SimAPI—A common programming interface for simulation

by G. G. Hallock
E. J. Kaminski, Jr.
K. M. Lasko
M. P. Mullen

This paper describes SimAPI, a common programming interface for cycle simulators, and how SimAPI was used to test the S/390® Parallel Enterprise Server Generation 4. SimAPI provides a rich set of functions useful for programs and test cases to drive and monitor a simulation model. Support exists for multithreading, event detection, checkpoint/restart, and accessing objects in the model. Most of the code which implements this interface is shared among all simulators, with only a small amount of simulator-specific code needed to connect a simulator to SimAPI. This makes it easy to run a new simulator with SimAPI.

Introduction

A cycle simulator is a type of logic simulator which ignores the detailed timing of the logic circuits and calculates the output of the logic only on clock cycle boundaries. This tends to be much faster than event-based simulators, which simulate the timing of the logic. Typically, some form of timing analysis must also be done during the design cycle. However, most functional testing of a logic design can be performed with a cycle simulator.

In recent years the number of cycle simulators has increased significantly, with each simulator having its own special niche. For example, a simulator with low model-build time and source-level debugging is ideal for testing small pieces of logic in the early stages of a design. The same simulator may be much too slow for large sections of logic. On the other end of the scale, a hardware accelerator provides very high performance, but model-build time is large and debugging is difficult.

For the S/390* Parallel Enterprise Server Generation 4 (S/390 G4), four different simulators were used. For designer-level simulation a VHDL event simulator was used, since it provided fast model construction and source-level debugging. However, this simulator proved to be much too slow for element-level models. (An element model consists of a medium-size chunk of logic, typically with well-defined interfaces such as a processor or memory controller.) For small element-level models, the TEXSIM software cycle simulator was used. This simulator provided much higher speed than the VHDL simulator, but at the expense of slower model build and no source-level debug. Larger element models used ZFS, a software-event-driven cycle simulator. This simulator provided much higher performance than TEXSIM for large models, but ran slower for small models. Finally, system-level models were simulated on the EVE hardware accelerator. This simulator has the longest model-construction time of the four simulators, and debugging is difficult. However, the simulation performance was much better than that of any

0018-8646/97/$5.00 © 1997 IBM

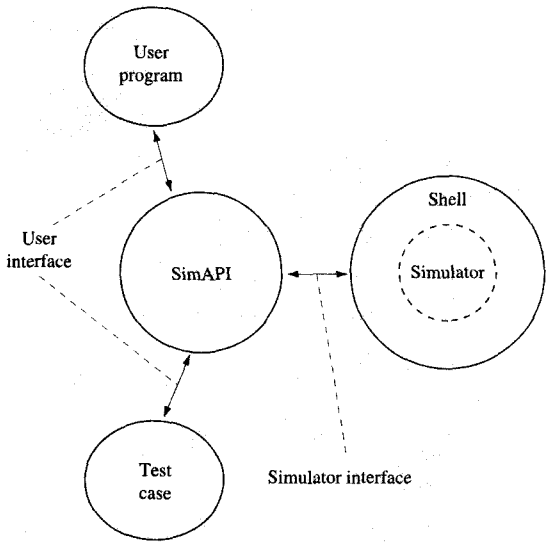

Figure 1

SimAPI interfaces.

other simulator. These simulators provided a wide range of performance—from 10 seconds per cycle to more than 300 cycles per second.

Each simulator has its own programming interface. In addition, two of the simulators ran only on AIX*, the EVE hardware accelerator ran only on VM/CMS, and the ZFS software cycle simulator would run on both AIX and VM/CMS. This made it very difficult to write code and test cases to drive a simulation model and then move to a different simulator. Rather than rewrite code for each simulator, a common programming interface called SimAPI was developed, and all user code and test cases were written against SimAPI. SimAPI determines which simulator is running and calls the appropriate native simulator functions, thereby insulating the user code from the differences between simulators.

Attempts had been made in the past, when a new simulator was developed, to mimic the interface of its predecessor. However, this was limited to cases in which the new simulator was designed as a new and improved replacement for an old one. In addition, these interfaces typically exposed too much of the detail of the simulator to the user, making it difficult or impossible to use the interface efficiently with another cycle simulator. SimAPI is the first interface that addresses the wide differences between cycle simulators and runs efficiently on both software simulators and hardware accelerators.

