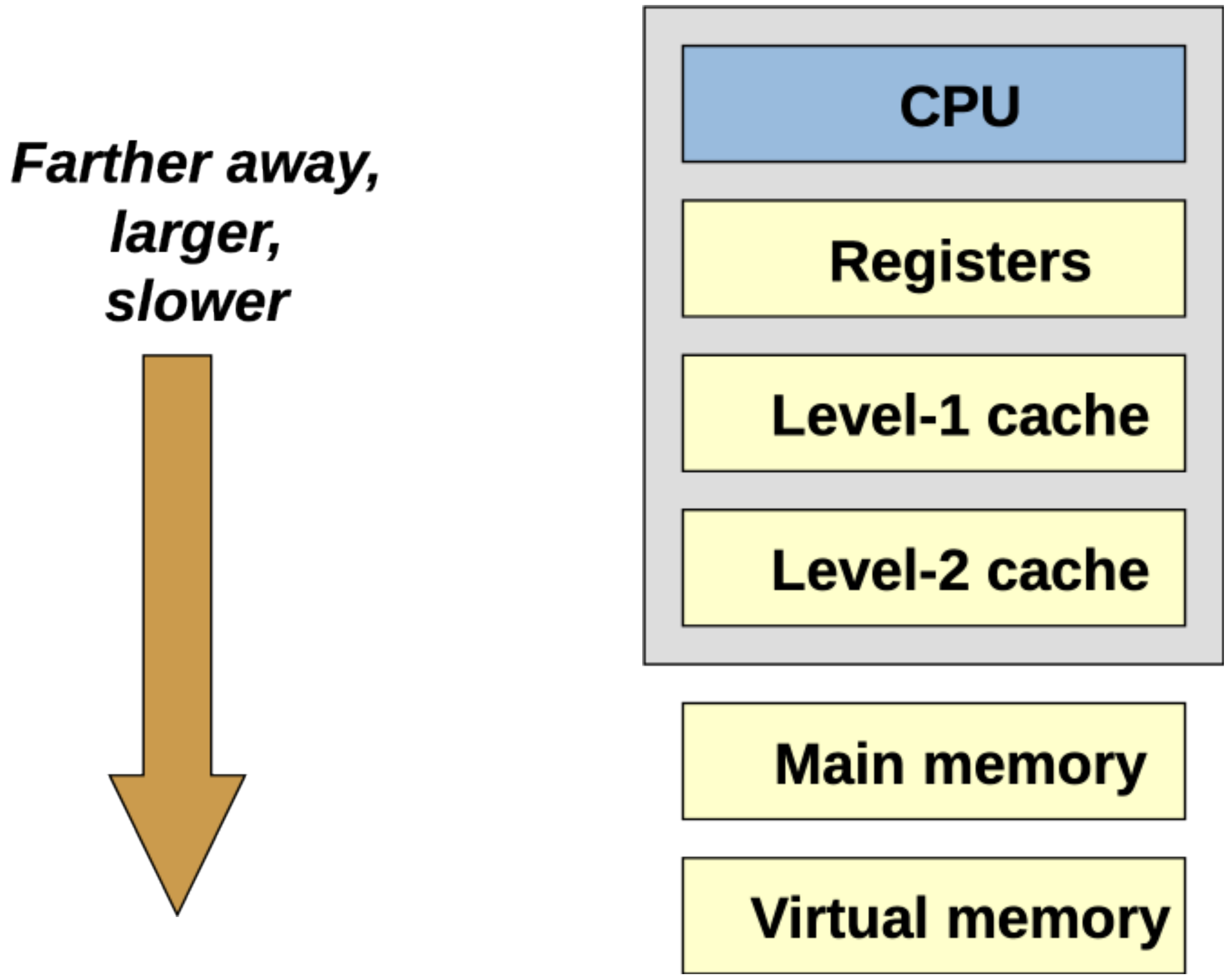
CS614: Advanced Compilers

Cache Optimizations

Manas Thakur
CSE, IIT Bombay

Spring 2025

Remember the hierarchy?



Pentium 4 3.2 Ghz	*Core 2 Duo*	*Athlon 64*
1 cycle	1 cycle	1 cycle
2 cycles (16 KB)	3 cycles (64 KB)	3 cycles (128 KB)
19 cycles (2 MB)	14 cycles (2 MB)	13 cycles (1 MB)
204 cycles	180 cycles	125 cycles
millions of cycles	millions of cycles	millions of cycles

➤ We have worked with registers; now is the time to move down (cache memory).

Data Prefetching

- We have cache memory for *temporal* and *spatial* locality.
  - Temporal: Same data may be accessed again.
  - Spatial: Nearby data may be accessed soon.

`prefetch` is a hardware-supported instruction, and needs to be preserved across compiler transformations.

