

# Cache Locality for Non-numerical Codes

---

María Jesús Garzarán  
CS 498: Compiler Optimizations  
Fall 2006

University of Illinois at Urbana-Champaign



# Outline

---

- ◆ Cache-conscious Structure Definition, by Trishul M. Chilimbi, Bob Davidson, and James Larus, PLDI 1999.
  - Structure Splitting
  - Field Reordering
- ◆ Cache-conscious Structure Layout, by Trishul M. Chilimbi, Mark D. Hill and James Larus, PLDI 1999.



# Cache Conscious Structure Definition

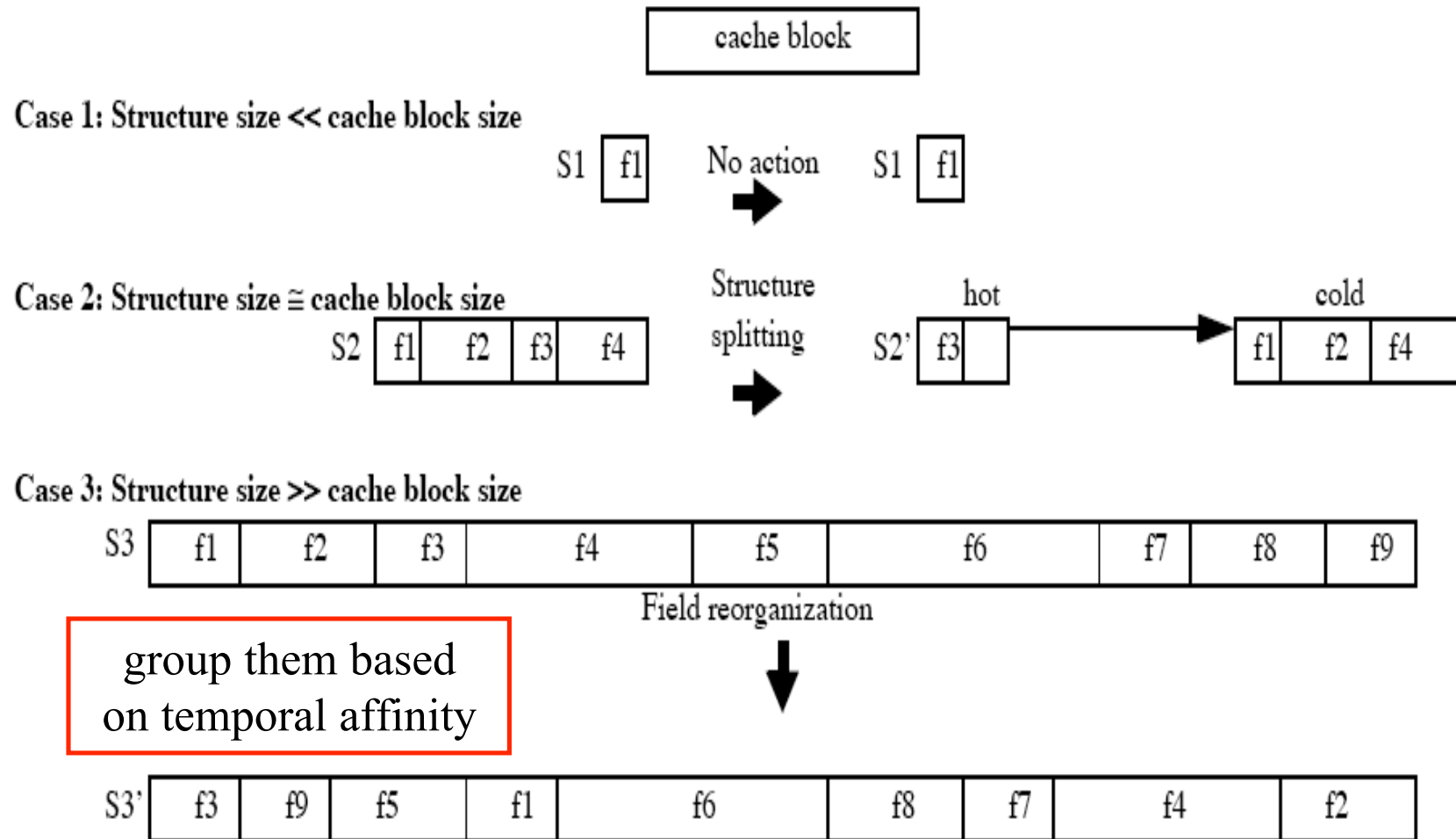


Figure 2. Cache-conscious structure definition.

# Structure Splitting

---

- ◆ Size of Java objects  $\approx$  Size of a cache block
- ◆ Split java classes:
  - hot (frequently accessed)
  - cold (rarely accessed)

It can be done based on profiled field access frequencies, or applied manually by the programmer
- ◆ Splitting allows more hot object instances to be packed into a cache block and kept in the cache at the same time.



# Size of heap allocated Java objects

---

- ◆ Most heap allocated Java objects are on average smaller than a cache block (consider objects < 256 bytes)

Table 2: Most heap allocated Java objects are small.

Program	# of heap allocated small objects (< 256 bytes)	Bytes allocated (small objects)	Avg. small object size (bytes)	# of heap allocated large objects (>= 256 bytes)	Bytes allocated (large objects)	% small objects
cassowary	958,355	19,016,272	19.8	6,094	2,720,904	99.4
espresso	287,209	8,461,896	29.5	1,583	1,761,104	99.5
javac	489,309	15,284,504	31.2	2,617	1,648,256	99.5
javadoc	359,746	12,598,624	35.0	1,605	1,158,160	99.6
pizza	269,329	7,739,384	28.7	1,605	1,696,936	99.4



# Live object statistics

---

- ◆ But small objects die fast and the technique is only effective for longer-lived objects, which survive scavenges.
- ◆ Table shows the # of small objects live after each scavenge, averaged over the entire program execution

Table 3: Most live Java objects are small.