A common interface

SimAPI provides a rich set of high-level functions for use by user programs and test cases to drive a simulation model. It does not attempt to be yet another programming language, since there are plenty of languages to choose from already. There are no if/then/else or loop constructs; instead, SimAPI simply provides bindings to existing languages. Currently C/C++, PL/X, and Rexx are supported. Typically, programs that are used often and require high performance are written in C/C++ or PL/X. When ease of use is needed or when performance is not critical, SimAPI provides a command-level interface for Rexx. SimAPI runs on three operating systems—AIX/6000, VM/CMS, and Sun Solaris.

SimAPI actually consists of two well-defined interfaces, as shown in **Figure 1**. The first is a high-level interface for use by user programs and test-case drivers with nearly eighty functions. The second is a low-level interface connecting SimAPI to a simulator consisting of about thirty functions. Only about half of these low-level functions are essential to basic SimAPI operation. The remaining functions can be implemented at the simulator's discretion to provide improved performance and support functions that are less often used. Typically, a small amount of code called the simulator shell must be written for each simulator that plugs into SimAPI. This shell code converts the low-level simulator calls from SimAPI to the native simulator API. When possible, this shell code is merged into the simulator to effectively render the native simulator API the same as the SimAPI simulator interface and to improve performance. Currently, shell code exists to connect eight different cycle simulators to SimAPI.

It may seem at first that SimAPI could be significantly slower than the native simulator interface, since a function call must go through at least one and typically two additional layers of code (SimAPI and the specific simulator shell). However, as is shown later, SimAPI provides great flexibility as to which functions are provided by the simulator and shell and which are handled by SimAPI. Also, SimAPI user-level functions are typically higher-level than native simulator functions. This means that fewer calls to SimAPI functions are needed. For these reasons, SimAPI is typically no slower than the native simulator interface and can actually be faster.

Threads

SimAPI provides support for multithreading. All threads run in the same address space and are non-preemptive. A thread gives up control to another thread only when it must wait for some event. On UNIX**-based systems, SimAPI implements threads with POSIX threads.

There are two types of threads—overlap and non-overlap. An overlap thread has no need to access the simulation model and can run in parallel with running

simulation cycles. A non-overlap thread requires access to the simulation model and can run only when simulation is stopped. An overlap thread can temporarily gain access to the simulation model by calling the `spi_halt_sim` function. This causes simulation to stop so that the thread can have a constant view of the model. The thread can later call the `spi_resume_sim` function to tell SimAPI that it no longer needs access to the model. Simulation cycles begin to run again only when all threads have indicated that they no longer need access to the model. There are functions in SimAPI to create a thread, cancel a thread, and cause a thread to wait for some event.

Use of multiple threads can simplify writing of complex test cases (e.g., multiple threads, each driving a separate port in the model).

Events

SimAPI can be used to define various types of events, which can be used in two ways. First, a thread can wait for an event. The thread suspends, giving control to other threads, until the event occurs; this is done with the `spi_clock` and `spi_wait` functions. Second, an event can be used to trigger the start of a new thread using the `spi_after` function. Whenever the event occurs, a new instance of the thread will be created.

In addition to defining events, SimAPI has functions to delete, activate, and deactivate an event. When an event is deactivated, it will not occur even if the conditions are such that the event would normally occur if activated. This can help improve performance in two ways. First, ignoring an event after it occurs is more time-consuming than not setting the event in the first place. Second, deactivating and activating an event is much faster than deleting and later redefining the event.

- *Cycle events* A cycle event occurs when some number of cycles have been run. Cycle events can be defined to be relative to the current cycle or based on absolute cycle numbers.
- *Program events* A program event provides a way for threads to synchronize and pass messages. A thread can set a program event and associate a message with the event. If some other thread is waiting for the event, it will "wake up" and can then read the message.
- *Object events* An object event occurs when one or more objects in the model reach certain values. An object event is composed of an expression containing any number of terms which are combined with AND and OR functions. Each term compares an object in the model with a value. Object events can be either level-sensitive or edge-triggered. A level-sensitive event occurs for every cycle for which the expression is true. An edge-triggered event occurs when the expression changes from false to true.

