# SALT: The Simulator for the Analysis of LWP Timing

Paul Springer, Arun Rodrigues, Jay Brockman

March 13, 2006

**Abstract**

With the emergence of new processor architectures that are highly multi-threaded, and support features such as full/empty memory semantics and split-phase memory transactions, the need for a processor simulator to handle these features becomes apparent. This paper describes such a simulator, called SALT.

## 1 Introduction

In the design of future high performance computers, one of the main obstacles to overcome is the processor to memory bottleneck. One approach that is receiving increasing attention is to use memory that has computing capability built into it. This kind of architecture is referred to as *processing-in-memory*, or PIM, and has been used in a number of recent projects[4]. SALT simulates most of the features found in the PIM Lite chip, as well as extended memory semantics to support tagged memory using full/empty bits, similar to that used by the Cray MTA. Unique aspects of this architecture are highlighted below.

*1) Hardware Parallelism:* The target hardware simulated by SALT consists of multiple PIM chips, each of which contains a small number of Lightweight Processors (LWPs) together with memory.

*2) Lightweight Multithreading:* At one end of the spectrum of multithreading lies Unix style pthreads, which have large amounts of state and high overheads in thread synchronization, forking and scheduling activities. By contrast, lightweight threads can be forked in just a few cycles, scheduled in a single cycle, and their states can be encapsulated by a single register set.

*3) Frames:* Each thread in the target architecture has an associated dedicated register set that carries its state. This register set is called a frame, and is physically part of the memory on the PIM chip. In our simulation each frame consists of 32 64-bit registers.

*4) Distributed Shared Memory:* Memory is distributed evenly among the PIM chips being simulated. Within a chip, LWPs share on-chip memory in the sense that each one has the same view of and access to the local memory as well as the remote memory.

*5) Parcels:* Communication between PIM chips is done using specialized packets called *parcels* (parallel communication elements). Parcels support remote reads and writes, as well as spawning and atomic memory operations to remote memory.

*6) Locality Awareness:* Whether or not memory is local to a chip is something that is visible all the way up to the application level, allowing an application at run time to make decisions about data distribution and load balancing. The architecture causes threads to be spawned to an LWP for which a specified memory location is local.

*7) Tagged Memory:* In the target architecture, each double word of memory includes an extra bit, called an *extension bit*. The state of the extension bit affects how the memory is handled by the processor when reads or writes are targeted to that memory location. A set of extended memory semantics determines what the outcome of the requested operation will be.

*8) Split Phase Memory Transactions:* All memory accesses are done by means of transactions that are split into two phases: a request phase and a response phase. For a read request, the value is returned as a response. The response to a write request is an acknowledgment that is returned to a designated acknowledgment register. This is similar to the functionality built in to the Split-C compiler[2].

With the unique hardware capabilities necessary to simulate, as well as the need for flexibility in the front-end instruction input stream, we decided to develop our own simulator. To enable the user to run in either a functional mode or a cycle-accurate mode, SALT was built on top of a low overhead discrete event simulation engine, Enkidu. This engine is detailed in section 3.6. The memory controller is described in section 4.4, and the design of the front-end in section 3.4. Section 3.5 covers the back-end performance model.

As of the writing of this paper, the functional version of SALT has been completed, but not the cycle-accurate version. This paper will focus primarily on the former, though some mention will be made of the latter. The MTA-style front-end currently has the most functionality programmed into it, which is why the examples included here refer to it.

# 2 Related Work

SPIM is a widely used program constructed to simulate the MIPS processor[5]. It proved to be a valuable teaching tool for this project, and some of its parsing code was incorporated into SALT. SuperScalar is another software package often used as a basis for simulating various kinds of processors[1]. SuperScalar is particular efficient in decoding instructions to be executed by the processor, and we took the same approach of using a large table of macros, one for each instruction, that SuperScalar uses.

The Zebra simulator at Cray was built to simulate the MTA computer, as part of the design process. Zebra simulates multiple processors with multiple threads running on each processor, as well as full/empty bit memory semantics[6].

