# MEMORY OPTIMIZATION

**Christer Ericson**
**Sony Computer Entertainment, Santa Monica**
**(christer_ericson@playstation.sony.com)**

# Talk contents 1/2

- **Problem statement**
  - Why “memory optimization?”
- **Brief architecture overview**
  - The memory hierarchy
- **Optimizing for (code and) data cache**
  - General suggestions
  - Data structures
    - Prefetching and preloading
    - Structure layout
    - Tree structures
    - Linearization caching
- **…**

# Talk contents 2/2

- …
- **Aliasing**
  - Abstraction penalty problem
  - Alias analysis (type-based)
  - ‘restrict’ pointers
  - Tips for reducing aliasing

# Problem statement

- **For the last 20-something years…**
  - CPU speeds have increased ~60%/year
  - Memory speeds only decreased ~10%/year
- **Gap covered by use of cache memory**
- **Cache is under-exploited**
  - Diminishing returns for larger caches
- **Inefficient cache use = lower performance**
  - How increase cache utilization? Cache-awareness!

# Need more justification? 1/3

Instruction parallelism:

**SIMD instructions consume data at 2-8 times the rate of normal instructions!**

# Need more justification? 2/3

**Proebsting's law:**

**Improvements to compiler technology double program performance every <u>~18 years</u>!**

Corollary: Don't expect the compiler to do it for you!

# Need more justification? 3/3

## On Moore's law:

- Consoles don't follow it (as such)
  - Fixed hardware
  - 2nd/3rd generation titles must get improvements from somewhere

# Brief cache review

- **Caches**
  - Code cache for instructions, data cache for data
  - Forms a memory hierarchy
- **Cache lines**
  - Cache divided into cache lines of ~32/64 bytes each
  - Correct unit in which to count memory accesses
- **Direct-mapped**
  - For n KB cache, bytes at k, k+n, k+2n, … map to same cache line
- **N-way set-associative**
  - Logical cache line corresponds to N physical lines
  - Helps minimize cache line thrashing

# The memory hierarchy

| | Roughly: |
|---|---|
| **CPU** | 1 cycle |
| **L1 cache** | ~1-5 cycles |
| **L2 cache** | ~5-20 cycles |
| **Main memory** | ~40-100 cycles |

# Some cache specs

| | L1 cache (I/D) | L2 cache |
|---|---|---|
| PS2 | 16K/8K† 2-way | N/A |
| GameCube | 32K/32K‡ 8-way | 256K 2-way unified |
| XBOX | 16K/16K 4-way | 128K 8-way unified |
| PC | ~32-64K | ~128-512K |

- †16K data scratchpad important part of design
- ‡configurable as 16K 4-way + 16K scratchpad

# Foes: 3 C's of cache misses

- **Compulsory misses**
  - Unavoidable misses when data read for first time
- **Capacity misses**
  - Not enough cache space to hold all active data
  - Too much data accessed inbetween successive use
- **Conflict misses**
  - Cache thrashing due to data mapping to same cache lines

# Friends: Introducing the 3 R's

- **Rearrange (code, data)**
  - Change layout to increase spatial locality
- **Reduce (size, # cache lines read)**
  - Smaller/smarter formats, compression
- **Reuse (cache lines)**
  - Increase temporal (and spatial) locality

| | Compulsory | Capacity | Conflict |
|---|---|---|---|
| Rearrange | **X** | **(x)** | **X** |
| Reduce | **X** | **X** | **(x)** |
| Reuse | **(x)** | **X** | |

# Measuring cache utilization

- **Profile**
  - CPU performance/event counters
    - Give memory access statistics
    - But not access patterns (e.g. stride)
  - Commercial products
    - SN Systems’ Tuner, Metrowerks’ CATS, Intel’s VTune
  - Roll your own
    - In gcc ‘-p’ option + define _**mcount()**
    - Instrument code with calls to logging class
  - Do back-of-the-envelope comparison
- **Study the generated code**

# Code cache optimization 1/2

- **Locality**
  - Reorder functions
    - Manually within file
    - Reorder object files during linking (order in makefile)
    - **__attribute__ ((section ("xxx")))** in gcc
  - Adapt coding style
    - Monolithic functions
    - Encapsulation/OOP is less code cache friendly
  - Moving target
  - Beware various implicit functions (e.g. fptodp)

# Code cache optimization 2/2

- **Size**
  - Beware: inlining, unrolling, large macros
  - KISS
    - Avoid featuritis
    - Provide multiple copies (also helps locality)
  - Loop splitting and loop fusion
  - Compile for size (‘-Os’ in gcc)
  - Rewrite in asm (where it counts)
- **Again, study generated code**
  - Build intuition about code generated

# Data cache optimization

- **Lots and lots of stuff…**
  - "Compressing" data
  - Blocking and strip mining
  - Padding data to align to cache lines
  - Plus other things I won't go into
- **What I will talk about…**
  - Prefetching and preloading data into cache
  - Cache-conscious structure layout
  - Tree data structures
  - Linearization caching
  - Memory allocation
  - Aliasing and "anti-aliasing"

# Prefetching and preloading

- **Software prefetching**
  - Not too early – data may be evicted before use
  - Not too late – data not fetched in time for use
  - Greedy
- **Preloading (pseudo-prefetching)**
  - Hit-under-miss processing

# Software prefetching

```
// Loop through and process all 4n elements
for (int i = 0; i < 4 * n; i++)
    Process(elem[i]);
```

```
const int kLookAhead = 4; // Some elements ahead
for (int i = 0; i < 4 * n; i += 4) {
    Prefetch(elem[i + kLookAhead]);
    Process(elem[i + 0]);
    Process(elem[i + 1]);
    Process(elem[i + 2]);
    Process(elem[i + 3]);
}
```

# Greedy prefetching

```
void PreorderTraversal(Node *pNode) {
    // Greedily prefetch left traversal path
    Prefetch(pNode->left);
    // Process the current node
    Process(pNode);
    // Greedily prefetch right traversal path
    Prefetch(pNode->right);
    // Recursively visit left then right subtree
    PreorderTraversal(pNode->left);
    PreorderTraversal(pNode->right);
}
```

# Preloading (pseudo-prefetch)

```
Elem a = elem[0];
for (int i = 0; i < 4 * n; i += 4) {
    Elem e = elem[i + 4]; // Cache miss, non-blocking
    Elem b = elem[i + 1]; // Cache hit
    Elem c = elem[i + 2]; // Cache hit
    Elem d = elem[i + 3]; // Cache hit
    Process(a);
    Process(b);
    Process(c);
    Process(d);
    a = e;
}
```

(NB: This code reads one element beyond the end of the elem array.)

