18-600, Fall 2018
Architecture Lab
Assigned: Tuesday, Sep. 25, 11:59PM PDT
Due: Tuesday, Oct. 9, 11:59PM PDT
Last Day to Hand in: Friday, Oct. 12, 11:59PM PDT

1 Introduction

In this lab assignment, you will learn about performing optimizations both below and above the Instruction Set Architecture (ISA), and exploring the design space of superscalar out-of-order (O3) processors. You will be modifying both high level code of several benchmarks and parameters of an superscalar O3 CPU model using a simulation tool to identify bottlenecks, and improve performance. When you have completed the lab assignment, you will gain keen appreciation for the interactions between object code and execution hardware that affect the performance of your programs.

The lab is organized into three parts. In Part A you will write some simple Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you will learn to optimize the Y86-64 benchmark program. In Part C, you will optimize two specific parameters of a superscalar O3 CPU model in order to improve the performance and performance/cost (efficiency) for running two specific benchmarks.

2 Logistics

You will work on this lab alone.

Any clarifications and revisions to the assignment will be posted on Piazza.

3 Handout Instructions

1. Start by downloading the handout `arclab-handout.tar` from autolab to a (protected) directory in which you plan to do your work.
2. Then give the command: `tar xvf arclab-handout.tar`. This will cause the following files to be unpacked into the directory: `README`, `mm.c`, `daxpy.c`, `sim.tar`, `Makefile`, `simguide.pdf`, `o3_mm.py`, `o3_daxpy.py`, `o3_star.py`, `arclab.pdf`, and `driver.py`.

3. Next, give the command `tar xvf sim.tar`. This will create the directory `sim`, which contains your personal copy of the Y86-64 tools. You will be doing all of your work inside this directory.

4. Finally, change to the `sim` directory and build the Y86-64 tools:

```
unix>  cd sim
unix>  make clean; make
```

4 Part A

(30 points)

You will be working in directory `sim/misc` in this part.

Your task is to write and simulate the following three Y86-64 programs. The required behavior of these programs is defined by the example C functions in `examples.c`. Be sure to put your name and ID in a comment at the beginning of each program. You can test your programs by first assembling them with the program YAS and then running them with the instruction set simulator YIS.

Example:

```
unix> ./yas sum.ys
unix> ./yis sum.yo
```

In all of your Y86-64 functions, you should follow the x86-64 conventions for passing function arguments, using registers, and using the stack. This includes saving and restoring any callee-save registers that you use.

We recommend checking out the example programs in the `y86-code` folder for inspiration.

`sum.ys`: Iteratively sum linked list elements

Write a Y86-64 program `sum.ys` that iteratively sums the elements of a linked list. Your program should consist of some code that sets up the stack structure, invokes a function, and then halts. In this case, the function should be Y86-64 code for a function (`sum_list`) that is functionally equivalent to the C `sum_list` function in Figure 1. Test your program using the following three-element list:

```
# Sample linked list
.align 8
ele1:
        .quad 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0
```

```c
/* linked list element */
typedef struct ELE {
    long val;
    struct ELE *next;
} *list_ptr;

/* sum_list - Sum the elements of a linked list */
long sum_list(list_ptr ls)
{
    long val = 0;
    while (ls) {
        val += ls->val;
        ls = ls->next;
    }
    return val;
}

/* rsum_list - Recursive version of sum_list */
long rsum_list(list_ptr ls)
{
    if (!ls)
        return 0;
    else {
        long val = ls->val;
        long rest = rsum_list(ls->next);
        return val + rest;
    }
}

/* copy_block - Copy src to dest and return xor checksum of src */
long copy_block(long *src, long *dest, long len)
{
    long result = 0;
    while (len > 0) {
        long val = *src++;
        *dest++ = val;
        result ^= val;
        len--;
    }
    return result;
}
```

Figure 1: **C versions of the Y86-64 solution functions.** See `sim/misc/examples.c`

`rsum.ys`: Recursively sum linked list elements

Write a Y86-64 program `rsum.ys` that recursively sums the elements of a linked list. This code should be similar to the code in `sum.ys`, except that it should use a function `rsum_list` that recursively sums a list of numbers, as shown with the C function `rsum_list` in Figure 1. Test your program using the same three-element list you used for testing `list.ys`.

