Escola Tècnica Superior d’Enginyeria Informàtica

Universitat Politècnica de València

PINT, HERRAMIENTA DE SIMULACIÓN BASADA EN TRAZAS PIN

Final Year Project

Computer engineering

Author: Francisco Blas Izquierdo Riera

Director: Julio Sahuquillo Borrás - UPV
Per Stenström - CTH
Abstract

In the course of this project we have developed a set of programs to improve the correction and execution time of the gem5 simulator.

For this, we moved the functional simulation step out of gem5 into an independent instrumented process to ensure correction in the functional stage and to provide a good execution speed (since the code will then be natively executed). This instrumentation is done by Pin.

Also, in order to allow efficient communication between the processes despite the limitations imposed by Pin to the available tools, an IPC framework to allow message passing between the processes was developed. This framework uses lockless fifo queues over shared memory so the resulting slowdown is minimal.

Keywords: hardware, simulator, x86, Pin, Pintool, gem5, ipc, fifo, C++
Contents

1 Introduction ........................................ 4
   1.1 Project rationale ................................... 4
   1.2 Project objectives .................................. 5
   1.3 Project strengths ................................... 5
   1.4 Memory structure ................................... 7

2 State of the art ...................................... 8
   2.1 Architectural simulators ............................... 8
      2.1.1 Graphite .......................................... 10
      2.1.2 Multi2Sim .......................................... 10
      2.1.3 gem5 ............................................. 11
   2.2 Instrumentation systems ............................... 11
      2.2.1 gprof ............................................ 12
      2.2.2 Pin ............................................. 12
   2.3 Benchmarks .......................................... 13
      2.3.1 SPEC CPU2006 .................................... 13
      2.3.2 SPLASH-2 ......................................... 14

3 System description ................................... 16
   3.1 Mead: a message passing framework ................. 16
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Pint: a Pin based trace generator</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Schnapps: a simple consumer of the traces</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Gin5: a gem5 trace player</td>
<td>20</td>
</tr>
<tr>
<td>4 System design</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Mead: a message passing framework</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Pint: a Pin based trace generator</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Schnapps: a simple consumer of the traces</td>
<td>25</td>
</tr>
<tr>
<td>4.4 Gin5: a gem5 trace player</td>
<td>26</td>
</tr>
<tr>
<td>5 Results</td>
<td>28</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>30</td>
</tr>
<tr>
<td>6.1 Improvements for next release</td>
<td>31</td>
</tr>
<tr>
<td>A User manual</td>
<td>32</td>
</tr>
<tr>
<td>A.1 Building</td>
<td>32</td>
</tr>
<tr>
<td>A.2 Pint</td>
<td>32</td>
</tr>
<tr>
<td>A.3 Schnapps</td>
<td>33</td>
</tr>
<tr>
<td>A.4 Gin5</td>
<td>33</td>
</tr>
<tr>
<td>B Relevant source code</td>
<td>34</td>
</tr>
<tr>
<td>Bibliography</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Project rationale

Despite the vast amount of hardware simulators that exist nowadays, most of them either lack flexibility on the simulations or are slow since they simulate the code execution instead of instrumenting the natively executed executable. As a related issue, since code is simulated and not executed it is common to find bugs where the simulator will not set the processor state properly which cause corner cases where not acting as the processor causes execution issues with some programs.

Also many simulators lack support for parallel execution and those who do tend to add big overheads when running the simulation in a single machine and will not support instruction level simulation granularity.

Finally current simulators tend to add big overheads to the functional simulation step which makes it unfeasible to run large tests even when simulating simple systems.

Program instrumentation solves all these shortcomings by running the code natively (modified so it will also execute the instrumentation code),
Chapter 1. Introduction

1.2. Project objectives

allowing it to run in parallel and, since code is executed natively in the processor, providing a completely native execution.

Given the limitations of the current simulators we consider that the community needs a flexible and fast instrumentation based tracer able to be used with a broad range of programming languages who will handle local simulations in parallel, with small overheads and with instruction level granularity.

1.2 Project objectives

Our main objective is providing an instrumentation based tracer that can be used with other simulators. Given the problems with the size these traces can have we will feed them in a lively fashion.

In order to see whether these objectives are met or not we will measure the slowdown compared to the non instrumented program with a simple trace consumer (to ensure it is not the bottleneck). Our objective is getting at least similar slowdowns to the ones of Graphite [9], but removing the caveats it has at least on single core processors.

Also we intend to design an architecture which can later be expanded to support multiple simultaneous execution threads. This will be done on a later version of the project though due to timing constraints.

1.3 Project strengths

The biggest problem with instrumentation-based systems is that the instrumentation code is limited heavily by the instrumentation API of the instrumentation system (for example the POSIX thread API can not be used with Pin), this also reduces vastly the number of languages that can be used, in
order to overcome these limitations, we use FIFO queues placed in shared memory to extract the data from a process to another, using the operating system’s process separation to execute the simulation in a different processor. As a result, the simulator overcomes the restrictions caused by the instrumentation framework since these will only apply to the process where the program is being instrumented.

As a side effect of this approach, the resulting simulators will be segmented since the functional simulation can be done on a different processing unit than the one running the simulation itself. As the number of processor cores increases it is likely that hardware simulators will use the segmentation approach more extensively in order to increase performance. Also, when shared the cache accesses caused by the shared memory communication cause slowdowns, hyperthreading processors can be used and proper processor affinity to the processes can be set so the critical simulation parts (i.e., those responsible of bottlenecks) will be set along with the previous part on the different threads of a single core so the reads from the FIFO queue are likely to be on the level 1 cache.

To ensure real parallelism each thread of the instrumented program can use a different FIFO queue to extract its traces. This also allows the user to limit instruction granularity by setting an appropriate queue size since the instrumentation will stop execution once the queue is full.

As an example of how Pint can be used we also created Gin5 a slightly modified version of the gem5 simulator which uses Pint’s instrumentation as the source of the memory access information during the simulation.

Another known problem is that simulators tend to be very good on sim-
ulating a specific part of the system whilst having issues on others. We consider that in the future this FIFO system may be useful to interconnect simulators so the best of them can be gotten.

1.4 Memory structure

In this introduction we presented the problem we are trying to fix, our objectives and our strengths.

On the next section, we will explain the state of the art at the time of our publication in the topics of Architectural Simulators, Instrumentation systems and Benchmarks.

Afterwards we will analyze the four modules we developed for the project and we will continue later with the design decisions.

We will finally present our benchmarking results and our conclusions.

Annexed you will find a brief user manual in case you want to try our system and the referred bibliography.

On the Annex folder you will find the sources we developed in this project.
Chapter 2

State of the art

Of the many simulators currently available, we have chosen three to explain which is the current state of the art for being the ones on which most work is being done nowadays: gem5, Multi2Sim and Graphite.

Also, as instrumentation tools we will cover gprof based profiling and Pin.

Finally, as benchmarks we will cover the SPEC CPU2006 and the SPLASH-2 benchmarks.

2.1 Architectural simulators

Architectural simulators are tools used to see how a proposed processor design would work without the need of building the processors themselves. Despite these share a some similarities with virtual machines in that they execute programs and that the main focus in both is the correct execution of the program; virtual machines have their main focus in providing a speedy execution of the program, whilst architectural simulators focus on providing good statistics of the program execution and executing the program in the same way the architecture would use.
Architectural simulators tend to be structured in a set of stages, disassembly, functional simulation and cycle by cycle simulation.

During the disassembly stage the machine code to be executed is transformed into a set of structures that can be understood by the simulator, the set of structures used is critical for an efficient simulation.

During the functional simulation the resulting set of structures is interpreted by the simulator to modify the internal state of the processor structures and the representation of the simulated program’s memory space.

Finally during the cycle by cycle simulation the represented architecture and system are simulated in a cycle by cycle basis so the timing results are precise.

Simulators may have these stages clearly differentiated or not but all of the do have these stages.

Also some simulators emulating only the memory system (and further processor structures) are based on memory traces. A memory trace is a description of the memory accesses made by a particular program when run which is then replayed on the simulated memory system.

In general trace based simulators tend to be fast since they will not only remove the execution step but also use a more simplified model for the processor. But traces have a few problems: on one side the programs being run need to be run in a way in which they will be generated, for example with a dynamically instrumented program, and when big enough they can take a lot of space, for example a trace of the SPLASH-2 LU with contiguous blocks trace would take around 1.5GiB if each access could be stored in only 32 bits.

Anyway there are some nice works in trace generation with Pin for simulators like Dinero IV [7], an example of which can be found in the dinerotool [1] by Kenneth Barr.
2.1. Architectural simulators

When using traces it is hard to overcome the requirement of using traces, but, it is possible to overcome the space limitation restrictions by feeding them live into our memory simulator. This was the approach chose by us.

2.1.1 Graphite

Graphite [18] [9] is a multicore simulator also written over Pin designed to provide real multithreading both when run locally and when run over a large number of computers. In order to do this, graphite hijacks some syscalls of the syscalls which will then be sent either to the local kernel or to the central kernel or to both. A similar procedure is used to track memory access es and an internal "MMU" tracks which machine has which copy of the memory.

In order to synchronize threads Graphite provides a few different synchronization ways of which the fastest is the lax synchronization method.

Given the popularity of this simulator nowadays Graphite was the simulator chosen as the reference against which we will compare the speed of our system.

Saddly, one of the major caveats with Graphite is that it is very system specific and, as a result, it was impossible for us to run it on our testing equipment.

2.1.2 Multi2Sim

Multi2Sim [20] [3] is a simulator supporting a big set of targets to emulate different architectures, both CPU and GPU.

As a simulator it is split in different components, a disassembler intended to convert the input programs into something the simulator can understand and use, a functional simulator which maintains the CPU and memory state
and runs the code and a cycle by cycle simulator which does the execution. It also provides some visual tools for checking how the simulation is run.

2.1.3 gem5

gem5 \[16\] \[2\] is the result of the merge of two powerful simulators: M5 and GEMS. gem5 is a simulator able to emulate some architectures both in Full Mode (this is, running the kernel as part of the simulation) and in Syscall Emulation mode (as the two aforementioned simulators by emulating the kernel for the provided binary).

As a full system simulator it is known for the flexibility it has for emulating different systems, not only by the number of architectures it supports but also by the number of devices it can emulate and the flexibility it provides in doing so.

The main problem it has is that although the modules are written in C++, they are usually run by a python script which complicates the system.

This flexibility gem5 was the reason for choosing this simulator as a target for implementing our system.

2.2 Instrumentation systems

Instrumentation systems provide ways to know how is the code running, either for later statistic generation and performance checking or for other uses like memory trace generation.

Instrumentation can be dynamic if the code that will control how the program is running is added when it is executed or static if this code is interleaved when building the program with the compiler. Normally dynamic instrumentation is preferable since it will allow us to instrument also propri-
etary programs and will not require a modified compiler.

2.2.1 gprof

gprof \cite{17} \cite{6} is a profiling system used along with programs compiled with special flags by gcc \cite{14} \cite{13}. For this gcc will embed the profiling code and mix it with the compiled sources before assembly. This technique is called static instrumentation since it is done in compilation time.

Programs compiled with profiling flags will generate when run binary file, called gmon.out, containing the execution statistics. Afterwards a call to gprof can be used to interpret the generated file.

Although traces could be also generated by using these techniques the requirement of having to compile the programs with a particular compiler is an impediment in some cases thus the ideas provided by this system where discarded.