- ***Data prefetching*** aims to utilize temporal and spatial locality (through the compiler).

```
for (i = 0; i < N; i++) {
    sum = sum + A[i];
}
```

$\Rightarrow$

```
for (i = 0; i < N; i++) {
    prefetch(A[i+pd]);
    sum = sum + A[i];
}
```

- Prefetch distance (`pd`) specifies the number of iterations before prefetching a data element and its access.
  - Needs to be determined carefully based on access pattern and latency.

Accuracy and Timeliness

- Is there a guarantee that the prefetched data will be accessed in future?
- What if the prefetch distance crosses the data structure boundary?
- Should we always attempt to compute the prefetch distance?

```
for (i = 0; i < N; i++) {
    if (A[foo(i)] > 100)
        sum = sum + B[i];
}
```

- `foo` may have side effects
- Call to `foo` may be expensive

- What if the prefetch is done so late that its benefits do not even matter?
  - Most trivial (but highly likely useless) `pd` value is 0!
- What if prefetch is done so early that the prefetched data gets evicted before use?
  - But we should be able to finish the prefetch before starting with the iteration that needs the prefetched value.

Profitability

- No. of cache misses without prefetching = `MAX/64`
- Prefetching overhead = `MAX` cycles
- Even if all the misses are prefetched,
  - Cycles saved by prefetching = `MAX/64 * c`
- For prefetching to be beneficial,
  - `MAX/64 * c > MAX ==> c > 64`
- What if we ***unroll the loop*** body 64 times?
  - Prefetching overhead reduces to `MAX/64` cycles!
- Thus, prefetching is beneficial when,
  - `MAX/64 * c > MAX/64 ==> c > 1`
  - Thus, even a cache-miss cost of >1 cycle would imply that prefetching is beneficial.

L1 cache line size: 64 Bytes
Issue width: 1
Cache miss cost: c cycles

```
char A[MAX];
for (i = 0; i < MAX; i++) {
   prefetch(A[i+pd]);
   sum += A[i];
}
```

Prefetching *affine* array accesses

```
for (i = 0; i < N; i++) {
    sum = sum + A[i];
}
```

1. Perform locality analysis.
   - A is *reused* in each iteration with an increment of 1.
2. Identify accesses requiring prefetch.
   - Assuming L1 cache line is 64 Bytes and A is stored in row-major order, we may prefetch every 64 iterations.
   - Identify *leading references*: e.g. 0, 64, 128, etc.
3. Perform ***loop decomposition*** to insert prefetch instructions at the right place.

Avoids unnecessary prefetch every original iteration.

```
for (i = 0; i < N; i += 64){
   // prefetch here
   sum += A[i];
   for(ii = i+1; ii < min(N, i+64); ii++){
       sum += A[ii];
   }
}
```

Prefetching non-array accesses

- Arrays in loops often have regular access patterns (i.e., constant *strides*).
- How about arrays with non-affine indices?
- Traversal of a recursive data structure such as a linked list?
- Compilers may need *profiling feedback* to insert suitable prefetch instructions.
- If we know accesses are regular but the stride keeps changing after some time, we can compute pd dynamically:

```
while (ptr != NULL) {
   sum += *ptr;
   ptr = ptr->next;
}
```

→

```
prev = ptr;
while (ptr != NULL) {
   sum += *ptr;
   stride = ptr - prev;
   prefetch(ptr+K*stride);
   prev = ptr;
   ptr = ptr->next;
}
```

K needs to be chosen carefully based on the profile.

Data Layout Transformations

- Data prefetching aims to bring future accesses to the cache ahead of time.
- What is the **underlying idea**?
    - Bring in ***frequently-accessed-together*** data items in the cache at the same time.
    - Needs to be done because such data items may not be allocated together in memory.
- How about already keeping such data items **together in the memory** itself?
- Leads way to compilers changing the layout of data structures at the source-code level itself.

Field layout

- Group hot fields together and separate them from cold fields.
  - Can be found statically, dynamically, or with a hybrid strategy.
- Optimization called ***structure splitting***:

```
struct S1 {
   char hot[4];
   char cold[60];
};
S1 s1[512];
```

```
struct S1_cold {
   char cold[60];
};
struct S1 {
   char hot[4];
   struct S1_cold *cold_link;
};
S1 s1[512];
```

- Good: `s1` may now fit within the L1 cache.
- (Slightly) Bad: Additional level of indirection to access the cold fields.

Field layout (Cont.)

- We can even get rid of the indirection cost by ***structure peeling***:

```
struct S1 {
   char hot[4];
   char cold[60];
};
S1 s1[512];
```

```
struct S1_cold {
   char cold[60];
};
struct S1_hot {
   char hot[4];
};
S1_hot s1_hot[512];
S1_cold s1_cold[512];
```

- Addendum: Requires program refactoring to modify the usages of `s1`.
- Difficult when the structure type has a pointer to itself.

