DAT105 Computer Architecture
Study Period II, 2017
Department of Computer Science and Engineering
Chalmers University of Technology, Göteborg, Sweden

Laboratory Assignment 1:
Exploring the Impact of Cache Hierarchy on Processor Performance

Contributors: Mehrzad Nejat, Mohammad Waqar Azhar, Per Stenstrom

Learning outcome: The objective of this laboratory assignment is to explore the impact of cache design on overall processor performance. You will use the SimpleScalar toolset [1] and the MiBench benchmark suite [2] to perform the experiments. At the end of this assignment, you should know how to:

- measure and report the performance of a processor model
- compare the relative performance of various processor models
- evaluate the impact of cache parameters on processor performance

Suggested reading: It is strongly recommended that you go through the following study material ([1] and [2] are available in Ping Pong Repository) before you start this assignment:

[1] D. Burger and T. Austin, "*The SimpleScalar Tool Set Version 2.0*", University of Wisconsin-Madison, Computer Sciences Department, Technical Report 1342, 1997.

[2] M. R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", In *Proceedings of IEEE International Workshop on the Workload Characterization (WWC-4)*, pages 3 – 14, 2001.

Evaluation: You must write a report that includes: A brief description of the problem and the method you used, important assumptions if you had, simulation results and observations, your design choices and the reason behind them. Detailed report guidelines will be available on Ping-Pong.

Simulation Environment

As depicted in Figure 1, the simulation environment basically consists of the following sections:

- **Benchmark:** This is a test program that is executed on the simulated processor. It can have its own inputs and outputs. You can run any program on the simulated processor; but in order to have a standard base for comparison, people usually use programs and input sets from well-known benchmark suites.
- **Configuration File:** the architectural simulator uses this file to set the configuration and parameters of the processor in simulation. This is not the only method for modifying the processor model. You can use parameter-specific command-line options or change the simulator code. But we will not use these methods in this lab. We provide a base configuration file. You can create different processor models by modifying this file.

- **Simulator:** This is the simulator program. It is executed on a real computer referred to as the *host machine*. The host machine can be your laptop, a local computer in the Lab, or a server. The simulator is a software model of a computer system and runs as software on the host machine. It includes probes for measurements while simulating the execution of benchmark on the computer system model being simulated with the specified configuration. The number of executed instructions, the number of loads and stores, and the average cycles per instruction are just a few examples of many parameters that the simulator can measure.
- **Results:** The simulator eventually reports the measurement results in some output files. After simulating different processor models, you can compare these results to evaluate the performance of the models when running a specific benchmark program with a specific input set.

In summary, you must remember that you are running a benchmark program on a simulated computer system model with the specified configuration on a real computer. Therefore, for example you have two different run times: program run time on the simulated processor, and run time of the simulation on the host machine. Simulation time is usually several orders of magnitude larger than program run time.

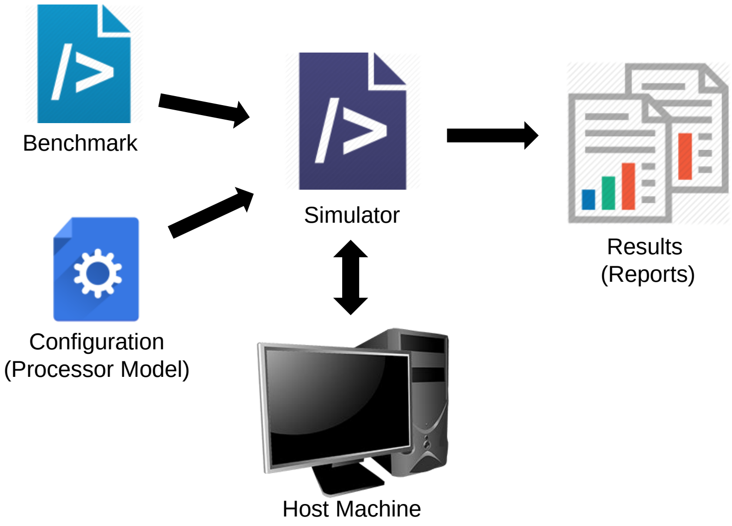

Figure 1: Simulation Environment

Running a simulation

You must run the simulator in a Linux system. You can use computers in the lab or connect to Chalmers server with Secure Shell (SSH) connection[1].
The required files can be found in the documents section of the course page on Ping-Pong. You must download '*simhome.tar.gz*'. After uncompressing, you can see three

[1] You can connect to the server even from a windows operating system

folders: '*apps*', '*bin*', and '*configs*'. The first one contains 5 benchmark applications from MiBench. We are using the small input sets to reduce the simulation time. The second folder contains the executable files of two Simple Scalar simulators. '*sim-profile*' is used for fast functional simulation of benchmark applications to provide statistics on instruction mix. We will not use this simulator in this lab; but you can try it yourself. '*sim-outorder*' performs detailed architectural simulation of an out of order processor. This is the main simulator that you will use. '*configs*' folder is where you place different processor models and simulation results are stored in the corresponding sub-folders. Besides these folders, you can find the base configuration file '*base1.txt*' and two scripts for running out of order simulator and profiling simulator: '*runsim_sim*' and '*runsim_profile*' respectively. Open these scripts and see how each simulation is initiated[1]. You can pass the name of the configuration as a command line argument to the script. You can deactivate any benchmark application by simply changing the corresponding lines to comment.
Remember, after each simulation, two different sets of outputs can be generated: Application outputs in the same location of the application executables, and simulation outputs in the same location as the configuration files. These locations are set in the script.