# 3 Simulator Design

## 3.1 Challenges

The hardware being simulated motivated the addition of features into SALT that are not found in most other processor simulators. The parallelism of the hardware requires SALT to handle not just one processor, but many interacting with each other. Multiple threads have to be supported on each of the parallel LWPs, together with a means for synchronizing and scheduling those threads. Multiple registers sets, one for each thread, have to be allocated appropriately out of memory, associated with the thread, and deallocated once the thread completed. The simulator must handle parcel communication between PIM chips, and must be aware of what memory is assigned to what chip. Probably the most complex part of SALT is the handling of the extended memory semantics, with extra tag bits for each memory location, and splitting up a memory transaction into response and reply phases that may be widely separated in time.

An additional challenge for the design of SALT came out of our motivation to support the system architects in making design tradeoffs. As the design changed, SALT needed to change as quickly and easily as possible. This necessitated encapsulating major parts of the simulator, so that one part could be changed without the change rippling through to other parts of SALT.

## 3.2 Architecture

The overall architecture of SALT is shown in figure 1. The operational parameters of the simulation are controlled by a configuration file. The particular front-end feeder to be used for input is selected by a configuration parameter. The back-end components of the performance model are invoked by the program stream coming through the selected front-end, and the Enkidu engine handles communications and models time between the components. The simulator can be run in a multi-processor mode, with the number of processors specified in the configuration file. In this mode, SALT can assign multiple LWPs to an chip, and can also simulate a system with multiple chips, with memory partitioned between chips, and shared among the LWPs on a single chip. SALT is written to handle the interactions between these components.

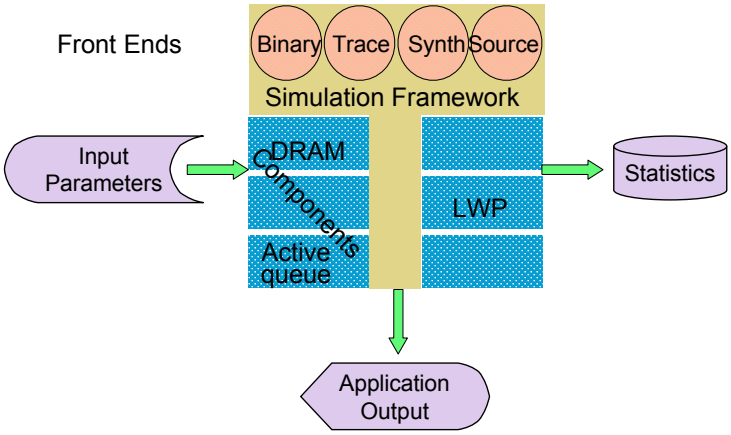


Figure 1: **SALT Architecture**

## 3.3 Configuration Input

At the outset of a simulation run, the configuration file is read by SALT, and configuration parameters are set up. This approach was taken to maintain a high degree of flexibility in running SALT, so that a large parameter space can be explored (in real time or in batch mode) without recompiling SALT. The parameters can be roughly

grouped into the four areas of interest they control: front-end control, target topology, simulation behavior, and statistical output. The front-end control parameters specify which feeder SALT will use for program stream processing, as well as the file name containing the program stream. The target topology parameters control characteristics of the target topology such as the number of LWPs per PIM chip, total number of chips, and memory size and distribution. Other parameters control whether the simulation will run in functional mode or timing-accurate mode, and how much debugging and statistical output will be produced.

## 3.4 Front End Feeders

In order to cleanly separate the performance model of the simulation from the instruction execution, SALT is structured so that feeder modules read the program stream file, and communicate to the performance model through a defined interface[3]. Feeder modules have been written for MIPS-like assembly source, MTA-like assembly and ELF binary. The simulator design is flexible enough to work with a trace file type of feeder as well.

