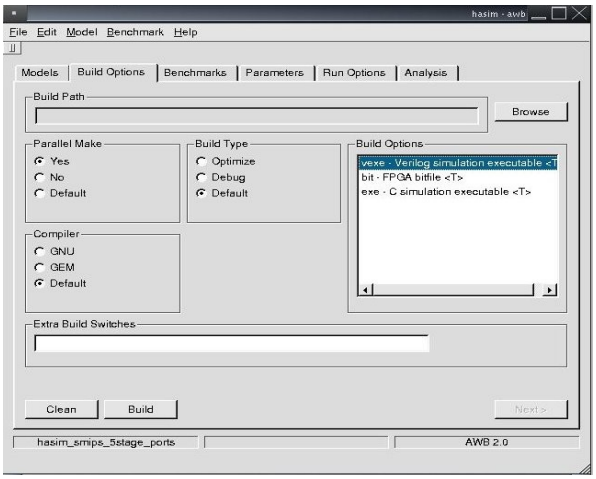

Figure 5: AWB - Build Tab

poor man's UNIONFS [UNIONFS] in which the directory trees corresponding to the packages in the *awb-search-path* can be thought of as overlaid on each other in the order specified in the *awb-searchpath.* The result leads to more convenient references to package files and the ability to easily move files from one package to another, but can lead to confusion when a file in one package masks a file with the same name in the same relative location in another package.

5.0 Model Building and Running

After a user has checked out a set of packages the user needs be able to manipulate and run models. An easy way to do this is using the GUI tool `awb`. `Awb` allows a user to visually configure and build models, and set up and run benchmarks. To perform nearly all these operations `awb` just runs a command line operation, which is shown at the top of the log window displayed when an operation is executed.

Figure 4 shows the `awb` opening screen. In the panel to the upper left, one sees the directory tree containing models. This directory tree is actually the overlaid union of the directories in the directory `config/pm` in all packages in the *awb-searchpath.* Selecting a directory results in a listing in the upper right panel of all the model (*.apm) files in that unionFS-style directory. Selecting a specific model results in a brief listing of information about that model in the lower panel. Now, we are ready to configure the model.

As noted above, because the Asim/AWB environment does not contain a single massively reconfigurable model there is no single Makefile (or equivalent) that can be used to build a model. Instead there is an operation provided by the Asim/AWB tools to *configure* a model. Thus, after a user selects a model he/she must request the system to *configure* that model. The result is a build tree, which by default is placed in a directory of the workspace named:

`build/default/<model>/pm`

In this build tree will be a ordinary Makefile and the source files for the build.

More precisely, the process proceeds by taking the specification of the model, which is contained in an .apm file that describes the set of modules (each of which is specified by an .awb file) that comprise the model. The top module of that hierarchy with *awb-type* **model** will invariably contain a specification of one or more Makefile templates. These templates will be text substituted with the names of the source files for the modules as well as for libraries or sub-targets of the build process. They then will be placed in the build target which is further populated with all the source files of the modules (actually symbolic links to the original files) which are obtained from a traversal of the module tree. Also added will be some synthesized files that include parameter values and have well-known filenames for things like #include operations. This process can be invoked by clicking on the "configure" button on the first tab of `awb`, and the end result is a ordinary looking build tree for the model of interest. The "Nuke" button on this tab can be used to completely erase any prior configured versions of this model.

Moving to the second tab of `awb` as shown in Figure 5, the build control screen for the selected model can be seen. A variety of build options are presented, which simply correspond to various command-line arguments to `make`. Clicking on "Build" simply invokes the `make`