# Structures

- **Cache-conscious layout**
  - Field reordering (usually grouped conceptually)
  - Hot/cold splitting
- **Let use decide format**
  - Array of structures
  - Structures of arrays
- **Little compiler support**
  - Easier for non-pointer languages (Java)
  - C/C++: do it yourself

# Field reordering

```
struct S {
    void *key;
    int count[20];
    S *pNext;
};
```

```
struct S {
    void *key;
    S *pNext;
    int count[20];
};
```

```
void Foo(S *p, void *key, int k) {
    while (p) {
        if (p->key == key) {
            p->count[k]++;
            break;
        }
        p = p->pNext;
    }
}
```

- Likely accessed together so store them together!

# Hot/cold splitting

Hot fields:

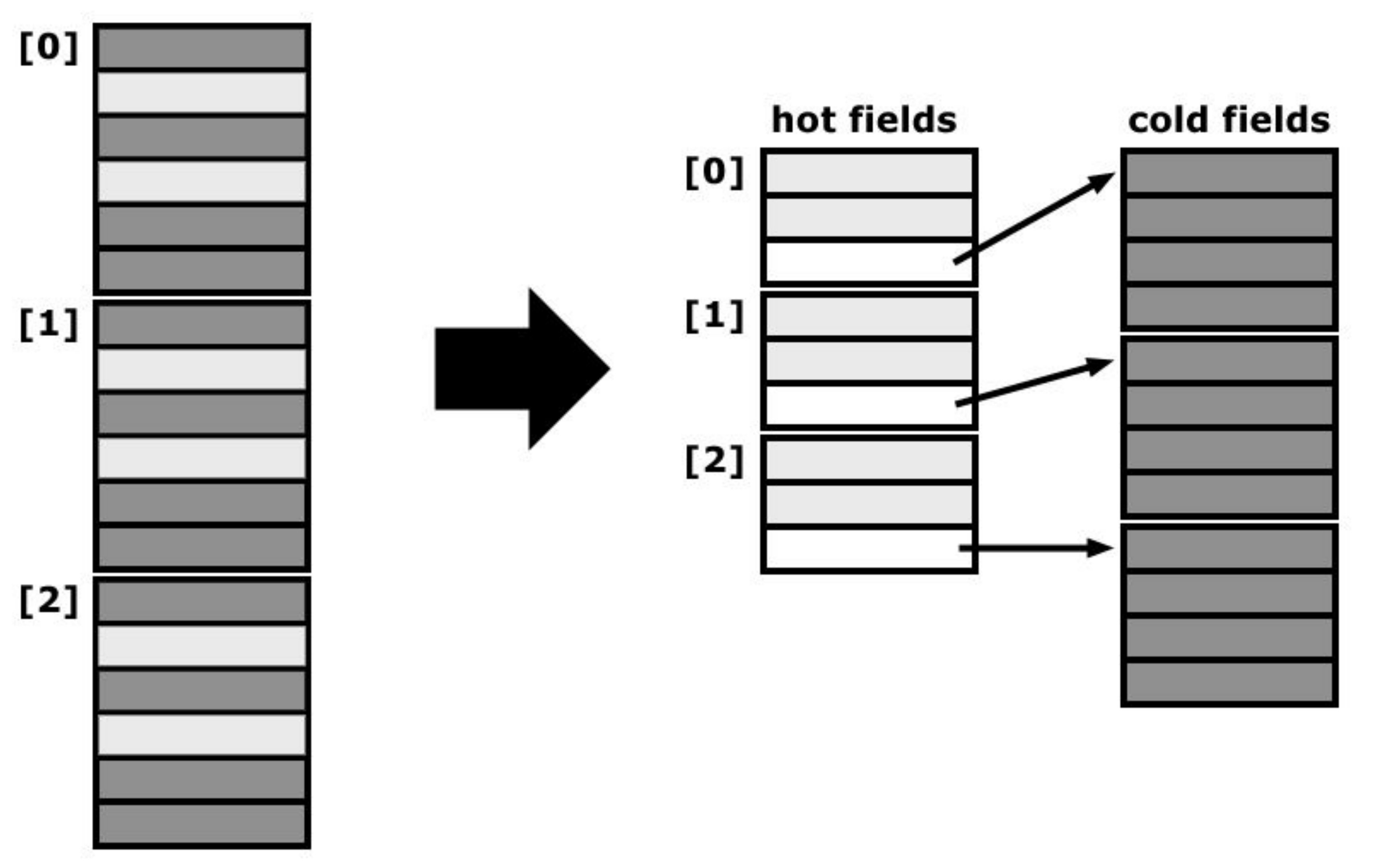
```
struct S {
    void *key;
    S *pNext;
    S2 *pCold;
};
```

Cold fields:

```
struct S2 {
    int count[10];
};
```

- Allocate all 'struct S' from a memory pool
  - Increases coherence
- Prefer array-style allocation
  - No need for actual pointer to cold fields

# Hot/cold splitting



# Beware compiler padding

```
struct X {
    int8 a;
    int64 b;
    int8 c;
    int16 d;
    int64 e;
    float f;
};
```

```
struct Y {
    int8 a, pad_a[7];
    int64 b;
    int8 c, pad_c[1];
    int16 d, pad_d[2];
    int64 e;
    float f, pad_f[1];
};
```

```
struct Z {
    int64 b;
    int64 e;
    float f;
    int16 d;
    int8 a;
    int8 c;
};
```

Decreasing size!