Program	Avg. # of live small objects	Bytes occupied (live small objects)	Avg. live small object size (bytes)	Avg. # of live large objects	Bytes occupied (live large objects)	% live small objects
cassowary	25,648	586,304	22.9	1699	816,592	93.8
espresso	72,316	2,263,763	31.3	563	722,037	99.2
javac	64,898	2,013,496	31.0	194	150,206	99.7
javadoc	62,170	1,894,308	30.5	219	148,648	99.6
pizza	51,121	1,657,847	32.4	287	569,344	99.4



# Results

---

- ◆ Most java objects are small
- ◆ Average live object size is smaller than a cache block (64 bytes)



# Hot/Cold Class Splitting Algorithm

---

- ◆ Hot/cold class splitting class based on field access counts has a precise solution only if the program is rerun on the same input data.
- ◆ Unfortunately, field access frequencies for different program inputs are unpredictable.
- ◆ So, the class splitting algorithms uses heuristics.  
(field refers to class instance variable, i.e. non-static class variables)



# Structure Splitting

---

- ◆ The algorithm only considers classes that
  - total field accesses exceed a specific threshold, which are called the live classes.
  - are larger than 8 bytes
  - contain more than two fields.
- ◆ Label fields in the live classes as hot or cold
  - See heuristics on the paper on how to determine if a field is hot or cold.
  - Notice that smaller hot partitions:
    - more cache-block colocation
    - but increases access to cold fields (extra indirection)



# Program Transformation

---

- ◆ Cold fields are placed in a new class and contain
  - cold fields, labelled with public access modifier
  - constructor method
- ◆ Hot fields are in the original class
  - contains an extra field which is a reference to the new cold class
- ◆ Compiler modifies the code
  - Hot class constructors need to create a cold class instance and assign it to the class cold reference field
  - Access to cold fields requires an extra indirection through the cold class reference in the hot class



# Program Transformation. Example

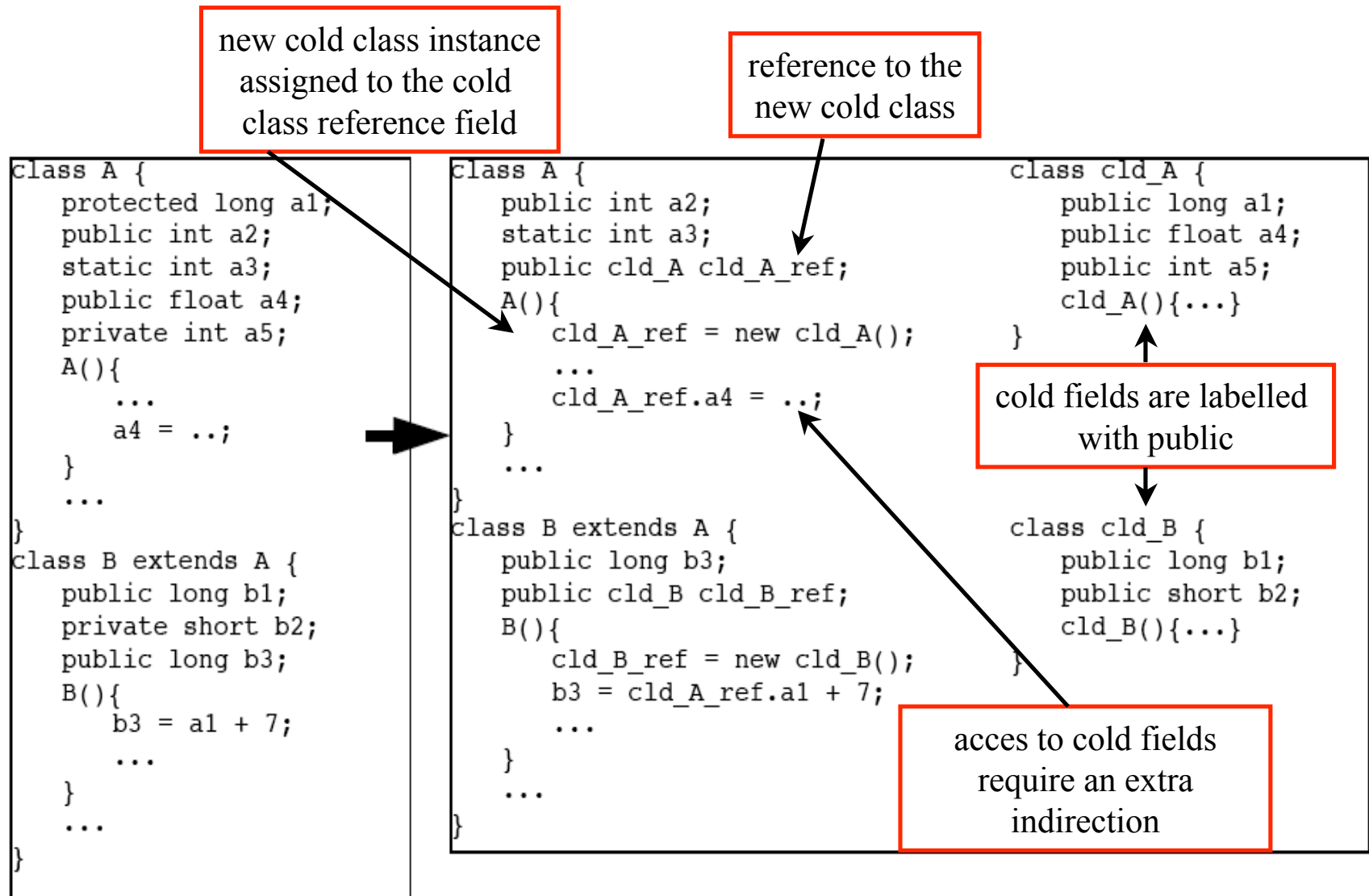


Figure 5. Program transformation.