Field layout (More considerations)

- Structure peeling may have performance penalties when the size of the record structure is large:

```
struct S2 {
   char hot1[32];
   char hot2[32];
   char not_so_hot[32];
};
struct S2 s2[128];
```

```
for (i = 0; i<128; i++) // Invoked N times
    x += foo (s2[i].hot1);
...
for (i = 0; i<128; i++) // Invoked N times
    x += foo(s2[i].hot2);
...
for (i = 0; i<128; i++) // Invoked N/10 times
    sum3 += foobar(s2[i].not_so_hot) + foo(s2[i].hot1);
```

- Fields `hot1` and `hot2` are accessed more frequently than the field `not_so_hot`, but `not_so_hot` is always accessed before `hot1` (*so, they should be placed together*).
- Benefit of structure peeling depends on the cost of additional misses incurred in the third `for` loop.

Data object layout

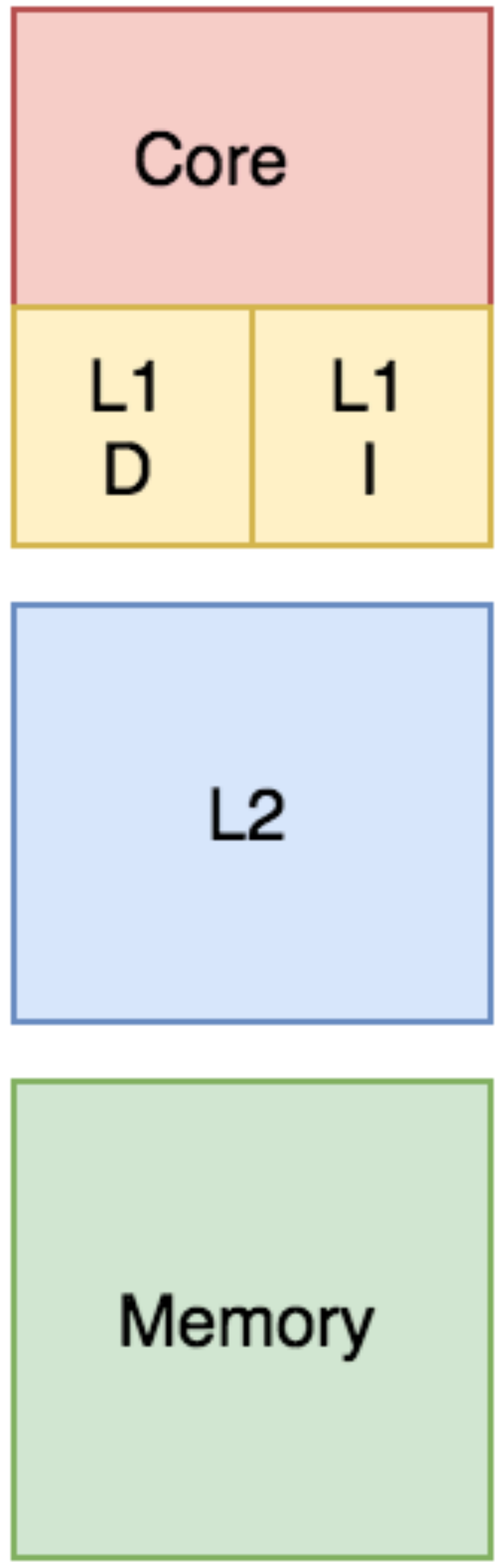
- Size of a single structure may rarely exceed the size of a cache block, but what about laying out (potentially large) objects themselves together?
- We need to be careful about introducing cache-line conflicts:
  - Two distinct cache lines containing two different data objects may conflict in the cache.
  - If those two data objects are accessed together frequently, a large number of conflict misses may result.
- **Example:** Different data-structure instances, if accessed frequently in individuality, may be ***pooled*** together in memory.

```
struct list;
struct tree;
struct linked_list{
    int n;
    struct linked_list *next;
};
struct tree{
    int n;
    struct tree *left;
    struct tree *right;
};
struct linked_list *l1, *l2;
struct tree *t;
...
```

`l1`, `l2` and `t` may be pooled together, based on access patterns.