SimAPI gives the simulator a choice of how object events are detected. By default, SimAPI detects object events by fetching the objects every cycle and evaluating the expression. The simulator can choose, however, to detect the events itself. This can improve performance greatly for hardware accelerators such as EVE and for distributed simulation where the simulator is running on multiple processor nodes. For example, EVE detects object events by actually dynamically modifying the logic being simulated. The simulator can choose to detect some events while leaving others for SimAPI to handle. This is an example of how SimAPI can actually improve performance even though it introduces extra layers of code in the calling chain. Without the concept of events, a program would be forced to sit in a loop fetching objects and running a cycle. This provides acceptable performance for some software simulators, but performance would be very poor for a hardware accelerator such as EVE. By providing the higher-level concept of events, the simulator now has the opportunity to choose a method of event detection that works efficiently.

- *Built-in events* SimAPI also defines some built-in events. For example, the START event is set by SimAPI just before the first simulation cycle is run. The END event is set when SimAPI is ready to end simulation.

Accessing model objects

SimAPI has functions to get, set, stick, and unstick an object in the simulation model. The function `spi_get_object` gets the value of an object, and `spi_set_object` sets an object to a particular value while still allowing the logic in the model to modify the object. The function `spi_stick_object` sticks an object to a particular value. This differs from *set* in that the logic in the model is prevented from modifying the object. Normally `spi_set_object` is used to set the state of objects which have memory, such as registers and memory arrays, while `spi_stick_object` is used to tie model input signals or to inject errors. The function `spi_unstick_object` reverses the effect of *stick*, allowing an object to be modified by the logic in the model.

For performance reasons, these functions do not access the model directly. Instead, the requests are buffered using a *model action list*, which is simply a list of gets, sets, sticks, and unsticks. Calls to such functions as `spi_get_object` and `spi_set_object` can be used to add requests to a list. The list is not actually executed until a call to `spi_flush_list` is made. This buffering can improve performance significantly for hardware accelerators such as EVE and for distributed simulators, since it reduces the number of I/O operations required.

There are two basic types of model action lists—temporary and permanent.

- *Temporary lists*

A temporary model action list is emptied after the commands on the list are executed. Temporary lists are useful when one-time references to objects are required. By default, a temporary list has a fixed-size buffer. When the buffer becomes full, SimAPI issues an implicit call to `spi_flush_list` to free up the buffer. It is possible to tell SimAPI not to implicitly flush a list. In this case the list buffer grows as needed and is flushed only by an explicit call to `spi_flush_list`. Programs can define any number of temporary lists, but a program will typically use the default temporary list defined by SimAPI.

- *Permanent lists*

A permanent model action list can be executed many times and can be emptied only by an explicit call to the `spi_empty_list` function. Permanent lists are useful when the same operations must be performed on a group of objects many times during a simulation. Unlike a temporary list, the buffer for a permanent list grows as needed. SimAPI never implicitly flushes a permanent list. Using a permanent list can significantly improve performance. All error checking on inputs is done when the list is created, so this time is saved on each execution of the list. Also, SimAPI provides the simulator an opportunity to preprocess the list after all requests have been added to the list. This processing can include such things as formatting the list for faster, later execution and moving the list closer to the actual simulator. For example, in a distributed simulator which runs on multiple nodes connected by a network, the list can be split into pieces, with each piece sent to the node that must process it. Then, when the list is executed, there is relatively little data that must be transferred across the network. This has proved to provide a significant improvement in performance.

- *Conditional set and stick*

SimAPI has conditional set and stick functions—`spi_cond_set_object` and `spi_cond_stick_object`. These are similar to set and stick, but the execution of the command is dependent on a change flag. When the list is executed by calling `spi_flush_list`, only those commands whose change flag is set will be executed. When used with a permanent list, conditional set and stick can significantly improve the performance of driving interface lines which change infrequently. A program would typically set only those change flags for objects whose value must be modified and clear all other change flags. It would then call `spi_flush_list`. This prevents time from being wasted setting or sticking objects whose value is not changing, while at the same time exploiting the performance advantage of permanent lists.

- *Shadow*

Sometimes it is necessary to get the value of an object every cycle. This can be done with `spi_get_object` and a permanent list, but SimAPI provides a more efficient method. The function `spi_shadow_object` is very similar to `spi_get_object`. However, shadow makes certain assumptions which allow it to be fast:

- The object is going be fetched every cycle (or at least very often), so shadow optimizes for this case.
- Changes to an object occur relatively infrequently.
- Shadow is allowed to update the data area where the value of the object will be stored at any time (not just due to a flush).

This last assumption is critical to shadow's speed. Since shadow allows the data area to be updated as soon as the object is changed without waiting for a call to `spi_flush_list`, quite often, by the time the list is flushed, there is nothing much left to do. Shadow will also optionally set a change flag if the object has changed value. This allows a program to flush a list containing a number of shadows and then quickly determine what has changed and what needs further processing.

