Massachvsetts Institvte of Technology

Department of Electrical Engineering and Computer Science

6.313 Contemporary Computer Design

Final has been scheduled for Monday Morning December 18 in Room 26-310/314

Problem Set #9 Fall, 1995, Due Wed 22 November

1. Many computer systems use multiple levels of cache structures to provide good effective latency to memory systems. Level 1 and level 2 caches are one example of such an organization. Level 1 caches are typically on chip and intimately connected to the processor. Level 2 caches are off chip and can be larger. Larger caches have a correspondingly longer latency but larger size, and hence a lower miss rate.

a) Derive an expression for the expected memory access latency as a function of the miss rates and latencies of L1 and L2 caches and for memory.

b) You collect the following statistics (real) on a variety of computer systems.

System	L1 cache lat. ns	size	L2 cache lat. ns	size	Memory lat. ns
Linux i586@120	8	8K	107	256K	150
FreeBSD i586@133	7	8K	111	256K	181
Linux i686@167	12	8K	36	256K	194
UNIX SV x86at@165	14	8K	34	256K	196
AIX rs6000-990@71	–	–	13	256K	260
SunOS-5.5 sun4u@167	6	16K	42	512K	270
OSF1 alpha@147	12	8K	67	512K	291
Linux alpha@275	6	8K	46	96K	357
AIX powerpc@134	6	16K	164	512K?	394
OSF1 alpha@303	3	8K	66	4M	400
HP-UX 9000/819@99	–	–	10	1M?	430
IRIX IP22@250	8	16K	64	2M	1170
IRIX IP19@250	8	16K	64	4M	1189
SunOS-5.5 sun4d@50	20	8K	140	1M	1236

Assume the following miss rates as a function of cache size and calculate the expected latency of a memory access read for each of these systems.

Size	Miss Rate
8K	4%
16K	3%
256K	0.5%
512K	0.4%
1M	0.3%
4M	0.25%

c) Most operating systems also page their memory accesses by using the disk as an additional stage in the memory hierarchy. Extend the cache miss model we have discussed by thinking of the page faults as a form of cache miss. Given that a page fault has a typical latency of 15 milliseconds, derive the maximum allowable page memory miss rate for memory accesses which is required to allow a processor to run at no less than half of its nominal performance when operating in a paged environment. Assume a processor makes a memory reference every 6 ns on average, and that the effective latency of these accesses is 4 ns as seen through all of the caches (not including paging).

2. As we saw in class, the lack of prior knowledge about the expected direction of conditional branch instructions led to branch delay slots in simple FRAMW pipelined architectures, and to performance problems even in more sophisticated processors, because we wish to overlap the fetching and execution of subsequent instructions during execution of the branch.

a) Invent a strategy based on the relative addresses of the branch instruction and the address being conditionally branched to (the so-called *branch target*) which can be used by the processor to guess whether it should start executing instructions following the branch, or instructions at the branch target.

b) Compilers sometimes know through programmer provided information or profiling program execution which branches are likely to be taken and which are likely to fall through. What modifications or additions are needed to the processor instruction set to allow the compiler to express this knowledge in a way such the processor could make use of it.

c) Cache techniques can be used to rely on temporal behavior consistency to predict branch directions effectively. What information should a processor store to help it guess the direction of future branches? Where should this information be stored? What additional implementation resources are required? When during a FRAMW pipeline is this information available for writing?

d) Another approach used to assist in correct branch prediction is to replace uses of simple conditional branches with conditionally executing instructions. Explain how the inclusion of an absolute value instruction, for example, could improve the pipeline performance of a FRAMW machine.

e) Certain high level language features, such as method dispatches in object oriented languages are difficult to execute effectively in standard pipelined processors, because the branch target of the method dispatch is dynamically computed. What architectural features would you add to improve the performance of a processor for such branch instructions?

f) The whole idea of caches is to improve the *latency* of memory access. Bandwidth is relatively cheap. In executing uninterrupted sequential instructions, the location of the next instruction is easily predictable. How can we make use of this easy predictability to reduce the size of the instruction cache, while maintaining much of the performance, at the cost of increased memory bandwidth into the processor?