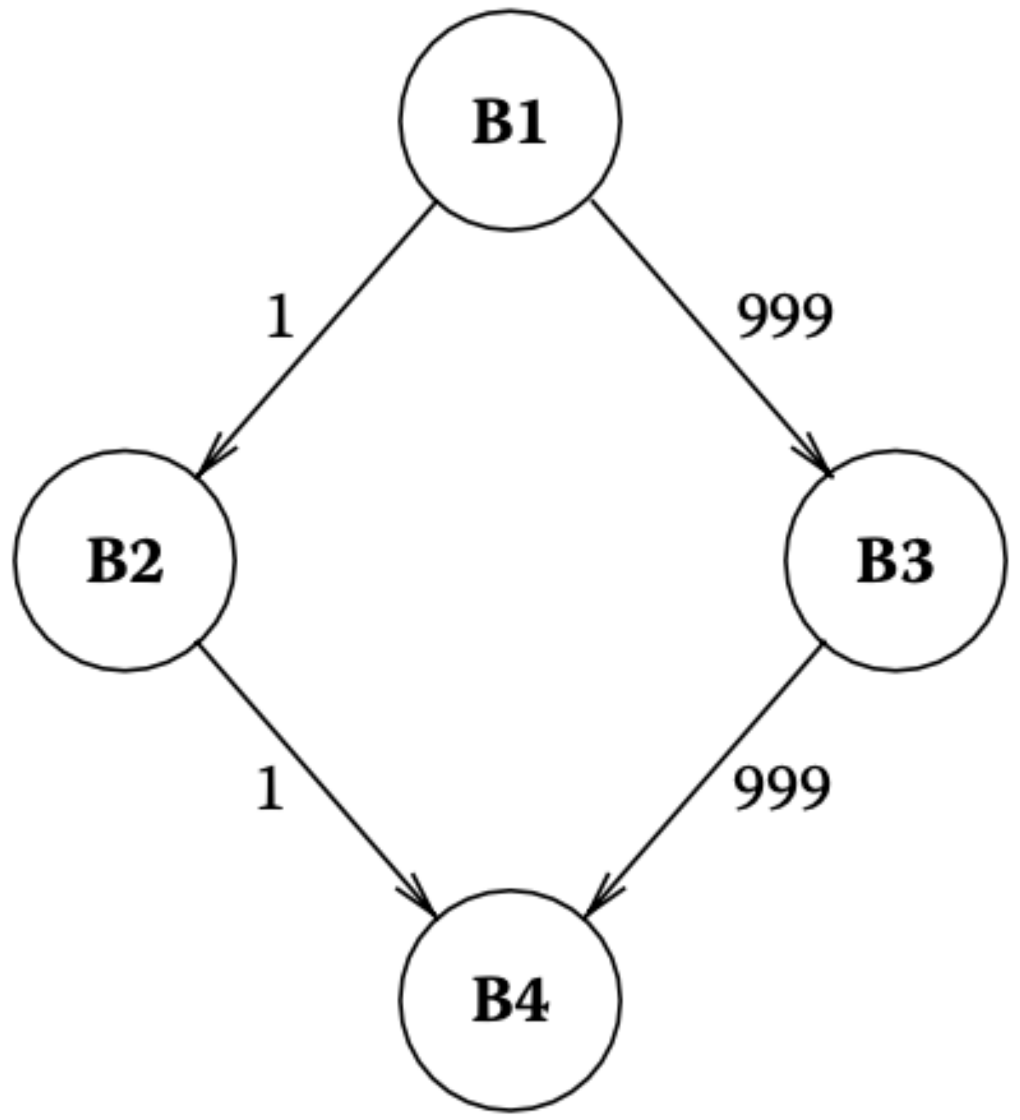
But we have both D-cache and **I-cache**

- Most processors have two separate L1 caches for data (**L1D**) and instructions (**L1I**).
- Anything if needs to be fetched from the main memory has higher cost than from the cache.
- Data prefetching could be done based on data-access patterns; how about keeping the right set of future instructions in the instruction cache?
- Requires finding access patterns in the control flow.



Code layout in procedures

- Consider the following "profiled" CFG:



- Often, B3 is executed after B1 (e.g., in loops the branch is taken most of the times).
- We may want to place the instructions in B1, B3 (and if fits, even the ones in B4) together in the virtual-address space.
- Optimization called *procedure splitting*.

Cache-line coloring

- Even with hot-code basic-block separation, we may incur cache misses if a caller and callee are mapped to the same cache line.
- `foobar` calls `foo` (and transitively, `bar`) in a hot loop.
- If `foo` (or `bar`) are mapped to the same cache line as `foobar`, we may incur an I-cache miss in each iteration.
- Cache-line coloring takes a program's "call graph" as input, and colors the same such that K levels of callers-callees do not get assigned the same colour.
- Differently colored procedures get different cache lines assigned during execution.

```
void foo(){
   bar();
   ...
}
int foobar(){
   for (i=0; i<n; i++)
      foo();
   ...
}
```

Optimizing cache behaviour without prefetching

- Almost all kinds of software (compiler) prefetching require hardware support.
- Layout transformations require changes in the data structures.
- Can we *change computations* instead, to take advantage of locality?
- **Next class onwards:** Loop Transformations (probably the most involved as well as powerful optimizations modern compilers perform).
- Techniques naturally lead way to loop *parallelization* and *vectorization*.