Task 1: Evaluating the base configuration

In this assignment you study a very simple system presented in Figure 2. Usually, there are two or three levels of cache between Processor and main memory. One reason is that bigger caches come with longer access time and each level tries to hide the access time of the next level. However, we want to start with a comprehensive yet interesting system. Hence, here we only have one level of cache with one cycle access time. But, we keep the instruction and data caches separated as it is in most of the real processors for level one.

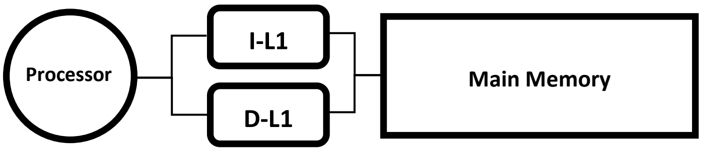

Figure 2: Architecture of the base system

We know that the execution time of a code is modeled as:

$$Execution\_Time = IC \times CPI \times T_{cycle} \qquad (1)$$

Where IC is instruction count, CPI is cycles per instruction and T_{cycle} is cycle time[2]. Because the processor frequency is constant in our case, we can remove cycle time from both sides of the equation and express all the times in processor cycles.

[1] If you are not familiar with Linux scripts, you should check at least some basic structures and simple examples on internet. It will not take a lot of your time, but it can be a powerful tool.

[2] The inverse of processor frequency

Furthermore, since we run the same program on different computer system models with the same ISA, IC will be the same. Therefore, we can divide both sides by IC and consider only CPI – Cycles Per Instruction. This is especially useful for comparing the performance of multiple applications with different number of instructions. This means CPI could be used as an important parameter for measuring the performance of a processor.

CPI can be divided into two components[1]:

$$CPI = CPI_0 + MPI \times MP \tag{2}$$

Here MPI and MP stand for number of misses per instruction and miss penalty in cycles. This is a simple model that applies to the system in Figure 2. The miss penalty component in (2) corresponds to the extra waiting time when a data or an instruction is not found in the data and instruction cache, respectively, and the processor needs to bring it from main memory. If we remove this component from CPI, what remains (we call it CPI_0) is the number of cycles per instruction related to processor operations and even if we had a perfect cache with no misses, it would not be affected.

Now, let's start by simulating the base model and measuring its performance.

- Place the provided "base1.txt" configuration file in the corresponding folder in '*simhome/configs*'. Open it and see how different processor configurations are set in this file. Specially check the configurations for caches. Use Simple Scalar user guide [1] to understand the parameters. You will change these parameters to create new processor models.
- Execute '*runsim_sim*' for the base model[2].
- Read the results from txt files starting with "*Stats_*" and fill out the first five rows of Table 1.
- Create another model and change memory access time to 1 instead of 200 (cache access times are already 1 cycle). This represents a processor with an ideal memory subsystem.
- Simulate the ideal model and collect the CPI results as an estimation of CPI_0 and fill out the corresponding row in Table 1.
- Using CPI_{base} and CPI_0, calculate the upper bound on speedup (SP_{ideal}). Use the following formula for calculating the speed up of any configuration x:

$$SP_x = \frac{CPI_{base}}{CPI_x} \tag{3}$$

- Choose the two applications that suffer the most from accessing memory. You will focus on these applications in the next task[3].

[1] Usually in a multi-level cache and when cache access time is longer than one cycle, the second component of this equation is expressed in more details that include the multiplication of accesses per instruction to each cache or memory by the corresponding access time.

[2] Hint: All the files that need to be executed, should have executable permissions.

[3] Hint: You can deactivate the simulations of each benchmark by changing the corresponding lines in the simulation script into comment. This way, you can easily reactivate it later.

Table 1: Simulation results for base configuration

Application	dijkstra	qsort	stringsearch	gsm-untoast	jpeg-cjpeg
Instruction Count*					
Execution Time in cycles					
CPI_{base}					
MPI I-L1					
MPI D-L1					
CPI_0					
SP_{ideal}					

* use number of committed instructions

Think about these question:

- How much is memory subsystem responsible for each application's execution time?
- If you could only optimize one cache to improve the performance, which cache would you choose?
- Is the effect of memory sub system on execution time similar for different applications?