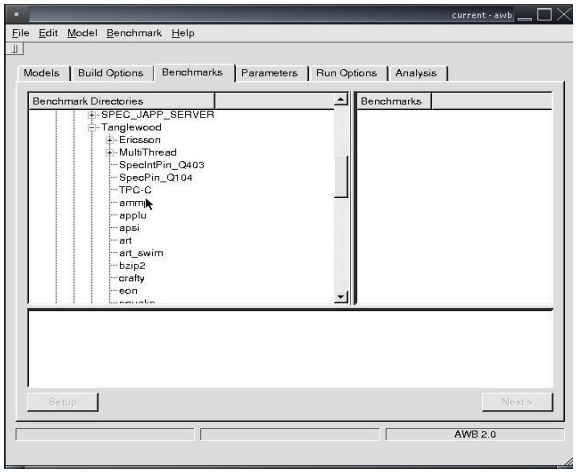

Figure 6: AWB - Benchmark Tab

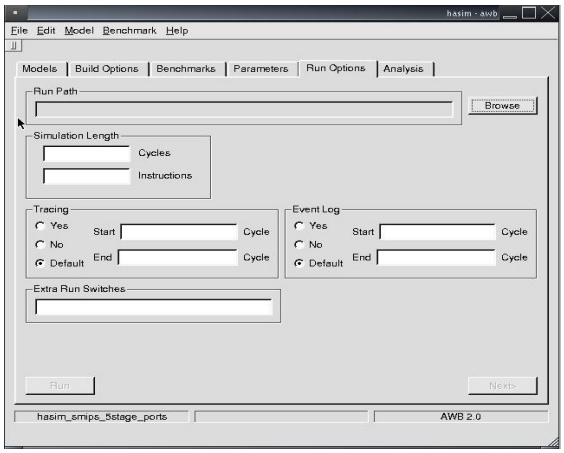

Figure 7: AWB - Run Tab

command in the appropriate directory with the selected arguments. As a convenience the "Clean" button will invoke a `make clean` on the model.

The third tab of `awb` (as shown in Figure 6) displays a view of all the benchmark directory tree in a manner analogous to the models in the model tab. These benchmarks are found in the overlaid directories at `config/bm`. A benchmark configuration file contains a bit of information about the benchmark along with the name of a script to set up the benchmark. Clicking on the "Setup" button causes that script to be invoked and a directory with all the files needed to run the benchmark is created. By default this directory is:

```
build/default/<model>/bm/<benchmark>
```

which will contain a script creatively named `./run` to run the benchmark on the model.

Asim models support configuration parameters that are set at both compile-time and run-time (with default values). For those models with run-time parameters they can be set in the "Parameters" tab. And finally the "Run" tab in Figure 7 allows one to run the benchmark. Again the options on this page correspond directly to command-line options of the `./run` script. With luck, clicking "Run" will result in an interactive run of the benchmark on the model.

Other tools for submitting vast quantities of simulations are also available. Most low-level manipulation of Asim/AWB objects, such as workspaces, models, modules and benchmarks are handled through a common library written in object-oriented Perl, so it is relatively easy to add additional functionality to the system.

6.0 Model Configuration and Creation

Creating a model consists of using modules already in the module pool, i.e., in the `modules/` directory of a package, or creating new ones. Creating a module consists of placing in a directory a new module description (.awb) file and the corresponding source files.

Creating (or modifying) a model can be done from `awb`'s model tab. The "Model" menu, or the right click menu on a model, brings up a model editor. The model editor is illustrated in Figure 8. The main information in the model editor is the module hierarchy in the top panel of the screen. Each node in the tree is identified on the left by its *awb-type* and (if present) the name of the particular module implementation that has been selected for this model. As a convenience, the right click menu on a node permits one to "edit" the module, i.e., open the files comprising that module in your favorite text editor.

When a particular node of the tree is selected, the alternative implementations of that *awb-type* that are available are listed in the lower left panel. These alternatives come from the module pool formed by the union of all modules in all available packages. Selecting one of the alternatives will result in a more detailed description of the module appearing in the panel to the right. Double clicking on an alternative will instantiate that module into the model. After a module has been selected, the default values for the module's parameters can be overridden for this model by selecting and updating the parameter value in the module description dialog box.

Note that Asim/AWB maintains a cache of all the modules in the module pool. So if a module you expect to see is missing, then clicking on the "Refresh" button will update the cache.

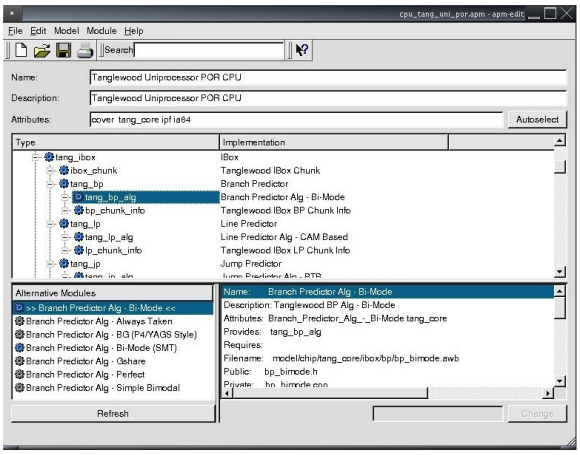

Figure 8: Model Editor

Creating models by consecutively clicking on the nodes of the model tree can be rather tedious, so the model editor provides a way to narrow the selection or select a best match for each node. The selection process is based on the idea of *module-attributes.* Each module in its specification (.awb) file is allowed to provide a prioritized list of attributes that best describe the module. For example, a module might be part of a particular variant V, of a particular design, D, of a part of the family of designs, F, and used for a particular architecture, A. Similarly, the model itself can contain a prioritized list of attributes that best describes that machine that is being modeled. The model editor can automatically select the module that is the best match for the model's attributes, and will highlight or automatically fill in that module into your model. Future work includes allowing further refinements, for example by using 'goes with' or 'conflicts with' specifications.