The strategy of using feeders as part of the architecture worked very well for us. After the binary feeder was developed we realized that the new processor architecture being simulated would require additions to the MIPS instruction set we originally supported. The encoding of the new instruction set would not be worked out for some time, so that support of assembly source allowed the simulation to proceed without having to wait. Later on in the project the decision was made to switch from a MIPS flavored instruction set to an MTA style one. A new MTA feeder was written, and very few modifications were required to the rest of SALT to support this new feeder. Of course any new instruction set that depends of different architectural characteristics, for example, a different number of registers, or a different register size, will require changes made to the back-end performance model.

## 3.5 Back End Performance Model

Interactions between the front-end and back-end portions of SALT are illustrated by figure 2. Thread objects (in the front-end) and processor objects (in the back-end) each have public interfaces. Encapsulating each object makes it easy to move a thread from one processor to another in the course of the simulation, or to have one processor run multiple threads. Examining some of the steps involved in executing a load instruction will serve as a simplified illustration of how the different pieces interact.

A processor first selects a thread to activate. The activated thread's FetchInstruction() routine is then called by the processor, followed by the IssueInstruction() and then CommitInstruction() thread routines. The latter routine in turn may call upon the processor's ReadMem() routine to get the contents of a memory location. The processor sends a request to the memory's Read() routine, which in turn call's the thread's SetRegister() routine to store the memory contents into a register.

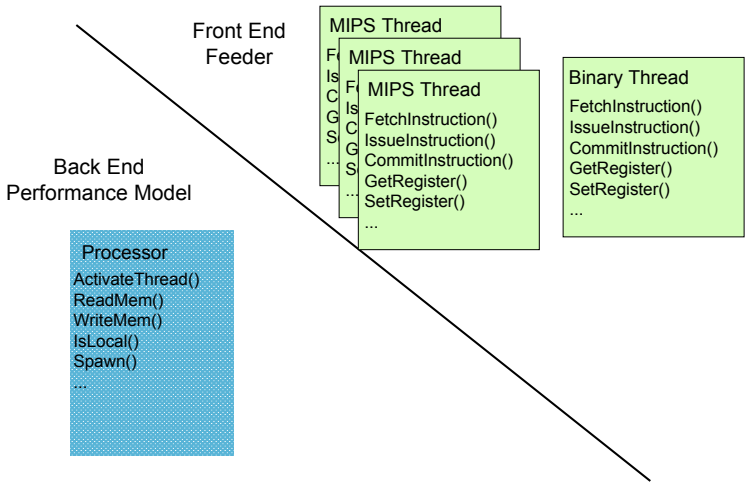


Figure 2: **SALT Internal Interface**

## 3.6 Enkidu Simulation Engine

The underlying framework for the SALT simulator is Enkidu, a component-based hybrid simulation engine. In Enkidu, components represent the various physical components of the system. In the original version of Enkidu, every component is evaluated every clock cycle, allowing the component to advance its internal state. The components interact by passing event notifiers to each other through a discrete event framework. As modified for SALT, Enkidu skips over time steps for which no events are waiting, which provides the efficiency of a discrete event model.

Modern processors can be represented as a series of buffers which store instructions and data, separated by logic which acts upon those instructions. Data flows from buffer to buffer according to a strict centralized clock. For processor architectural simulation, it is possible to say that all events take place in synchronization

with this clock. The processors simulated by SALT can have dozens of threads in various stages of execution during each processor clock cycle. As a result, more than one transition event can occur each cycle.

Enkidu defines a component class, which is meant to be a base class for application objects that need any functionality that is to be provided by the simulation engine. Any such application object can be scheduled for running by Enkidu, and will be able to call Enkidu routines for sending and receiving messages.

# 4 Functionality

## 4.1 Instruction Interpretation

The front-end currently being used, the MTA-style assembly source front-end, uses a large macro table to implement the body of the switch statement used in executing the different instructions. One case statement exists for each instruction, and these are contained in a single file, similar to the way the SimpleScalar processor modeling software handles this. This approach benefits both the performance of the simulation as well as the flexibility and cleanness of the code, though at the cost of making debugging of this part of the code more difficult. The macro table is referenced in both the issue cycle and commit cycle, and by means of secondary macros invoked by the primary macros, the primary macros have a different functionality in the two cycles.