# Results for structure splitting

Table 4: Class splitting potential.

Benchmark	# of classes (static)	# of accessed classes	# of 'live' classes	# of candidate classes (live & suitably sized)	# of split classes	Splitting success ratio (#split/#candidates)
cassowary	27	12	6	2	2	100.0%
espresso (input1)	104	72	57	33	11 (8)	33.3%
espresso (input2)	104	69	54	30	9 (8)	30.0%
javac (input1)	169	92	72	25	13 (11)	52.0%
javac (input2)	169	86	68	23	11 (11)	47.8%
javadoc (input1)	173	67	38	13	9 (7)	69.2%
javadoc (input2)	173	62	30	11	7 (7)	63.6%
pizza (input1)	207	100	72	39	10 (9)	25.6%
pizza (input2)	207	95	69	36	10 (9)	27.8%

# Results for structure splitting

Table 5: Split class characteristics

Benchmarks	Split class access / total prog. accesses	Avg. pre-split class size (static)	Avg. pre-split class size (dyn)	Avg. post-split (hot) class size (static)	Avg. post-split (hot) class size (dyn)	Avg. reduction in (hot) class size (static)	Avg. reduction in (hot) class size (dyn)	Avg. normalized temperature differential	Additional space allocated for cold class field ref (bytes)
cassowary	45.8%	48.0	76.0	18.0	24.0	62.5%	68.4%	98.6%	56
espresso (input1)	55.3%	41.4	44.8	28.3	34.7	31.6%	22.5%	79.2%	74,464
espresso (input2)	59.4%	42.1	36.2	25.7	30.1	39.0%	16.9%	79.5%	58,160
javac (input1)	45.4%	45.6	26.3	27.2	21.6	40.4%	17.9%	75.1%	50,372
javac (input2)	47.1%	49.2	27.2	28.6	22.4	41.9%	17.6%	79.8%	36,604
javadoc (input1)	56.6%	55.0	48.4	29.3	38.1	46.7%	21.3%	85.7%	20,880
javadoc (input2)	57.7%	59.4	55.1	33.6	44.0	43.4%	20.1%	85.2%	12,740
pizza (input1)	58.9%	37.8	34.4	22.9	27.3	39.4%	20.6%	79.4%	55,652
pizza (input2)	64.0%	39.4	30.9	23.7	24.4	39.9%	21.0%	82.1%	38,004



# Results for structure splitting

Table 6: Impact of hot/cold object partitioning on L2 miss rate.

Program	L2 cache miss rate (base)	L2 cache miss rate (CL)	L2 cache miss rate (CL + CS)	% reduction in L2 miss rate (CL)	% reduction in L2 miss rate (CL + CS)
cassowary	8.6%	6.1%	5.2%	29.1%	39.5%
espresso	9.8%	8.2%	5.6%	16.3%	42.9%
javac	9.6%	7.7%	6.7%	19.8%	30.2%
javadoc	6.5%	5.3%	4.6%	18.5%	29.2%
pizza	9.0%	7.5%	5.4%	16.7%	40.0%

Table 7: Impact of hot/cold object partitioning on execution time.

Program	Execution time in secs (base)	Execution time in secs (CL)	Execution time in secs (CL + CS)	% reduction in execution time (CL)	% reduction in execution time (CL + CS)
cassowary	34.46	27.67	25.73	19.7	25.3
espresso	44.94	40.67	32.46	9.5	27.8
javac	59.89	53.18	49.14	11.2	17.9
javadoc	44.42	39.26	36.15	11.6	18.6
pizza	28.59	25.78	21.09	9.8	26.2



# Field Reordering

---

- ◆ Applied for large structures with many fields
- ◆ The goal is to reduce cache pressure by grouping fields that have high temporal locality in a cache block
- ◆ Description of bbcache
  - Tool that produces structure field recommendations



# Results for field Reordering

---

Table 8: **bbcache** evaluation metrics for 5 active SQL Server structures.

Structure	Cache block utilization (original order)	Cache block utilization (recommended order)	Cache pressure (original order)	Cache pressure (recommended order)
ExecCxt	0.607	0.711	4.216	3.173
SargMgr	0.714	0.992	1.753	0.876
Pss	0.589	0.643	8.611	5.312
Xdes	0.615	0.738	2.734	1.553
Buf	0.698	0.730	2.165	1.670



# Cache-conscious Structure Layout. Trishul M. Chilimbi, M. D. Hill and James R. Larus. PLDI 99.

---

- ◆ Locality can be improved by:

1. changing program's data access pattern

Applied to scientific programs that manipulate dense matrices:

- uniform, random accesses of elements
- static analysis of data dependences

2. changing data organization and layout

Pointer programs have neither of these properties

They have locational transparency: elements in a structure can be placed at different memory (and cache) locations without changing a program's semantics.

