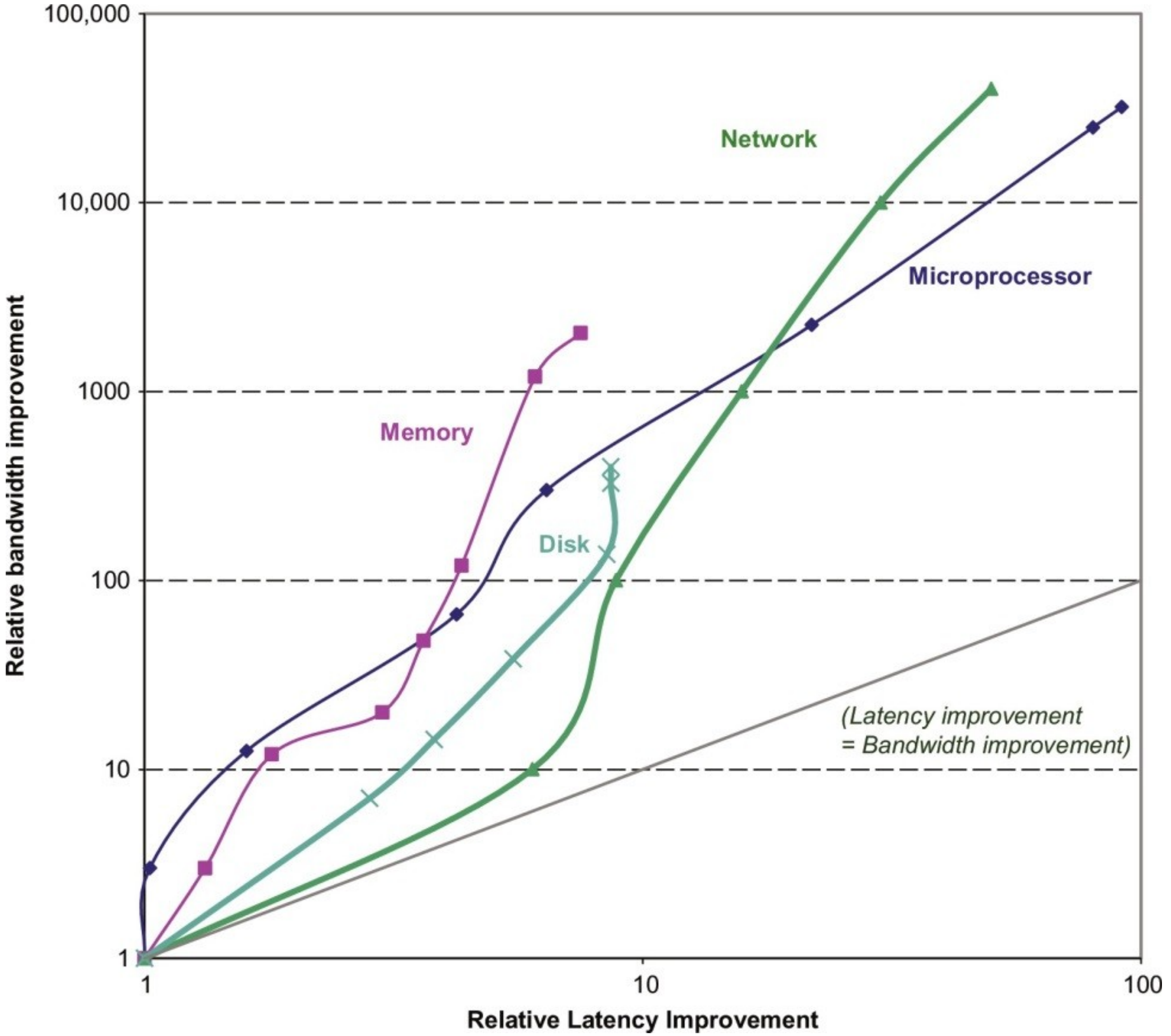
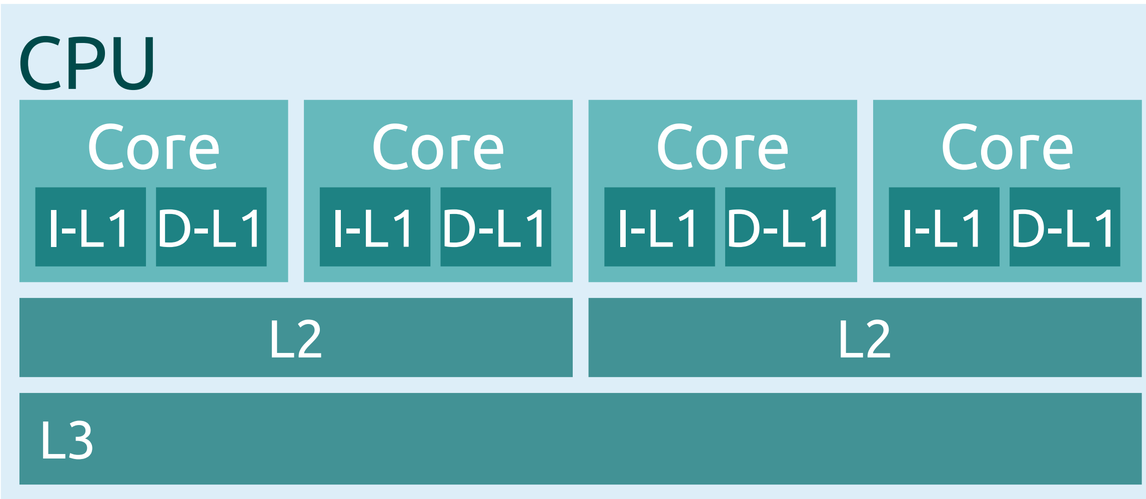
Why memory management matters

