C152 Laboratory Exercise 3

Professor: Krste Asanovic
GSI: Henry Cook
Department of Electrical Engineering & Computer Science
University of California, Berkeley

March 4, 2008

1 Introduction and goals

The goal of this laboratory assignment is to allow you to conduct some simple virtual experiments in the Simics simulation environment. Using the x86-tlb module, you will collect virtual statistics and make some architectural recommendations based on the results.

The lab has two sections, a directed portion and an open–ended portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The open–ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task or the arguments you provide in support of your ideas.

Students are encouraged to discuss solutions to the lab assignments with other students, but must run through the directed portion of the lab by themselves and turn in their own lab report. For the open-ended portion of each lab, students can work individually or in groups of two or three. Any open-ended lab assignment completed as a group should be written up and handed in separately. Students are free to take part in different groups for different lab assignments.

You are only required to do one of the open ended assignments. These assignments are in general starting points or suggestions. Alternatively, you can propose and complete your own open ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the TA or professor.

This lab assumes you have completed the earlier laboratory assignments. However, we will re-include all the relevant files from past labs in this lab's distribution bundle for your convenience. Furthermore, we will assume that you remember all the commands used in earlier labs for controlling Simics simulation. If you feel any confusion about these points, feel free to consult the first lab guide or the Simics User Guide.

1.1 Simics TLB Overview

For this lab we will be using the x86-tlb module, as it is the most customizable and accessible of the processor TLBs available in Simics. In order to use this module, we must simulate an x86 processor. Simics can actually simulate many x86 processors (Intel Pentium II, III, IV and AMD Hammer64) using a single target module (x86–440bx). We will use the Tango machine provided with Simics, which is a Pentium IV running Fedora Linux.

The x86-tlb module is not especially well documented in the materials available from Virtutech, so we will provide you with a summary of the relevant information here. We will also provide additional details about specific commands as they are needed throughout the lab.

1.1.1 Background: the x86 Page Tables

The x86 architecture uses a three level page table hierarchy, similar to the one discussed in class. A first level page is referred to as a page directory, and a second level page is referred to as a page table. A page directory contains 1024 page directory entries (PDEs), each of which can point to a page table. A page table contains 1024 page table entries (PTEs), each of which can point to a page frame. Therefore, a virtual address is divided into three components, each of which indexes one of these structures: a 10-bit directory index, a 10-bit page index and a 12-bit offset.

Actually, the system is slightly more complicated, as large pages are also allowed. These pages can be either 2 or 4 megabytes depending on operating system settings. The version of Fedora Linux running on our target machine requires that these large pages be 4MB (it also calls them huge pages).

On a TLB miss this page table is walked by the hardware.

1.1.2 The Simics x86–tlb TLBs

Our simulator actually has four TLBs. Each TLB caches a different sort of address translation (4KB data, 4KB instruction, 4MB data, 4MB instruction). The size and associativity of the 4KB TLBs can be configured separately from the size and associativity of the 4MB TLBs, but the instruction and data versions of each must be the same size. The default values are 64 entries, 4–way set associative for the 4KB TLBs and 8 entries, 4–way set associative for the 4MB TLBs. The values can be changed by recompiling the module.

1.1.3 TLB statistics and logging

Unfortunately, the TLBs do not come with statistics gathering commands like those we used to measure cache performance. However, they do generate certain types of simulation events (which Simics calls *haps*). You will be provided with scripts which monitor these events and then use them to provide you with a summary of TLB statistics. You can also change the log–level attribute of the TLB module to see notifications of certain types of events (e.g. flushes). The following statistics will be tracked for you:

TLB Fills. Triggered when a TLB entry is filled after a tablewalk.

TLB Invalidates. Triggered when a TLB entry is invalidated. TLB flushes generate multiple invalidate haps.

TLB Replaces. Triggered when one TLB entry is replaced by another.

TLB Misses. Triggered when a TLB miss occurs. Not tracked separately for 4KB and 4MB TLBs.

These statistics record only the total number of these events, so it is up to you to keep track of the number of instructions executed during which these events were recorded. If you do not, any comparative analysis you undertake will have no meaning.

1.2 Graded Items

You will turn a hard copy of your results to the professor or TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

1. Problem 2.3: TLB statistics for each configuration and answers
2. Problem 2.4: TLB statistics for both sizes and answers
3. Problem 2.5: TLB statistics during boot and answers
4. Problem 2.6: TLB statistics for both programs and answers
5. Problem 3.1/3.2/3.3/3.4 modifications and evaluations (include source code if required)

2 Directed Portion

2.1 General methodology

Unlike g–caches, TLBs are a core part of the x86 target's functionality, and operate constantly. It is therefore unnecessary to 'warm' them the way we had to with the caches. While you must insure you are capturing a representative portion of the program's execution, you can simply begin recording statistics whenever you like. Thereafter, you can run scripts to view the statistics collected so far or to reset the counters if you wish to begin a new measurement. Resetting the counters does not flush the contents of the TLBs.

