Laboratory 11
Cache Memory and Processes
Computer Science 240

Caches = small, fast memories which contain current and recently/likely to be used data from the large, slower, main memory

Temporal and spatial locality make this possible

When programs have poor locality or characteristics that cause frequent cache misses, very poor (slow) performance can occur

The performance impact in such cases can be used to measure the dimensions of the cache

Cache Experiments on Real Hardware

- reps may be any positive number
- stride must be a power of 2 no larger than SIZE
- both functions perform the same task, but using different order, which affects the performance/time to complete the task

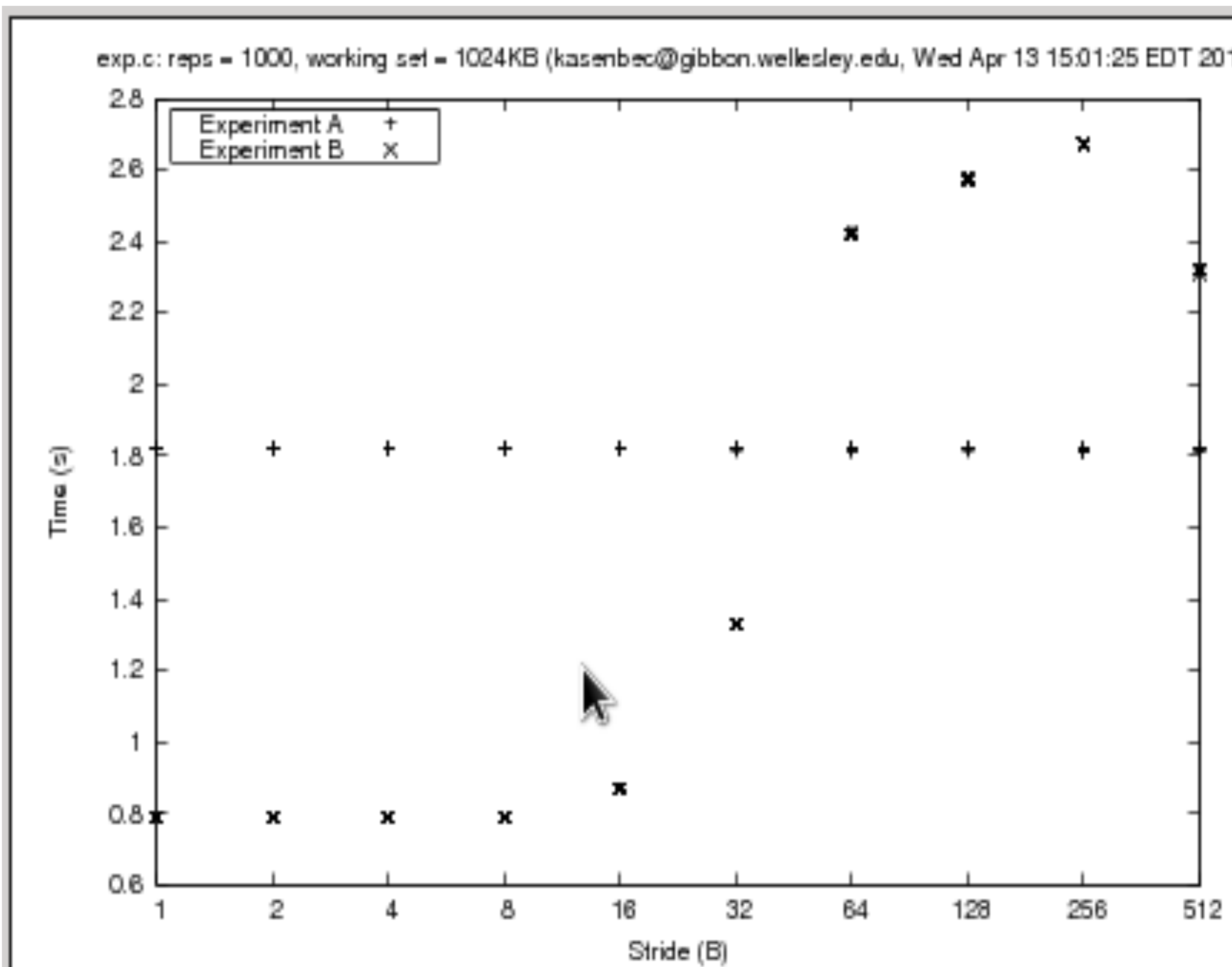
```
int experimentA(const int reps, const int stride) {

  assert(stride <= SIZE);

  int result = 0;

  for (int i = 0; i < SIZE; i += stride) {

    for (int j = 0; j < stride; j++) {

      for (int r = 0; r < reps; r++) {

        result += array[i+j];
        array[i+j]++;
      }
    }
  }

  return result;

}
```