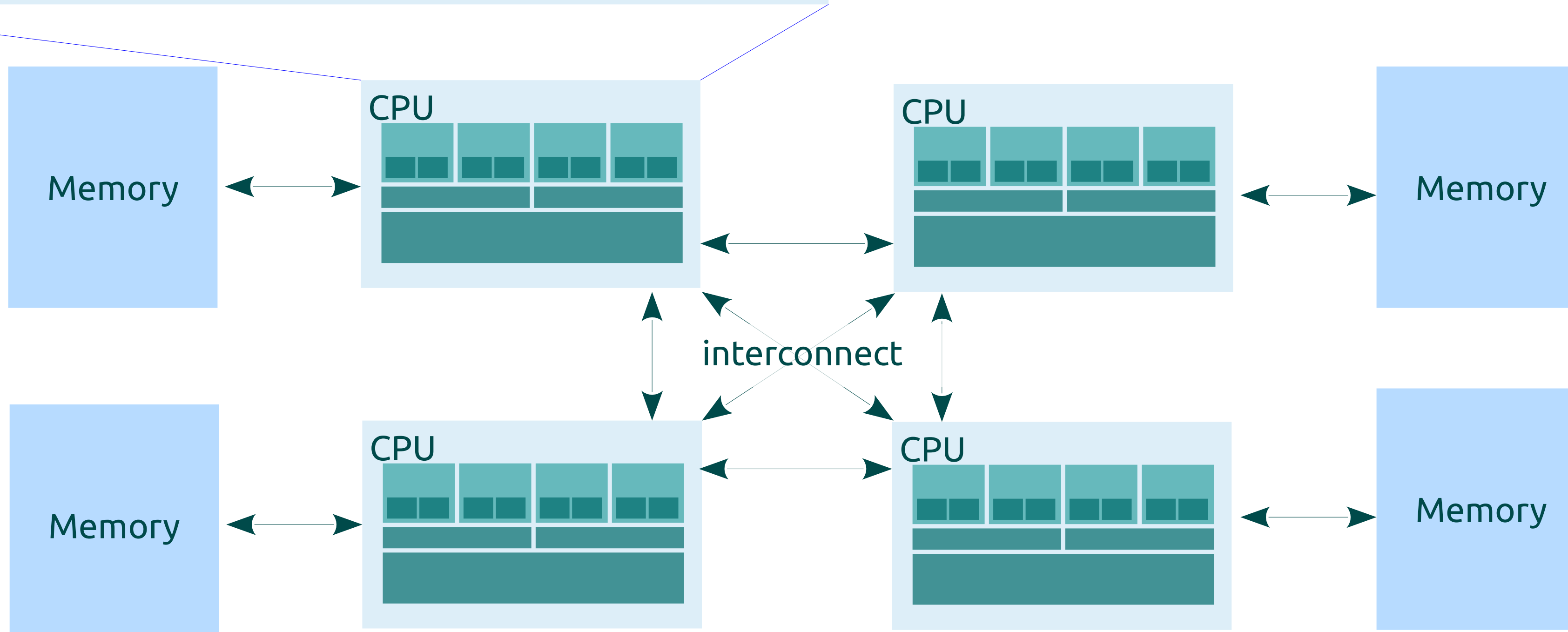


Hennessy, Patterson “Computer Architecture: A Quantitative Approach”

Introduction



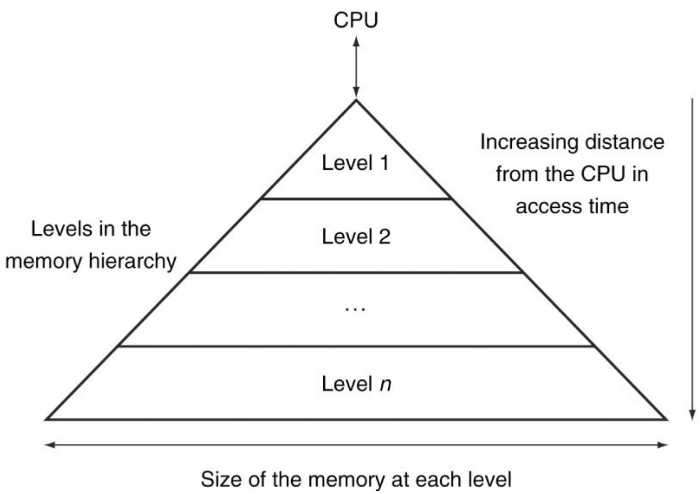