SimAPI gives the simulator a choice as to whether it will implement shadow. Event simulators, such as ZFS and the VHDL event simulator, can implement `spi_shadow_object` much more efficiently than `spi_get_object`, since they must do something only if the object changes value. Other simulators such as TEXSIM and EVE cannot implement `spi_shadow_object` any faster than `spi_get_object`. If a simulator tells SimAPI that it does not handle shadow, SimAPI will convert `spi_shadow_object` calls to `spi_get_object`. This allows shadow to work on all simulators while at the same time exploiting the special features of some simulators to improve performance.

Operational (utility) functions

- *Object name prefixing*

Because of the many levels of hierarchy that can occur in a large system model, object names can become quite long. A particular thread may have to access objects only in a local section of the hierarchy. It is also sometimes necessary to drive multiple copies of a piece of logic which are in different parts of the hierarchy with the same code. SimAPI supports object name prefixing to simplify the sharing of code for this purpose. Each thread has an associated prefix which is added to all references to objects. When a thread is initially started, the prefix is null. A thread

can then set the prefix using the `spi_set_prefix` function. Typically, the prefix would be passed as a parameter when the thread is started. So, for example, to have a single piece of code drive multiple copies of a piece of logic, all that need be done is to start up multiple threads, passing a different object name prefix for each thread.

• *Virtual and alias objects*

SimAPI permits virtual and alias objects to be defined. To a program, a virtual object looks just like any real object in the model. It can be set and its value can be fetched. Since a virtual object is not connected to any real logic, its value changes only because of programs calling SimAPI functions. Virtual objects can be useful when a subset of a model is being run and a program expects an object to exist which is not in the subset.

An alias object is a special kind of virtual object. Like a virtual object, it does not exist in the model itself; however, unlike a virtual object, an alias object is backed by real objects. Alias objects are useful when the bits of a register are actually represented by many smaller objects. For example, a 32-bit register might consist of 32 one-bit latches, each with its own name. An alias object can be defined that is then mapped onto the individual latches; thus, a program can refer to a single 32-bit object instead of many small latches. A single alias object bit can be mapped onto multiple real object bits. This is useful when there are multiple copies of an object in the model. Setting the alias object then causes all copies to be set. It is also possible to leave gaps in an alias. For example, a register might logically be 32 bits wide but have only 31 bits used. Any alias bits that are not mapped to a real object return a value of 0 when the object is fetched. When set, the data for any unmapped bits are ignored. SimAPI provides functions to define and delete alias objects and to map alias objects to real objects.

• *Running cycles*

SimAPI provides two functions to allow a thread to run simulation cycles. The `spi_clock` function causes the thread to suspend until a specified number of cycles have run. The `spi_wait` function allows the thread to wait either for some number of cycles to be run or until one or more events occur.

In both cases, simulation cycles are not run directly as a result of the thread calling `spi_clock` or `spi_wait`. Instead, the thread simply waits for cycles to be run or an event to occur. SimAPI causes cycles to be run only when all non-overlap threads have suspended.

• *Checkpoint/restart*

SimAPI provides functions to checkpoint the state of the simulation model and to restart the model from a checkpoint.

When a thread calls the `spi_checkpoint` function, a checkpoint is not immediately taken. Instead, the thread is suspended until it is safe to take a checkpoint. SimAPI considers it a safe time to take a checkpoint only when all non-overlap threads have either suspended or ended. This is typically just before the next simulation cycle is run. In this way a thread can call `spi_checkpoint` and be assured that the checkpoint will not be taken while some other thread is in the middle of making changes to the model state. Similarly, a call to the `spi_restart` function causes the thread to suspend until all non-overlap threads have suspended or ended. The control of the time at which a checkpoint or restart is done is performed by SimAPI, so the simulator need not concern itself with this. The low-level simulator functions for checkpoint and restart perform the function immediately.

One use of checkpoint is to save the reset state of a model. Typically, when a model is first built, an initialization test case is run to bring the model to the initial state used for all other test cases. This initial state is recorded in a checkpoint. After that, whenever a test case is run, the model is first restored to this initial checkpoint. This can be much faster than going through the reset process before each test case.