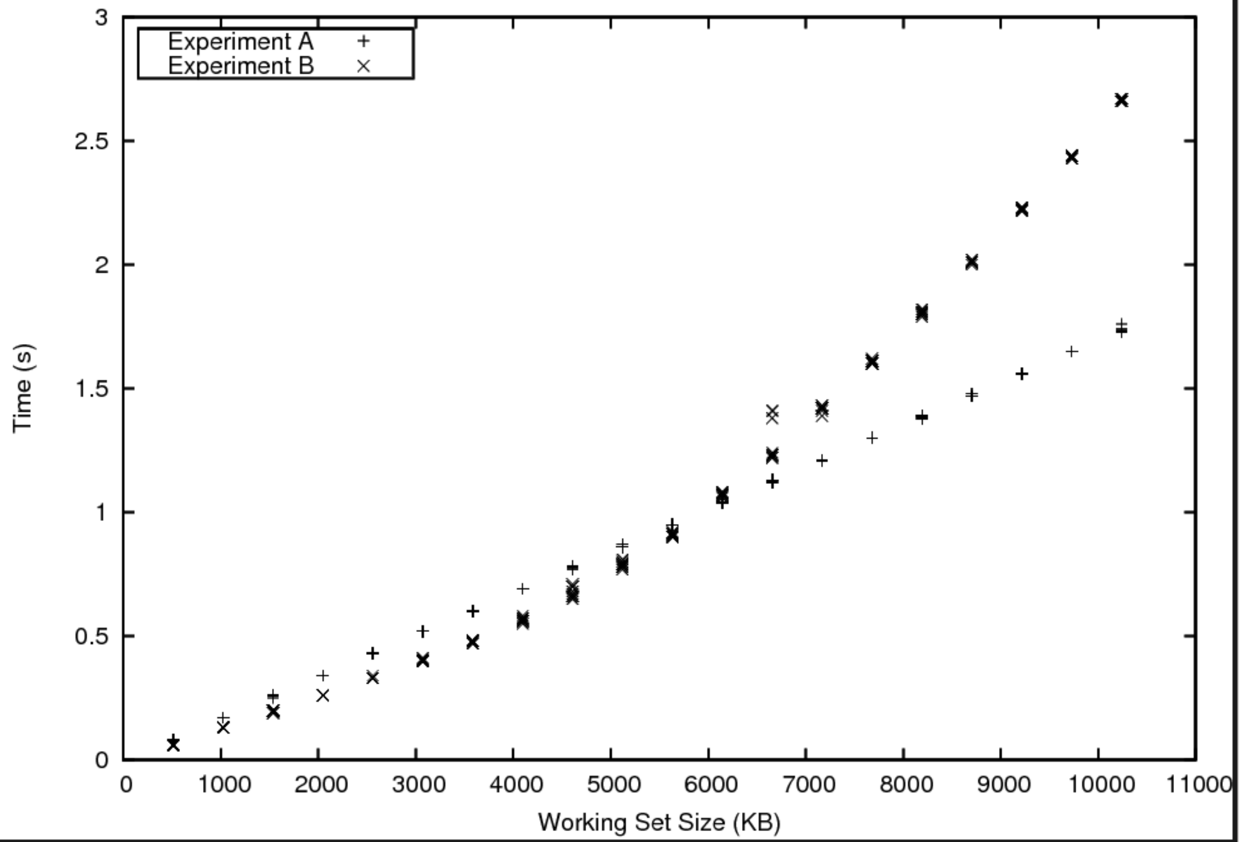
```
int experimentB(const int reps, const int stride) {

  assert(stride <= SIZE);

  int result = 0;

  for (int r = 0; r < reps; r++) {

    for (int j = 0; j < stride; j++) {

      for (int i = 0; i < SIZE; i += stride) {

        result += array[i+j];
        array[i+j]++;
      }
    }
  }

  return result;

}
```