- Typical, simplified, CPU and system layout
  - Non Uniform Memory Access

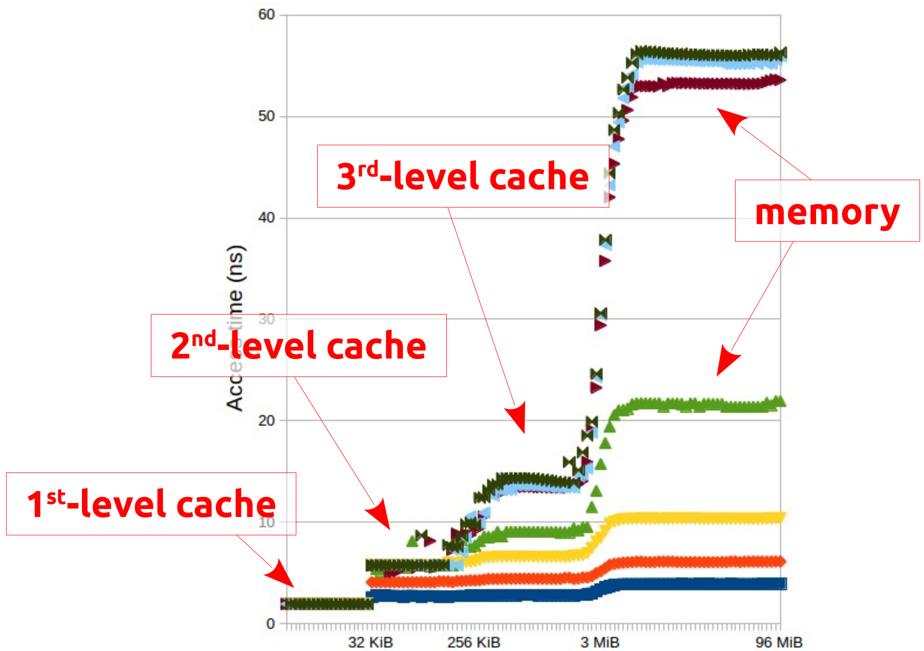


What's the ideal memory?