Another use of checkpoint is to record the intermediate state of the model during a long-running simulation. By taking a checkpoint every N cycles, debugging can be easier if a problem is discovered. The simulation can be restarted from the last checkpoint taken prior to the failure to enable detailed debugging. It is also possible to transfer checkpoints between simulators. Thus, if a problem is found while running on the EVE simulator, the last checkpoint can be converted to a ZFS checkpoint and transferred to the ZFS simulator for debugging. This has proved very useful, since debugging on ZFS is much easier than on EVE. Also, EVE is a limited and valuable resource; debugging on a software simulator such as ZFS means that the EVE resource is available to do what it does best—high-speed simulation.

• *Message logging*

SimAPI provides its own log file for recording messages and a function, `spi_simlogf`, to format and log messages. This is independent of any log file that the simulator itself may generate.

• *Random-number generation*

Randomness must often be introduced into simulation to improve test coverage or to randomly inject errors. At the same time, any randomness must be repeatable in case an error is detected, to permit debugging the problem. SimAPI provides a random-number generator that creates a new random number each cycle. An initial seed is used to start the random-number

generator, and the sequence of random numbers is repeatable, given the seed.

- *Global data*

The `spi_entry` function allows programs to operate on global data by name; `spi_entry` maintains a table of eight-character names and four-byte values. Any thread can set or get the value for a named variable. SimAPI defines some built-in entry variables. For example, the variable APICYCLE contains the current cycle number.

Limitations

SimAPI is designed for use with cycle simulators. It can be used with event simulators as long as the simulator can be made to look like a cycle simulator. This was done with the VHDL event simulator used for S/390 G4 by passing the cycle time to the simulator shell. So, when SimAPI told the simulator to run a cycle, this was translated by the shell to a certain number of nanoseconds. However, SimAPI has no functions to directly control time.

SimAPI currently limits the names of objects to 256 characters. However, this limit could easily be changed if longer object names were needed.

SimAPI usage on S/390 G4

- *Processor chip verification*

Environment

The processor chip level simulation was introduced in an earlier paper in this journal [1]. The model consisted of a single processor, with a program representing the L2 chip. The primary stimulus was a set of architectural verification programs (AVPs) generated by AVPGEN [2]. An AVP is a test consisting of initial values for architected S/390 registers, an instruction stream, storage operands, and the expected results of storage and registers after the instructions are executed.

The simulation environment for processor verification used SimAPI extensively. Both ZFS and the VHDL event simulator were used for performing the simulation. SimAPI made the choice of simulator essentially transparent.

In order to control the simulation environment for the processor, a run-time manager (RTM) was created. This was a Rexx exec with SimAPI commands for establishing the overall flow of control: initialization, event detection, thread control, and termination. The RTM called other C programs to do many of these functions.

Initialization

Two functions were implemented to do the model initialization: one for internal registers and one for architected (S/390) registers. The internal registers were initialized by using reset logic built into the hardware (for "power-on reset" support). This required setting some internal reset signals in each of the units via the `spi_set_object` SimAPI function, and clocking the model several cycles using the `spi_clock` SimAPI function. The architected registers were set to values specified as initial values in the AVP, using the `spi_set_object` function.

Aliases

During the design process, register names can often change (e.g., because of a change in hierarchy in the design). To avoid having to change our programs whenever these name changes occurred, we made use of "alias" support in SimAPI. Aliases provide an association of logical names with real names. The SimAPI command `spi_map_alias` was used to establish the aliases. All of the mappings were done in one command file, which facilitated rapid changes.

The object names for a given register can vary among simulators. To make these name differences transparent to our programs, an alias file was created for each simulator.

Aliases were also used to map a single logical register to multiple actual registers. This feature was used to initialize certain architected registers (e.g., control registers), where copies existed in multiple units in the processor. A single `spi_set_object` command, using an alias name, initialized all of the copies.

Checkpointing

Whenever a new processor model was built, a reset state checkpoint was created. This was done for performance reasons. Since each AVP starts from the same initial state, the use of a checkpoint eliminated much of the initialization overhead (i.e., the simulation cycles needed to reset internal registers). To create the checkpoint, the SimAPI `spi_checkpoint` function was used. The checkpoint was created after the internal registers were initialized, but before the architected registers were initialized. At the start of each AVP, the model was reset to the checkpoint state via the `spi_restart` function, and the architected register initialization was then performed.

Events

As part of the processor simulation environment, many events were defined. When the events were detected, various actions were taken. Each of the events was declared in the RTM. Some examples are the following:

1. *AVPBegin* This was an event set when a new AVP was about to begin. It triggered the program which performed the architected initialization for the test case.