An important note: you can run Simics in fast mode all the time, but you must generally run for longer intervals to see OS–level effects that we are interested in.

The general methodology of this lab is:

1. Compile the x86–tlb module in the configuration you want.
2. Upon starting Simics, run a script (begin.simics) to set the appropriate simulator variables
3. Run the code to the point you are interested in.
4. Run a script (start-tlb-tracking.py) to begin collecting TLB statistics
5. Run the benchmark, keeping track of the number of instructions executed.
6. Run a script (report-tlb-tracking.py) to see the collected statistics.
7. To begin a new study, run a script (reset-tlb-tracking.py) to reset the counters.

You can use any of the `ilinux{1,2,3}.eecs.berkeley.edu` instructional servers to complete this lab assignment. The compiling issue from Lab 1 has been resolved! Even so, do not wait until the night before the assignment is due, because you will face resource contention that may significantly increase the time it takes to complete the assignment.

2.2 Setup

You will use a modified version of the `x86_tlb` module to track the instruction mixes of several benchmark programs provided to you. The first step is to copy the source code of the original `trace` module into your personal Simics workspace:

```
host$ cd ~/simics-workspace
host$ /share/instsww/pkg/virtutech/simics-3.0.30/bin/workspace-setup --copy-device=x86_tlb
```

Copy the files `assoc.c` and `x86_tlb.c`, provided with this lab into the `~/simics-workspace/modules/x86_tlb` directory. This will replace the original source code with code that provides more accurate statistics. Compile the new `x86_tlb` module with `make`. In the next section you will modify `assoc.c` to change the TLB associativities.

Start Simics, using the `targets/x86-440bx/tango-common.simics` script. Make sure the host filesystem or workspace is mounted and that the benchmark binary is copied into the target machine. To save time in the following sections, you should create a checkpoint that has all the files loaded, or possibly a checkpoint at each of the initial magic breakpoints in the benchmark programs.

2.3 Collecting statistics from TLBs of varying associativity

The binary file used as a benchmark in this lab assignment is GEneralized Matrix Multiply code, the same as was used in the open–ended portion of Lab 2. We are using this code because it is both CPU intensive and can access arbitrarily large amount of memory over the course of its execution. The binary precompiled for this section is called `gemm_x86` (for other sections you may have to compile this or other code from source on the target machine itself). The matrices used in this binary are 512KB each.

The default settings of the TLBs are to be 4-way set associative. In this section, you will collect statistics for a 4-way set associative, a direct mapped, and a fully associative TLB. Keep the TLB sizes constant for these studies. The associativity can be modified by changing the values of the `#define` statements at the top of `assoc.c`. Remember to recompile the module with make after you have modified the TLB parameters.

For each TLB configuration:

- Once Simics is started run:
  `simics> run-command-file begin.simics`
- Run the benchmark program (`gemm_x86`).
  `target# ./gemm_x86`
- It will quickly reach a magic breakpoint and the simulation will pause.
- Turn on TLB statistics tracing
  `simics> run-python-file start-tlb-tracking.py`
- Run for at least 100,000,000 instructions.
  `simics>  c 100_000_000`
- Display the recorded statistics
  `run-python-file report-tlb-tracking.py`

- Examine the current contents of the TLBs, and answer the questions below.
  `simics> cpu0_tlb.status`
- `Continue` the simulation, and halt the benchmark. If you want to collect more data, reset the counters with
  `run-python-file reset-tlb-tracking.py`

For each configuration, record the counts of each type of occurrence. What differences do you see between configurations in terms of these occurrences? Which page types seemed to face the most contention? Which types seem to be flushed most often?

Based on what you saw with `.status`, how many translations of the different page types are retained in TLBs? Characterize whether pages of a certain type seem to be mostly global or local (`G` or –) and supervisor or user (`supr` or `user`).

2.4 Collecting data about the effect of multiprogramming on TLB performance

Using a 4-way associative TLB, run multiple instances of `gemm_x86`, i.e.:

```
target# ./gemm_x86 &
target# ./gemm_x86 &
target# ./gemm_x86 &
target# ./gemm_x86 &
```

Collect the same data you collected for the last section for this new workload (but only for this TLB configuration). Begin collecting data after the magic breakpoint in the fourth instance of gemm is reached. What statistics seem to have changed? What can you conclude about the effect of multiple active processes on the TLBs?

Change the TLB configuration so that all TLBs are double the size they were previously (leave the associativity at 4). Run the benchmark again and collect more statistics. To what degree does doubling TLB size affect the frequency of the various events? Given that TLBs are expensive (in terms of chip resources) low–latency memory, but that TLB misses require expensive (in terms of time) page table walks, does this modification seem worthwhile?