Two placement techniques:

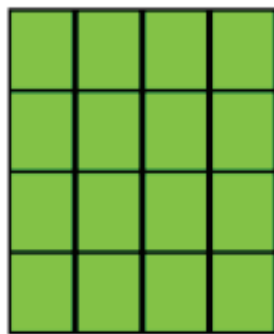
- coloring
- clustsering



# Arrays and Structures

## Arrays

- Random Access
- Strong Program Analysis

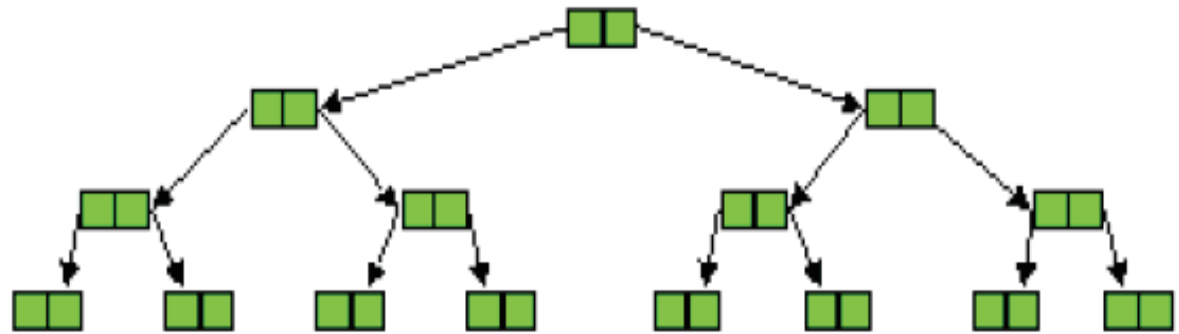


- Data Fixed

Change program

## Structures

- Pointer Access
- Pointer Analysis



- Data Movable

Change data

# Clustering

---

- ◆ Packs data structure elements likely to be accessed contemporaneously into a cache block.
- ◆ Improves spatial and temporal locality and provides implicit prefetch.
- ◆ One way to cluster a tree is to pack subtrees into a cache block.

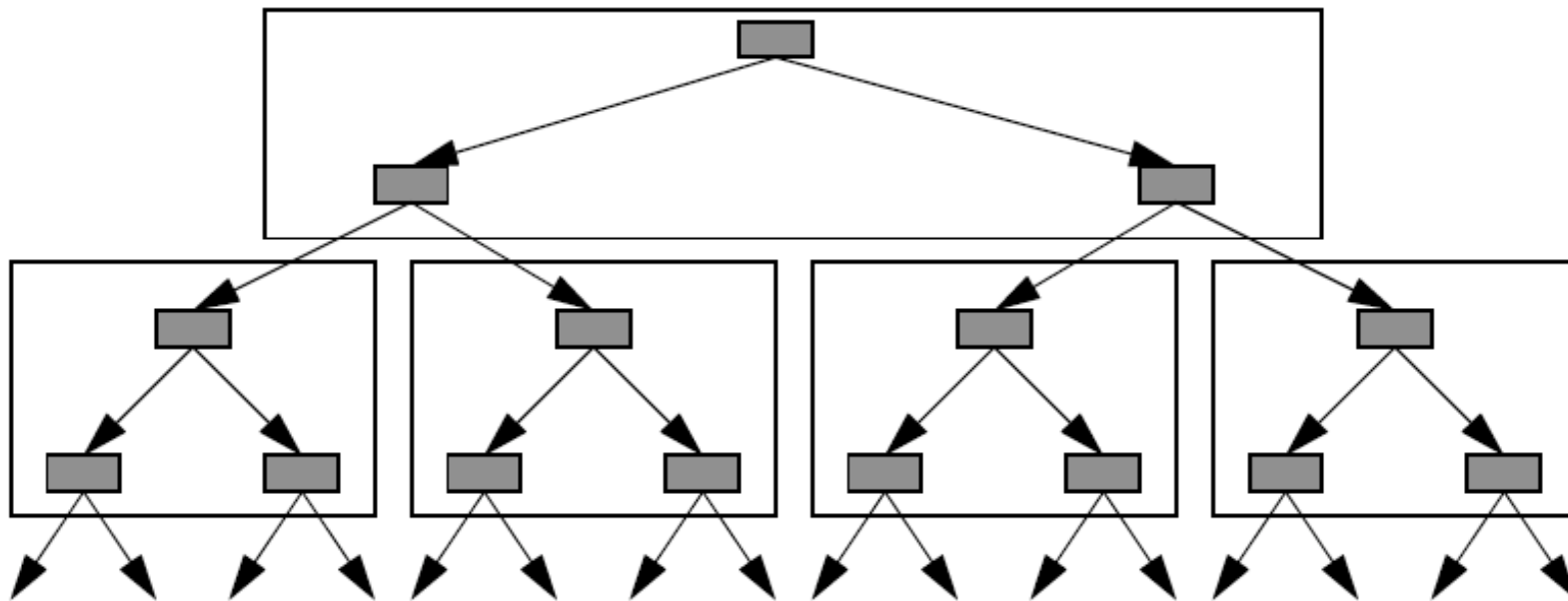


Figure 1. Subtree clustering.