2.2.2 Pin

Pin \cite{21} \cite{10} \cite{8} on the other side is a dynamic instrumentation framework, this means that instrumentation code is added dynamically. For this Pin hijacks with ptrace the program to be run, as a debugger like gdb \cite{15} \cite{12} would do, and then loads the Pintools’ code and the Pin framework into the running program and modifies the process so it will run the code produced by the JIT generator provided by PIN.

For this to work, Pin provides a modified version of the C++ runtime which has some features stripped down in order to prevent incompatibilities with the program being run. Anyway most of the C++ features can still be used by the tools and for those that can not Pin provides alternatives (for example locks).
The main problem with Pin is that running it on hardened systems is complicated since the default method used by Pin to attach to the program via ptrace is considered dangerous by these kernels (since it is not a parent attaching to its child but the other way around), also the JIT compiler provided by Pin causes problems because it tries to have mapPings which are both writable and executable which is another technique restricted by hardened systems.

Despite these issues Pin was the system chosen for providing the instrumentation framework.

2.3 Benchmarks

Benchmarks are programs with standardized inputs that are used to measure and compare the performance of different systems running them. Depending of the component being measured different metrics can be used: power consumption, execution time, number of frames per second generated, etc. Of these in this project we care the most about execution time.

Benchmarks can be synthetic when they emulate the load caused by typical programs of a particular type, examples of which are Dhrystone \cite{25} and Whetstone \cite{5}; or application when they run one or more real world programs like the two we have analyzed. In general real world benchmarks provide more meaningful results since they allow you to see how will real applications behave.

2.3.1 SPEC CPU2006

The SPEC CPU2006 \cite{22} \cite{11} benchmark is a set of programs from the real world which are provided along with some inputs to test the speed of a
system and with a main focus on the CPU execution speed. Despite being there since the 2006 these benchmarks are widely used and understood in the academic and real world.

Most of the programs provided with the benchmark are licensed with GPL style licenses and are well known in the free software world, for example gcc or perl, whilst others come from different research projects. It is because of this that the copyright is held over the input files in this benchmarks.

The main problem with these benchmarks is that they focus on single threaded processes.

2.3.2 SPLASH-2

The SPLASH-2 [23] [26] benchmark was developed by the Flash research group at the Stanford university to provide a set of benchmarks that could be used on shared memory multiprocessor systems. Although the benchmarks are quite old and require modifications to work properly they can still be used and have the advantage of running in a short time.

The applications provided are related to the scientific world with examples of 3 body gravity simulators or some kernels like the LU decomposition of a matrix.

Since the original tests will not run, we used a modified version of the SPLASH-2 benchmark [19]. Even more modifications were required for the null macro to work properly and for the tests to be able to be run with Pin on hardened systems, these modifications are provided as a patch file in the source distribution.

The main reason for choosing these was that the relative performance results of these tests (although with more than one processor) were provided on [9] so we did not need to run the benchmarks again for Graphite and thus
set up the required Debian environment.
Chapter 3

System description

Our application will be divided in 4 modules: Mead, a framework for providing an efficient message passing interface between different processes; Pint, a Pin based trace generator; Schnapps, a simple consumer of the traces; and Gin5, a gem5 trace player for the memory system.

The traces will be generated by Pint and then fed through Mead to either Schnapps or Gin5 which will process it and provide some simulation statistics.

3.1 Mead: a message passing framework

The pattern of message passing is not new and it conforms the base of some Object Oriented views. Mead will provide a fast and simple way of passing around the traces as messages stating that something has happened (for example the program made an execution memory access of size x at position y). These messages may contain the thread identifier of the thread that caused them and also attached data, for example in the case of a memory write the data available before writing and the data being written.

Although the API provided by mead is quite agnostic of the message
passing system being used we have chosen producer-consumer FIFO queues. FIFOs are used since they are a known pattern which allows for easy implementation and migration over other interprocedural communication systems, if interprocess shared memory is not an option, like POSIX message queues or datagram sockets.

Our FIFO model differs slightly from the standard model since it allows for two communication types, on one hand you have the event communication system which can queue many events for further handling by the receiving side. On the other you will find a command interface able of holding a single command. The command interface requires acknowledging the sent command and is used to indicate important events which require specific handling by the queue system like the death of the FIFO or the beginning and ending of the simulation procedures.

The main difference between events and commands are that events are unidirectional (from the producer to the consumer) whilst commands can be used bidirectionally (as long as the absence of collisions is guaranteed by the programmer) and are more easily handled with the futex syscall which makes them very useful for events which will require a really heavy processing on the other side by allowing the other thread to preempt the CPU while this is done.

The FIFO architecture is based over a central FIFO (called the main FIFO) which is used to send global events which are supposed to stall the simulation until attended (so the simulator can decide whether it should clear or not the per thread queues before processing the aforementioned event), this queue handles at least the thread creation and deletion events where a new FIFO queue is negotiated between both sides, but it can also be used to process events like the creation and deletion of new mappings amongst
3.2. Pint: a Pin based trace generator

Chapter 3. System description

others.

Given the importance of the main FIFO in the architecture it is important that both processes know where to access it beforehand and are able to negotiate its creation independently of who arrived first (since synchronization is impossible before the FIFO creation).

The framework also features a per thread FIFO which can be used to send events which are not of global significance to the listener on the other side. This lets the programmer communicate information fast since the queues can be then lockless and, as a result, as long as there are at least two processors available the current process will not be changed by the kernel preventing expensive context switches. The creation of these FIFOs should be negotiated over the main FIFO when implementing the multithreaded version.

3.2 Pint: a Pin based trace generator

Pint by itself it is not a simulator but a framework providing efficient ways to extract the data from the instrumented program through Mead. The version presented with this project is single threaded (although designed to be multithreaded and with part of the work for that already done) and relies on Mead for communicating with the simulator itself. As an example the provided instrumentation will study memory accesses made by the program (of any type ranging from prefetches to execution fetches) and sends them out with Mead so the simulator can prevent the issues associated with Pin tracing tools.

The code here is focused heavily on speed and thus the user must have the option to choose the features that should to be used.

The granularity of the execution can be easily tunned by setting an ap-
propriate queue size. For example for instruction by instruction execution the queue must have size one.

Also, a simulation started event must be the first one to be queued so you can discard old elements when you want the execution to be done.

Since all instructions will start (and contain) with a single fetch event it is possible to use this event as the differentiator between instructions. Anyway it is a good idea to integrate at least the number of events the instruction will cause to make tracing easier. This may be done on future versions.

Pint also provides a way to specify the number of instructions that must be executed before switching to the next simulation mode and thus you can provide the number of instructions that must be executed (by the sum of threads) before switching to another simulation mode.

The mode automaton allows for three simulation modes which are switched in the following order, the fast forward, the warm up and the simulation mode, which will then go back to the fast forward.

In the fast forward mode instructions are just accounted and executed but no data is generated which allows for near native speed execution. In the warm up mode and simulation mode instructions will generate events for filling the caches but the entrance and exit of the simulation status are notified to the consumer so it can handle statistics properly.

3.3 Schnapps: a simple consumer of the traces

Schnapps is intended to be used mainly for analyzing the performance of the instrumentation code by consuming the events generated whilst trying to avoid causing any bottlenecks in execution, and also as an example program of how to extract the generated traces.
3.4. Gin5: a gem5 trace player

Schnapps reads the traces generated by Pint and outputs the map changes as they happen (in a diff like format) and some statistics for the current simulation and for the total run, in particular, amount of data read or written by the different memory access types, the number of said accesses that has happened and an execution mark made by xoring the different accesses’ addresses together the idea being that different marks imply different traces being generated but the same mark does not necessarily imply the same trace being generated.

3.4 Gin5: a gem5 trace player

Although previous versions of gem5 came with a trace player supporting different formats, these modules stopped being maintained long time ago and those stopped building, as a result and despite being a good base for starting the work given the big amount of changes the memory system has suffered since then a different base was necessary.

As a result we have set again a generic CPU for playing traces (missing the TLB) which will ask for memory access request to the queues via a clear interface so it can be used also with other types of trace formats including the old ones if the classes containing them are updated.
Chapter 4

System design

4.1 Mead: a message passing framework

Mead has a macro of particular interest: USE_YIELD which will enable the use of the yield system call to let other process use the processor when waiting.

On mead, we have chosen to implement a lockless buffer ring over shared memory for our FIFOs since it is a well known pattern [4] [24].

The other reasons for choosing such an structure was speed and independence. By being lockless we avoid expensive spinlocks that would hinder performance whilst avoiding also having to either use the ones provided by Pin everywhere or building our own. Also having the data in shared memory will prevent us from making expensive system calls to have the messages passed and will allow the usage of cache for that.

It should be taken into account that the lockless queue will only work properly if a single thread acts as reader and a single thread acts as writer. In case of having more threads at either side they require a lock to work properly.
Our implementation is based on templates so it can be used with different classes although it should be taken into mind that the same class (i.e. no inheritance) should be used on the whole queue.

Also one of the current major caveats is that the shared memory address is currently hardcoded and, as a result, only a single instance of the program can be started at the same time. We expect to fix this in future versions by providing a launcher that will allocate an anonymous shared memory segment and pass its identifier to both Pint and the trace reader being used.

The command types are defined by the shmstatus enum. Since newly allocated shared memory is filled with 0s we assume the 0 value as the initial state (NONE). The server will then write a SERVER_STARTED command and wait for a CLIENT_ACK then. Finally when dying the server is expected to send the SERVER_DIED command so the client will not wait forever for data.

Of the many methods provided, those of special relevance for the programmer are the gethead and gettail methods used to be able to access the data we want to insert or extract from the queue, the push and pop methods used for adding or removing an element from the queue and the full and empty methods used to check for these states.

Also some methods for waiting in case the queue is empty/full are provided but these must be used carefully since if the consumer is singlethreaded it could hang waiting forever for the queue to match the condition. In this case special waits monitoring the main queue status too are recommended instead. Also a wait_push method is provided that will wait until a push can be done.

For control handling we provide the send_control, receive_control and ack_control methods. In particular send_control will wait until an ACK is
sent back to state the condition was taken care of.

Finally the wait_start and tell_Start methods are provided for initialization and instead of yields they use calls to the futex syscall to lock the thread until they have been attended to reduce the processor load in some situations.

4.2 Pint: a Pin based trace generator

In order to ensure unwanted features will not hinder performance preprocessor based switches can be used to disable those you are not interested in using, also some other options can be set in this way.

The macros of interest here are PADSIZE which defines the amount of bytes of the cache line in order to prevent false sharing, MAXMEMSIZE which defines the maximum size a single memory access may have (used for amongst other setting the size of the buffers), USE_DATA which will enable the infrastructure for fetching and sending the accessed data in the events, MULTITHREADED which will enable the still incomplete multithreaded code, USE_STATES which will enable the fast forward, warm up and simulation state machine and DTRACE which will make pint output some debugging information.

Given the impact these features can have we decided to allow the user disable them at compile time. Also some of these features can be disabled at run time although they will still have some impact on the execution, in particular, USE_STATES will still cause the slowdowns of the conditionals introduced before the instrumentation calls to handle the state machine and USE_DATA will make the event size, and thus the queues larger.

A final option that can be disabled is mapping tracing after a context
change (disabled by default) and after a syscall. The reason for this is the great slowdown caused by this operation since it requires at least 3 system calls in order to be executed and parsing a large text file.

In Pint the instrumentation is added by the Instruction function, when given the choice between adding complexity here or in the instrumentation functions we should add it here since this function is executed with much less frequency than the instrumentation code. As can be seen this function just tells Pin to add calls to the proper instrumentation functions, either with previous conditionals if the state machine is being used or without them otherwise.