Assuming 4-byte floats, for most compilers sizeof(X) == 40, sizeof(Y) == 40, and sizeof(Z) == 24.

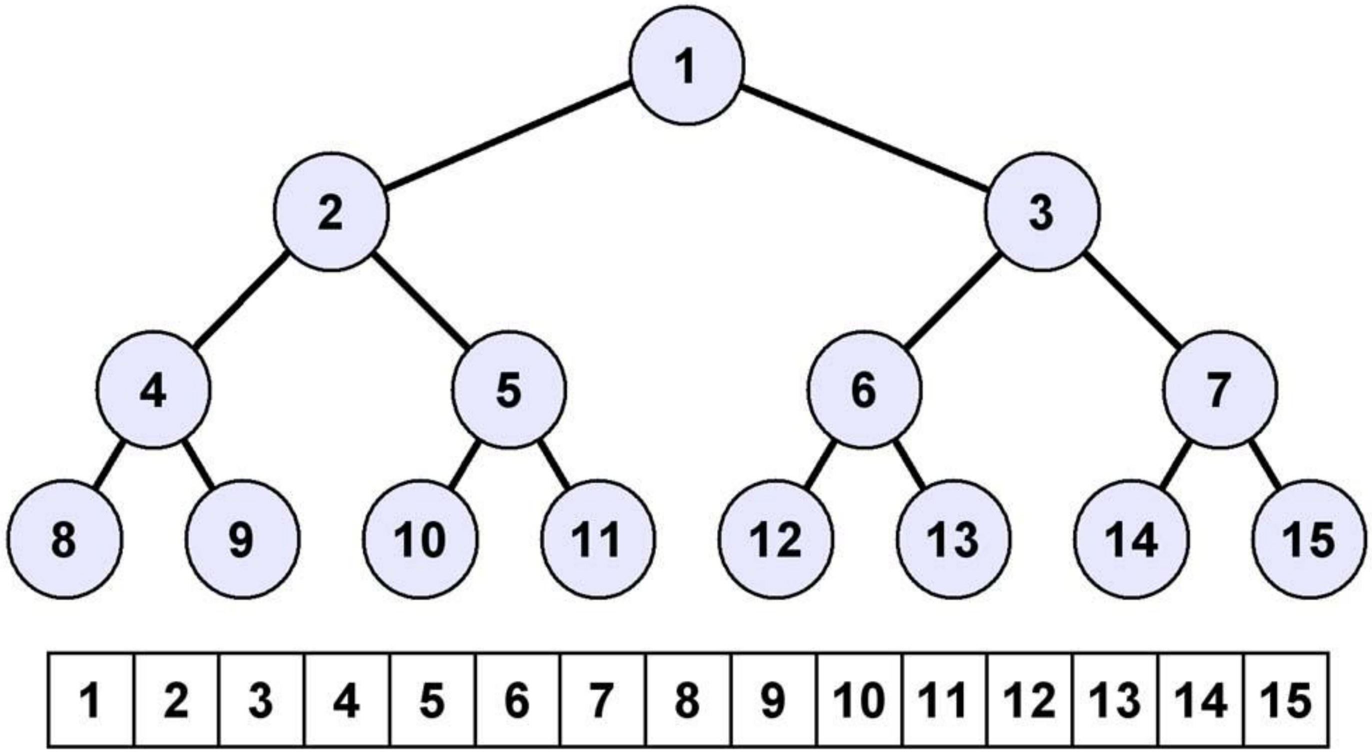
# Cache performance analysis

- **Usage patterns**
  - Activity – indicates hot or cold field
  - Correlation – basis for field reordering
- **Logging tool**
  - Access all class members through accessor functions
  - Manually instrument functions to call Log() function
  - Log() function…
    - takes object type + member field as arguments
    - hash-maps current args to count field accesses
    - hash-maps current + previous args to track pairwise accesses