# Clustering

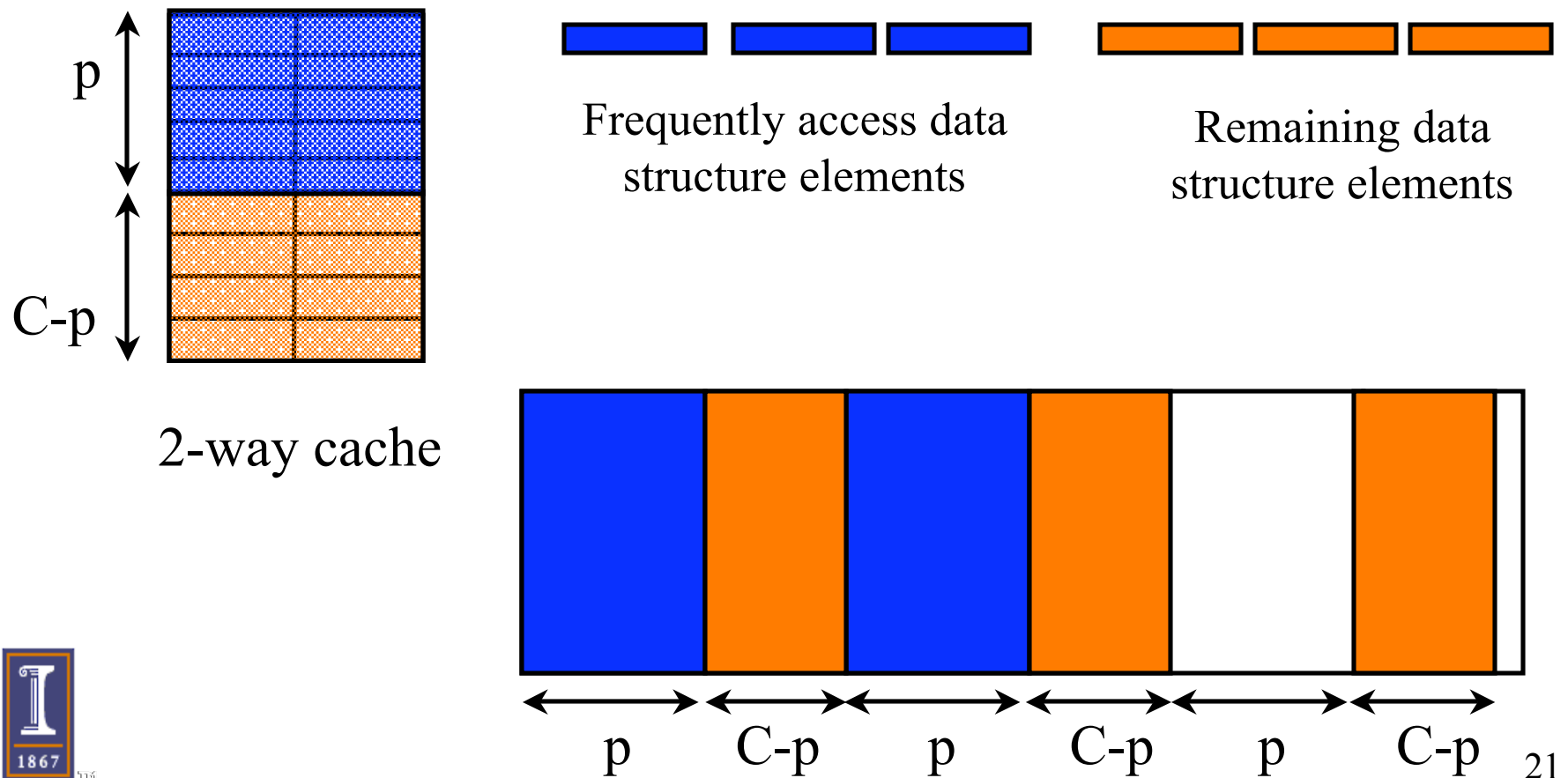
---

- ◆ Why is this clustering for binary tree good?
  - Assuming random tree search, the probability of accessing either child of a node is  $1/2$ .
  - With  $K$  nodes of a subtree clustered in a cache block, the expected number of accesses to the block is the height of the subtree,  $\log_2(k+1)$ , which is greater than 2 when  $K > 3$ .
- ◆ With a depth-first clustering, the expected number of accesses to the block is smaller.
  - Of course this is only true for a random access pattern.



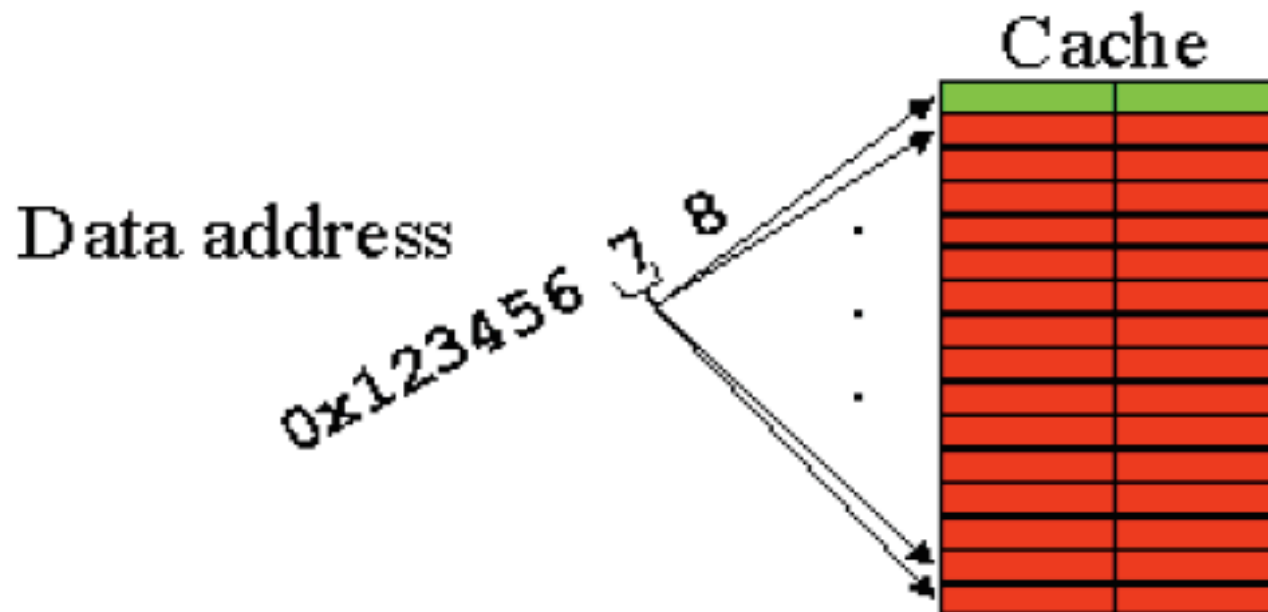
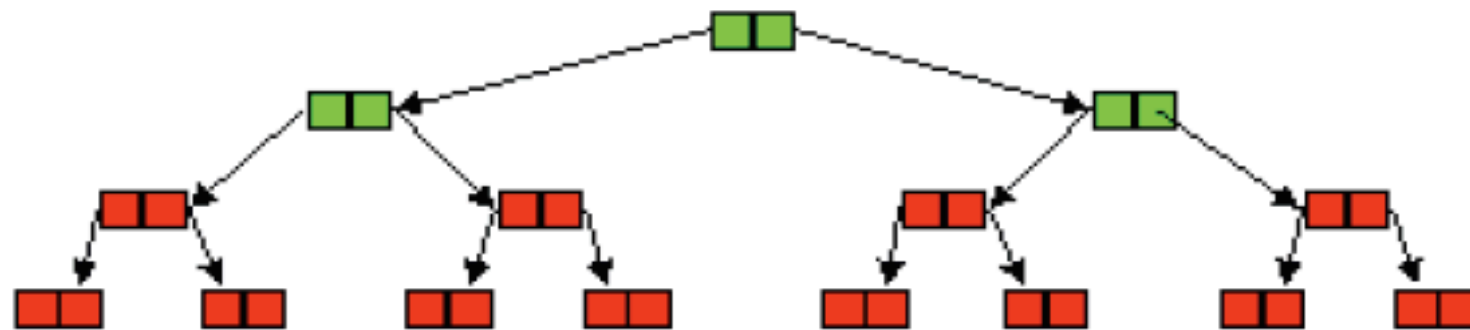
# Coloring