Here we should consider all the parsemaps functions which are wrappers around the original parsemaps function that will take care of generating the events may maps be added or deleted. Also, as it can be seen, this function will consume quite a lot of resources given the way in which it works. Sadly the PIN framework on which pint is based does not provide any API in order to distinguish the mappings made by the instrumentations (including the JIT caches and the instrumentation code itself) as a result a lot of events will be generated on the simulation status queue. In order to reduce this overhead we assume mappings may only change after either coming back from a context change (as is the case when the application is being ptraced by a debugger) or coming back from a system call, this reduces the overhead greatly but still generates a lot of spurious mapping changes that may pollute the simulator assumptions. We expect this issue to be fixed with the addition of a proper API on future versions of PIN. Unlike memory access information given the importance of the mapping information it is sent independently of the simulation mode as it is generated.

In order to take track of the memory accesses the RecordMemExec,
Chapter 4. System design  4.3. Schnapps: a simple consumer of the traces

RecordMemRead, RecordMemPrefetch and RecordMemPreWrite functions are used. Also when the user is interested in the data generated by these functions, RecordMemPreWrite changes its behavior so it can access the memory information provided before the access and a new function called RecordMemWrite and executed after the instruction finishes is added, the reason for this is that the written data can not be known otherwise.

In order to handle the state machine we have an enum called state which contains the current simulation state, a function called nextState which takes care of handling the previous variable and the one with the instruction counter, and is called only when the instruction counter reaches zero, we also have the StateCounter method that will decrease the instruction counter by one and say whether we have processed the last instruction or, we also have CounterDone which sends the events for starting or ending a simulation and finally we have the Instrument function which check whether instrumentation code should or not be run in the current state.

We finally have a few callbacks, ThreadStart used to notify the creation of new threads, ThreadFini used to notify its destruction and Fini which is called before the instrumented program exits and will generate the SERVER_DIED event.

4.3 Schnapps: a simple consumer of the traces

The code on Schnapps is all written on the main function given its simplicity.

First, the queues are negotiated with Pint, afterwards, the variables holding the stats are initialized to 0 and we state we are not simulating anything.

With that done we enter the main loop that will process information until the trace generator reports that it has died. In this loop, the data from
the thread queue is extracted and added to the statistics. Afterwards, the main queue is checked for events like mappings being added/removed and these changes are printed. And finally control signals are handled properly, including the beginning of a simulation (by setting the stats to 0) and the end (by printing the simulation stats).

Finally, once outside of the loop and with the simulation finished, we print the total stats.

### 4.4 Gin5: a gem5 trace player

The biggest amount of coding is likely to have been written in these classes since we had to revamp the trace readers and the trace CPUs so they would work with the current memory system used by gem5.

The MemTraceReader class is a very simple class providing a single method called getNextRequest that will provide either a pointer to the next Request to be played on the memory system or a NULL pointer along with the reason why it was provided.

The memory requests are represented by the MemTraceRequest which returns packets through the getNextPkt method.

The PinReader class is derived from the MemTraceReader class and aside from handling the Pint queues also adds some callback to delete the queues when done.

Finally the TraceCPU class provides the MemPort classes and the Tick-Event classes which are required by the simulator and is the responsible of requesting the data to the reader when necessary and sending the requests to the memory system through the proper port. From a CPU point of view it emulates a system without a TLB (we basically take the LSBs of the address...
to convert the virtual addresses we get into physical addresses) with ports for an instruction and a data cache.

An example gem5 configuration using this class is also provided in the pintrace.py file.
Chapter 5

Results

The benchmark results can be seen in the following table (extracted from the annexed .ods file).

It surprises us to get a slowdown as high as 804x in the case of LU and also the fact that fmm only got a 94x slowdown in the Graphite benchmarks. Anyway, if we discard the fmm benchmark we can see that our system performs better than graphite in all cases using a single processor.
Chapter 5. Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Graphite slowdown</th>
<th>Pint slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes</td>
<td>N/A</td>
<td>287</td>
</tr>
<tr>
<td>cholesky</td>
<td>346</td>
<td>361</td>
</tr>
<tr>
<td>fft</td>
<td>3978</td>
<td>284</td>
</tr>
<tr>
<td>fmm</td>
<td>94</td>
<td>322</td>
</tr>
<tr>
<td>lu_cont</td>
<td>4007</td>
<td>557</td>
</tr>
<tr>
<td>lu_non_cont</td>
<td>3061</td>
<td>804</td>
</tr>
<tr>
<td>ocean_cont</td>
<td>515</td>
<td>317</td>
</tr>
<tr>
<td>ocean_non_cont</td>
<td>433</td>
<td>360</td>
</tr>
<tr>
<td>radiosity</td>
<td>N/A</td>
<td>498</td>
</tr>
<tr>
<td>radix</td>
<td>1648</td>
<td>199</td>
</tr>
<tr>
<td>raytrace</td>
<td>N/A</td>
<td>279</td>
</tr>
<tr>
<td>volrend</td>
<td>N/A</td>
<td>404</td>
</tr>
<tr>
<td>water_nsquared</td>
<td>2465</td>
<td>509</td>
</tr>
<tr>
<td>water.spatial</td>
<td>966</td>
<td>683</td>
</tr>
</tbody>
</table>

Table 5.1: Slowdown comparison between Graphite with 8 cores and Pint with one
Chapter 6

Conclusions

The project development has taken a long time given the research components it had yet, its development helped us have a good insight on how to improve simulators speed.

Also, given the promising results obtained with the benchmarks (worst case of 804 when simulating, best case of 199 with a mean of 360.5 and an average of 419) run during the development and testing of this fairly limited version we think that ideas like simulation segmentation and instrumentation based simulation on independent process may help to the development of faster and more powerful simulators and will continue with its development.

We expect to see in the future heavily multithreaded simulators where each processor has its own group of threads each handling the different stages independently in order to speed up execution times on multiprocessor machines.

We also expect to see in the future more simulators used the process based separation between the data collection routines responsible of the execution of the program and the simulation itself in order to allow for the usage of higher level languages with less restrictions whilst still providing high performance
and native execution of the simulated programs.

6.1 Improvements for next release

In the next release we intend to have a fully parallel instrumentation framework, we will also reimplement the trace simulator as a full gem5 CPU so it can have proper TLB handling and can be extended internally with more complex models. Finally we will change the queuing system so the simulator knows how many events will be generated by the instruction being executed before these events are handled down. We will also interconnect Multi2sim with gem5 in order to prove the powerfulness of Mead.

Once we release the next version we intend to publish a paper on a publication on this topic.
Appendix A

User manual

A.1 Building

In order to build the sources it suffices with running the `make` command on the sources directory.

A.2 Pint

Running the Pint pintool is quite easy and for that it is enough to run:

```
./pin -t source/tools/SimpleExamples/obj-intel64/pinatrace.so - command arguments
```

Options can be set by setting the desired switches between pinatrace.so and the –

Currently the following options are available:

- `-f number`: adds the set number of instruction to be run in the fast forward state (used many times it will set more instruction counts to be run the next time we go back to said state)
- `-w number`: adds the set number of instruction to be run in the warm up
state (used many times it will set more instruction counts to be run the next
time we go back to said state)
-s number : adds the set number of instruction to be run in the simulation
state (used many times it will set more instruction counts to be run the next
time we go back to said state)
-syscallmap {0,1} : disables, if 0, or enables, if 1, the checking of process
mappings after returning from a syscall
-ctxchangemap {0,1} : disables, if 0, or enables, if 1, the checking of process
mappings after a context change
-values {0,1}: disables, if 0, or enables, if 1, the copying of data along
with the memory events

A.3 Schnapps

For running Schnapps just run ./consumer

A.4 Gin5

Gin5 requires a python file setting the system to be emulated. An example
of such system can be found in the pintrace.py file. Once you have set
up your system on a python file you just need to run the gem5.fast binary
followed by the file containing the system being defined.

Scripts to set up systems may take arguments from the command line if
introduced after the script file. Our example file does not make use of this
feature but others may.
Appendix B

Relevant source code

```c
#include <linux/futex.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <csignal>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <new>

#define likely (x) __builtin_expect ( (!!x), 1)
#define unlikely (x) __builtin_expect ( (!!x), 0)

// Compile time configuration
#define PADSIZE 64 // 64 byte line size as per intel specs
#define MAXMEMSIZE 32 // 256 bits as per AVX, needs to be increased on the future
#define MULTITHREADED 1 // Add codepaths to obtain the data on the memory accesses
#define USE_STATES 1 // Use a state machine to allow for fast forward and warm up states
#define DTRACE 1 // Add debugging output
#define USE_YIELD 1 // Use yield() or the pin equivalent when waiting for the other thread
#define QSIZE_ 1024 // Default FIFO queue size

#define USE_DATA 1 // Add codepath to obtain the data on memory accesses
#define MULTITHREADED 1 // Add codepaths to allow for multithreaded applications
#define USE_STATES 1 // Use a state machine to allow for fast forward and warm up states
#define DTRACE 1 // Add debugging output
#define USE_YIELD 1 // Use yield() or the pin equivalent when waiting for the other thread
#define QSIZE_ 1024 // Default FIFO queue size

// Compile time configuration
#define PADSIZE 64 // 64 byte line size as per intel specs
#define MAXMEMSIZE 32 // 256 bits as per AVX, needs to be increased on the future
#define MULTITHREADED 1 // Add codepaths to obtain the data on the memory accesses
#define USE_STATES 1 // Use a state machine to allow for fast forward and warm up states
#define DTRACE 1 // Add debugging output
#define USE_YIELD 1 // Use yield() or the pin equivalent when waiting for the other thread
#define QSIZE_ 1024 // Default FIFO queue size

enum DataType { INVALIDDATA, STARTTH, ACCMEM };
```
Appendix B. Relevant source code

```c
enum AccessType {ACCEXEC, ACCREAD, ACCWRITE, ACCPREFETCH};

// #define PAD(n) (((n) + (PADSIZE - 1)) / PADSIZE) * PADSIZE)
#include <cassert>

#ifdef DTRACE
#define dcprintf(c,...) if(c) fprintf(stderr, __VA_ARGS__)
#define dprintf(...) fprintf(stderr, __VA_ARGS__)
#define dcputs(c,a) if(c) fputs((a), stderr)
#define dputs(a) fputs((a), stderr)
#else
#define dcprintf(...) 
#define dprintf(...) 
#define dcputs(c,a) 
#define dputs(a) 
#endif

//TODO: use alignments instead of paddings
//TODO: use other padded struct for the data from read to write
class MemAccess {
private:
    // We are not going to use derive classes here for efficiency
    AccessType type;
    VOID * ea; // Effective address of the access
    #ifndef USE_DATA
    char data[MAXMEMSIZE]; // Contains either the data executed/read or the data contained before writing
    char wdata[MAXMEMSIZE]; // This is valid only when the data access is a write contains the written data
    #endif
    UINT32 size; // Size of the access
    #ifdef MULTITHREADED
    UINT32 tid; // The ID of the thread generating the access
    #endif
    #ifndef USE_DATA
    inline void setData() {
        assert (PIN_SafeCopy(data , ea , size ) == size );
    }
    #endif
    inline void copyData(const MemAccess &ma) {
        memcpy(data , ma.data , size );
        if (type == ACCWRITE )
            memcpy(wdata , ma.wdata , size );
    }
    #endif
public:
    #ifndef USE_DATA
    inline void setWdata() {
        assert (PIN_SafeCopy(wdata , ea , size ) == size );
    }
    #endif
    inline void MemAccessSet (AccessType type , VOID * ea , UINT32 size
        , UINT32 tid
        ) {
        this->type = type;
        this->ea = ea;
        this->size = size;
        #ifndef MULTITHREADED
        this->tid = tid;
        #endif
        #ifdef USE_DATA
        if (type != ACCPREFETCH) {
            setData();
        }
        #endif
    }
    inline void MemAccessSet (const MemAccess &ma) {
        type = ma.type;
        ea = ma.ea;
        size = ma.size;
        #ifdef MULTITHREADED
        tid = ma.tid;
        #endif
        #ifdef USE_DATA
        if (type != ACCPREFETCH) {
            copyData(ma);
        }
        #endif
    }
    void show(FILE *f); // Requires the C LOCK
    inline AccessType getType() const { return type;}
    inline VOID * getEA() const { return ea;}
};
```
Appendix B. Relevant source code

```c
#ifdef USE_DATA
inline const void * getData() const {
  return data;
}

inline const void * getWData() const {
  return wdata;
}
#endif

inline UINT32 getSize () const {
  return size ;}

#ifdef MULTITHREADED
inline UINT32 getTid () const {
  return tid ;}
#endif

} cachealigned ;

union SimDataU {
  class MemAccess ma;
};

class SimData {
private:
  DataType type;
  SimDataU data;
public:
  SimData() : type(INVALDATA) {
  }
  SimData(DataType _type) : type(_type) {
  }
  inline DataType getType () const {
    return type ;}
  inline void setType (DataType _type) {
    type = _type;
  }
  inline MemAccess & getMa () {
    return data.ma;
  }
  inline const MemAccess & getCMa () const {
    assert (type == ACCMEM);
    return data.ma;
  }
};

eenum InstEventType {
  ADDMAPPING, //A mapping was added during the last context change/syscall
  REMOVEMAPPING, //A mapping was removed during the last context change/syscall
  ADDTHREAD, //A new execution thread has been spawned
  REMOVERTHREAD, //An execution thread has ceased existing
};

struct range {
  unsigned long int b; // begin
  unsigned long int e; // end
  inline bool operator < (const struct range &r) const {
    // There shouldn’t be overlapping ranges (at least in theory);
    return b < r.b;
  }
};

union InstEventData {
  range r ;
  //TODO: handle per thread access queue creation here
};

class InstEvent {
private:
  InstEventType type;
  InstEventData data;
public:
  inline InstEvent() { }
  inline void SetInstEvent (InstEventType _type , range _r) {
    type = _type;
    data.r = _r;
  }
  inline InstEventType getType () const {
    return type ;
  }
  range & getRange () {
    assert (type == ADDMEM || type == REMOVERTHREAD);
    return data.r ;
  }
};

class InstEvent {
private:
  InstEventType type;
  InstEventData data;
public:
  inline InstEvent() { }
  inline void SetInstEvent (InstEventType _type , range _r) {
    type = _type;
    data.r = _r;
  }
  inline InstEventType getType () const {
    return type ;
  }
  range & getRange () {
    assert (type == ADDMEM || type == REMOVERTHROUGH);
    return data.r ;
  }
};
```

