More Cache Aware Programming on Multicores

Contents of Lecture 11

- Cache Misses
- Reduce Communication
- Impove Locality
- Data Prefetching

Cache memories

- Faster but smaller memories than normal RAM
- When a variable is in the cache (a *cache hit*), reading it is fast
- At a *cache miss*, a block with e.g. 128 bytes is copied from RAM
- A cache miss and can take hundreds of clock cycles
- Except in Sequential Consistency, writing to the cache is also fast
- In SC it depends on if the cache already owns the cache block
- Recall cache block ownership in cache coherence protocols
- The time it takes to copy data from RAM is called the cache miss latency

Locality of references

- *Temporal locality*: After a variable `X` has been used, it is likely it will be used again soon
- *Spatial locality*: After a variable at address `&X` has been used, it is likely a variable at address `&X+1` will be used soon
- Caches are for programs with locality of references
- Fast programs need to have locality of references

Cache misses in multicores

- Misses in uniprocessors:
  - compulsory misses (cold misses),
  - capacity misses, and
  - conflict misses
- In addition to those found in sequential programs, we also have:
  - True sharing miss: essential miss since it communicates data
  - False sharing miss: non-essential miss.
- False sharing misses are due to using a large cache block size
- If only one variable at a time would be copied from RAM they would disappear
- But that would be inefficient

False Sharing Miss

- Assume a cache block size of two words.

Access	*Processor 1*	*Processor 2*	*Comment*
1	`Load 0`		Cold miss
2		`Load 1`	Cold miss
3		`Store 1`	Invalidation
4	`Load 0`		False sharing miss

Effects of larger cache block size:

- Increased benefit from spatial locality (prefetching within block)
- The larger risk of suffering from false sharing.

True Sharing Miss

Access	*Processor 1*	*Processor 2*	*Comment*
1	`Load 0`		Cold miss
2		`Load 1`	Cold miss
3		`Store 1`	Invalidation
4	`Load 0`		True sharing miss
5	`Load 1`		Reads a new value

- While we *cannot* know it at the time of Access 4, that miss is a true sharing miss (which we realize at Access 5).

Reducing false sharing

- Suppose each thread should count something.
- The following will result in false sharing

```
int      count[NUM_THREADS];

/* .... */

count[thread->index] += 1;
```

- It is better to collect the variables a thread should use in a struct that only that thread will modify.

Reduce also true sharing

- Ideally, each thread should work on its own data and no other should be involved. No communication and no true sharing.
- This is normally impossible for most algorithms, though.
- True sharing can be reduced with clever decisions of which thread should work on which data

Examples of tricks to exploit caches better

- Use smaller data structures: an `int` instead of a pointer.
- Use arrays instead of linked-lists if possible
- If a node's neighbors never change you can do:

```
struct node_t {
        edge_t* a;      /* array of edges. */
        int     n;      /* neighbors.      */
};
struct edge_t {
        int     v;      /* the other node.    */
        int     i;      /* edge number.       */
        int     b;      /* direction from lab0 */
};
```

- Keep track of the capacities and flows somewhere else.

Examples of tricks to exploit caches better

- Pad structs to fit cache blocks better — to avoid multiple cache misses per struct
- This can be done with a cache array with a suitable size if you know the cache block size.
- Put struct fields used at nearly the same time near each other
- Avoid putting smaller and larger struct fields next to each other in a struct to avoid padding between them.

Cachegrind

- `valgrind --tool=cachegrind ./a.out < 4huge.in`

```
f = 9924
==2250753==
==2250753== I   refs:      182,135,320
==2250753== I1  misses:          2,006
==2250753== LLi misses:          1,916
==2250753== I1  miss rate:        0.00%
==2250753== LLi miss rate:        0.00%
==2250753==
==2250753== D   refs:       79,372,178  (51,287,248 rd   + 28,084,930 wr)
==2250753== D1  misses:      1,690,859  ( 1,510,713 rd   +    180,146 wr)
==2250753== LLd misses:      1,416,910  ( 1,239,883 rd   +    177,027 wr)
==2250753== D1  miss rate:         2.1% (       2.9%     +        0.6%  )
==2250753== LLd miss rate:         1.8% (       2.4%     +        0.6%  )
==2250753==
==2250753== LL refs:         1,692,865  ( 1,512,719 rd   +    180,146 wr)
==2250753== LL misses:       1,418,826  ( 1,241,799 rd   +    177,027 wr)
==2250753== LL miss rate:          0.5% (       0.5%     +        0.6%  )
```