2. *MilliCheckstop* This event was caused when the millicode set a particular register after detecting an illegal condition. The event caused a message to be logged and the test case to be terminated.
3. *DecodeSuccess* Whenever an instruction was successfully decoded in the I-unit, this event was set. A program was then (optionally) called to put an entry in a trace file, which was used for debug purposes.
4. *ZeroPSW* This was an event set whenever a program status word (PSW) with a value of all zeros was detected. This signaled the end of an AVP.

Multiple threads

To support the stand-alone processor chip simulation environment, two programs were written to represent the L2 and the MBA chips. The L2 program provided a behavioral simulation for the second-level cache and memory hierarchy, and the MBA program provided the timing facility reference clocks. Each of these programs was run as a separate thread, allowing them to run simultaneously with the actual simulation. The `spi_create_thread` SimAPI command was used in the RTM to establish the multiple threads.

Termination

The end of a test case was typically detected with a ZeroPSW event (an indication that an interrupt occurred), or with an error condition (e.g., the MilliCheckstop event). A program was then called which retrieved the final state of the architected registers and compared those values to the expected values in the test case.

- *L2 chip verification*

Environment

The L2 chip-level simulation used a model containing the hardware model of the L2 with drivers and monitors attached to the CP and BSN interface ports. For the most part, the drivers and monitors were C++ programs that shared information. An exception to this was the BSN driver, which may have been some combination of VHDL macro or hardware model for its three components, BSN, STC, and memory. In all cases, a C++ driver representing a remote L2 or MBA was talking to the BSN interface ports not connected to the real L2.

As in the processor chip verification, the L2 simulation used SimAPI services to provide portability across simulator platforms. In the early stages of the project, SimAPI was not available, and two separate compiles were required to support the simulators in use at that time, a VHDL event simulator and TEXSIM. By using SimAPI, management of the executables for the different simulators was simplified, since only one version of the compiled code was required, and the ability to use the ZFS simulator was then available at no extra cost.

Besides the portability between simulators, SimAPI offered a number of other conveniences which were used in the L2 environment. These included language portability, control events, permanent get-object lists, object alias files, and multithreading of special test cases that ran along with the main environment.

Language portability

While the majority of the L2 verification was written in C++, programs that did environment initialization before the C++ code was started were written in Rexx. These had all of the SimAPI functions available to them, yet remained independent of the simulator being used. The two most used were the RTM, responsible for loading and running the main L2 environment, and the SIMINIT file, which processed a list of objects and initial values to initialize latch objects. (There was no reset on the L2, and scanning was used to initialize the latches, so the SIMINIT provided a quick shortcut for initialization.)

Control events

For some of the simulators that were used for L2 verification, a performance penalty was incurred when the getting and setting of objects were intermixed. This was because the simulator would reevaluate some or all of the model when an attempt was made to get an object after one or more objects had been set in the same cycle.

To avoid this situation, the cycle was managed as having two times within it, a "begin cycle" and an "end cycle." The setting of objects was done at the beginning of a cycle, and the getting of object values was done at the end of the cycle. To effect this management, SimAPI provided the `spi_wait` function and predefined `BEGIN_CYCLE` and `END_CYCLE` events. The call to `spi_wait` was handled correctly for simulators that support this notion, and took the appropriate action for simulators that did not.

Permanent get-object lists

In the L2 chip verification, checking correct operation required monitoring a substantial number of objects on the interfaces and internally by getting their values from the simulators. Therefore, in the interest of execution performance it was important to make these accesses as fast as possible. SimAPI provided a mechanism, the permanent list, that could be used for obtaining all of the values for objects which were to be fetched a number of times. The L2 used only one list that was executed every cycle, but multiple lists were possible.

The overhead of building the list was incurred only once, and the optimization of getting values for a number of objects was contained within SimAPI. Because SimAPI

handled the entire list at once, the overhead of multiple calls to SimAPI was removed. Another benefit was the ability to exploit special features of a given simulator that allowed for accelerated access to object values, such as the z-shadow function of ZFS. These features were easily accessible because SimAPI supported them.

Object aliases

Because the verification code had to work with several different configurations of the L2 model over the life of the entire project, the names of referenced objects were likely to change quite often. To avoid having to maintain separate versions of the verification code, it was important to remain as flexible as possible on this point. Two things were used to increase flexibility in this area.

Soft-coded object names, which were not a function of SimAPI, were used to support hierarchy differences encountered in the various configurations used. However, this did not address the problems caused by partitioning arrays which arose when physical layout became a serious factor.