36
Appendix B. Relevant source code

```c
// Currently we use them for two purposes, passing events related to memory
// mappings and threads between the instrumentation and the simulator and
// passing around the memory access of each thread.
// We only use QSIZE−1 thus there is always one element free for processing before queueing.
#define NEXTQELEM(v) (((v) + 1) % QSIZE)

enum shmstatus {NONE = 0, // Initial state
               CLIENT_ACK=1, // The client confirms reception of previous state
               SERVER_STARTED=2, // The server has just started
               SERVER_DIED=3, // The server has died
               SERVER_SIM_START=4, // We are going to jump into simulation reset stats
               SERVER_SIM_END=5 // We have ended simulation reset stats
               };

// A lockless single producer single consumer queue, with more than 1 you will need locks

template <class T, int QSIZE=QSIZE> class SHMQ {  
private:
  T queue[QSIZE] cachealigned;

  volatile sig_atomic_t qhead cachealigned;
  volatile sig_atomic_t qtail cachealigned;

  // Elements are inserted on the head and removed from the tail like a snake.
  volatile sig_atomic_t control cachealigned;

public:
  inline SHMQ () : qhead(0), qtail (0) {
  }

  inline T & gethead () { return queue[qhead] ; }
  inline T & gettail () { assert (!empty()); return queue[qtail]; }

  inline bool full () { return NEXTQELEM(qhead) == qtail; }
  inline bool empty() { return qtail == qhead; }

  inline void wait_full () { while ( unlikely ( full())) YIELD(); }

  inline bool wait_empty_cond() { return (empty() && control == CLIENT_ACK); }
  inline void wait_empty() { while ( unlikely (wait_empty_cond())) YIELD(); }

  inline void push() { assert (! full()); qhead = NEXTQELEM(qhead ); }

  inline void wait_push() { wait_full (); push (); }

  inline void pop() { assert (!empty()); qtail = NEXTQELEM(qtail ); }

  inline enum shmstatus receive_control () { if (control == CLIENT_ACK) return NONE;
          return (enum shmstatus) control ; }

  inline void ack_control () { while ( unlikely (control == CLIENT_ACK)) YIELD();
                              control = CLIENT_ACK; syscall(SYS_futex, &control ,FUTEX_WAKE,1 ,NULL,NULL,0); }

  inline void send_control(enum shmstatus st) { assert (st != CLIENT_ACK); // For this we should use ack_control instead
                                                     control = st; // Wait for the ACK
                              while ( unlikely (control != CLIENT_ACK)) YIELD(); }

  inline void wait_start () { sig_atomic_t control;
          while ( (control = control) != SERVER_STARTED) syscall(SYS_futex, &control ,FUTEX_WAIT,control_ ,NULL,NULL,0);
                              control = CLIENT_ACK;
                              syscall(SYS_futex, &control ,FUTEX_WAKE,1 ,NULL,NULL,0); }

  inline void tell_start () { sig_atomic_t control;
Appendix B. Relevant source code

```c
control = SERVER_STARTED;
syscall(SYS_futex, &control, FUTEX_WAKE, 1, NULL, NULL, 0);
while ((control_ = control) != CLIENT_ACK) syscall(SYS_futex, &control, FUTEX_WAIT, SERVER_STARTED, NULL);
```

```c
typedef SHMQ<SimData> SimDataq;
typedef SHMQ<InstEvent> InstEventq;

SimDataq *server_init2();
SimDataq *client_init2();
void server_fini2(SimDataq *q);
void client_fini2(SimDataq *q);

SimDataq *server_init2() {
  SimDataq *q = get_q2(shmid);
  new(q) SimDataq(); // We use a placement new so we have the SimDataq in the shared memory
  q->tell_start();
  return q;
}

SimDataq *client_init2() {
  int shmid;
  SimDataq *q = get_q2(shmid);
  q->wait_start();
  // Since we are connected we tell the OS the segment can be deleted
  if (shmctl(shmid, IPC_RMID, NULL) < 0)
    perror("shmctl");
  return q;
}

void server_fini2(SimDataq *q) {
  q->send_control(SERVER_DIED);
}

void client_fini2(SimDataq *q) {
  q->ack_control();
  q->~SimDataq();
}
```
Appendix B. Relevant source code

```
server_init () {
    int shmid;
    InstEventq *q = get_q(shmid);
    new (q) InstEventq(); // We use a placement new so we have the InstEventq in the shared memory
    q->tell_start();
    return q;
}

client_init () {
    int shmid;
    InstEventq *q = get_q(shmid);
    q->wait_start();
    // Since we are connected we tell the OS the segment can be deleted
    if (shmctl(shmid,IPC_RMID,NULL) < 0)
        perror("shmctl");
    return q;
}

server_fini (InstEventq *q) {
    q->send_control(SERVER_DIED);
}

client_fini (InstEventq *q) {
    q->ack_control();
    q->~InstEventq();
}
```

#endif
Appendix B. Relevant source code

unatrace.cpp

```cpp
#include "pin.H"
#include "pintrace.H"
#include <iostream>
#include <algorithm>
#include <set>
#include <fstream>

// Use when calling C and C++ library functions

PIN_LOCK c_lock;

// FILE *TraceFile;
FILE *StatsFile;

// KNOB<cstring> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "pintrace.out", "specify trace file name");
KNOB_COMMENT(comment("pintool.trace", "Options for the tracing behaviour"));

#define USE_STATES
KNOB<UINT64> Knob(KNOB_MODE_APPEND, "pinatrace", "use", "", "Data, Output, and Business Interruption")
KNOB_COMMENT(comment("use", "Options for structure warming")

#define USE_DATA
KNOB<STRING> KnobValues(KNOB_MODE_WRITEONCE, "pintool", "values", "1", "Output_memory_values_read_and_written");

#define MULTITHREADED
```

© 2011 Intel Corporation. All rights reserved.
Appendix B. Relevant source code

```c
static SimDataq *q;
static InstEventq *iq;
static FILE *mout;

void parsemaps(void) {

  FILE *f;
  static set<range> prev;
  set<range> s, rem, add;
  range r;

  //TODO: we’ll need to have a lock here to ensure threads don’t collide
  f = fopen("/proc/self/maps","r");
  while ( fscanf(f, "%lx-%lx", &(r.b),&(r.e)) == 2 ) {
    s.insert(r);
    int c;
    while ((c = fgetc(f)) != ‘\n’ ) putchar(c);
    while ( (c = fgetc(f)) != ‘\n’ ) ;
  }
  //We don’t need the file any more so release the FD
  fclose(f);

  set_difference(prev.begin(), prev.end(), s.begin(), s.end(), inserter(rem, rem.begin()));
  for (set<range>::iterator it = rem.begin(); it != rem.end(); it++) {
    iq->gethead().SetInstEvent(REMOVEMAPPING, *it);
    iq->wait_push();
    // printf("-%lx-%lx\n", it->b, it->e);
  }

  set_difference(s.begin(), s.end(), prev.begin(), prev.end(), inserter(add, add.begin()));
  for (set<range>::iterator it = add.begin(); it != add.end(); it++) {
    iq->gethead().SetInstEvent(ADDMAPPING, *it);
    iq->wait_push();
    // printf("+%lx-%lx\n", it->b, it->e);
  }

  //Ok finally we’ll make the old set this one
  prev = s;
}

void parsemaps1(THREADID _1, CONTEXT *_2, SYSCALL_STANDARD _3, VOID *_4) {
  fputs("Syscall!
",mout);
  parsemaps();
}

void parsemaps2(THREADID _1, CONTEXT_CHANGE_REASON _2, const CONTEXT *_3, CONTEXT *_4, INT32 _5, VOID *_6) {
  fputs("Context/uni2423change!
",mout);
  parsemaps();
}

void parsemaps3(void) {
  fputs("Initial\n",mout);
  parsemaps();
}

static char AccessType2Char(AccessType at) {

  switch (at) {
  case ACCEXE: return ‘X’;
  case ACCEXCEL: return ‘X’;
  case ACCREAD: return ‘R’;
  case ACCWRITE: return ‘W’;
  case ACCPREFETCH: return ‘P’;
  default: return ‘?’;
  }
}

#define USE_DATA
static VOID EmitMem(FILE *f, VOID *data, UINT32 size) {
  switch(size) {
  case 0: break;
  }

  /* TODO: Here we do some assumptions about sizes, a proper program should fill them properly*/
```

Appendix B. Relevant source code