`copy.ys`: Copy a source block to a destination block

Write a program (`copy.ys`) that copies a block of words from one part of memory to another (non-overlapping) area of memory, computing the checksum (Xor) of all the words copied.

Your program should consist of code that sets up a stack frame, invokes a function `copy_block`, and then halts. The function should be functionally equivalent to the C function `copy_block` shown in Figure Figure 1. Test your program using the following three-element source and destination blocks:

```
.align 8
# Source block
src:
        .quad 0x00a
        .quad 0x0b0
        .quad 0xc00

# Destination block
dest:
        .quad 0x111
        .quad 0x222
        .quad 0x333
```

Evaluation

Part A is worth 30 points, 10 points for each Y86-64 solution program. Each solution program will be evaluated for correctness, including proper handling of the stack and registers, as well as functional equivalence with the example C functions in `examples.c`.

The programs `sum.ys` and `rsum.ys` will be considered correct if the graders do not spot any errors in them, and their respective `sum_list` and `rsum_list` functions return the sum `0xcba` in register `%rax`.

The program `copy.ys` will be considered correct if the graders do not spot any errors in them, and the `copy_block` function returns the sum `0xcba` in register `%rax`, copies the three 64-bit values `0x00a`, `0x0b`, and `0xc` to the 24 bytes beginning at address `dest`, and does not corrupt other memory locations.

5 Part B

(50 points)

```
1 /*
2  * ncopy - copy src to dst, returning number of positive ints
3  * contained in src array.
4  */
5 word_t ncopy(word_t *src, word_t *dst, word_t len)
6 {
7     word_t count = 0;
8     word_t val;
9
10    while (len > 0) {
11        val = *src++;
12        *dst++ = val;
13        if (val > 0)
14            count++;
15        len--;
16    }
17    return count;
18 }
```

Figure 2: **C version of the `ncopy` function.** See `sim/pipe/ncopy.c`.

You will be working in directory `sim/pipe` in this part.

The `ncopy` function in Figure 2 copies a `len`-element integer array `src` to a non-overlapping `dst`, returning a count of the number of positive integers contained in `src`. Figure 3 shows the baseline Y86-64 version of `ncopy`.

Your task in Part B is to modify `ncopy.ys` with the goal of making `ncopy.ys` run as fast as possible.

You will be handing in `ncopy.ys` with a header comment with the following information:

- Your name and ID.
- A high-level description of your code, and describe how and why you modified your code.

Coding Rules

You are free to make any modifications you wish, with the following constraints:

- Your `ncopy.ys` function must work for arbitrary array sizes. You might be tempted to hardwire your solution for 64-element arrays by simply coding 64 copy instructions, but this would be a bad idea because we will be grading your solution based on its performance on arbitrary arrays.
- Your `ncopy.ys` function must run correctly with YIS. By correctly, we mean that it must correctly copy the `src` block *and* return (in `%rax`) the correct number of positive integers.
- The assembled version of your `ncopy` file must not be more than 1000 bytes long. You can check the length of any program with the `ncopy` function embedded using the provided script `check-len.pl`:

```
1  ##################################################################
2  # ncopy.ys - Copy a src block of len words to dst.
3  # Return the number of positive words (>0) contained in src.
4  #
5  # Include your name and ID here.
6  #
7  # Describe how and why you modified the baseline code.
8  #
9  ##################################################################
10 # Do not modify this portion
11 # Function prologue.
12 # %rdi = src, %rsi = dst, %rdx = len
13 ncopy:
14
15 ##################################################################
16 # You can modify this portion
17         # Loop header
18         xorq %rax,%rax          # count = 0;
19         andq %rdx,%rdx          # len <= 0?
20         jle Done                # if so, goto Done:
21
22 Loop:   mrmovq (%rdi), %r10     # read val from src...
23         rmmovq %r10, (%rsi)     # ...and store it to dst
24         andq %r10, %r10         # val <= 0?
25         jle Npos                # if so, goto Npos:
26         irmovq $1, %r10
27         addq %r10, %rax         # count++
28 Npos:   irmovq $1, %r10
29         subq %r10, %rdx         # len--
30         irmovq $8, %r10
31         addq %r10, %rdi         # src++
32         addq %r10, %rsi         # dst++
33         andq %rdx,%rdx          # len > 0?
34         jg Loop                 # if so, goto Loop:
35 ##################################################################
36 # Do not modify the following section of code
37 # Function epilogue.
38 Done:
39         ret
40 ##################################################################
41 # Keep the following label at the end of your function
42 End:
```

Figure 3: **Baseline Y86-64 version of the `ncopy` function.** See `sim/pipe/ncopy.ys`.

```
unix>  ./check-len.pl < ncopy.yo
```

You may make any semantics preserving transformations to the `ncopy.ys` function, such as reordering instructions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions.