# Tree data structures

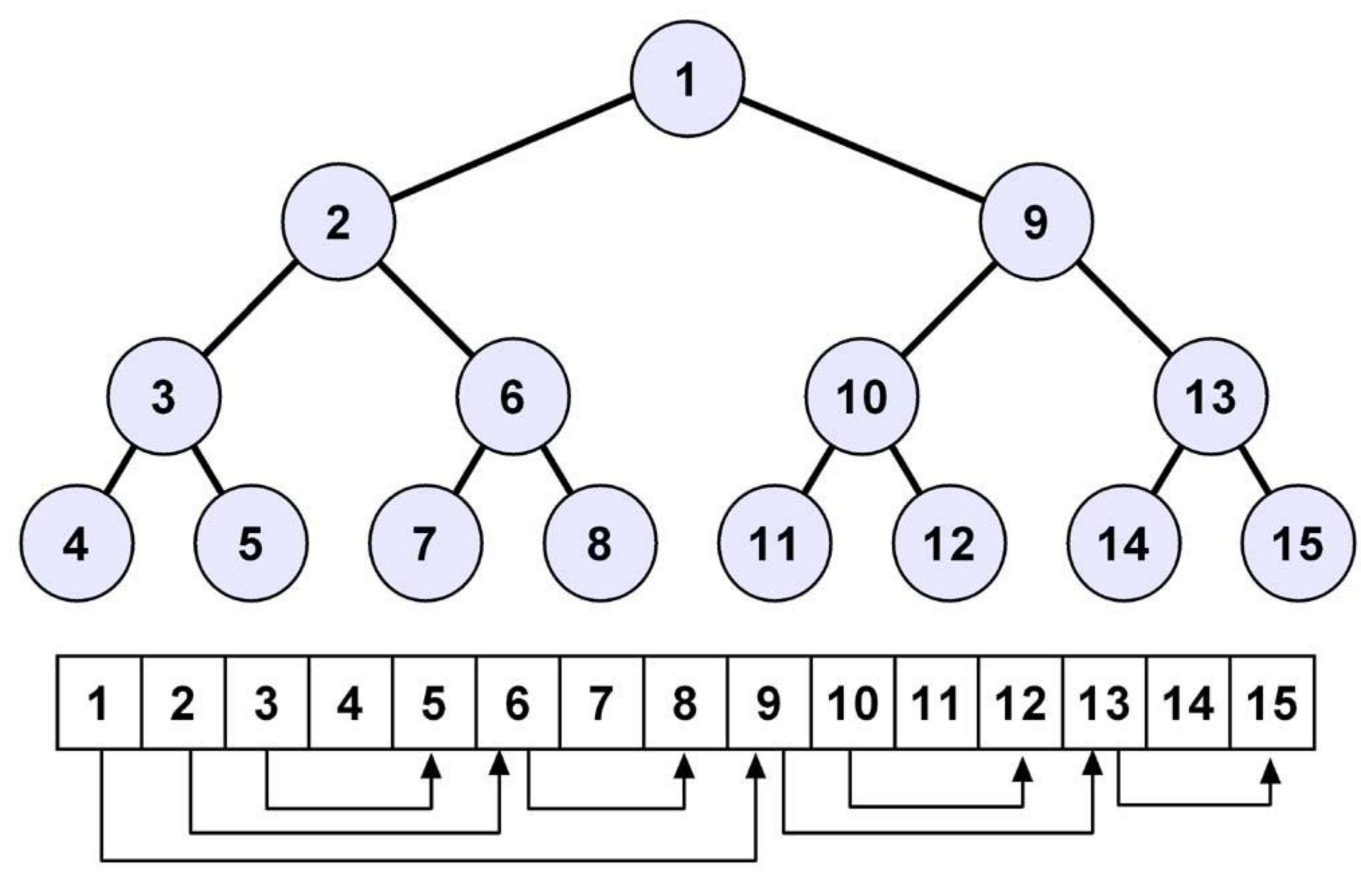
- **<u>Rearrange</u> nodes**
  - Increase spatial locality
  - Cache-aware vs. cache-oblivious layouts
- **<u>Reduce</u> size**
  - Pointer elimination (using implicit pointers)
  - "Compression"
    - Quantize values
    - Store data relative to parent node

# Breadth-first order



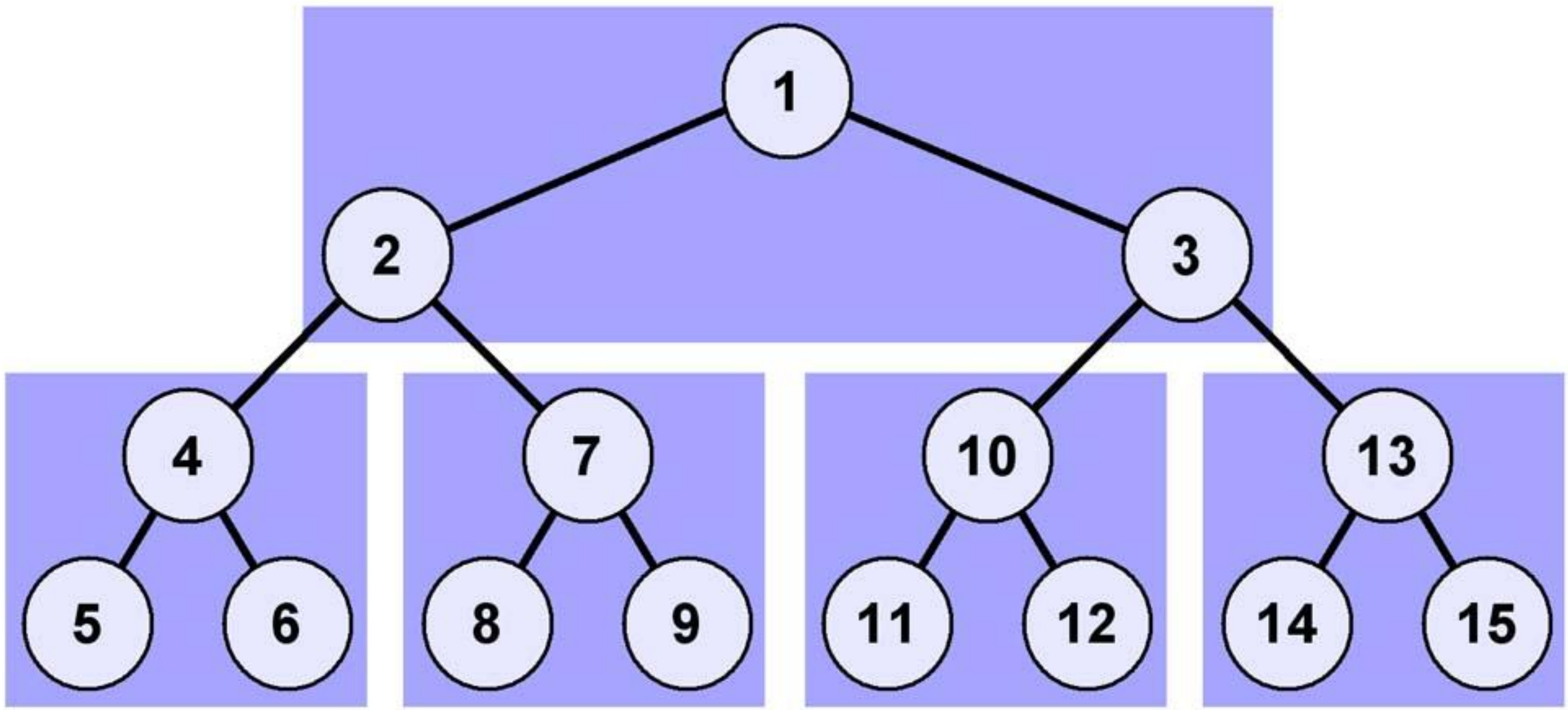

- Pointer-less: Left(n)=2n, Right(n)=2n+1
- Requires storage for complete tree of height H