```c
166 case 1:
167     fprintf(f, "0x%02hhx", (static_cast<UINT8*>(data)));
168     break;
169
case 2:
170     fprintf(f, "0x%04hx", (static_cast<UINT16*>(data)));
171     break;
172
case 4:
173     fprintf(f, "0x%08x", (static_cast<UINT32*>(data)));
174     break;
175
case 8:
176     fprintf(f, "0x%016lx", (static_cast<UINT64*>(data)));
177     break;
178
default:
179     if (size > 0) {
180         fprintf(f, "0x%02hhx", (static_cast<UINT8*>(data)));
181         for(UINT32 i = 1; i < size; i++)
182             fprintf(f, "%02hhx", (static_cast<UINT8*>(data)[i]));
183     }
184     break;
185 }
186
187 #endif
188
189 void MemAccess::show(FILE *f) {
189     fprintf(f, #ifdef MULTITHREADED
190         "%10u"
191     #endif
192     "/uni2423%c/uni2423%#016lx%/uni24233d/uni2423" ,
193     #ifdef MULTITHREADED
194         (UINT32) tid ,
195     #endif
196    AccessType2Char(type) , (unsigned long int) ea , size);
197     #ifdef USE_DATA
198     if (KnobValues) {
199         if (type != ACCPREFETCH) {
200             EmitMem(f, data, size);
201             if (type == ACCWRITE) {
202                 fputs("−>/uni2423" , f);
203                 EmitMem(f, wdata, size);
204             }
205         }
206     }
207     fputs("\n" , f);
208 }
209
210 static INT32 Usage() {
211     fputs(
212         "This/uni2423tool/uni2423produces/uni2423a/uni2423memory/uni2423address/uni2423trace .
213         For/uni2423each/uni2423memory/uni2423access/uni2423(execute/read/write/prefetch )/uni2423the/uni2423ea/uni2423is/uni2423recorded
214         \n", stderr);
215     fputs(KNOB_BASE:: StringKnobSummary().c_str() , stderr);
216     /* UNISIM_GetOS() */
217     return -1;
218 }
219 static VOID PIN_FAST_ANALYSIS_CALL RecordMemExec(VOID *ip, UINT32 size
220     #ifdef MULTITHREADED
221     , THREADID tid
222     #endif
223     ) {
224     GetQLock(tid+1);
225     q->gethead().getMa().MemAccessSet(ACCEXEC, ip , size
226     #ifdef MULTITHREADED
227     , tid
228     #endif
229     ) ;
230     q->wait_not_empty();
231     q->wait_push();
232     ReleaseQLock();
233 }
234 #ifdef USE_DATA
235     #endif
236 }
```

42
Appendix B. Relevant source code

```c
static VOID PIN_FAST_ANALYSIS_CALL RecordMemRead(VOID * ea, UINT32 size)
#
ifdef MULTITHREADED
, THREADID tid
#endif
{
  //TODO: use a per thread lockless queue
  GetQLock(tid+1);
  q->gethead().getMa().MemAccessSet(ACCREAD, ea, size
  #ifdef MULTITHREADED
  , tid
  #endif
  ;
  iq->wait_not_empty();
  q->wait_push();
  ReleaseQLock();
}

static VOID PIN_FAST_ANALYSIS_CALL RecordMemPrefetch(VOID * ea, UINT32 size
#ifdef MULTITHREADED
, THREADID tid
#endif
{
  //TODO: use a per thread lockless queue
  GetQLock(tid+1);
  q->gethead().getMa().MemAccessSet(ACCPREFETCH, ea, size
  #ifdef MULTITHREADED
  , tid
  #endif
  ;
  iq->wait_not_empty();
  q->wait_push();
  ReleaseQLock();
}

static VOID PIN_FAST_ANALYSIS_CALL RecordMemPreWrite(VOID * ea, UINT32 size
#ifdef MULTITHREADED
, THREADID tid
#endif
{
  #ifdef USE_DATA
  #ifdef MULTITHREADED
  MemAccess *ma = static_cast<MemAccess *>(PIN_GetThreadData(wMemAccess, tid));
  ma->MemAccessSet(ACCWRITE, ea, size, tid);
  #else
  q->gethead().getMa().MemAccessSet(ACCWRITE, ea, size);
  #endif
  #endif
  GetQLock(tid+1);
  q->gethead().getMa().MemAccessSet(ACCWRITE, ea, size
  #ifdef MULTITHREADED
  , tid
  #endif
  ;
  iq->wait_not_empty();
  q->wait_push();
  ReleaseQLock();
  #endif
}

#ifdef USE_DATA
static VOID PIN_FAST_ANALYSIS_CALL RecordMemWrite(
#ifdef MULTITHREADED
THREADID tid
#endif
{
  #ifdef MULTITHREADED
  MemAccess *ma = static_cast<MemAccess *>(PIN_GetThreadData(wMemAccess, tid));
  ma->setWdata();
  #else
  q->gethead().getMa().setWdata();
  #endif
  #endif
}
```

43
Appendix B. Relevant source code

```c
#if defined MULTITHREADED

//TODO: use a per thread lockless queue
GetQLock(tid+1);
#endif
q->gethead().getMa().MemAccessSet(*ma);
#endif
eq->wait_not_empty();
q->wait_push();
ReleaseQLock();
#endif

#if defined USE_STATES
static UINT64 inscount = -1;
static enum state { FASTFORWARD = 0, WARMING = 1, SIMULATION = 2} state = SIMULATION;
static UINT32 fIndex = 0;
static UINT32 wIndex = 0;
static UINT32 sIndex = 0;

inline VOID nextState() {
    do {
        switch ( state ) {
            case FASTFORWARD:
                inscount = Knobw.Value(wIndex);
                wIndex++;
                state = WARMING;
                break
            case WARMING:
                inscount = Knobs.Value(sIndex);
                sIndex++;
                state = SIMULATION;
                break
            case SIMULATION:
                // If we are done simulating stop
                if (Knobf.NumberOfValues() == fIndex)
                    PIN_ExitApplication(0);
                inscount = Knobf.Value(fIndex);
                fIndex++;
                state = FASTFORWARD;
                break;
            default:
                puts("Unknown/uni2423state\n", stderr);
                PIN_ExitApplication(1);
                break;
        }
    } while (inscount == 0);
}

static ADDRINT PIN_FAST_ANALYSIS_CALL StateCounter(VOID *ip
#endif MULTITHREADED

} #ifdef MULTITHREADED

{ inscount--;
    dprintf("ins. %p\n", ip);
    if (inscount == 0)
        puts("switch\n");
    return inscount == 0;
}

static ADDRINT PIN_FAST_ANALYSIS_CALL Instrument {
#endif MULTITHREADED

}

static ADDRINT PIN_FAST_ANALYSIS_CALL CounterDone{
#endif MULTITHREADED

} #ifdef MULTITHREADED

{ enum state orig = state;
    if (orig == SIMULATION) {
        q->send_control(SERVER_SIM_END);
    }
    nextState();
}
```
Appendix B. Relevant source code