Hints for Part B

- You may find it useful to read about loop unrolling in Section 5.8 of CS:APP3e.
- You may find it useful to read about jump tables or other methods of control in Section 3.6

Building and Running Your Solution

In order to test your solution, you will need to build a driver program that calls your `ncopy` function. We have provided you with the `gen-driver.pl` program that generates a driver program for arbitrary sized input arrays. For example, typing

```
unix>  make drivers
```

will construct the following two useful driver programs:

- `sdriver.yo`: A *small driver program* that tests an `ncopy` function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 2 in register `%rax` after copying the `src` array.
- `ldriver.yo`: A *large driver program* that tests an `ncopy` function on larger arrays with 63 elements. If your solution is correct, then this program will halt with a value of 31 (`0x1f`) in register `%rax` after copying the `src` array.

Each time you modify your `ncopy.ys` program, you can rebuild the driver programs by typing

```
unix>  make drivers
```

You can build the default pipeline simulator for this lab by typing

```
unix>  make psim
```

If you want to rebuild the simulator and the driver programs, type

```
unix>  make
```

For your initial testing, we recommend running your solution in TTY mode, comparing the results against the ISA simulation:

```
unix>  ./psim -t sdriver.yo
```

If the ISA test fails, then you should debug your implementation by single stepping the simulator in GUI mode (more info on using GUI mode in hints at end of handout). To test your solution in GUI mode on a small 4-element array, type

```
unix>  ./psim -g sdriver.yo
```

To test your solution on a larger 63-element array, type

```
unix>  ./psim -g ldriver.yo
```

Once your simulator correctly runs your version of `ncopy.ys` on these two block lengths, you also need to perform the following additional tests:

- *Testing your driver files on the ISA simulator.* Make sure that your `ncopy.ys` function works properly with YIS:

  ```
  unix>  make drivers
  unix>  ../misc/yis sdriver.yo
  ```

- *Testing your code on a range of block lengths with the ISA simulator.* The Perl script `correctness.pl` generates driver files with block lengths from 0 up to some limit (default 65), plus some larger sizes. It simulates them (by default with YIS), and checks the results. It generates a report showing the status for each block length:

  ```
  unix>  ./correctness.pl
  ```

  This script generates test programs where the result count varies randomly from one run to another, and so it provides a more stringent test than the standard drivers.

  If you get incorrect results for some length K, you can generate a driver file for that length that includes checking code, and where the result varies randomly:

  ```
  unix>  ./gen-driver.pl -f ncopy.ys -n K -rc > driver.ys
  unix>  make driver.yo
  unix>  ../misc/yis driver.yo
  ```

  The program will end with register `%rax` having the following value:

  `0xaaaa` : All tests pass.

  `0xbbbb` : Incorrect count

  `0xcccc` : Function ncopy is more than 1000 bytes long.

  `0xdddd` : Some of the source data was not copied to its destination.

  `0xeeee` : Some word just before or just after the destination region was corrupted.

- *Testing your program against the default pipeline.* To ensure that your program is using instructions that are supported by the default pipeline model, execute

  ```
  unix> ./correctness.pl -p
  ```

  This performs the same set of tests as running `./correctness.pl` but using the default pipeline model.

Evaluation

This part of the Lab is worth 50 points: **You will not receive any credit if your code for** `ncopy.ys` **fails any of the tests described earlier.**

- 5 points for your description in the header of `ncopy.ys` and the quality of the description.
- 45 points for performance. To receive credit here, your solution must be correct, as defined earlier. That is, `ncopy` runs correctly with YIS.