Memory technology	Typical access time	$ per GiB
SRAM	0.5 ns – 2.5 ns	500 – 1000
DRAM	50 ns – 70 ns	3 – 6
Magnetic disk	5 ms – 20 ms	0.01 – 0.02

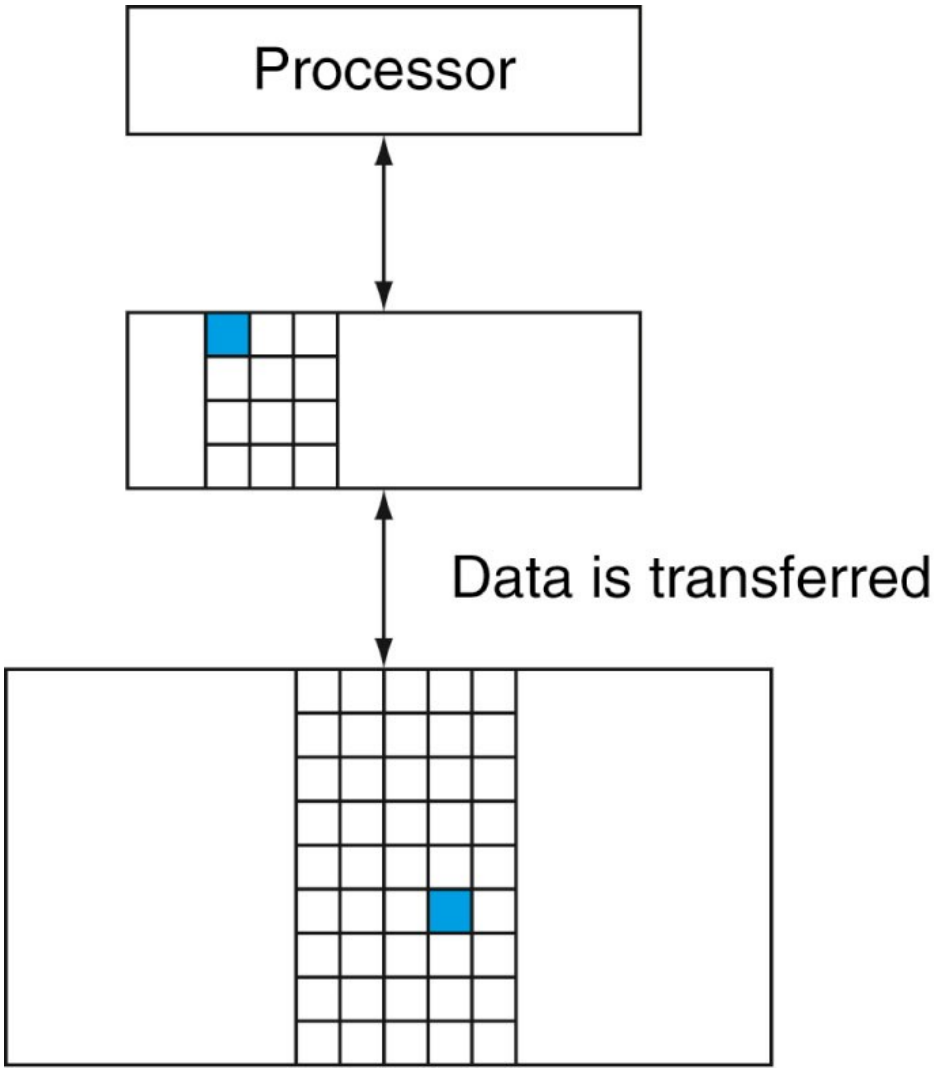
- Access time of SRAM, $/GiB and capacity of disk
- The ideal situation can be approximated with a hierarchy of different memory types