# Depth-first order




- Left(n) = n + 1, Right(n) = stored index
- Only stores existing nodes

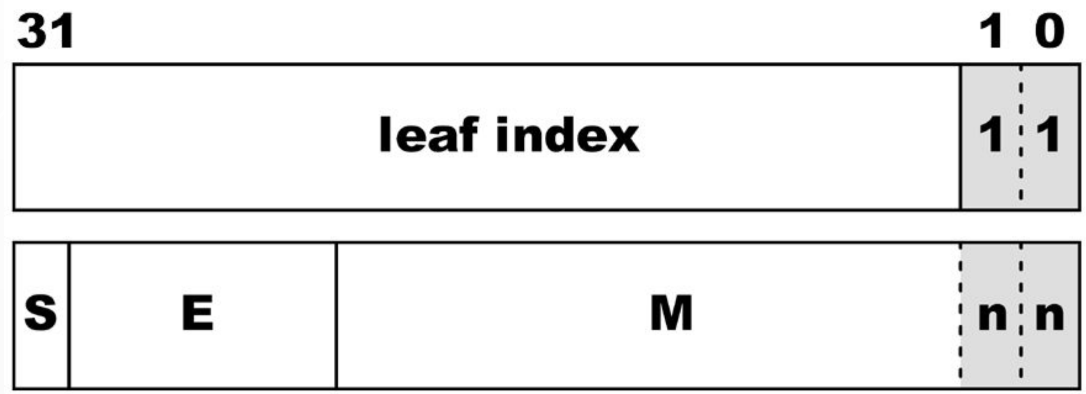
# van Emde Boas layout

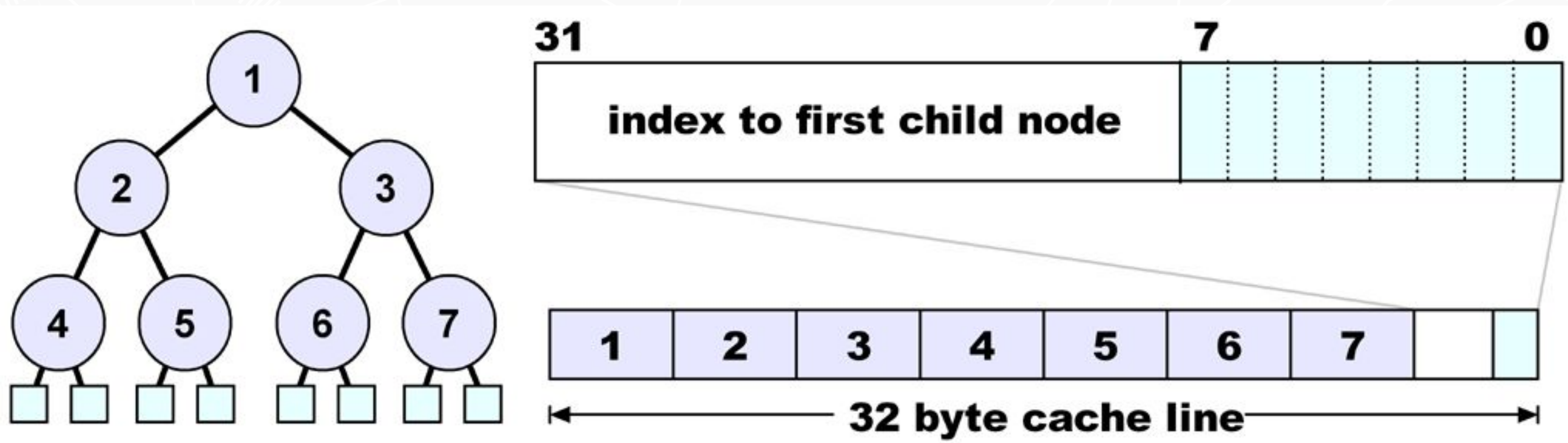



- "Cache-oblivious"
- Recursive construction

# A compact static k-d tree

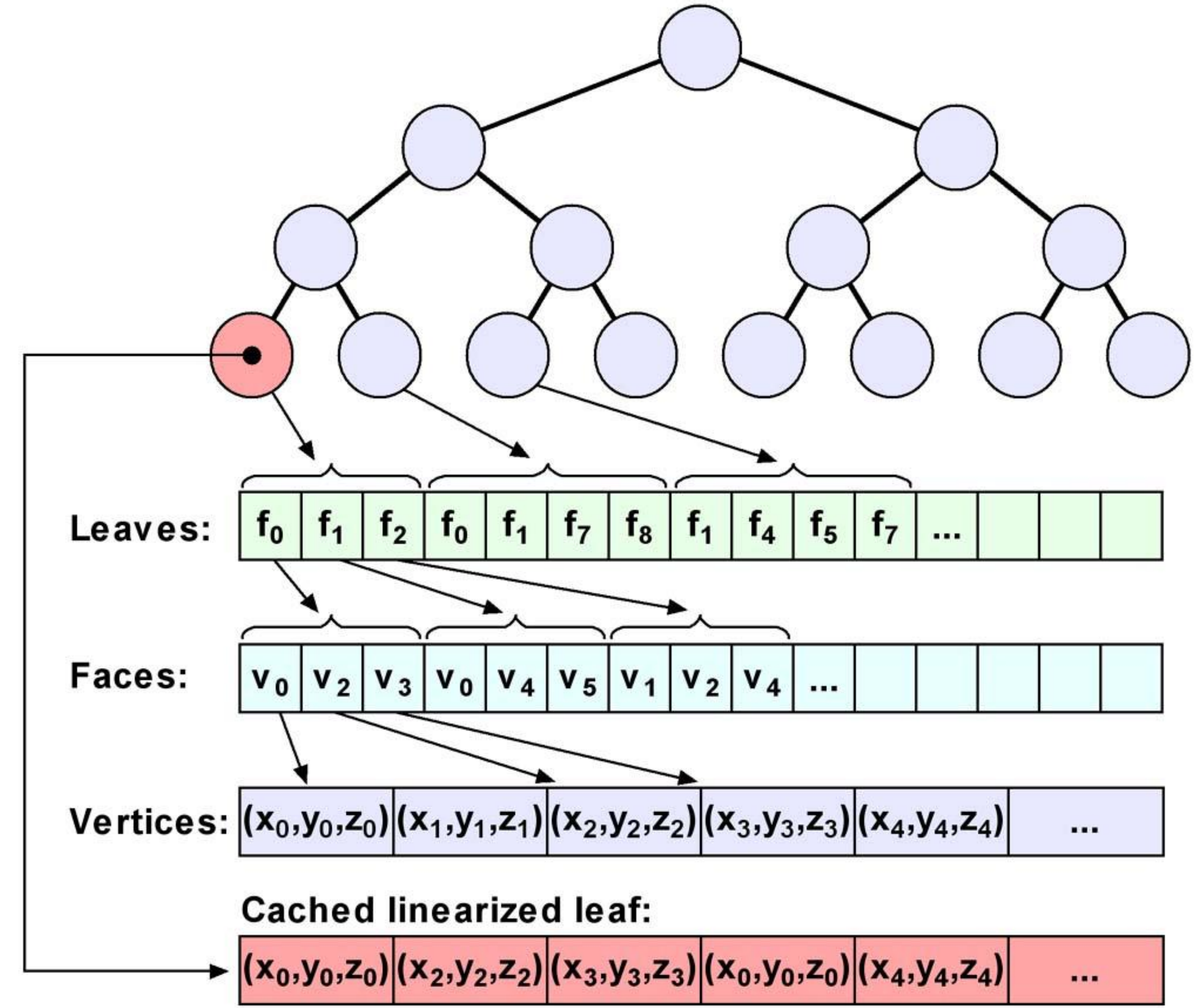
```
union KDNode {
    // leaf, type 11
    int32 leafIndex_type;
    // non-leaf, type 00 = x,
    // 01 = y, 10 = z-split
    float splitVal_type;
};
```