SimAPI aliasing allowed the verification code to remain unchanged and supported the one-cycle and two-cycle models. The two-cycle model was closely tied to the physical layout, and the arrays were split and named accordingly, whereas the one-cycle model had only to be functionally correct, and a few large arrays were used for the cache data array. Alias objects were flexible enough to allow the twelve smaller arrays of the two-cycle model to be mapped to the same name and addressability as the single large array used in the one-cycle model. This was accomplished by running a SimAPI Rexx program from the RTM at cycle 0.

Multithreading

Although L2 verification was based on randomness, it was often necessary to create a specific event or sequence of events. This was most often required for debugging and verifying problems that were found at the system level or on the test floor.

The random environment was generally capable of creating the necessary conditions, but needed some fine tuning in some instances. This was accomplished by using another SimAPI test case running at the same time as the random environment. These test cases were used to inject errors or commands at specific times, or to manipulate internal objects to obtain a desired action. Because SimAPI was also available to programs written in Rexx, an interpreted language, these test-case threads were quickly created and tuned in an interactive way, which was very useful when trying to pin down an exact timing, by trial and error, with very little information.

- *System verification*

Environment

System verification was introduced in an earlier paper in this journal [1]. The model consisted of multiple processor chips, L2 chips, bus control chips, memory control/memory adapter chips, and main memory. S/390 instruction streams (generated by SAK) were executed on the processors to verify the implementation of the system from an architectural perspective. EVE 1.5 (a hardware accelerator) was used to provide greater throughput than is achievable on software simulators running large models. EVE 1.5 executed hundreds of simulation cycles each second, in contrast to the same model on a software simulator running 10–15 cycles per second. A single simulation run executed millions of cycles in a reasonable amount of time. A drawback to EVE 1.5 is its limited capability to gather trace information about the state of the model, which made problem debug cumbersome and slow. ZFS (a software simulator) has the ability to generate full traces of the model state and was used for model bring-up and problem debug.

SimAPI was used extensively in the system environment and allowed one set of programs (written in C and Rexx) to interact seamlessly with models on either simulator. Many of the SimAPI features used at the processor and L2 verification levels were also used at the system level. The run-time environment was controlled by an RTM, which has been described in the processor and L2 chip verification sections. The `Needwork` event was used to initiate data transfers between the simulation model and the SAK host system when a modeled processor had completed its tasks. Events declared in the RTM behaved identically on EVE 1.5 and ZFS from a user perspective. Some differences or unique applications are discussed below.

Checkpointing

Checkpoints [3] were taken periodically (typically at one million cycles) while running simulation on an EVE, capturing not only the state of the EVE model but also all of the data transfers that took place between checkpoints. This enabled failures to be restarted on EVE 1.5 in a relatively small window of simulation cycles without having to return to the starting point, as runs of 1 to 40 million or more cycles until a failure was detected were observed. A subsequent checkpoint was then taken in close proximity to the failure. The new EVE 1.5 checkpoint was converted seamlessly to a checkpoint for the ZFS software simulator. After the conversion, the user specified the ZFS simulator instead of EVE 1.5 and made a slight modification to the RTM which enabled full trace. Full traces for debug were generated during the ZFS

simulation runs which matched the EVE 1.5 runs on a cycle-by-cycle basis.

Data buffering

The `spi_flush_list` function allows users to control the point at which a data transfer is actually propagated to the simulation model. The user sets bit 0 of the flags parameter of a SimAPI function such as `spi_set_object` to a '0'. This tells SimAPI to buffer this request. SimAPI holds the request until an explicit `spi_flush_list` function executes or until its buffer is full. Request(s) are then propagated. The following paragraph describes a use of data buffering.

SAK test cases were moved between the simulation model's memory and the SAK host via a program known as Memmove. When data movement occurred to or from the simulation model, simulation was stopped. The data were transferred between an S/390 EVE host and the EVE 1.5 hardware via an S/390 channel connection and placed in a memory object (array) or taken from a memory object by the EVE's I/O processor. A transfer of a single line of data (128 bytes) required 161 calls (or more) to `spi_set_object` to ensure that the storage hierarchy was properly updated. Large data transfers resulted in thousands of requests being generated. Each request was a unique S/390 I/O request, which is a relatively slow process. So much time was consumed completing all of the requests that if each request had been propagated uniquely, significantly more time would have been spent transferring data than running simulation. (This phenomenon has been observed at times as a result of coding errors.) If Memmove were coded in this manner, software simulators would actually provide higher cycle throughput than the hardware accelerator, wasting the speed advantage of EVE 1.5. To solve this problem, Memmove took advantage of the buffering capability of SimAPI when transferring data. Efficiency ratios (percent of the time EVE is actually running cycles) of 70–90% have been observed.