Another convenient feature of the Asim/AWB environment is the concept of sub-models. Instead of selecting an individual module for a node of the model tree, one can select (via the right click menu or module selection box) an entire model (.apm) file with a tree rooted at a module of the required *awb-type.* Creation of a sub-model rooted at something other than **model** is accomplished through the right click menu on an intermediate node in an existing model. These sub-models allow one to quickly propagate changes in some portion of a design, such as a CPU, though all models that use that CPU.

7.0 Conclusion

This paper has described the Asim Architect's Workbench, AWB, which aims to provide a structured framework for performance modeling that supports modularity and reuse at two levels. First, Asim/AWB supports a representation of a model as a hierarchical tree of modules, where each module has a type, its *awb-type* that informally specifies the module's interface requirements. Substitution of a module with another module of the same *awb-type* permits alternative implementations that represent different designs with the same interface or alternative fidelity representations of the same design. Using this plug-n-play style mechanism a user can create a large variety of models from a pool of modules. Since it is distributed across multiple independent source-code repositories, this pool forms the second level of support for modularity and reuse. Much of this functionality is accessed through a simple command-line tool and a corresponding GUI-based program.

The Asim/AWB infrastructure have been in use for almost 10 years at DEC, Compaq and Intel. A wide variety of processor designs have been modeled and over 20 distinct repositories are in use. And recently, this infrastructure has been applied to pure hardware design.

The Asim/AWB infrastructure has repeatedly demonstrated the opportunities for module reuse at the fine grain, e.g., branch predictor algorithms, and the coarse grain where entire CPU pipelines or memory hierarchies are reused. Also plug-n-play modules have facilitated the creation of models driven alternatively by a wide range of inputs from random instruction generators to traces to full-system simulation. Overall, this reuse has often resulted in greatly enhanced model development time for high-fidelity models.

Acknowledgments

The authors would like to thank the great many people who have contributed to various aspects of Asim over the years. More specifically, the original implementation of the plug-n-play framework was written by David Goodwin. Michael Adler, Artur Klauser and Martha Mercaldi have also made notable contributions to the core Asim/AWB functionality. Over the years, innumerable conversations with Toni Juan, Srilatha Manne, Shubu Mukherjee, Angshuman Parashar, Ramon Matas and Nate Binkert and more recently with Arvind have helped immeasurably to refine the Asim/AWB design. Currently, Saila Parthasarathy, Krishna Rangan and Brian Slechta are participating in the design and implementation of the system.

Bibliography

[ASIM] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, T. Juan, Asim: A Performance Model Framework, IEEE Computer, vol. 35, no. 2, pp. 68-76, February, 2002.

[LIBERTY] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, The Liberty Simulation Environment: A Deliberate Approach to High-Level System Modeling, ACM Transactions on Computer Systems, vol. 24, no. 3, pp. , August, 2006.

[M5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, S. K. Reinhardt, The M5 Simulator: Modeling Networked Systems., IEEE Micro, vol. 26, no. 4, pp. 52-60, July/August, 2006.

[FASTMP] S. Kanaujia, I. Esmer Papazian, J. Chamberlain, J. Baxter, FastMP, MOBS, 2006.

[RAMP] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform. Technical report, 2005.

[FAST] Derek Chiou. *FAST: FPGA-based Acceleration of Simulator Timing Models*. In Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11, February 2005.

[HASIM] N. Dave, M. Pellauer, J. Emer, Arvind, Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA, the 2nd Workshop on Architecture Research using FPGA Platforms (WARFP), 2006.

[BLUESPEC] Bluespec, Inc..Bluespec Language Reference Manual.

[APORTS] M. Pellauer, J. Emer, A-Ports: Simulating Synchronous Digital Circuits with Synchronous Digital Circuits, Submitted to ICCAD, 2007.

[UNIONFS] D Quigley, J Sipek, CP Wright, E Zadok, UnionFS: User-and Community Oriented Development of a Unification File System, Ottawa Linux Conference, 2006.