- Coloring maps contemporaneously-accessed elements to non-conflicting regions of the cache.



# Coloring, Cont'd

Keep heavily referenced items in cache



# Strategies

---

- ◆ Cache-conscious Data Reorganization: ccmorph
- ◆ ccmorph operates on tree-like structures without external pointer into the middle of the structure.
- ◆ ccmorph requires the pointer to the root of the data structure, a function to traverse the structure, and cache parameters.
- ◆ ccmorph applies coloring and clustering as explained above

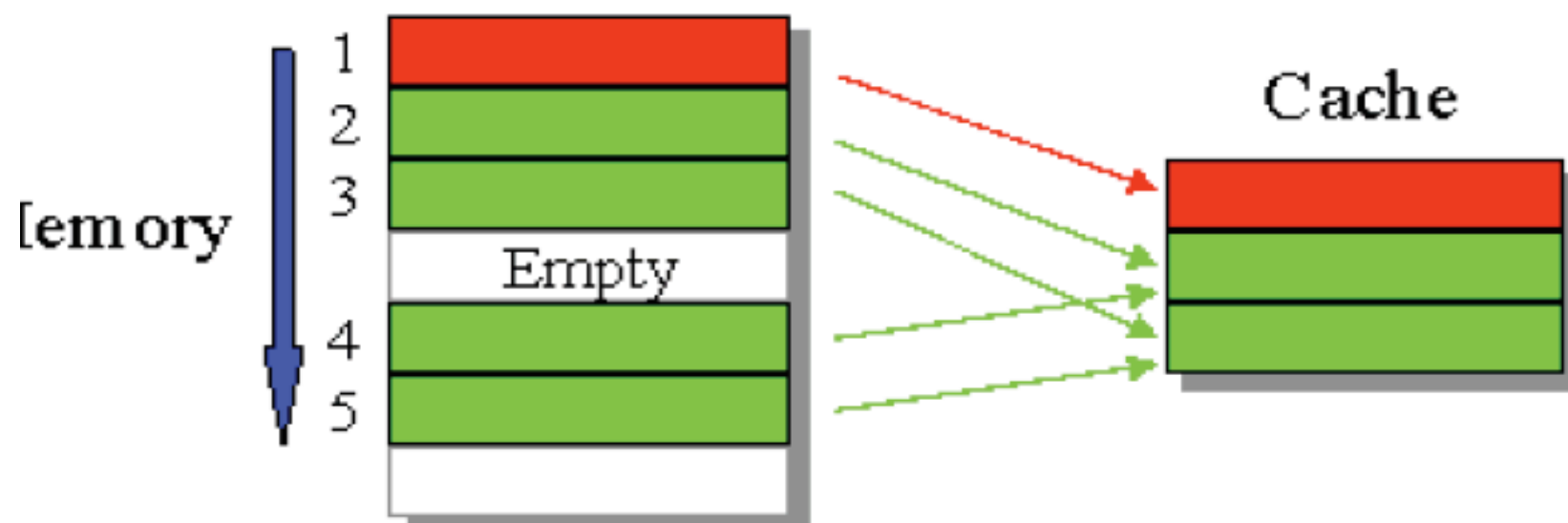
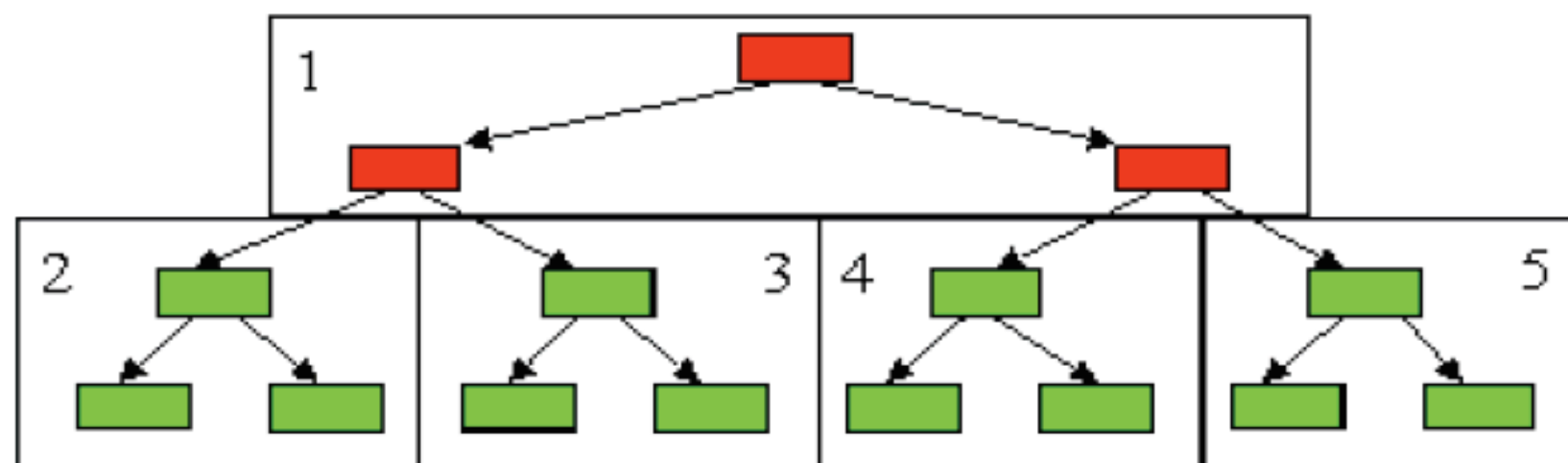