Each entry in the macro table consists of a number of fields, of which only certain ones are used in a given cycle. For example, before an instruction can be initiated, the registers it uses must be in a certain state with regards to their full/empty bits. The macro used to check the required state for each register is inserted into one of five possible condition fields available in the table entry for that instruction.

## 4.2 Syscall Support

Experience with the MIPS-style front-end motivated us to include the SYSCALL (system call) interface into the MTA-style front-end as well. To support that functionality, a SYSCALL instruction is defined in the instruction macro table. The contents of register 2 determine which system call is invoked. A single argument (if required) is passed in register 4, and any return value is put into register 2. With no system or library code for the simulator to execute, it has built-in support for a limited number of SYSCALLs. These include routines to print a string or to print a number in various formats, and to generate a random number. The SYSCALL

routines have proved very useful in lieu of library support code for printf and related functions, to help in application debugging. Application I/O support is not yet built in but is high on the priority list of future enhancements.

## 4.3 Threads and Frames

Lightweight threads are supported by the target processor directly in hardware. Because of the need for high performance handling of these threads as well as direct control over the scheduling and priority algorithms for them, they could not be mapped into a POSIX thread library, and instead support for them was built directly into the simulator.

Each thread has its own set of registers, in a block called a frame, and each frame is mapped to a location in memory. SALT handles the allocation of memory (from a frame pool) for frame usage, and also assigns frames to new threads, and frees frames when threads are deleted. Eventually most of this functionality is expected to be supported by the hardware.

Each thread is a C++ class, and a unique thread class exists for each front-end. All of these front-end thread classes inherit from a thread base class. The base class contains support for such common functions as frame assignment, and register access. The derived class for each front-end contains the support for fetching an instruction, forking a new thread, and thread object initialization.

A single thread is initially created by SALT once a program is loaded into memory. That thread begins execution at the assigned start location. Other threads can be created either explicitly by means of fork or spawn instructions, or implicitly by certain events such as access of memory locations that are in certain states. One type of implicitly generated thread is the handler thread, used to supply added functionality to the hardware.

A processor object is responsible for selecting which of the threads assigned to it should execute. Upon selecting the thread, it gives control to the thread, allowing it to execute a single instruction. Once the instruction is executed, control is relinquished back to the processor, which then chooses the next thread for execution.

## 4.4 Memory Controller

A primary concern of the processor architects was to hide memory latency. This desire helped motivate the split-phase memory design that resulted. In this design, threads reading from or writing to memory do not have to wait for the memory operation to complete, unless it is necessary to do so. This allows thread execution

and memory latency to proceed in parallel.

The architecture is designed so that all memory operations send back an acknowledgment signal when the operation has completed. In the case of a load operation, the data from the memory acts as the acknowledgment signal. For a store operation, a special acknowledgment signal is sent to the register designated by the instruction to receive that signal. The requesting thread does not know or care whether the memory operation has completed, until the acknowledgment register is accessed by an instruction. At this point, if the acknowledgment has not been received, the thread is removed from the queue of active threads, and blocked until the memory operation completes, at which time it is returned to the active queue. To maintain consistency with this paradigm, all memory operations are between registers and memory; there are no memory to memory operations.

These requirements mandate a smart memory controller, one that is capable of receiving requests, sending results and acknowledgments back to thread registers, and moving threads on and off the active queue. The memory controller must also be capable of understanding the concept of local and remote memory. If the memory address of the request is mapped to memory that is local to this chip's memory controller, it executes the request; otherwise the request is forwarded to an external controller. Similarly, the memory controller must also be able to respond to requests from threads running on remote chips that were forwarded by that thread's remote memory controller.

The split phase memory transactions greatly simplify the interface between threads and memory. The asynchronous interface between the two is clean–threads don't run an indetermine number of cycles before blocking–they only block (and block immediately) when the response register is accessed by a thread instruction. The extent to which a thread can continue to execute following a memory operation depends only on whether the response has arrived by the time the register is accessed.