  We will express the performance of your function in units of *cycles per element* (CPE). That is, if the simulated code requires C cycles to copy a block of N elements, then the CPE is C/N. The PIPE simulator displays the total number of cycles required to complete the program. The baseline version of the `ncopy` function running on the standard PIPE simulator with a large 63-element array requires 897 cycles to copy 63 elements, for a CPE of $897/63 = 14.24$.

  Since some cycles are used to set up the call to `ncopy` and to set up the loop within `ncopy`, you will find that you will get different values of the CPE for different block lengths (generally the CPE will drop as N increases). We will therefore evaluate the performance of your function by computing the average of the CPEs for blocks ranging from 1 to 64 elements. You can use the Perl script `benchmark.pl` in the `pipe` directory to run simulations of your `ncopy.ys` code over a range of block lengths and compute the average CPE. Simply run the command

  ```
  unix>  ./benchmark.pl
  ```

  to see what happens. For example, the baseline version of the `ncopy` function has CPE values ranging between 29.00 and 14.27, with an average of 15.18. Note that this Perl script does not check for the correctness of the answer. Use the script `correctness.pl` for this.

  You should be able to achieve an average CPE of less than 10. Our (TAs) best version averages 6.26. If your average CPE is c, then your score S for this portion of the lab will be:

$$S = \begin{cases} 0\,, & c \geq 12.5 \\ 11.25 \cdot (12.5 - c)\,, & 8.5 \leq c < 12.5 \\ 45\,, & c \leq 8.5 \end{cases}$$

  By default, `benchmark.pl` and `correctness.pl` compile and test `ncopy.ys`. Use the `-f` argument to specify a different file name. The `-h` flag gives a complete list of the command line arguments.

6 Gem5

For Part C, we will be using a simulator Gem5 [1] with x86 CPU model to collect performance metrics for two benchmark programs to analyze their performance bottlenecks. Gem5 is widely used by researchers and developers in academia and industry, supports the standard PC platform, and provides plenty of useful

[1] `http://www.gem5.org/Main_Page`

metrics for real workloads. You will run the binaries of the benchmarks using the Gem5 simulator with several CPU model configurations. A Gem5 tutorial will be conducted in recitation.

Among the many metrics provided by Gem5, we mainly will focus on Cycles Per Instruction (CPI) as the performance measure, and a *hardware cost* (C) to evaluate the efficiency of the machine model. We define *efficiency* (E) as $E = CPI \times C^{1/3}$. For this lab we will calculate C based on the sizes of two specific buffers (ROB and IQ) used in Out-of-Order (O3) pipeline.

6.1 Superscalar O3 Processor Pipeline

In this part you will be working with different parameters of a superscalar out-of-order (O3) pipeline to optimize performance for a few benchmarks.

This pipeline[2] is composed of 5 stages :

Fetch : fetch instruction and handle branch prediction.

Decode : decodes instructions, handle unconditional branches.

Rename : Rename the instruction using physical registers from the free list, will stall when the ressources for that have filled up. (This can then cause the previous stage to stall).

IEW : Issue Execute Write, this stage handles waiting until all the operands are available and executing the instruction. Gem5 doesn't show how long the instruction actually executes.

Commit : Handles instruction retirement in order. It also handles signalling the front end in case of branch misprediction.

6.2 O3CPU parameters

Below is a list of parameters which can be tweaked to improve performance using Gem5.

1. `numROBEntries`: Number of reorder buffer entries.
2. `numIQEntries`: Number of instruction queue entries (reservation station entries).

To check the CPI using the default set of parameters, run the `driver.py`:

```
unix>  python driver.py -C
```

- All parameters and their values for the simulation run can be found in a configuration file `config.ini` created in the output folder. For example the output folder for naive matrix multiplication is: `${ARCLAB-ROOT}/out/o3_mm_naive`.

[2]Official documentation at `http://gem5.org/O3CPU`

- ${ARCLAB-ROOT}/out/o3_{benchmark} also contains statistics for each run in a file named stats.txt, and a detailed trace of the pipeline content, named trace.out.gz. These file should help you analyze how changing each parameter improves or degrades performance. Since you have to provide reasoning for why changing one or multiple parameters improves performance, please monitor stats.txt while modifying the CPU model parameters. The trace can be visualized using a graphical tool named konata. The tool can be run on the shark machine as:

  /afs/cs.cmu.edu/academic/classes/18600-f18/konata-linux-x64/konata, if you connect with X11 forwarding.