# Using ccmorph

```
main()
{
    .....
    root = maketree(4096, ..., ... );
    ccmorph(&root, next_node,
    Num_nodes, Max_kids,
    Cache_blk_size, Cache_size,
    Cache_assoc, Color_const);
    .....
}
```

```
Quadtree next_node (Quadtree node, int i)
{
    /* Valid values for i are -1, 1 .. Max_kids */
    switch(i) {
        case -1: return (node->parent);
        case 1: return (node->nw);
        case 2: return (node->ne);
        case 3: return (node->sw);
        case 4: return (node->se);
    }
}
```

# Tree Reorganizer (ccmorph)



# ccmalloc: cache-conscious heap allocator

---

- ◆ Memory allocator similar to malloc
- ◆ Takes as argument an existing data structure element likely to be accessed contemporaneously (e.g. the parent of a tree node).
- ◆ ccmalloc tries to locate the new data item in the same cache block as the existing item.



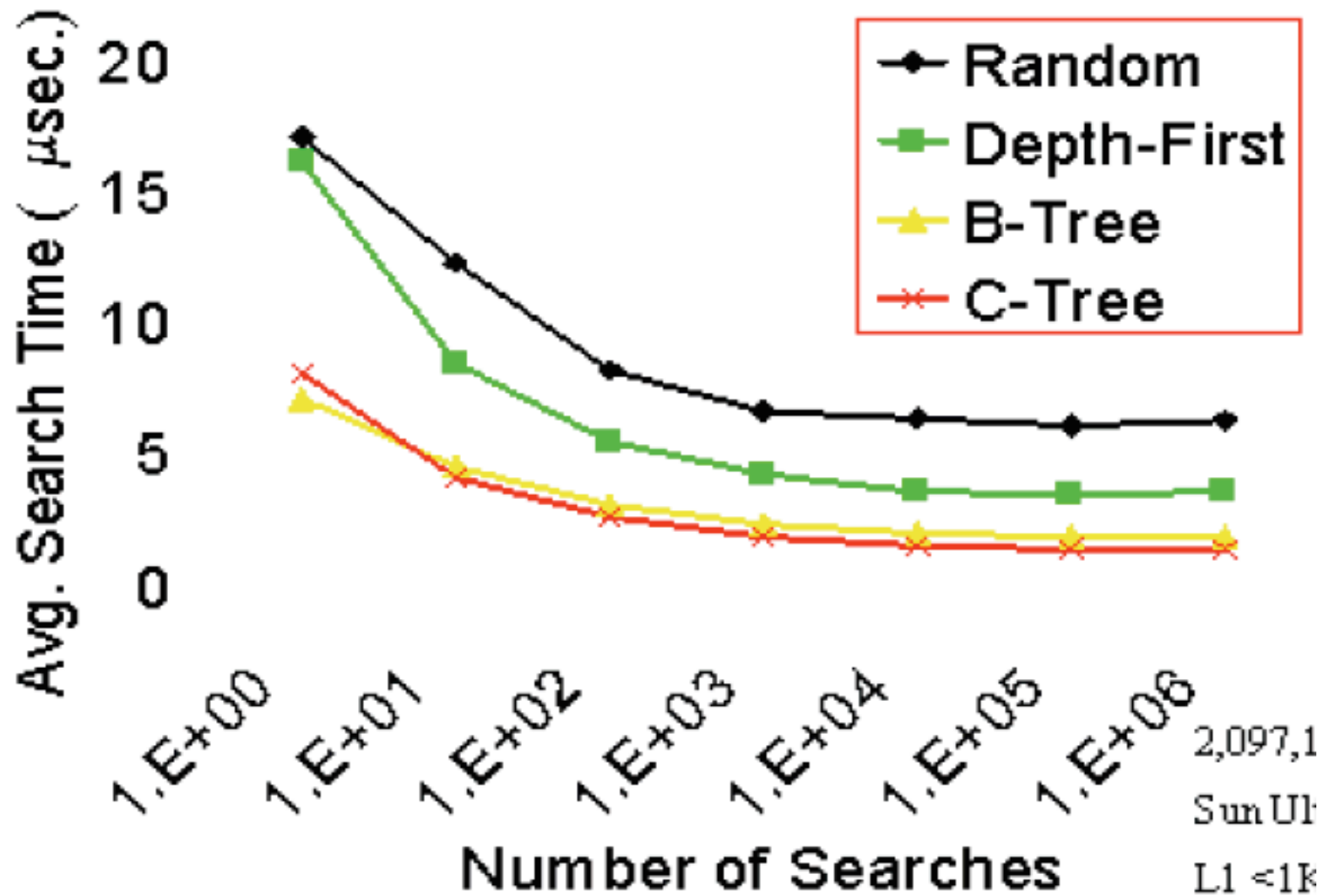
# Cache-Conscious Heap Allocation (ccmalloc)

```
/* Add an element to the end of a list */  
while (list != NULL) {  
    b = list;  
    list = list->forward;  
}  
list = (struct List *) ccmalloc(sizeof(struct List), b);
```

