Eleven advanced cache optimizations:

1. **Small and simple caches** [hit time reduction technique]
   - → Small cache helps the hit time and hence smaller hardware is faster.
   - → Keep the cache small enough to fit on the same chip as the processor.
   - → Avoid time penalty of going off chip.
   - → Keep the cache simple.
   - → It overlaps the tag check with the transmission of data.

2. **Way prediction:**
   - → It is a method to combine fast hit time of direct mapped & have the lower conflict misses of 2-way SA cache.
   - → Block predictor bits are added to each block of cache.
   - → These bits select which of the blocks to try on next cache access.
   - → Mux is set early to select desired block.
   - → Only a single tag comparision is performed.
     - * if a tag matches, predicted block returned in 1 cycle.
     - * if tag fails, rest of the block checked in second cycle.

3. Trace caches:

→ Higher ILP needs instructions every cycle. So, the solution for this is trace caches.
→ Trace caches contains dynamic trace of executed instructions.
→ Branch prediction is now implemented by cache.
→ Better utilization of long blocks in instruction trace caches.
→ Drawback of trace cache is that instructions may appear multiple times in multiple dynamic trace due to different branch outcomes.
→ key idea is to pack multiple non contiguous basic blocks into one continuous cache trace line.

4. Pipelined Cache Access:

→ pipelined cache access to maintain bandwidth, but to have higher latency.
→ Instruction cache pipeline stages:

Pentium: 1 stage
Pentium Pro through pentium III: 2 stages
Pentium 4: 4 stages.

→ Greater penalty on mispredicted branches.
→ More clock cycles between issue of load & use of data

5. Non-blocking caches to (increase cache bandwidth)

→ Pipelined computers that allow out of order completion, the processor need not stall on a data cache miss.
→ Processor would continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data.

→ Thus non-blocking cache allows data cache to continue to supply cache hits during a miss.
→ cache may further lower the effective miss penalty if it can overlap multiple misses.
→ But it is hard to measure miss penalty due to overlap of hits & misses.

6. Multibalanced caches:
→ Divide cache into independent banks that can support simultaneous accesses.
→ Banking works best when the accesses naturally spread themselves across the banks.
→ mapping of addresses to banks:
  → spread the addresses of the banks sequentially across the banks.
  → called as 'sequential interleaving'.

7. Critical word first: [to reduce miss penalty]
→ cache block size tends to increase to exploit spatial locality.
→ Any given reference needs only one word from a multi word block.
→ CWF fetches requested word first & sends it to processor.
→ Processor continues execution while rest of the block is fetched.
→ Early restart: Fetches words in the order stored in the block as soon as critical word arrives, sends to processor & processor restarts.

8) Merging write buffer: [To reduce miss penalty]

→ Processor blocks on write if write buffer is full.

→ Processor checks write address with address in write buffer.

→ Processor merges writes to some address if address is present in write buffer.

→ victim caches: Tiny cache holds evicted cache block.

→ On a subsequent cache miss, check the victim cache for the desired data before going to lower level memory.

9) Compiler optimizations:

→ code and data rearrangement:-

code can easily be rearranged without affecting correctness. code optimization aims for better efficiency from ~~language~~ long cache blocks. Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.

→ Loop interchange:

Simply exchanging the nesting of loops can make the code access the data in the order they are sorted. This technique reduces cache miss by improving spatial locality.

Ex: (Before)

```
for(j=0; j<100; j++)
    for(i=0; i<1000; i++)
        x[i][j] = 2+x[i][j]
```

(after)

```
for(i=0; i<1000; i++)
    for(j=0; j<100; j++)
        x[i][j] = 2+x[i][j].
```

10) Hardware prefetching of instructions:

→ Both instructions and data can be prefetched, either directly to cache or to external buffer. Typically processor fetches two blocks on a miss, the required block & the executive block. The required block is placed in the executive code che When it returns, prefetched block is placed in the instruction stream buffer, If the requested instruction is there, in the stream buffer & the next prefetch request is issued.

11) Compiler Controlled prefetching:

→ compiler inserts prefetching instructions to request data before the processor needs it.

→ Non-faulting: Prefetch doesnot cause exceptions.

→ 2 types:

i) Register prefetch:
loads data into register
Exc:- HP, PA-RISC loads.

ii) cache prefetch:
loads data into cache.
EX:- MIPS.

```
Exc:- for (i=0; i<N; i++)
      {
          prefetch(&a[i+P]);
          prefetch(&b[i+P]);
          sum+= a[i] * b[i];
      }
```