Task 2: Optimization of the Caches

You have studied a base processor model in task 1. In this task, you will try to find a better model that improves the performance by modifying the cache parameters. But first, you are going to study the effect of each parameter individually on each cache. These parameters are cache size, associativity and block size. It is important that you change only one parameter at a time to isolate the effect of other parameters. For example, if you change associativity and cache size[1] together and observe some performance variations, you cannot determine how much each parameter in isolation impacts on the result.

Every time you want to move to the next step in simulations, create a new folder and configuration file in "*configs*" folder. You can start by copying '*base1.txt*' and then modifying it. In each step, alter only one parameter of one cache and derive MPI and CPI from the simulation result. Then calculate the speedup relative to base configuration and compare it with the upper bound you found in the previous task for an ideal memory subsystem. You can also elaborate the results by generating curves and graphs. Perform the simulations for the two applications you chose in the previous task. Whenever you make a decision, you should consider both applications.

[1] Hint: In configuration file, cache size is not directly available. It is the multiplication of number of sets, associativity and block size. Therefore, you must be careful when you want to change each parameter. Check simulator user guide.

2.1. Optimize D-Cache

2.1.1. Cache Size: First you will study the effect of cache size. Simulate different cache sizes while associativity and block size are unchanged and fill out Table 2. You might notice a point after which any further increase in size will not result in any noticeable improvement. Choose it as the optimum size. Otherwise, you can choose the maximum size. Compare these results with the upper bound on speedup.

Table 2: Performance Results for different D-Cache sizes

	4KB			8KB			16KB			32KB		
	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP
APP1												
APP2												

2.1.2. Associativity: Next, you change the associativity while the size remains equal to the optimum size you found in the previous step and the same unchanged block size. Similarly, find an optimum associativity after which improvements are not considerable.

Table 3: Performance Results for different D-Cache associativities

	1 way			2 way			4 way			8 way		
	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP
APP1												
APP2												

2.1.3. Block size: Finally, set cache size and associativity equal to the optimum values you found in the previous steps and change the block size.

Table 4: Performance Results for different D-Cache block sizes

	16 B			32 B			64 B			128 B		
	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP
APP1												
APP2												

2.2. Optimize I-Cache

You have found an optimum configuration for D-Cache so far. Start from base configuration and perform the same steps in task 2.1 for the I-Cache.

Table 5: Performance Results for different I-Cache sizes

	4KB			8KB			16KB			32KB		
	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP
APP1												
APP2												

Table 6: Performance Results for different I-Cache associativities

	1 way			2 way			4 way			8 way		
	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP
APP1												
APP2												

Table 7: Performance Results for different I-Cache block sizes

	16 B			32 B			64 B			128 B		
	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP	CPI	MPI	SP
APP1												
APP2												

2.3. Compare the results

By now, you should have found an optimum configuration for each cache. Put it together and make one final configuration named OPT1. But, choose one block size for both caches. Even though it is not impossible to have different block sizes, it complicates the system since these blocks of data move through different components of memory subsystem. Perform the simulations for the rest of the applications with OPT1, calculate speedup compared to base and fill out Table 8. Compare the geometric mean of the speed ups.

Table 8: Comparing the speedup of OPT1 model with an Ideal memory system

Application	**dijkstra**	**qsort**	**stringsearch**	**gsm-untoast**	**jpeg-cjpeg**	**GM**
SP_{OPT1}						
SP_{ideal}						

Think about the following questions:

- Do you have better answers for the final questions in Task 1 now?
- Which cache parameters have larger effect on performance? Or are the effects in a similar range?

- How close did you get to an Ideal memory system? What causes are preventing you from reaching the maximum speed up?

Task 3: A more realistic system

So far, we assumed that changing the cache size and associativity doesn't have any cost. But, this is not true in a real system. Other than area cost, larger caches with higher associativity comes with longer access time and higher power consumption. This is one of the main reasons of having multiple levels of a cache in real computers. We do not want to complicate the problem by looking into area or power costs. But, we can have a simple case with a more realistic assumption on performance cost of increasing size and associativity. Use the values[1] in Table 9 and update the cache access time in cycles in OPT1 and name the updated model OPT2. Run the simulation on all the benchmarks and calculated the speedups compared to base and fill Table 10. Note that the base model remains unchanged by applying the values in Table 9.

Think about these questions:

- How much is the effect of considering latency costs of your proposed cache architecture?
- Is it still worth making those changes? If yes, how much further increase in the latency costs do you think would destroy the improvement you achieved?

Table 9: Cache Access latency in cycles for different sizes and associativities

		Size			
		4 KB	8 KB	16 KB	32 KB
Associativity	1	1	1	2	2
	2	1	2	2	3
	4	2	2	3	3
	8	2	3	3	4

Table 10: Comparing the speedup of OPT1, OPT2 and Ideal memory system

Application	dijkstra	qsort	stringsearch	gsm-untoast	jpeg-cjpeg	GM
SP_{OPT1}						
SP_{OPT2}						
SP_{ideal}						

[1] Note that these values are hypothetical and not based on real measurements or data. They are only useful for this experiment.