  Alternatively you can download the binaries from https://github.com/shioyadan/Konata/releases [3] for macOS, Windows, or GNU/Linux distributions, and run it on your laptop. In this case you will need to retrieve the trace file using scp or rsync before opening it with the tool.

After saving your changes, you can run:

```
unix>  make rmgem && python driver.py -C
```

to get a new set of results.

7 Part C

40 points

First if you haven't done it yet, run make in the root directory of the lab to compile the various benchmarks.

7.1 Part C.1 Getting use to Gem5

In this part you will try to optimize a simple piece of code for the out of order processor of Gem5, in order to get familiar with how the processor work and with the visualization tool usage.

We will use the metric *number of cycles* required to run the interesting section of the benchmark, which is a couple of hundred instruction long with our initial naive implementation.

Your goal is to reduce the number of cycles required to run the computation, by modifying simple.c between the ROI_BEGIN and ROI_END calls.

To evaluate the number of cycles we will run ./driver.py -c1 which will run simple on the gem5 simulator. You can also use the run_gem5.py script we provide using the default values for IQ and ROB, if you want to specify a different output directory while you are making changes. In that case you should run

```
unix>  make simple && ./run_gem5.py --directory=<somename> --cmd=./simple
```

[3] You may have a look at http://learning.gem5.org/tutorial/presentations/vis-o3-gem5.pdf which presents the tool

For this part you will not tweak the micro architecture. To guide yourself and understand where the `simple.c` code is not efficient you may use the O3 pipeline trace support. The Out of Order processor output a view of its pipeline named `trace.out.gz` in the output directory. You can use Konata (preferably on your personal machines after retrieving the trace) to open the trace and visualize the pipeline. You should spend a couple of minutes looking at the trace to get an understanding of what is happening as this will help you with the next tasks. You will likely want to display the disassembly of the binary (with gdb or objdump) to map the various micro instruction to the corresponding instruction in the disassembly using their addresses.

You will be graded on the number of cycle that your implementation requires. (Proposed scale $> 720 : 0$pt, $< 620 :$ max points, linear in between).

7.2 Part C.2

In this part you will start tinkering with some CPU parameters to optimize the CPU configuration for individual benchmarks. This question will be hand graded along with part C.3, from a short report that you should write.

You will have to test various configuration and report in `o3_configs.py` the optimal configuration. In addition you have to report the results for the various configurations in your report, along with any insights that may explain the optimal configuration for each benchmark. These insights should prove useful for the next question.

The four benchmarks are, with between parenthesis the command structure to test a configuration (replace all the `<placholders>` by a proper value) :

daxpy : double precision A*X + Y, run

```
./run_gem5.py --directory=<somename> --cmd=./daxpy --options=-u
--IQ=<n> --ROB=<m>
```

mm : naive matrix multiply, run

```
./run_gem5.py --directory=<somename> --cmd=./mm --options=-nu --IQ=<n>
--ROB=<m>
```

modexp : modular exponentiation using the squaring method, run

```
./run_gem5.py --directory=<somename> --cmd=./modexp --IQ=<n> --ROB=<m>
```

tree : build an AVL balanced binary tree then traverse it, run

```
./run_gem5.py --directory=<somename> --cmd=./tree --IQ=<n> --ROB=<m>
```

The four configuration are showed in table 1.

Execute `./driver.py -c2` to run the four benchmarks with the configuration specified in `o3_configs.py`.

CPU Model Name	{numROBEntries, numIQEntries}
1. Small	$\{8, 4\}$
2. Medium	$\{16, 8\}$
3. Large	$\{32, 16\}$
4. XLarge	$\{64, 32\}$

Table 1: CPU Models

7.3 Part C.3

In this part you will set the `star` configuration in `o3_configs.py`. Your aim is to find the best configuration for running all the 4 benchmarks in terms of efficiency. For this question, the report with the reasoning for your star configuration is far more important than the actual values you provided in the script. You should also provide data in support of the configuration you propose. Also explain your methodology and how you reached your result. It is important to think about the various benchmark and results; interpret of these behaviour of the pipeline and document it in the document.