Vary either size of array (working set size) or size of element (stride), and predict results: how do size of array and size of element (stride) affect the performance of either program?

Next use tools that allow you to run multiple trials, store the time for each experiment, and plot results.

The shape of the graphs helps you understand cache block size and cache size.




exp.c: reps = 100, stride = 32 (kasenbec@gibbon.wellesley.edu, Wed Apr 13 15:25:03 EDT 2016)
Experiment A
Experiment B
Time (s)
Working Set Size (KB)
0
0.5
1
1.5
2
2.5
3
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000

Cache Sleuth

*U*se a cache **simulator** and attempt to automatically discover the dimensions of mystery caches based only on observations of hits and misses for a stream of memory accesses.

> **getLineSize(...)** determines the line size of the cache.
>
> **getCapacity(...)** determines the capacity of the cache.
>
> **getAssociatvity(...)** determines the associativity of the cache.

Assumptions:

All caches use an LRU (least recently used) replacement policy.

All caches start empty (cold).

Processes

Operating System

The set of software that controls the overall operation of a computer system, typically by performing such tasks as memory allocation, job scheduling, and input/output control.

Kernel

The part of the operating system that runs in *privileged* or *supervisory* mode (has access to all instructions and memory in the system). It does all sorts of things like interact with hardware, do file I/O, and spawn off processes. The kernel is the center of the operating system that manages everything.

Shell

A user interface for access to an operating system's services, which translates user commands to low-level calls to the kernel.

Process

Instance of a program in execution. A process provides the illusion that the program has exclusive use of the processor and exclusive use of the memory system. In Linux, when you run a program by typing the name of an executable object file to the shell, the shell creates a new process with the help of the kernel.

Context
A program runs in the context of some process, where the context is the *state* needed to run correctly. State consists of:

- Program's code and data stored in memory
- Stack
- Registers
- Program Counter
- Environment variables
- Set of open file descriptors

Context Switch
The kernel maintains a context for each process. When the kernel pre-empts the running process with a new process or a previously running process, it is called a context switch: the context of the current process must be saved, the context of the new process must be asserted, and then control is passed to the preempting process.

System Calls
The **execve()** function replaces the current process' code and context (registers, memory) with that of a different program.

The **fork**() function is called by a *parent* process to create a new running *child* process. The child process is almost identical to the parent (it inherits an identical (but separate) copy of the address space, and all open files). The main difference is that the child has a different PID (process ID).

Fork is called by the parent process, but returns twice: once to the parent process, returning the value of the child PID, and once to the child, with a return value of 0.

The parent and child processes run concurrently, and their instruction flows can be interleaved by the kernel in an arbitrary way.

The **waitpid(pid)** function pauses execution of the process which calls it, and waits until the process with the specified pid terminates. It can be used to enforce a given order of execution for different processes.

The **getpid()** function returns the pid of the process which calls it.

Zombies

When a process terminates, it is not immediately removed from the system by the kernel. Instead, it is kept until the parent *reaps* the terminated child, at which point the kernel passes the child's exit status to the parent. Until it is reaped, it is called a zombie.

A zombie is not running, but does use memory resources to maintain some of its state.

Diagrams for Understanding Process Execution