# Linearization caching

- **Nothing better than linear data**
  - Best possible spatial locality
  - Easily prefetchable
- **So linearize data at runtime!**
  - Fetch data, store linearized in a custom cache
  - Use it to linearize…
    - hierarchy traversals
    - indexed data
    - other random-access stuff



Leaves:
f0 f1 f2 f0 f1 f7 f8 f1 f4 f5 f7 ...
Faces:
v0 v2 v3 v0 v4 v5 v1 v2 v4 ...
Vertices:
(x0,y0,z0) (x1,y1,z1) (x2,y2,z2) (x3,y3,z3) (x4,y4,z4) ...
Cached linearized leaf:
(x0,y0,z0) (x2,y2,z2) (x3,y3,z3) (x0,y0,z0) (x4,y4,z4) ...

# Memory allocation policy

- **Don't allocate from heap, use pools**
  - No block overhead
  - Keeps data together
  - Faster too, and no fragmentation
- **Free ASAP, reuse immediately**
  - Block is likely in cache so reuse its cachelines
  - First fit, using free list

# The curse of aliasing

What is aliasing?

```
int n;
int *p1 = &n;
int *p2 = &n;
```

Aliasing is multiple references to the same storage location

Aliasing is also missed opportunities for optimization

```
int Foo(int *a, int *b) {
    *a = 1;
    *b = 2;
    return *a;
}
```

What value is returned here? Who knows!

# The curse of aliasing

- What is causing aliasing?
  - Pointers
  - Global variables/class members make it worse
- What is the problem with aliasing?
  - Hinders reordering/elimination of loads/stores
    - Poisoning data cache
    - Negatively affects instruction scheduling
    - Hinders common subexpression elimination (CSE), loop-invariant code motion, constant/copy propagation, etc.

# How do we do ‘anti-aliasing’?

- What can be done about aliasing?
  - **Better languages**
    - Less aliasing, lower abstraction penalty†
  - **Better compilers**
    - Alias analysis such as type-based alias analysis†
  - **Better programmers (aiding the compiler)**
    - That’s you, after the next 20 slides!
  - **Leap of faith**
    - -fno-aliasing

† To be defined

# Matrix multiplication 1/3

Consider optimizing a 2x2 matrix multiplication:

```
Mat22mul(float a[2][2], float b[2][2], float c[2][2]){
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            a[i][j] = 0.0f;
            for (int k = 0; k < 2; k++)
                a[i][j] += b[i][k] * c[k][j];
        }
    }
}
```

How do we typically optimize it? Right, unrolling!

# Matrix multiplication 2/3

Staightforward unrolling results in this:

```
// 16 memory reads, 4 writes
Mat22mul(float a[2][2], float b[2][2], float c[2][2]){
    a[0][0] = b[0][0]*c[0][0] + b[0][1]*c[1][0];
    a[0][1] = b[0][0]*c[0][1] + b[0][1]*c[1][1]; //(1)
    a[1][0] = b[1][0]*c[0][0] + b[1][1]*c[1][0]; //(2)
    a[1][1] = b[1][0]*c[0][1] + b[1][1]*c[1][1]; //(3)
}
```

- But wait! There's a hidden assumption! a is not b or c!
- Compiler doesn't (cannot) know this!
  - (1) Must refetch b[0][0] and b[0][1]
  - (2) Must refetch c[0][0] and c[1][0]
  - (3) Must refetch b[0][0], b[0][1], c[0][0] and c[1][0]

# Matrix multiplication 3/3

A correct approach is instead writing it as:

```
// 8 memory reads, 4 writes
Mat22mul(float a[2][2], float b[2][2], float c[2][2]){
    float b00 = b[0][0], b01 = b[0][1];
    float b10 = b[1][0], b11 = b[1][1];
    float c00 = c[0][0], c01 = c[0][1];
    float c10 = c[1][0], c11 = c[1][1];

    a[0][0] = b00*c00 + b01*c10;
    a[0][1] = b00*c01 + b01*c11;
    a[1][0] = b10*c00 + b11*c10;
    a[1][1] = b10*c01 + b11*c11;
}
```