In functional mode if an instruction needs to access memory, the instruction execution code calls the memory controller routines directly, as needed, to handle load, store, fork, or AMO instructions. In this case, for purposes of speed, the simulation engine is bypassed. Once invoked, the appropriate memory controller routine checks to see if the targeted memory location is local to the chip the thread is operating on. If not, the controller creates a message with the contents of this memory request, and passes the message to Enkidu for delivery to the appropriate memory controller. Enkidu will deliver the message and invoke the receiving controller.

After determining that the memory location targeted by the request is local, the responding memory controller routine checks the state of the memory target to determine how it should respond. Some requests may place a precondition on the

memory state, and this must be taken into account as well. If the state of the memory is appropriate for the request, the request is fulfilled. If the request was in response to a load instruction, the memory value is returned to the appropriate register of the requesting thread. If a store instruction was executed, memory is modified and an acknowledgment is returned to the designated acknowledge register. If an AMO operation was executed, memory is modifed and a value returned to a register. In the case of a request for a fork, a new thread is created, register contents are copied into it, the new thread is put onto the active thread queue, and an acknowledgment is sent back to the designated register of the requesting thread.

Read, write, or AMO operations will not themselves block threads. In response to these types of requests, the memory controller will return a value or acknowledgment back to the designated register of the requesting thread, or an indicator to that register (using the full/empty bit of the register) that the operation can't be completed immediately. In the cycle accurate mode this can happen if there is a cache miss and the memory can not be accessed in a single cycle. In either the cycle accurate or functional mode the operation may not complete if, for example, a load instruction was executed against an empty memory locaion. In any case, before a thread executes the next or any subsequent instruction, the registers needed by the new instruction are checked, and at that time if the register state indicates an uncompleted operation is pending, the thread is then removed from the active queue.

This prompts the question of how the thread is tracked once it is no longer in the active thread queue, and how it is reactivated. In the cycle accurate case, where a cache miss has occurred, the original memory request is forwarded to the appropriate module at the proper time. The request packet contains all necessary information, including information about which thread made the request. The responding module then generates a response packet which will put the requested value or acknowledge indicator into the requesting register, and restore the thread to the active queue. Both the architecture and the simulator make use of the contents of the request parcel to find the appropriate thread.

A thread can also be removed from the active thread queue if a memory location is not in the state that is expected by an instruction making a memory request. In the simplest case of this, when a single load instruction needs to wait until the target memory location has a value stored into it, a pointer to the thread is stored in the memory location by the hardware architecture. For the sake of efficiency, the simulator stores threads in an STL map container, which associates the thread object with the address of its frame, for easy retrieval when needed.

The more difficult case occurs when several threads are all waiting for a memory location to change state. As only one thread address can be stored in a memory

location, this case can not be handled in an obvious way. This difficulty is resolved by forking a new thread which begins execution of special purpose handler code.

## 4.5 Cycle Accurate Memory Access

When SALT is operating in functional mode, the code is structured so that each PIM chip has a single memory controller to handle the memory requests coming from the LWP processors on that chip, and there is no simulation of cache. In contrast, cycle accurate mode allows for a much more complex memory structure, with multiple cache banks assigned to a PIM chip, a memory controller assigned to each cache bank, and off-chip DRAM memory. To facilitate this, two interface classes were created, a memoryIF class to handle memory reads and writes, and a controllerIF class to handle controller memory requests. The cache and memory classes inherit from the memoryIF class, and the memory controller class inherits from the controllerIF class. The other class derived from the controllerIF class is a pimRouter class. As derivative classes of the controllerIF class, both controller and pimRouter objects can handle requests from the processors, and then use the memoryIF class to implement reads and writes to memory.