operf on Power

- ophelp lists all events that can be sampled
- `operf -e PM_LD_MISS_L1:100000 ./a.out < big/002.in`
- `opannotate -s a.out`

```
  83  0.8820 :                while (p != NULL) {
 551  5.8555 :                        e = p->edge;
5625 59.7768 :                        p = p->next;
             :
 455  4.8353 :                        if (u == e->u) {
 576  6.1211 :                                v = e->v;
             :                                b = 1;
             :                        } else {
 773  8.2147 :                                v = e->u;
             :                                b = -1;
             :                        }
             :
1221 12.9756 :                        if (u->h > v->h && b * e->f < e->c)
             :                                break;
             :                        else
  63  0.6695 :                                v = NULL;
```

Data Prefetching

- The purpose is to fetch data so that it is available in the cache when it's needed.
- Compilers and hardware can do this for matrix codes.
- This is very difficult on recursive data structures such as lists or trees.
- Suppose we have a loop which traverses a list or tree.
- To prefetch a node needed e.g. three iterations ahead, we need to dereference multiple pointers where each dereference can result in a cache miss.
- In a superscalar processor with out-of-order execution of load instructions (i.e. a relaxed memory consistency model), this can possibly be useful.
- In a processor with a blocking cache, the pipeline will halt at the first cache miss and make the prefetching almost useless.

An Approach to Prefetching Nodes

- A problem with lists and trees is that we usually do not know the address of a node needed in the future.
- This is true if we allocate memory with standard methods such as `malloc`
- However, assume the size of a data structure is fixed for some time.
- Then we can put pointers to the nodes in an array in the expected order of traversal, and then we may be able to prefetch nodes sufficiently in advance.
- This can be useful if we will traverse a data structure multiple times.

More difficulties

- For shared data we intend to modify, it can be useful to prefetch it in exclusive mode, meaning that we request ownership of the cache block.
- The effect of this is:
  - Reduced write penalty in a sequentially consistent machine.
  - Reduced write traffic in all machines.
- However, with the ownership requests, there is a risk that we introduce additional cache misses!
- Measurements are needed, but note they are dependent both on the
  - Input data
  - Machine parameters such as number of processors, cache sizes, and latency.

Prefetch with GCC

```
void __builtin_prefetch(const void *addr, int write, int loc);
```

- ```
  for (i = 0; i < n; i++) {
          a[i] = a[i] + b[i];
          __builtin_prefetch(&a[i+j], 1, 1);
          __builtin_prefetch(&b[i+j], 0, 1);
  }
  ```
- The `loc` has values in 0..3 with 0 no temporal locality and 3 most temporal locality
- Some CPU's have extra buffers to save temporary data there instead of polluting the cache
- Data prefetch does not generate a segmentation fault if the address is invalid.
- The expression computing the address obviously must be valid.
  ```

Data Prefetching on Power

- Several processors, including Power, do prefetching of array references in hardware
- Of course, the CPU does not know it is arrays
- They work by discovering a constant stride (or distance between used addresses) and then predict which blocks will be required.
- Modern processors (including Power) have prefetch instructions: `dcbt` and `dcbtst`
- Power also supports software programmable prefetch engines.

Software Controlled Stream Prefetch on Power

- Four data streams can be prefetched concurrently
- The basic instruction is `dst` — data stream touch
- One of the instruction fields is a two bit stream selector
- Other parameters:
  - Prefetch unit size S in 16-byte blocks: 0..31 where 0 means 32.
  - Number of units to prefetch
  - Distance D in bytes between two units (i.e. stride)

Cache-miss initiated software controlled prefetch engines

- Hardware knows what is happening now and the compiler what will happen in near the future
  - Treat L2 cache misses as light weight exceptions — there soon will not be much to do for the processor to do anyway.
  - Such exceptions do not involve the OS kernel but simply jump to a special place in the program.
  - For certain references in certain loops, the compiler has created an exception handler which will program a prefetch engine.
- The exception handler is a part of the function's control flow graph so it has access to all local variables which are register allocated both for the function and the exception handler.
- Therefore the exception handler can compute what to prefetch while the L2 cache miss is being serviced.
- The instruction overhead of always prefetching is removed.
- Knowing whether to insert prefetch instructions or not can be impossible, e.g. for `memcpy`.

Storing zeroes

- Consider a directed graph where each node has a set X, represented as a bitvector
- In each iteration of a certain loop the union of successor nodes' X is computed
- No member is ever removed from a set.
- $X = \bigcup X_i$
- Implemented as $X = X \cup X_i$ in a loop
- Why can it be better to start with setting X to zeroes?