Consume inputs…

…before producing outputs

# Abstraction penalty problem

- Higher levels of abstraction have a negative effect on optimization
  - Code broken into smaller generic subunits
  - Data and operation hiding
    - Cannot make local copy of e.g. internal pointers
    - Cannot hoist constant expressions out of loops

- Especially because of aliasing issues

# C++ abstraction penalty

- Lots of (temporary) objects around
  - Iterators
  - Matrix/vector classes
- Objects live in heap/stack
  - Thus subject to aliasing
  - Makes tracking of current member value very difficult
  - But tracking required to keep values in registers!
- Implicit aliasing through the **this** pointer
  - Class members are virtually as bad as global variables

# C++ abstraction penalty

Pointer members in classes may alias other members:

```
class Buf {
public:
    void Clear() {
        for (int i = 0; i < numVals; i++)
            pBuf[i] = 0;
    }
private:
    int numVals, *pBuf;
}
```

**numVals** not a local variable!

May be aliased by **pBuf**

**Code likely to refetch numVals each iteration!**

# C++ abstraction penalty

We know that aliasing won't happen, and can manually solve the aliasing issue by writing code as:

```
class Buf {
public:
    void Clear() {
        for (int i = 0, n = numVals; i < n; i++)
            pBuf[i] = 0;
    }
private:
    int numVals, *pBuf;
}
```

# C++ abstraction penalty

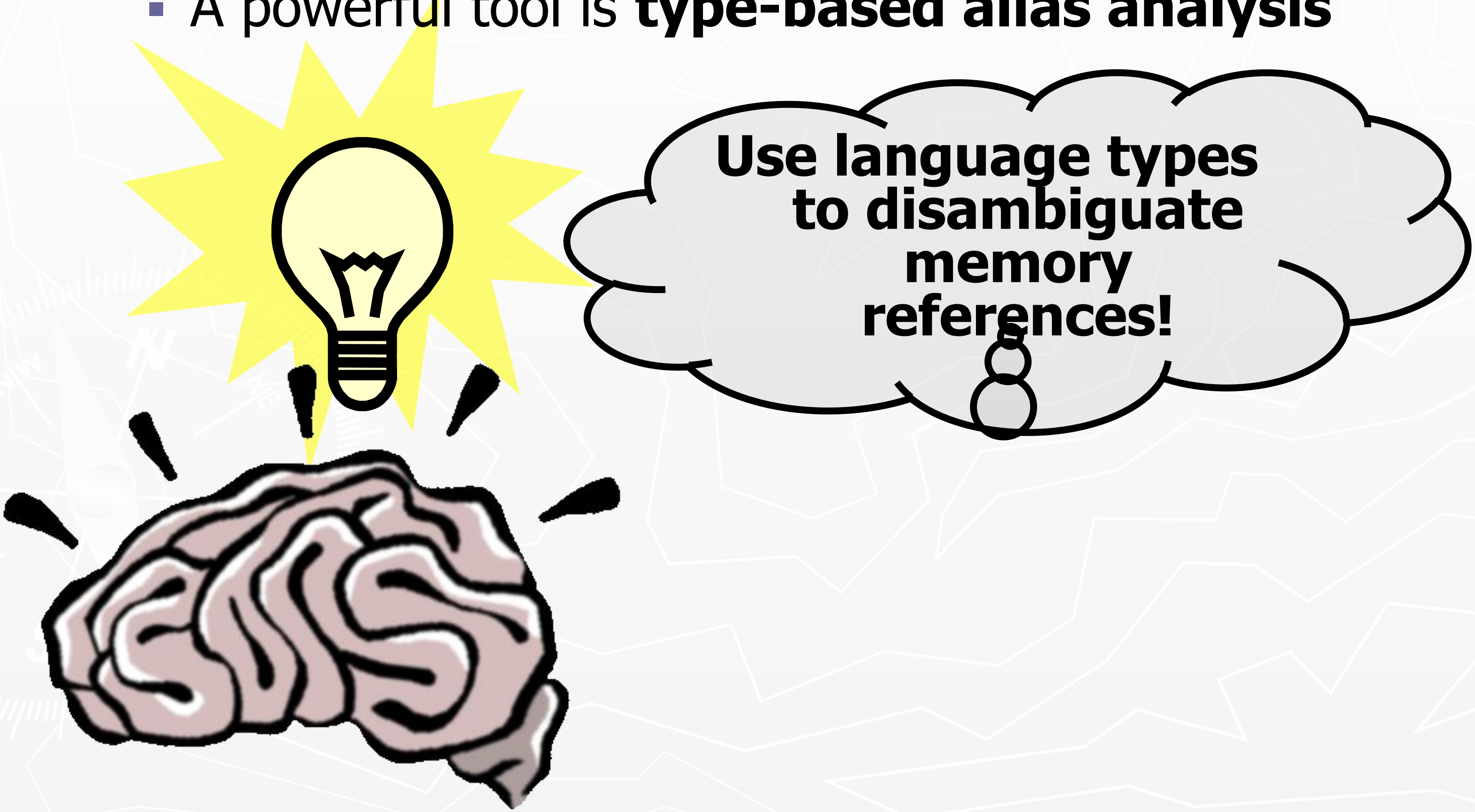
Since **pBuf[i]** can only alias **numVals** in the first iteration, a quality compiler can fix this problem by peeling the loop once, turning it into:

```
void Clear() {
    if (numVals >= 1) {
        pBuf[0] = 0;
        for (int i = 1, n = numVals; i < n; i++)
            pBuf[i] = 0;
    }
}
```

**Q: Does <u>your</u> compiler do this optimization?!**

# Type-based alias analysis

- Some aliasing the compiler can catch
  - A powerful tool is **type-based alias analysis**



# Type-based alias analysis

- ANSI C/C++ states that…
  - Each area of memory can only be associated with one type during its lifetime
  - Aliasing may only occur between references of the same <u>compatible</u> type
- Enables compiler to rule out aliasing between references of non-compatible type
  - Turned on with –fstrict-aliasing in gcc

# Compatibility of C/C++ types

- In short…
  - Types compatible if differing by **signed**, **unsigned**, **const** or **volatile**
  - **char** and **unsigned char** compatible with any type
  - Otherwise not compatible
- (See standard for full details.)