Summary

As the number of cycle simulators available continues to grow, the number of programming interfaces to simulation has also been growing. It has become very difficult to write code to run on multiple simulators. SimAPI solves this problem by providing a common high-level interface to cycle simulation. By also defining a common low-level interface to a simulator, SimAPI makes it to easy to run new simulators. It has been used with very good performance with such widely different simulators as a VHDL event simulator and a hardware accelerator. The functions available from SimAPI have proved useful for all levels of simulation from the designer level up to and including system-level models.

References

1. B. Wile, M. P. Mullen, C. Hanson, D. G. Bair, K. M. Lasko, P. J. Duffy, E. J. Kaminski, Jr., T. E. Gilbert, S. M. Licker, R. G. Sheldon, W. D. Wollyung, W. J. Lewis, and R. J. Adkins, "Functional Verification of the S/390 Parallel Enterprise Server G4 System," *IBM J. Res. Develop.* **41,** No. 4/5, 549–566 (1997, this issue).
2. A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "AVPGEN—A Test Generator for Architecture Verification," *IEEE Trans. VLSI Syst.* **3,** No. 2, 188 (June 1995).
3. Dennis F. Ackerman, David R. Bender, Salina S. Chu, George R. Deibert, Gary G. Hallock, David E. Lackey, Robert G. Sheldon, and Thomas A. Stranko, "Logic Simulation Using a Hardware Accelerator Together with an Automated Error Event Isolation and Trace Facility," U.S. Patent 5,146,460, 1992.

Received December 10, 1996; accepted for publication July 11, 1997

Gary G. Hallock *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (ghallock@vnet.ibm.com).* Mr. Hallock joined IBM in 1978 and is currently an Advisory Engineer. He has worked on the development of various simulation tools including SimAPI, the ZFS cycle simulator, and co-simulation. He received his B.S. in electrical engineering from Rensselaer Polytechnic Institute in 1976. Mr. Hallock received an IBM Outstanding Innovation Award in 1990, an IBM Outstanding Technical Achievement Award in 1991 for his work on SimAPI, and an IBM Division Award in 1995 for his work on co-simulation. He is a member of the IEEE.

Edward J. Kaminski, Jr. *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (eddiek@vnet.ibm.com).* Mr. Kaminski received a B.S. in electrical engineering from Rensselaer Polytechnic Institute in 1987, joining IBM that same year. He is currently a Staff Engineer, and has worked on verification of shared L2 cache and system controller elements of the IBM S/390 systems: 6-way, 8-way, and 10-way IBM ES/9000, and S/390 G4. Mr. Kaminski received IBM Team Awards for his work on the 10-way IBM ES/9000 in 1993 and S/390 G3 common chip verification in 1996, and an IBM Invention Achievement Award for his work on the TIMEDIAG/GENRAND tools in 1995.

Kevin M. Lasko *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (km_lasko@vnet.ibm.com).* Mr. Lasko is currently an Advisory Engineer in S/390 simulation, performing system verification of the IBM S/390 G4 system. He joined IBM in 1978 in Subproducts Manufacturing Engineering. Since 1981 he has simulated various S/390 systems at the element and system level. Mr. Lasko received a B.S. in electrical engineering from Union College in 1978 and an M.S. in computer engineering from Syracuse University in 1983. He received an IBM Outstanding Technical Achievement Award in 1987 for his work on random test-case development for element simulation of the 3090 storage controller. In 1996 he received an IBM Team Award for his work on S/390 G3 common chip verification.

Michael P. Mullen *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (mmullen@vnet.ibm.com).* Mr. Mullen is currently a Senior Programmer; he joined IBM in 1976. He received a B.S. degree in computer science from Union College in 1976, and an M.S. degree in computer/information sciences from Syracuse University in 1981. Mr. Mullen has worked on the development of several mainframe systems, and is currently responsible for the hardware design verification of IBM S/390 CMOS processors. He received IBM Outstanding Technical Achievement Awards for his work on the IBM 3090 processor controller microcode (1985), ES/9000 processor simulation (1991), and AVPGEN development (1994).