```c
415   if (state == SIMULATION) {
416     q->send_control(SERVER_SIM_START);
417     #endif
418   }
419   //We only want to change the instrumentation when switching from any state to the fast forward state or vice versa
420   // if (( orig != FASTFORWARD && state == FASTFORWARD) || (orig == FASTFORWARD && state != FASTFORWARD))
421   //TODO: this isn't working as expected :( (yet)
422   // PIN_RemoveInstrumentation (); //reinstrument the program
423 #endif
424 #endif
425
426 // Instrumentation
427 static VOID Instruction(INS ins, VOID *v)
428 {
429   // Also using the IF-then callback system make PIN more likely to inline the counter code
430 #ifdef USE_STATES
431   INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)StateCounter, IARG_FAST_ANALYSIS_CALL,
432     IARG_INST_PTR,
433     #ifdef MULTITHREADED
434     IARG_THREAD_ID,
435     #endif
436     IARG_END);
437   INS_InsertThenCall(ins, IPOINT_BEFORE, (AFUNPTR)CounterDone, IARG_FAST_ANALYSIS_CALL,
438     #ifdef MULTITHREADED
439     IARG_THREAD_ID,
440     #endif
441     IARG_END);
442 #endif
443   // if (state != FASTFORWARD) {
444 #ifdef USE_STATES
445   INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)Instrument, IARG_FAST_ANALYSIS_CALL,
446     IARG_INST_PTR,
447     #ifdef MULTITHREADED
448     IARG_THREAD_ID,
449     #endif
450     IARG_END);
451   #endif
452   #endif
453 #ifdef USE_STATES
454   INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)RecordMemExec, IARG_FAST_ANALYSIS_CALL,
455     IARG_INST_PTR,
456     #ifdef MULTITHREADED
457     IARG_THREAD_ID,
458     #endif
459     IARG_END);
460   #endif
461   if (INS_IsMemoryRead(ins))
462 {
463 #ifdef USE_STATES
464   INS_InsertIfPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)Instrument, IARG_FAST_ANALYSIS_CALL,
465     #ifdef MULTITHREADED
466     IARG_THREAD_ID,
467     #endif
468     IARG_END);
469   #endif
470   INS_InsertThenPredicatedCall
471   #else
472   INS_InsertPredicatedCall
473   #endif
474   (ins, IPOINT_BEFORE, (AFUNPTR)RecordMemExec, IARG_FAST_ANALYSIS_CALL,
475     IARG_MEMORYREAD_EA,
476     IARG_MEMORYREAD_SIZE,
477     #ifdef MULTITHREADED
478     IARG_THREAD_ID,
479     #endif
480     IARG_END);
481   #endif
482   if (INS_IsMemoryRead2(ins))
483 {
484 #ifdef USE_STATES
485   INS_InsertIfPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)Instrument, IARG_FAST_ANALYSIS_CALL,
486     #ifdef MULTITHREADED
487     IARG_THREAD_ID,
488     #endif
489     IARG_END);
490   #else
491   INS_InsertThenPredicatedCall
492   #endif
493   INS_InsertPredicatedCall
494   #else
495   INS_InsertPredicatedCall
496   #endif
497   (ins, IPOINT_BEFORE, (AFUNPTR)(INS_IsPrefetch(ins)?RecordMemPrefetch:RecordMemRead), IARG_FAST_ANALYSIS_CALL,
498     IARG_MEMORYREAD_EA,
499     IARG_MEMORYREAD_SIZE,
500     #ifdef MULTITHREADED
501     IARG_THREAD_ID,
502     #endif
503     IARG_END);
504   #endif
505   if (INS_HasMemoryRead2(ins))
506 {
507 #ifdef USE_STATES
508   INS_InsertIfPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)Instrument, IARG_FAST_ANALYSIS_CALL,
509     #ifdef MULTITHREADED
510     IARG_THREAD_ID,
511     #endif
512     IARG_END);
513   #else
514   INS_InsertThenPredicatedCall
515   #endif
516   INS_InsertPredicatedCall
517   #else
518   INS_InsertPredicatedCall
519   #endif
520   (ins, IPOINT_BEFORE, (AFUNPTR)(INS_IsPrefetch(ins)?RecordMemPrefetch:RecordMemRead), IARG_FAST_ANALYSIS_CALL,
521     IARG_MEMORYREAD_EA,
522     IARG_MEMORYREAD_SIZE,
523     #ifdef MULTITHREADED
524     IARG_THREAD_ID,
525     #endif
526     IARG_END);
527   #endif
528 }
529 
530 #endif
531
45"
Appendix B. Relevant source code

```c
#include <stdio.h>
#include <stdlib.h>

#define MULTITHREADED

typedef struct {
    int id;
    int value;
} MemAccess;

#define IARG_MEMORYREAD_SIZE
#define IARG_MEMORYWRITE_SIZE

#define IARG_THREAD_ID

#define IARG_END

#define USE_STATES

#define USE_DATA

#define USE_STATES

static MemAccess* ma = new MemAccess();

void ThreadStart(void* v) {
    MemAccess* ma = (MemAccess*)v;
    ma->value = 42;
}

void ThreadFini(void* v) {
    delete (MemAccess*)v;
}

void ProcessQueue(void* nothing) {
    while (!ending || !q->empty()) {
        q->pop();
    }
}

void main() {
    MemAccess ma;
    ma.value = 42;
    ProcessQueue(&ma);
}
```
Appendix B. Relevant source code

```c
581 // // Let others fill the queue
582 // YIELD();
583 // }
584 // }
585
586 static VOID Fini(INT32 code , VOID ∗v)
587 {
588   // ending = true;
589   // PIN_WaitForThreadTermination (processorid , PIN_INFINITE_TIMEOUT , NULL);
590   if (KnobSyscallMap || KnobCtxChangeMap)
591     fclose(mout);
592   server_fini2(q);
593   server_fini(iq);
594 }
595
596 int main(int argc, char ∗argv[])
597 {
598   if (PIN_Init(argc, argv))
599     return Usage();
600 }
601
602 #ifdef USE_STATES
603   if (!(Knobf.NumberOfValues() == Knobw.NumberOfValues() && Knobf.NumberOfValues()==Knobs.NumberOfValues()))
604     {
605       fputs("The number of occurrences of -f, -h and -s must be the same. ", stderr);
606       return Usage();
607     }
608 #endif
609
610 #ifdef MULTITHREADED
611   if (Knobf.NumberOfValues() >= 1)
612     nextState();
613   // This one is done due to the way instrumentation works
614   inscount+;
615   // Send the simu start command if necessary
616   if (state == SIMULATION)
617     q->send_control(SERVER_SIM_START);
618   INS_AddInstrumentFunction(Instruction , 0);
619   PIN_AddFiniUnlockedFunction(Fini , 0);
620 #endif
621
622 // Open the output file
623   if (KnobSyscallMap || KnobCtxChangeMap)
624     mout = fopen("maptrace.txt" , "w");
625
626 #ifdef USE_STATES
627   // Monitor syscalls and so for mapping changes
628   if (KnobSyscallMap)
629     PIN_AddSyscallExitFunction(parsemaps1 , NULL);
630   // Monitor also after context changes since if we are traced mappings may have changed
631   if (KnobCtxChangeMap)
632     PIN_AddContextChangeFunction (parsemaps2 , NULL);
633 #ifdef MULTITHREADED
634 
635 #endif
636   // This means the instruction addresses we get are mapped to the mappings corresponding to the libraries and not to
637   // code so we don’t have to worry about changes to these mappings, but, since we still can’t discern them from application
638   // mappings we still have to reserve space for them in the simulator space. This also means we’ll be having some movement
639   // in the map space almost always until PIN provides an api to discern pin/tool mappings from application ones.
640 
641 // Thread Callbacks
642   InitLock(&c_lock);
643 
644 #ifdef MULTITHREADED
645   InitLock(&k_lock);
646   wMemAccess = PIN_CreateThreadDataKey(0);
647  #endif
648   PIN_AddThreadStartFunction(ThreadStart , 0);
649   PIN_AddThreadFiniFunction(ThreadFini , 0);
650 #endif
651
652 // Start queue processor thread
653 
654 // processor = PIN_SpawnInternalThread (ProcessQueue , NULL , 0 , &processorid);
655 if (processor == INVALID_THREADID) return 1;
656 // Initial map loading
657 if (KnobSyscallMap || KnobCtxChangeMap)
658 parsemaps3();
659
660 PIN_StartProgram();
661
662```

Appendix B. Relevant source code

663     return 0;
664   }

Appendix B. Relevant source code

c consumer.cpp

1 #include <pinatrace.h>
2 #include <cstdint>
3 #include <cinttypes>

4

5 int main() {
6     InstEventq ∗iq;
7     iq = client_init();
8     SimDataq ∗q;
9     q = client_init2();
10    uint64_t nins = 0;
11    uint64_t nins2 = 0;
12    uint64_t nrea = 0;
13    uint64_t nrea2 = 0;
14    uint64_t nwri = 0;
15    uint64_t nwri2 = 0;
16    uint64_t npre = 0;
17    uint64_t npre2 = 0;
18    uint64_t sins = 0;
19    uint64_t sins2 = 0;
20    uint64_t srea = 0;
21    uint64_t srea2 = 0;
22    uint64_t swri = 0;
23    uint64_t swri2 = 0;
24    uint64_t spre = 0;
25    uint64_t spre2 = 0;
26    bool simulating = false;
27
28    while (q−>receive_control() != SERVER_DIED) {
29        while (!q−>empty()) {
30            assert (q−>gettail().getT() ! INVALIDDATA);
31            if (q−>gettail().getT() == ACCMEM) {
32                const MemAccess &ma = q−>gettail().getCMa();
33
34                mark ^= (uint64_t) ma.getEA();
35                mark2 ^= (uint64_t) ma.getEA();
36                switch (ma.getT()) {
37                    case ACCEXEC:
38                        nins ++;
39                        nins2++;
40                        sins += ma.getSize();
41                        sins2 += ma.getSize();
42                        break;
43                    case ACCREAD:
44                        nrea ++;
45                        nrea2++;
46                        srea += ma.getSize();
47                        srea2 += ma.getSize();
48                        break;
49                    case ACCWRITE:
50                        nwri ++;
51                        nwri2++;
52                        swri += ma.getSize();
53                        swri2 += ma.getSize();
54                        break;
55                    case ACCPREFETCH:
56                        npre ++;
57                        npre2++;
58                        spre += ma.getSize();
59                        spre2 += ma.getSize();
60                        break;
61                    default:
62                        puts("Unexpected_access_type!");
63                        break;
64                }
65            }
66        }
67        // Have we just emptied the buffer or has an event happened?
68        while (!iq−>empty()) {
69            if (iq−>gettail().getT() == REMOVIEWMAPPING) {
70                range r = iq−>gettail().getRange();
71                printf("%.5lx-%.5lx
", r.b, r.e);
72            } else if (iq−>gettail().getT() == ADDVIEWMAPPING) {
73                range r = iq−>gettail().getRange();
74                printf("%+.5lx-%.5lx
", r.b, r.e);
75            }
76            iq−>pop();
77        }
78    }
79
80    } // while (!q−>empty())
81    break;
82    case SERVER_SIM_END:
83
simulating = false;
puts ("Simulation_statistics:");
puts ("Number_of_accesses:");
printf ("%lu", nins);
printf ("%lu", nrea);
printf ("%lu", nwri);
printf ("%lu", npre);
printf ("%lu", sins);
printf ("%lu", srea);
printf ("%lu", swri);
printf ("%lu", spre);
printf ("%lu", sins2);
printf ("%lu", srea2);
printf ("%lu", swri2);
printf ("%lu", spre2);
printf ("%lu", mark);
printf ("%lu", mark2);
q->ack_control();
break;
case SERVER_SIM_START:
nins = 0;
area = 0;
npre = 0;
sins = 0;
area = 0;
area = 0;
area = 0;
swri = 0;
mark = 0;
simulating = true;
simulating = false;
puts ("Total_statistics:");
puts ("Number_of_accesses:");
printf ("%lu", nins2);
printf ("%lu", nrea2);
printf ("%lu", nwri2);
printf ("%lu", npre2);
printf ("%lu", sins2);
printf ("%lu", srea2);
printf ("%lu", swri2);
printf ("%lu", spre2);
printf ("%lu", mark2);
client_fini(q);
client_fini_iq();
return 0;
}
Appendix B. Relevant source code

```c
mem_trace_reader.hh

/*
 * Copyright (c) 2004-2005 The Regents of The University of Michigan
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met: redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer;
 * redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution;
 * neither the name of the copyright holders nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/* Authors: Erik Hallnør */

/* Definitions for a pure virtual interface to a memory trace reader. */

#ifndef __MEM_TRACE_READER_HH__
define __MEM_TRACE_READER_HH__
#include "mem/packet.hh"
#include "mem/request.hh"
#include "params/MemTraceReader.hh"
#include "sim/sim_object.hh"

class MemTraceRequest : public FastAlloc {
    Addr _paddr;
    unsigned _size;
    Request::Flags _flags;
    Tick _time;
    int _asid;
    Addr _vaddr;
    int _contextId;
    int _threadId;
    Addr _pc;
    MemCmd _cmd;

public:
    MemTraceRequest() {}
    MemTraceRequest(Addr paddr, int size, Request::Flags flags, MemCmd cmd)
        : _paddr(paddr), _size(size), _flags(flags), _time(curTick()), _cmd(cmd) {}
    MemTraceRequest(Addr paddr, int size, Request::Flags flags, Tick time, MemCmd cmd)
        : _paddr(paddr), _size(size), _flags(flags), _time(time), _cmd(cmd) {}
    ~MemTraceRequest() {} // for FastAlloc

    inline bool mustRun() {
        return _time <= curTick();
    }
    inline Tick time() {
        return _time;
    }
};
#endif
```

51
Appendix B. Relevant source code

```
inline bool isInstFetch () {
    return _flags.isSet(Request::INST_FETCH);
}

inline bool lastPacketSent () {
    return _size == 0;
}

/**
 * Get the next packet with proper bounds for this block size
 * Will return NULL when done
 *
 * @param lastPacketSent
 */
PacketPtr getNextPkt (int bsize, Packet::NodeID dest, MasterID mid) {
    if (lastPacketSent()) {
        return NULL;
    }
    // Base address of the block
    Addr base = (_paddr & ~(bsize - 1));
    // Current block maxsize
    int msize = bsize - (_paddr - base);
    // Minimum
    if (msize > _size) msize = _size;
    // Generate the request and the packet
    RequestPtr req = new Request(_paddr, msize, _flags, mid);
    PacketPtr pkt = new Packet(req, _cmd, dest);
    // Calculate the new base address and size
    _paddr += msize;
    _size -= msize;
    return pkt;
}

typedef MemTraceRequest * MemTraceRequestPtr;

/**
 * Pure virtual base class for memory trace readers.
 */
class MemTraceReader : public SimObject {
public:
    enum reason {EOT, STAT_RESET, STAT_DUMP};

    MemTraceReader(const MemTraceReaderParams *p) : SimObject(p) {}

    public:

    enum reason {EOT, STAT_RESET, STAT_DUMP};
    MemTraceReader(const MemTraceReaderParams *p) : SimObject(p) {}

    virtual MemTraceRequestPtr getNextRequest(enum reason &reason) = 0;

};
```

Appendix B. Relevant source code

```c++
/*
Copyright (c) 2004-2005 The Regents of The University of Michigan
All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met: redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer;
redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution;
nor the name of the copyright holders nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Authors: Erik Hallnor
*/

#ifndef __Pin_READER_HH__
#define __Pin_READER_HH__

#include "cpu/trace/reader/mem_trace_reader.hh"
#include "cpu/trace/reader/pin_atrace.hh"
#include "params/PinReader.hh"

class PinReader :
public MemTraceReader {
friend class DeleteQueuesCallback;
public:
friend class DeleteQueuesCallback;
SimDataq *q;
InstEventq *iq;
bool simulating;
//Whether we are in simulation state or not
bool drop;
//Should we drop the next element (has it been processed)
protected:
void removeQueues();

public:
/*
** Construct an M5 memory trace reader.
**
PinReader(const PinReaderParams *p);
-PinReader();
//TODO: redo doc pkt should contain time, request, command and data.
/**
* Read the next request from the trace. Returns the request in the
* provided RequestPtr and the cycle of the request to the return value.
@param req Return the next request from the trace.
* @return The cycle of the request, 0 if none in trace.
*/
virtual MemTraceRequestPtr getNextRequest(MemTraceReader::reason &reason);
};
#endif // __PIN_READER_HH__
```
Appendix B. Relevant source code

pin_reader.cc