# What TBAA can do for you

It can turn this:

```
void Foo(float *v, int *n) {
    for (int i = 0; i < *n; i++)
        v[i] += 1.0f;
}
```

Possible aliasing between **v[i]** and ***n**

into this:

```
void Foo(float *v, int *n) {
    int t = *n;
    for (int i = 0; i < t; i++)
        v[i] += 1.0f;
}
```

No aliasing possible so fetch ***n** once!

# What TBAA can also do

- Cause obscure bugs in non-conforming code!
  - Beware especially so-called “type punning”

```
uint32 i;
float f;
i = *((uint32 *)&f);
```

Illegal C/C++ code!

```
uint32 i;
union {
    float f;
    uint32 i;
} u;
u.f = f;
i = u.i;
```

Allowed By gcc

```
uint32 i;
union {
    float f;
    uchar8 c[4];
} u;
u.f = f;
i = (u.c[3]<<24L)+
    (u.c[2]<<16L)+
     ...;
```

Required by standard

# Restrict-qualified pointers

- **restrict** keyword
  - New to 1999 ANSI/ISO C standard
  - Not in C++ standard yet, but supported by many C++ compilers
  - A hint only, so may do nothing and still be conforming
- A restrict-qualified pointer (or reference)…
  - …is basically a promise to the compiler that for the scope of the pointer, the target of the pointer will only be accessed through that pointer (and pointers copied from it).
  - (See standard for full details.)

# Using the restrict keyword

Given this code:

```
void Foo(float v[], float *c, int n) {
    for (int i = 0; i < n; i++)
        v[i] = *c + 1.0f;
}
```

You really want the compiler to treat it as if written:

```
void Foo(float v[], float *c, int n) {
    float tmp = *c + 1.0f;
    for (int i = 0; i < n; i++)
        v[i] = tmp;
}
```

**But because of possible aliasing it cannot!**

# Using the restrict keyword

For example, the code might be called as:

```
float a[10];
a[4] = 0.0f;
Foo(a, &a[4], 10);
```

giving for the first version:

```
v[] = 1, 1, 1, 1, 1, 2, 2, 2, 2, 2
```

and for the second version:

```
v[] = 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

**The compiler must be conservative, and cannot perform the optimization!**

# Solving the aliasing problem

The fix? Declaring the output as **restrict**:

```
void Foo(float * restrict v, float *c, int n) {
    for (int i = 0; i < n; i++)
        v[i] = *c + 1.0f;
}
```

- Alas, in practice may need to declare **both** pointers restrict!
  - A restrict-qualified pointer can grant access to non-restrict pointer
  - Full data-flow analysis required to detect this
  - However, two restrict-qualified pointers are trivially non-aliasing!
  - Also may work declaring second argument as "float * **const** c"

# 'const' doesn't help

Some might think this would work:

```
void Foo(float v[], const float *c, int n) {
    for (int i = 0; i < n; i++)
        v[i] = *c + 1.0f;
}
```

Since ***c** is const, **v[i]** cannot write to it, right?

- Wrong! **const** promises almost nothing!
  - Says ***c** is const through **c**, **<u>not</u>** that ***c** is const in general
  - Can be cast away
  - For detecting programming errors, not fixing aliasing

# SIMD + restrict = TRUE

- **restrict** enables SIMD optimizations

```
void VecAdd(int *a, int *b, int *c) {
    for (int i = 0; i < 4; i++)
        a[i] = b[i] + c[i];
}
```

Stores may alias loads. Must perform operations sequentially.

```
void VecAdd(int * restrict a, int *b, int *c) {
    for (int i = 0; i < 4; i++)
        a[i] = b[i] + c[i];
}
```

Independent loads and stores. Operations can be performed in parallel!

# Restrict-qualified pointers

- **Important, especially with C++**
  - Helps combat abstraction penalty problem
- **But beware…**
  - Tricky semantics, easy to get wrong
  - Compiler won't tell you about incorrect use
  - Incorrect use = slow painful death!

# Tips for avoiding aliasing

- Minimize use of globals, pointers, references
  - Pass small variables by-value
  - Inline small functions taking pointer or reference arguments
- Use local variables as much as possible
  - Make local copies of global and class member variables
- Don't take the address of variables (with &)
- **restrict** pointers and references
- Declare variables close to point of use
- Declare side-effect free functions as **const**
- Do manual CSE, especially of pointer expressions

# That's it! – Resources 1/2


- Ericson, Christer. Real-time collision detection. Morgan-Kaufmann, 2005. (Chapter on memory optimization)
- Mitchell, Mark. Type-based alias analysis. Dr. Dobb's journal, October 2000.
- Robison, Arch. Restricted pointers are coming. C/C++ Users Journal, July 1999. http://www.cuj.com/articles/1999/9907/9907d/9907d.htm
- Chilimbi, Trishul. Cache-conscious data structures - design and implementation. PhD Thesis. University of Wisconsin, Madison, 1999.
- Prokop, Harald. Cache-oblivious algorithms. Master's Thesis. MIT, June, 1999.
- …

# Resources 2/2

- …

- Gavin, Andrew. Stephen White. Teaching an old dog new bits: How console developers are able to improve performance when the hardware hasn't changed. Gamasutra. November 12, 1999 http://www.gamasutra.com/features/19991112/GavinWhite_01.htm
- Handy, Jim. The cache memory book. Academic Press, 1998.
- Macris, Alexandre. Pascal Urro. Leveraging the power of cache memory. Gamasutra. April 9, 1999 http://www.gamasutra.com/features/19990409/cache_01.htm
- Gross, Ornit. Pentium III prefetch optimizations using the VTune performance analyzer. Gamasutra. July 30, 1999 http://www.gamasutra.com/features/19990730/sse_prefetch_01.htm
- Truong, Dan. François Bodin. André Seznec. Improving cache behavior of dynamically allocated data structures.