You can use the same `run_gem5.py` script as before just make sure to the four benchmark for each configuration. You can also use the driver with flag `-c3` to run the four benchmark using the `star` configuration in `o3_configs.py`.

Start from `numROBEntries=16`, and `numIQEntries=8`. Now assume you have an extra quota of 24 entries that you can use to increase the sizes of the ROB and/or IQ. This means that starting with $16 + 8$ entries you can now increase the total cost C from 24 to up to 48 (this total can be split between ROB and IQ per your choice).

Your task is to find the best configuration which works well for all four benchmarks in terms of efficiency E. Find the best balance between the two parameters, and report the best observed set of values in `o3_configs.py` as the star entry. The best configuration for each benchmark may not require using all 24 additional entries.

Evaluation

Part C is worth 40 points. For Part C.1, you will be rewarded points out of 10 by Autograder; 6 points will be given by TAs for the report submitted. For Part C.2, 12 points will be awarded for reporting the performance results for the four benchmarks (3 points per benchmark) on the four machine models, and identifying the best model for each benchmark; For Part C.3, 12 points for reporting the best star configuration, and analysis of the results while coming up with the best star configuration. Your analysis should contain empirical results like observed CPI values across different configurations, efficiency E values etc., and clear reasoning as to why a particular parameter or both are more relevant to reduce CPI in finding the star configuration.

You are welcome to refer and cite material from any of the course textbooks or lecture notes for the description.

Part	Points
1. C.1	$10 + 6$
2. C.2	12
3. C.3	12

Table 2: Distribution of points

8 The Driver

Finally, when you are ready to test your whole lab, navigate to your `arclab-handout` directory and run:

```
unix> python driver.py
```

This will run the same driver that autolab uses to grade your solution, and assign a score for both parts A and B. Part C will be graded manually by TAs.

9 Handin Instructions

- You will be handing in two sets of files:
  - Part A: `sum.ys`, `rsum.ys`, and `copy.ys`.
  - Part B: `ncopy.ys`.
  - Part C: `simple.c`, and `o3_configs.py`.
- Make sure you have included your name and ID in a comment at the top of each of your handin files.
- Each file submitted for parts A & B should also have a high-level description of your code. In each case, describe how and why you modified your code.
- For part C you will be submitting a PDF to gradescope of at most **3 pages** which will have description and analysis for all three parts.
- Detailed comments are to be provided for parameters modified as the case with star configuration. Do report the metrics observed using driver.py for the different CPU Models.
- To create a handin of your files for autolab, go to your `arclab-handout` directory and type:

  ```
  unix>  make handin
  ```

  This simply creates a arclab-handin.tar file containing the specified files, which then will be submitted to autolab.

10 Hints

- By design, both `sdriver.yo` and `ldriver.yo` are small enough to debug with in GUI mode. Some find it easiest to debug in GUI mode, and we suggest that you use it if you get stuck. GUI mode requires that secure X-forwarding be enabled for your ssh session with the shark machines. On unix machines, this generally means using `ssh -Y` instead of just `ssh`. Since this is largely dependent on your own computing environment, please be advised that the TAs may be no more competent than you at figuring out how to enable it on your machine. Below though are a collection of tips for getting it to work on various computing environments.
- If you have a Unix environment, you should be able to just use `ssh -Y` to connect
- If you have a Mac running a version of OS X, we recommend using XQUARTZ, as X11 is no longer included with OS X but might be if you have an older mac. Then, using the XQUARTZ terminal use `ssh -Y` to connect
- If you have a Windows machine, you have many random options. We have had success with Putty and Xming for Windows 10. A piazza post will be made for more info on this method.
- If you running in GUI mode on a Unix server, you may want to make sure that you have initialized the DISPLAY environment variable:

```
unix>  setenv DISPLAY myhost.edu:0
```

- With some X servers, the "Program Code" window begins life as a closed icon when you run `psim` or `ssim` in GUI mode. Simply click on the icon to expand the window.
- With some Microsoft Windows-based X servers, the "Memory Contents" window will not automatically resize itself. You'll need to resize the window by hand.
- The `psim` and `ssim` simulators terminate with a segmentation fault if you ask them to execute a file that is not a valid Y86-64 object file.
- CPI may vary slightly across the shark machines. Although it shouldn't change the best CPU model for each benchmark.