Patterson, Hennessy "Computer Organization and Design: The Hardware/Software Interface"

Hierarchy levels



Patterson, Hennessy "Computer Organization and Design: The Hardware/Software Interface"

- The data is **present** in the highest level
  - *hit*
    hit rate = hits / accesses
- The data is **not present** in the highest level
  - *miss*: data is looked for in the lower level
  - miss penalty: the cost of getting the data
  - likely causes stalls in the execution
- Data is moved in blocks (cache lines)

Locality principle

```
int strlen(char const* str)
{
  int len = 0;
  while (*str++) ++len;
  return len;
}
```

- Data
  - Multiple accesses to variable `len`
  - Scanning of array `str`
- Instructions
  - Repetition of the instructions corresponding to the expressions `*str++` e `++len`
  - Execution of consecutive instructions

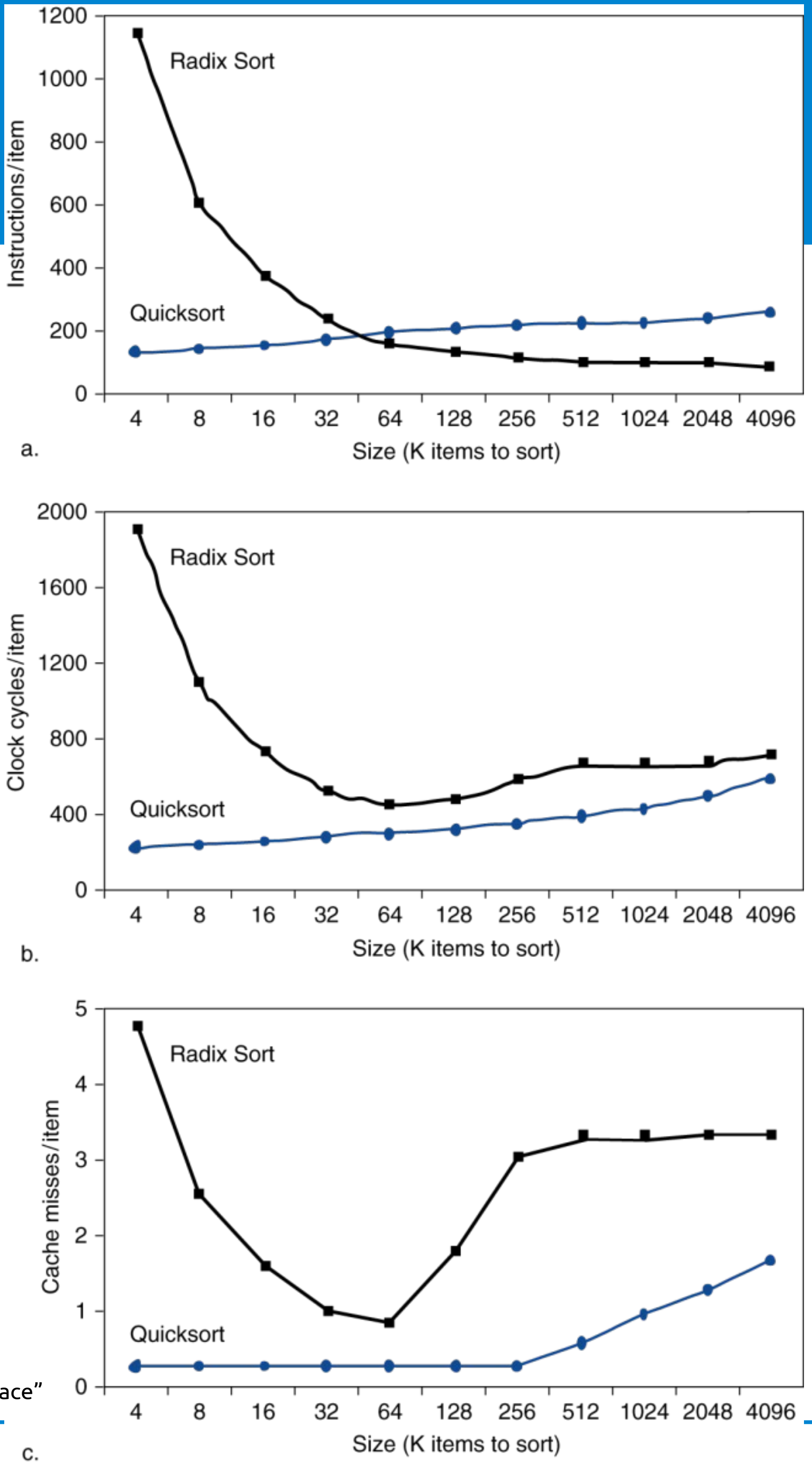
Locality principle

- In a limited time interval a program accesses only a small part of its whole address space
- Temporal locality
  - Memory locations recently accessed tend to be accessed again in the near future
    - e.g. instructions and counters in a loop
- Spatial locality
  - Memory locations near those recently accessed tend to be accessed in the near future
    - e.g. sequential access to instructions in a program or to data in an array
- Hardware components like caches and pipelines are justified by the locality principle

Cache effect

- The efficiency of a program does not depend only on the computational complexity of an algorithm…

Be friendly to the cache



Patterson, Hennessy "Computer Organization and Design: The Hardware/Software Interface"