```c
/* Copyright (c) 2004-2005 The Regents of The University of Michigan  
 All rights reserved. 
 * Redistribution and use in source and binary forms, with or without 
 * modification, are permitted provided that the following conditions are 
 * met: redistributions of source code must retain the above copyright 
 * notice, this list of conditions and the following disclaimer; 
 * redistributions in binary form must reproduce the above copyright 
 * notice, this list of conditions and the following disclaimer; 
 * neither the name of the copyright holders nor the names of its 
 * contributors may be used to endorse or promote products derived from 
 * this software without specific prior written permission. 
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS 
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT 
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR 
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT 
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, 
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT 
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, 
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY 
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT 
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE 
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. 
 */

Authors: Erik Hallnor

/** 
 * Declaration of a memory tracer reader for a pin memory trace. 
 */
#include "base/callback.hh"
#include "cpu/trace/reader/pin_reader.hh"
#include "sim/sim_exit.hh"
#include <set>

//TODO: look why the user interrupt received event doesn’t calls the Callback
/** Callback to clean the queues*/
class DeleteQueuesCallback : public Callback {
  public:
    DeleteQueuesCallback();
    void process();
  static DeleteQueuesCallback dqc;
}

/** List of PinReader elements for the queue deleting callback **/
static std::set<PinReader*> readers;

DeleteQueuesCallback::DeleteQueuesCallback() {
  registerExitCallback(this);
}

void DeleteQueuesCallback::process() {
  for (std::set<PinReader*>::iterator it = readers.begin(); it != readers.end(); it++) {
    (*it)->removeQueue();
  }
}

//TODO: Send client finalization events if necessary
void PinReader::removeQueue() {
  if (q) {
    client_fini2(q);
    q = NULL;
  } else if (iq) {
    client_fini(q);
    iq = NULL;
  }
  warn("Done");
}

PinReader::PinReader(const PinReaderParams* p) : MemTraceReader(p), simulating(false) {
  iq = client_init2();
  q = client_init2();
  //Wait for the initial event
```
while(q->empty()) { YIELD();
        drop = true;
        readers.insert(this);
    }

PinReader::~PinReader() {
    removeQueues();
    readers.erase(this);
}

MemTraceRequestPtr PinReader::getNextRequest(MemTraceReader::reason &reason) {
    MemCmd::Command cmd;
    MemTraceRequestPtr req;
    Request::Flags flags;
    if (drop) {
        assert(!q->empty());
        q->pop(); // Drop previous data
    }
    while(true) {
        // Wait for new traces if the server died just send NULL
        while(q->wait_empty_cond() && iq->wait_empty_cond()) YIELD();
        switch(q->receive_control()) {
        case SERVER_DIED:
            // The last dump should be made by m5 itself
            reason = MemTraceReader::EOT;
            drop = false;
            return NULL;
        case SERVER_SIM_END:
            simulating = false;
            q->ack_control();
            reason = MemTraceReader::STAT_DUMP;
            drop = false;
            return NULL;
        case SERVER_SIM_START:
            simulating = true;
            q->ack_control();
            reason = MemTraceReader::STAT_RESET;
            drop = false;
            return NULL;
        case NONE:
            break;
        default:
            warn("State not supported!");
            return NULL;
        }
        while(!iq->empty()) {
            switch(q->gettail().getType()) {
            case ACCMEM: {
                const MemAccess& ma = q->gettail().getCMa();
                switch (ma.getType()) {
                case ACCEXEC:
                    flags.set(Request::INST_FETCH);
                    cmd = MemCmd::ReadReq;
                    break;
                case ACCREAD:
                    cmd = MemCmd::ReadReq;
                    break;
                case ACCWRITE:
                    cmd = MemCmd::WriteReq;
                    break;
                case ACCPREFETCH:
                    flags.set(Request::PREFETCH);
                    cmd = MemCmd::ReadReq;
                    break;
                default:
                    panic("Access type unknown");
                    break;
                }
                Addr ea = (Addr)ma.getEA();
                ea &= (Addr)134217727; // 128Mb - 1 : P
                // By default time is set to 0
                req = new MemTraceRequest((Addr)ea, (int)magetSize(), flags, cmd);
                drop = true;
                return req;
                break;
            }
            case STARTTH:
            case INVALDATA:
                break;
            default:
                panic("Unexpected data type");
                break;
            }
        }
    }
}
Appendix B. Relevant source code

// Process mapping changes
if (iq->getTail().getType() == REMOVEMAPPING) {
  // TODO: remove mapping from TLB
} else if (iq->getTail().getType() == ADDMAPPING) {
  // TODO: add mapping from TLB
}

iq->pop();

PinReader * PinReaderParams::create()
{
  return new PinReader(this);
}
Appendix B. Relevant source code

```cpp
/* Copyright (c) 2004-2005 The Regents of The University of Michigan
 * All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met: Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer;
 * distributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer;
 * neither the name of the copyright holders nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

Authors: Erik Hallnor

```
virtual void recvRetry();

/∗∗ Port for instruction trace requests, if any. ∗/
MasterID _instMasterId;
MemPort icache;

/∗∗ Port for data trace requests, if any. ∗/
MasterID _dataMasterId;
MemPort dcache;

/∗∗ Data reference trace. ∗/
MemTraceReader *dataTrace;

/∗∗ Number of outstanding requests. ∗/
int outstandingRequests;

/∗∗ Next packet containing data, time, request, command, etc ∗/
MemTraceRequestPtr nextRequest;

/∗∗ Reason for the packet to be NULL ∗/
MemTraceReader::reason reason;

/∗∗ Next request ∗/
MemCmd::Command nextCmd;

/∗∗ Event to call the TraceCPU::tick ∗/
class TickEvent : public Event
{  
  private:
    TraceCPU *cpu;
  public:
    TickEvent(TraceCPU *c) : Event(CPU_Tick_Pri), cpu(c) {}
    void process() { cpu->tick(); }
    virtual const char *description() const { return "TraceCPU::tick"; }
    
    TickEvent tickEvent;

    inline Tick ticks(int numCycles) const { return numCycles; }

    public:
    /∗∗ Construct a TraceCPU object. ∗/
    TraceCPU(const TraceCPUParams *p);

    /∗∗ Perform all the accesses for one cycle. ∗/
    void tick();

    /∗∗ Handle a completed memory request. ∗/
    void completeRequest(PacketPtr req);

    virtual Port *getPort(const std::string &if_name, int idx = -1);
};
#endif// __CPU_TRACE_TRACE_CPU_HH__
Appendix B. Relevant source code

trace_cpu.cc

```c
/* Copyright (c) 2004-2005 The Regents of The University of Michigan
 * All rights reserved.
 */

/* Redistributon and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met: redistributions of source code must retain the above copyright
 notice, this list of conditions and the following disclaimer;
 redistributions in binary form must reproduce the above copyright
 notice, this list of conditions and the following disclaimer in the
 documentation and/or other materials provided with the distribution;
 neither the name of the copyright holders nor the names of its
 contributors may be used to endorse or promote products derived from
 this software without specific prior written permission.
 */

/* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/* Authors: Erik Hallnor */

using namespace std;

TraceCPU::TraceCPU(const TraceCPUParams *p) :
    MemObject(p),
    _instMasterId(p->sys->getMasterId(name() + " . inst"), icache("instructions", this),
    _dataMasterId(p->sys->getMasterId(name() + " . data"), dcache("data", this),
    dataTrace(p->trace), outstandingRequests(0), tickEvent(this)
{
    nextRequest = dataTrace->getNextRequest(reason);
    schedule(&tickEvent, curTick() + ticks(1));
}

//TODO: fixunalignedaccessesoutofblockboundaries

void TraceCPU::tick()
{
    assert(outstandingRequests >= 0);
    assert(outstandingRequests < 1000);
    int instReq = 0; //TODO convert to stats
    int dataReq = 0; //TODO convert to stats
    while(!nextRequest)
    {
        if (outstandingRequests) return;
        switch(reason) {
        case MemTraceReader::EOT:
            // No more requests to send. Finish trailing events and exit.
            exitSimLoop("end_of_memory_trace_reached");
        // } else {
        //   if (!tickEvent->scheduled())
        //     schedule(&tickEvent, nextTick() + ticks(1));
        // }
```
Appendix B. Relevant source code

```c
return;

  case MemTraceReader::STAT_RESET:
    nextRequest = dataTrace->getNextRequest(reason);
    Stats::reset();
    break;
  case MemTraceReader::STAT_DUMP:
    nextRequest = dataTrace->getNextRequest(reason);
    Stats::dump();
    break;
  }
  if (nextRequest->mustRun()) {
    int bsize = 0;
    if (nextRequest->isInstFetch()) {
      bsize = icache.peerBlockSize();
    } else {
      bsize = dcache.peerBlockSize();
    }
    // Rest of the request: get the new address and the new size
    if (nextRequest->isInstFetch()) {
      PacketPtr nextPkt = nextRequest->getNextPkt(bsize, 0, _instMasterId);
      tcpu->completeRequest(nextPkt);
    } else if (nextRequest->isRead()) {
      // DPRINTF( "id %d initiating %s read at addr %x (blk %x) expecting %x\n", 
               id, do_functional ? "functional" : "", req->getPaddr(), 
               blockAddr(req->getPaddr()), *result);
      icache.sendPkt(nextPkt);
    } else if (nextRequest->isWrite()) {
      // DPRINTF( MemTest, "initiating %s write at addr %x (blk %x) value %x\n", 
                 do_functional ? "functional" : "", req->getPaddr(), 
                 blockAddr(req->getPaddr()), data & 0xff);
      dcache.sendPkt(nextPkt);
    } else panic("CMD/uni2423not/uni2423implemented");
    delete nextRequest;
    nextRequest = dataTrace->getNextRequest(reason);
  }
  if (!tickEvent.scheduled())
    schedule(&tickEvent, max(curTick() + ticks(1), (nextRequest?nextRequest->time():0)));

Port *
TraceCPU::getPort(const std::string &if_name, int idx) {
  if (if_name == "data")
    return &dcache;
  else if (if_name == "instructions")
    return &icache;
  else panic("No/uni2423Such/uni2423Port\n");
}

bool TraceCPU::MemPort::recvTiming(PacketPtr pkt) {
  if (pkt->isResponse()) {
    tcpu->completeRequest(pkt);
  } else {
    if (pkt->isRequest())
      assert(pkt->isInstFetch());
    else panic("Atomic_accesses_not_supported");
  }
}

Tick TraceCPU::MemPort::recvAtomic(PacketPtr pkt) {
  panic("Atomic_accesses_not_supported");
```
Appendix B. Relevant source code