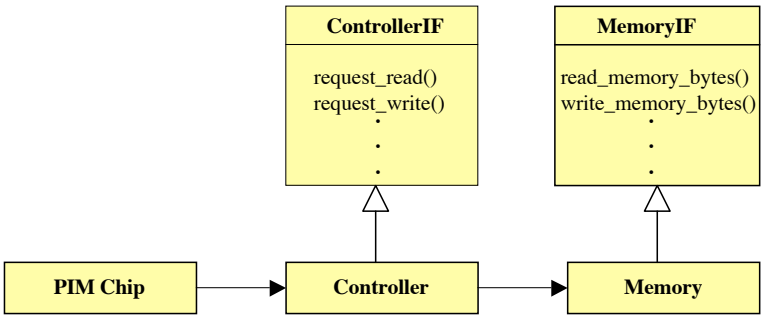


Figure 3: **Memory Model: Functional Mode**

SALT dynamically assigns a controllerIF-derived object to each PIM chip, to handle memory requests for that chip. In functional mode that assignment is simply a memory controller class, that in turn communicates to a memory object that has been assigned to that controller (figure 3). In cycle-accurate mode, an pimRouter object is assigned to the chip, and multiple controllers (one for each bank of cache) are assigned to the pimRouter. When the processor issues a memory request, the request is forwarded by the pimRouter to the appropriate memory controller, which in turn

issues a read or write to its assigned cache bank, using the memory interface. The cache bank communicates to DRAM as necessary, again using the memory interface (figure 4).

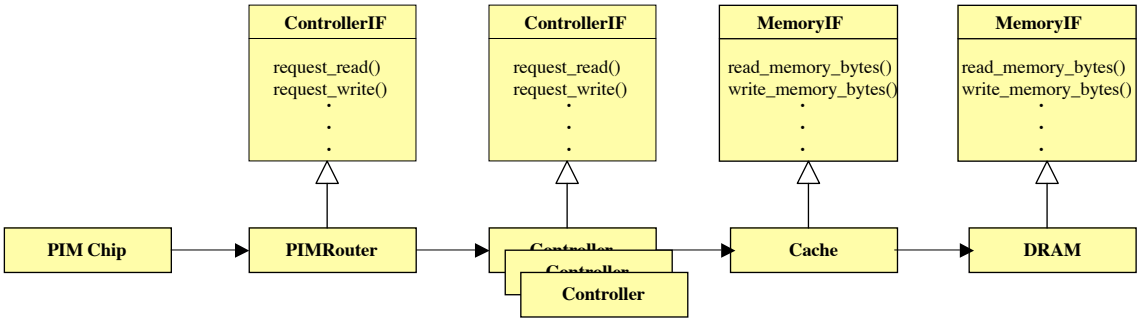


Figure 4: **Memory Model: Cycle-Accurate Mode**

The controllerIF-derived classes inherit from Enkidu's component class, and so are able to send and receive messages, and can be scheduled by the simulation engine. Because of the split-phase memory transactions, the interface is by its nature asynchronous, and the processor making the memory request must handle an asynchronous response whether running in functional or cycle-accurate mode. In the current version of SALT, calls between a controller and memory, by contrast, are handled as simple subroutine calls with a return value, consistent with the functional model. As development proceeds on SALT, the memoryIF-derived classes will be modified to be event driven.

## 4.6 Memory Partitioning

In the current version of the software, there is no implementation of virtual memory. All memory references are treated as physical addresses. The first 256MB of simulated memory space is divided evenly among the number of PIM chips requested for the simulation (see figure 5). Memory above 256MB is treated as a special shared data segment, that is considered local read-only memory by every chip. It is used to store data constants that come in through the program stream, and can be initialized by the front-end code.

The lowest 12K of local memory on a chip is currently unassigned. The next 8K is reserved for frame usage. All unused frame slots are tied together by means of a linked list. When a new thread is created, a frame is allocated from the free list,