Size of a type

- Determined statically (i.e. at compile time)
- Queried with the `sizeof` operator
  - returns multiples of `sizeof(char)`, which by definition is 1
  - typically a `char` is 1 byte, 8 bits
- For primitive types
  - on my laptop

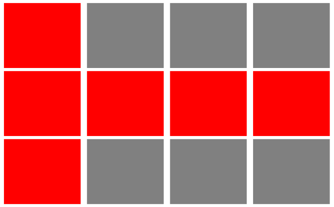
Type	`sizeof`
`bool`	1
`char`	1
`short`	2
`int`	4
`long`	8
`long long`	8
`float`	4
`double`	8
`long double`	16
`void*`	8

Layout

- Consider

```
struct S
{
  char c1;
  int  n;
  char c2;
};

static_assert(sizeof(S) == 12);
```



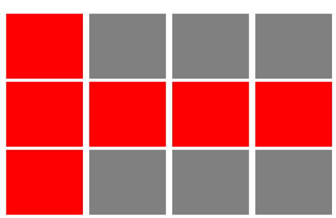
- The size is influenced by alignment constraints
  - the address of a variable of a certain type is typically a multiple of the size of that type
  - e.g. an `int` can reside only at an address multiple of 4

Does it matter?

- Try yourself, for example sorting a vector of structs with the same fields but different layouts

```
struct P
{
  char c1;
  int  n;
  char c2;
};

static_assert(sizeof(P) == 12);
```



```
struct P
{
  int  n;
  char c1;
  char c2;
};

static_assert(sizeof(P) == 8);
```

```
std::vector<P> v = …;
std::sort(v.begin(), v.end(), [](P const&, P const&) {…});
```

Cold data

- Consider

```
struct S
{
  int    n;
  float  f;
  double d;
};

static_assert(sizeof(S) == 16);
```

optimal layout

```
std::vector<S> v = …;
std::sort(v.begin(), v.end(), [](S const& l, S const& r) { return l.n < r.n; });
```

the order depends only on S::n

cache line (64 bytes)

- Data is brought into the cache, but it's not used
  - NB the “usefulness” depends on the specific operation

Does it matter?