```
166     // must be snoop upcall
167     assert ( pkt->isRequest () );
168     assert ( pkt->getDest () == Packet::Broadcast );
169     return  curTick ();
170 
171 void
172 TraceCPU::MemPort::recvFunctional(PacketPtr pkt)
173 {
174     // Do nothing if we see one come through
175     return;
176 }
177
178 void
179 TraceCPU::MemPort::recvRangeChange()
180 {
181 }
182
183 void
184 TraceCPU::MemPort::recvRetry ()
185 {
186     if (sendTiming(retryPkt )) {
187         DPRINTF ( MemTest , " access Retry setting to false[n"]);
188         accessRetry = false;
189         retryPkt = NULL;
190     }
191     accessRetry = true;
192     retryPkt = pkt;
193 }
194
195 void
196 TraceCPU::MemPort::sendPkt(PacketPtr pkt) {
197     if ( atomic ) {
198         cachePort . sendAtomic ( pkt );
199         completeRequest ( pkt );
200     } else {
201         tcpu->outstandingRequests++;
202         if (!sendTiming( pkt)) {
203             DPRINTF ( MemTest , " access Retry setting to true\n") ;
204             accessRetry = true;
205             retryPkt = pkt;
206         }
207     }
208 }
209
210 // TODO: handle stats
211 void
212 TraceCPU::regStats ()
213 {
214     using namespace Stats ;
215     numReadsStat . name ( name () + ", num_reads" )
216     . desc ( "number of read accesses completed" );
217     numWritesStat . name ( name () + ", num_writes" )
218     . desc ( "number of write accesses completed" );
219     numExecsStat . name ( name () + ", num_exec" )
220     . desc ( "number of execution accesses completed" );
221 }
222
223 void
224 TraceCPU::completeRequest(PacketPtr pkt)
225 {
226     Request *req = pkt->req ;
227     DPRINTF ( MemTest , " completing %s at address %s (blk %s) %s\n" ,
228             req->isWrite() ? "write" : "read" ,
229             req->getPaddr (),
230             req->isError () ? "error" : "success " ,
231             blkAddr ( req->getPaddr ()) );
232     if ( pkt->isError () ) {
233         warn("Access failed for %s at %s\n",
234             req->isWrite() ? "write" : "read",
235             blkAddr ( req->getPaddr ()) );
236     }
237 }
```
Appendix B. Relevant source code

```c++
249 } else {
250     // TODO: handle stats
251     if (pkt->isRead()) {
252         numReads++;
253         numReadsStat++;
254     } else {
255         assert(!pkt->isWrite());
256         numWrites++;
257         numWritesStat++;
258     }
259 }
260
261 pkt->deleteData();
262 delete pkt->req;
263 delete pkt;
264 if (!tickEvent.scheduled())
265     schedule(&tickEvent, max(curTick(), (nextRequest ? nextRequest->time() : 0)));
266 }
267
268 TraceCPU *
269 TraceCPUParams::create()
270 {
271     return new TraceCPU(this);
272 }
273 */ To convert */
274
275 void
276 MemTest::completeRequest(PacketPtr pkt)
277 {
278     Request *req = pkt->req;
279     if (issueDmas) {
280         dmaOutstanding = false;
281     }
282     DPRINTF(MemTest, "completing %s at address %x (blk %x) %s\n",
283             pkt->isWrite() ? "write" : "read",
284             req->getPaddr(), blockAddr(req->getPaddr()),
285             pkt->isError() ? "error" : "success");
286     MemTestSenderState *state =
287         dynamic_cast<MemTestSenderState *>(pkt->senderState);
288     uint8_t *data = state->data;
289     uint8_t *pkt_data = pkt->getPtr<uint8_t>();
290     //Remove the address from the list of outstanding std::set<unsigned>::iterator removeAddr =
291     outstandingAddrs.find(req->getPaddr());
292     if (removeAddr == outstandingAddrs.end())
293         outstandingAddrs.erase(removeAddr);
294     if (pkt->isError()) {
295         if (!suppress_func_warnings) {
296             warn("\"Functional Access failed for %s at %x\n",
297                 name(), req->getPaddr());
298         }
299     } else {
300         if (pkt->isRead()) {
301             if (memcmp(pkt_data, data, pkt->getSize()) != 0) {
302                 panic("%s: read of %x (blk %x) @ cycle %x returns %x, expected %x\n", name(),
303                     req->getPaddr(), blockAddr(req->getPaddr()), curTick(),
304                     *pkt_data, *data);
305             }
306             numReads++;
307             numReadsStat++;
308             if (numReads == (uint64_t)nextProgressMessage) {
309                 if (cmpっていう(nextProgressMessage) == progressInterval)
310                     nextProgressMessage += progressInterval;
311                 if (maxLoads != 0 && numReads >= maxLoads)
312                     exitSimLoop("maximum number of loads reached");
313             }
314         } else {
315             panic("%s: write of %x (blk %x) @ cycle %x\n", name(),
316                     req->getPaddr(), blockAddr(req->getPaddr()), curTick(),
317                     *pkt_data, *data);
318             numWrites++;
319             numWritesStat++;
320             if (numWrites == (uint64_t)nextProgressMessage) {
321                 if (cmpっていう(nextProgressMessage) == progressInterval)
322                     nextProgressMessage += progressInterval;
323             }
324         }
325     }
326     traceCPU->deleteData();
327     funcPort.writeBlob(req->getPaddr(), pkt_data, req->getSize());
328     numWrites++;
329 }
```
Appendix B. Relevant source code

```c
// numWritesStat++;

// }

// noResponseCycles = 0;
// delete state;
// delete [] data;
delete pkt->req;
delete pkt;
if (!tickEvent.scheduled())
    schedule(tickEvent, curTick() + ticks(1));
}

// void
// MemTest::tick()
{
    // make new request
    /*
    * unsigned cmd = random() % 100;
    * unsigned offset = random() % size;
    * unsigned base = random() % 2;
    * uint64_t data = random();
    * unsigned access_size = random() % 4;
    * bool uncacheable = (random() % 100) < percentUncacheable;
    */
    unsigned cmd = 0;
    offset ++;
    unsigned base = 0;
    uint64_t data = random();
    unsigned access_size = 0;
    bool uncacheable = false;
    unsigned dma_access_size = random() % 4;
    // If we aren't doing copies, use id as offset, and do a false sharing
    // mem tester
    // We can eliminate the lower bits of the offset, and then use the id
    // to offset within the bits
    // offset = blockAddr(offset);
    // offset += id;
    // access_size = 0;
    // dma_access_size = 0;
    Request *req = new Request();
    Request::Flags flags;
    Addr paddr;
    if (uncacheable) {
        flags.set(Request::UNCACHEABLE);
        paddr = uncacheAddr + offset;
    } else {
        paddr = ((base ? baseAddr1 : baseAddr2) + offset);
    }
    bool do_functional = false;
    if (issueDmas) {
        paddr = (((1 << dma_access_size) - 1);
        req->setPhys(paddr, 1 << dma_access_size, flags);
        req->setThreadContext(id, 0);
    } else {
        paddr = (((1 << access_size) - 1);
        req->setPhys(paddr, 1 << access_size, flags);
        req->setThreadContext(id, 0);
    }
    assert(req->getSize() == 1);
    uint8_t *result = new uint8_t[8];
    if (cmd < percentReads) {
        // read
        // For now we only allow one outstanding request per address
        // per test. This means we assume CPU does write forwarding
        // to reads that alias something in the cpu store buffer.
        if (outstandingAddrs.find(paddr) != outstandingAddrs.end()) {
            delete [] result;
            delete req;
            return;
        }
        outstandingAddrs.insert(paddr);
    }
```
Appendix B. Relevant source code

```cpp
// **** NOTE FOR RON: I'm not sure how to access checkMem. — Kevin
funcPort.readBlob(req->getPaddr(), result, req->getSize());
// 
// ∗∗∗∗∗
// cprintf(stderr, "id %id initiating %s read at addr %x (blk %x) expecting %x\n", id, do_functional ? "functional" : "", req->getPaddr());
// blockAddr(req->getPaddr()), *result);
// PacketPtr pkt = new Packet(req, MemCmd::ReadReq, Packet::Broadcast);
pkt->setSrc(0);
pkt->dataDynamicArray(new uint8_t[req->getSize()]);
MemTestSenderState *state = new MemTestSenderState(result);
pkt->setState(state);
if (do_functional) {
    assert(pkt->needsResponse());
    cachePort.sendFunctional(pkt);
    completeRequest(pkt);
} else {
    sendPkt(pkt);
}
}
} else {
    // write

// For now we only allow one outstanding request per address
// per tester. This means we assume CPU does write forwarding
// to reads that alias something in the cpu store buffer.
if (outstandingAddrs.find(paddr) != outstandingAddrs.end()) {
    delete [] result;
    delete req;
    return;
}
outstandingAddrs.insert(paddr);
DPRINTF(MemTest, "initiating %s write at addr %x (blk %x) value %x\n", do_functional ? "functional" : ", req->getPaddr(),
blockAddr(req->getPaddr()), data & 0xff);
// PacketPtr pkt = new Packet(req, MemCmd::WriteReq, Packet::Broadcast);
pkt->setSrc(0);
wint8_t *pkt_data = new uint8_t[req->getSize()];
pkt->dataDynamicArray(pkt_data);
memcpy(pkt_data, &data, req->getSize());
MemTestSenderState *state = new MemTestSenderState(result);
pkt->setState(state);
if (do_functional) {
    pkt->setSuppressFuncError();
    cachePort.sendFunctional(pkt);
    completeRequest(pkt);
} else {
    sendPkt(pkt);
}
```

Appendix B. Relevant source code

```
# Copyright (c) 2006−2007 The Regents of The University of Michigan
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are
# met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution;
# neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
Authors: Ron Drebinski

import optparse
import sys

import m5
from m5.objects import *

parser = optparse.OptionParser()

parser.add_option("-m", "--maxtick", type="int", default=m5.MaxTick,
                metavar="T",
                help="Stop after T ticks")

(options, args) = parser.parse_args()

if args:
    print "Error: script doesn't take any positional arguments"
    sys.exit(1)

# define prototype L1 cache
proto_l1 = BaseCache(size = '32kB', assoc = 4, block_size = 128,
                     latency = '1ns', tgts_per_mshr = 1)

proto_l1.mshrs = 1

pr = PinReader()

tcpu = TraceCPU(trace = pr)

# next comes L1 cache, if any
# prototypes.insert(0, proto_l1)

system = System(physmem = PhysicalMemory(latency = "100ns"))

new_bus = Bus(clock="500MHz", width=16)
system.physmem.cpu_side_bus = new_bus
system.physmem.port = new_bus.master

data_l1 = BaseCache(size = '32kB', assoc = 4, block_size = 64,
                    latency = '1ns', tgts_per_mshr = 8)
data_l1.mshrs = 1

data_l1.cpu = tcpu

tcpu.data = data_l1.cpu_side
```

Appendix B. Relevant source code

tcpu.instructions = ins_l1.cpu_side

def make_level(spec, prototypes, attach_obj, attach_port):
    parent = attach_obj
    # use attach obj as config parent too
    if len(spec) > 1 and (fanout > 1 or options.force_bus):
        new_bus = Bus(clock="500MHz", width=16)
        new_bus.port = getattr(attach_obj, attach_port)
        parent.cpu_side_bus = new_bus
        new_bus = attach_obj
        attach_port = "port"
        objs = [prototypes[0]() for i in xrange(fanout)]
        if len(spec) > 1:
            # we just built caches, more levels to go
            parent.cache = objs
            for cache in objs:
                cache.mem_side = getattr(attach_obj, attach_port)
                #make_level(spec[1:], prototypes[1:], cache, "cpu_side")
            else:
                # we just built the MemTest objects
                parent.cpu = objs
                for t in objs:
                    t.test = getattr(attach_obj, attach_port)
                    t.functional = system.funcmem.port
                #make_level(treespec, prototypes, system.physmem, "port")

# run simulation
root = Root(full_system=False, system=system)
root.system.mem_mode = "timing"
root.system.system_port = root.system.physmem.port

# Not much point in this being higher than the L1 latency
m5.ticks.setGlobalFrequency("1ns")
m5.instantiate()
exit_event = m5.simulate(m5.MaxTick)

print 'Exiting @ tick', m5.curTick(), 'because', exit_event.getCause()
Bibliography