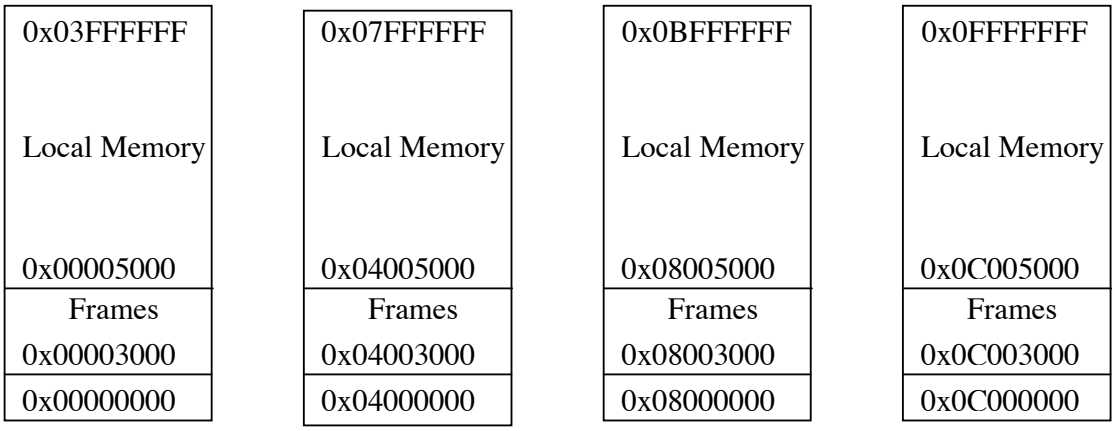


Figure 5: **Memory Partitioning for Four PIM Chips**

and when a thread is retired, its frame is returned to the head of the free list. The remainder of the local memory on the chip can be used by the application.

### 4.7 Simulator Output

SALT provides output at three levels. The first level is for application output, using the syscall mechanism, and is described elsewhere in this document. The next level of output is an instruction trace, useful for debugging the application. One line is output for each instruction executed. That line includes fields detailing the processor cycle number, the thread ID, program counter, source line number, and source line.

SALT also maintains statistical information internally, as directed by the configuration file parameters, and outputs that information into a file at the end of the run. Currently only a small number of statistics are output–those include one line for each clock cycle, detailing the PIM chip ID, the clock cycle count, and the number of active threads and total threads for that cycle.

## 5 Usefulness

SALT has been used and found to be useful for two different efforts so far. Because SALT uses as input an assembly instruction stream, it was decided to build a compiler for it. As the compiler was being developed, the simulator proved its worth in helping to debug the compiler.

SALT has also been put to use in fulfillment of its main purpose, namely to understand and analyze the operation and performance of the hardware architecture. A study has already been completed that uses statistical output from SALT to examine different thread synchronization strategies, and how execution time scales with the number of threads in use. Another effort is currently in progress that will use SALT to understand how well a neural network performs on this architecture in comparison to more traditional computer systems.

# 6 Future Work

In the near term we have plans to enhance development of the cycle-accurate part of SALT. As part of that, we intend to implement a full-fledged data cache module that includes an internal PIM chip bus for the cache banks on the chip. There are also plans to implement a frame cache, for which there are already some hooks in the current baseline.

Thread scheduling is currently very primitive, consisting of round-robin execution of threads. There has been some discussion of changing this, and perhaps allowing threads to execute with differing priorities.

Currently SALT decides what area of memory to use for frames, and sets up a linked list of free frames which can be allocated by threads. The ultimate intention is to add necessary instructions to allow the operating system to handle this task. Once that is accomplished, and a runtime module written to do this initialization, SALT will no longer need to initialize the frame memory.

Debugging is still fairly limited, with trace output and application output as the only available tools. We will probably want to add other capabilities, such as single-stepping, register and memory examination, active thread queue listing, and possibly breakpointing. These will combine to make it much easier to use this simulation tool.[1]

# References

[1] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[1]This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The funding for this research was provided for by the Defense Advanced Research Projects Agency under task order number NM0715612, under the NASA prime contract number NAS7-03001.

[2] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[3] Joel S. Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan L. Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, 2002.

[4] P. M. Kogge. EXECUBE - A new architecture for scalable MPPs. In Dharma P. Agrawal, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 1: Architecture*, pages 77–84, Boca Raton, FL, USA, August 1994. CRC Press.

[5] James R. Larus. SPIM S20: A MIPS R2000 SIMULATOR. Technical Report CS-TR-90-966, Computer Sciences Department, University of Wisconsin, Madison, WI, May 1990.

[6] Allan Porterfield. Private correspondence.