2.5 Collecting data about booting

You may have seen that certain page types appear to be underutilized. Why have a TLB for these types of pages at all?

In this section, you will record TLB statistics at boot time. Use the initial TLB configuration (64 entry, 4–way for 4KB, 8 entry, 4–way for 4MB). Start collecting statistics as soon as Simics opens. You should collect statistics for at least `1_000_000_000` cycles.

What type of pages are used while the machine boots? Why do you think this is the case?

2.6 Collecting data about effect of userland huge pages

In Linux, large pages are not by default available to user level processes. However, with a little effort, we can allow ourselves to make use of these pages and see how they affect performance for memory intensive applications. If you run into trouble in this section, refer to `hugetlbpage.txt`.

Examine the source code in the files `smallpage.c` and `hugepage.c`. The code maps a large amount of memory (64MB) and then writes and reads all of it sequentially. You can compile and

run `smallpage.c` as is on the target machine.

```
target# gcc -I/host/share/instsww/pkg/simics-3.0.30/src/include smallpage.c -o smallpage
target# ./smallpage
```

From the magic breakpoint you should collect statistics for `1_000_000_000` cycles.

For `hugepage.c`, we must make a few modifications to the running operating system on the target machine before we can run the code. First we must tell the kernel to allow the user to allocate large pages:

```
target# echo 60 > /proc/sys/vm/nr_hugepages
```

And we must create a filesystem that these huge pages can be allocated from (this is currently the best way to get access to these pages in Linux).

```
target# mount -t hugetlbfs nodev /mnt
```

Now you can compile and run `hugepage.c` just as you did for `smallpage.c`. Make sure you track its TLB statistics for the same number of instructions as you tracked `smallpage` so that you can fairly compare the two.

What affect do the large pages have on TLB performance? Are the miss, replace, and invalidate frequencies affected similarly or differently? How much space is saved in terms of PTEs that are no longer needed? In what cases do you think it is worthwhile to use the large pages?

3 Open-ended Portion

3.1 Rewriting GEMM code to use huge pages

Take the non–blocked gemm source code provided with this lab `gemm.c`, and rewrite it to use large user data pages. Look to `hugetlbpage.txt` and `hugepage.c` for guidance. Provide evidence that for a large matrix (say, greater than 16 MB) your code achieves better TLB performance. How much space does your new code save in terms of PTEs?

3.2 Study virtual caches

This is a project that didn't quite fit into the last lab. For it, you will use g-cache modules configured to be virtual caches (`add-twolevelcache.simics`). Your task is to study the performance of gemm or any other benchmark code running on a virtually indexed and virtually tagged two–level cache hierarchy, and compare its behavior in the cache to when it run on the same hierarchy, but physically indexed and tagged. Which has better miss rates? Submit your statistics and source code.

Something else that must be considered is that virtual caches must be flushed on every context switch! You can achieve this in Simics by monitoring the cr3 register and flushing the caches every time it is written to. A simple way to do this might be:

```
simics> break-cr register cr3=
simics> <cache name>.reset-cache-lines
```

Script branches that watch for the register to be written to are probably a more efficient way to accomplish this (see Chapter 8.1 of the Simics User Guide).

While we normally warm the caches so as to avoid detecting inaccurate compulsory misses, in this study it is primarily the effect of the true compulsory misses caused by process context switches that we are interested in. When these misses are taken into account, which cache hierarchy seems

to have better performance? Submit the statistics you gather, and state the benchmark you used and its parameters if any.

3.3 Create a program to generate TLB misses

This project is similar to the one included in the first laboratory assignment. You goal is to create a program with only 20 lines of C code (`tlbmiss_manufacuting.c`) that creates the maximum number of TLB misses possible. (Note that this is a count, not a percentage, so 1 instruction producing 1 miss is not a good solution.) The code should be run for `1_000_000` instructions or until it completes (report how many instructions you ran for). You can set up as many data structures or allocate as much memory as you like for free in advance of the magic breakpoint, but within the critical 20 lines you must not call any functions or jump to any external code. Submit your source code and measured TLB misbehavior.

3.4 Optimize blocked GEMM for a TLB

This project is similar to the one included in the second laboratory assignment. Use the binary file `blocked_gemm_x86` which is precompiled for the Tango machine (matrix size is 512KB per matrix, where each is 512x512) and come up with an optimal set of block dimensions (specified as 3 command line parameters) that reduce the miss rate of a **8 entry direct mapped data TLB**. Look at the source code from Lab 2 Problem 3.1 if you need to refresh on how the code operates. Remember to recompile you TLB module source to have the appropriate parameters. Report your chosen block dimensions and the results you have which indicate they are the most effective.