- Try yourself, for example sorting a vector of structs with a field of changing size which is not used
  - EXTSIZE can be passed with -DEXTSIZE=nn to the compilation command

```
struct S
{
  int  n;
  char ext[EXTSIZE]
};

std::vector<S> v = …;
std::sort(v.begin(), v.end(), [](S const& l, S const& r) { return l.n < r.n; });
```

Alternative design techniques

- Externalize cold data from the data structure

```
using Ext = char[EXTSIZE];
struct Particle {
  Vec position_;
  Ext ext_;
  void translate(Vec const& t) {
    position_ += t;
  }
};
```

```
using Ext = char[EXTSIZE];
struct ParticleExt { Ext ext; };
struct Particle {
  Vec position_;
  std::unique_ptr<ParticleExt> ext_;
  void translate(Vec const& t) {
    position_ += t;
  }
};
```

```
using Particles = vector<Particle>;
void translate(Particles& ps, Vec const& t) {
  for_each(ps.begin(), ps.end(),
    [=](Particle& p) { p.translate(t); }
  );
}
```

no impact on client code

- Try yourself

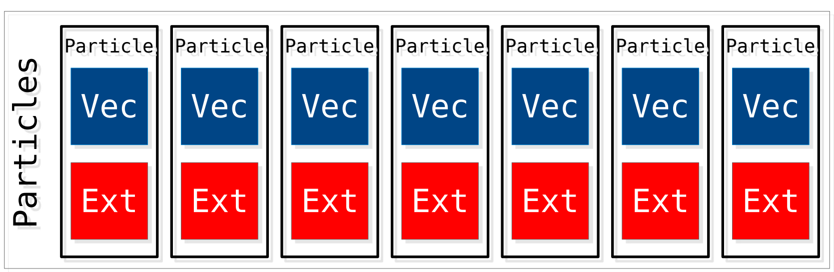
Alternative design techniques

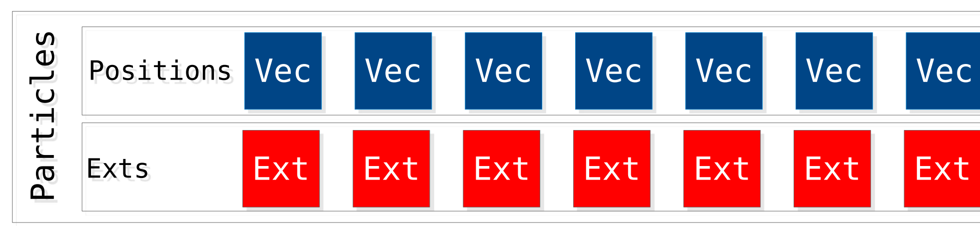
- Structure of Arrays instead of Array of Structures

```
struct Particle {
  Vec position;
  Ext ext;
  void translate(Vec const& t) {
    position += t;
  }
};

using Particles = std::vector<Particle>;
```

```
struct Particles {
  std::vector<Vec> positions;
  std::vector<Ext> exts;
};
void translate(Vec& position, Vec const& t) {
  position += t;
}
```





- The technique can be brought to the extreme, down to the primitive types

Alternative design techniques

- Structure of Arrays

```
struct Particle {
  Vec position;
  Ext ext;
  void translate(Vec const& t) {
    position += t;
  }
};

Particles v;
v[i].position;
```

```
struct Particles {
  vector<Vec> positions;
  vector<Ext> exts;
};
void translate(Vec& position, Vec const& t) {
  position += t;
}

Particles v;
v.positions[i];
```

some impact on client code

```
void translate(Particles& ps, Vec const& t) {
  std::for_each(ps.begin(), ps.end(),
    [&](Particle& part) { part.translate(t); }
  );
}
```

```
void translate(Particles& ps, Vec const& t) {
  auto& positions = ps.positions;
  std::for_each(positions.begin(), positions.end(),
    [&](Vec& pos) { translate(pos, t); }
  );
}
```

- Try yourself

Hands-on

- Inspect, build, run, measure, also through `perf`
  - `sort_packed.cpp`
  - `sort_cold.cpp`
  - `aos.cpp`
  - `aos_impr.cpp`
  - `soa.cpp`