# Evaluation Methodology

- Olden benchmarks (small, pointer-based codes)
  - RSIM (detailed cycle-by-cycle simulator)
- Macrobenchmarks (large, real-world applications)
  - Sun Ultraserver E5000
    - 167 MHz, L1 <1K, 4, 1> 6 cycles, L2 <16K, 16, 1> 70 cycles

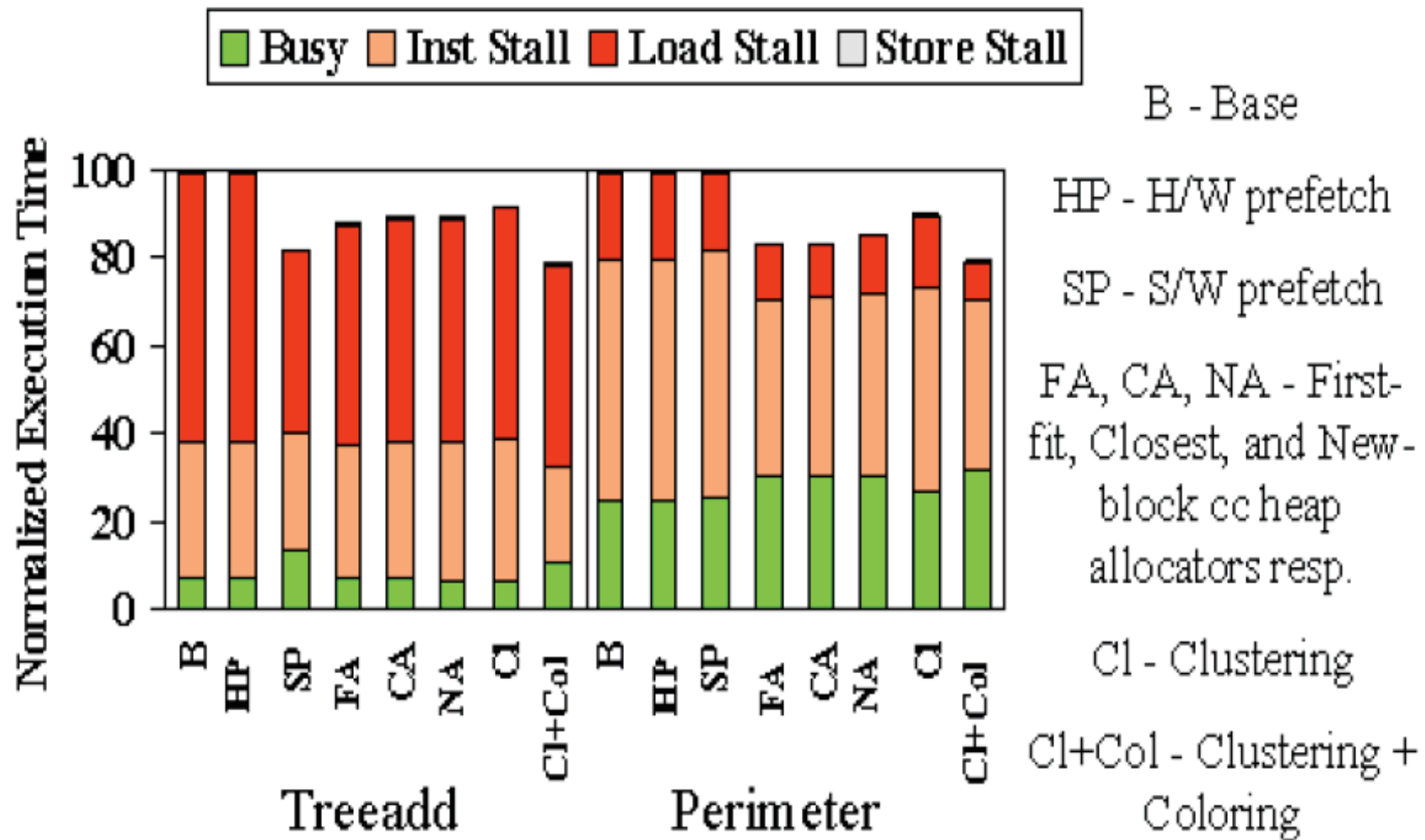
# Binary Tree Benchmark



2,097,1  
Sun UI  
L1 <1K



# Olden Benchmark Performance



# Olden Benchmark Performance

