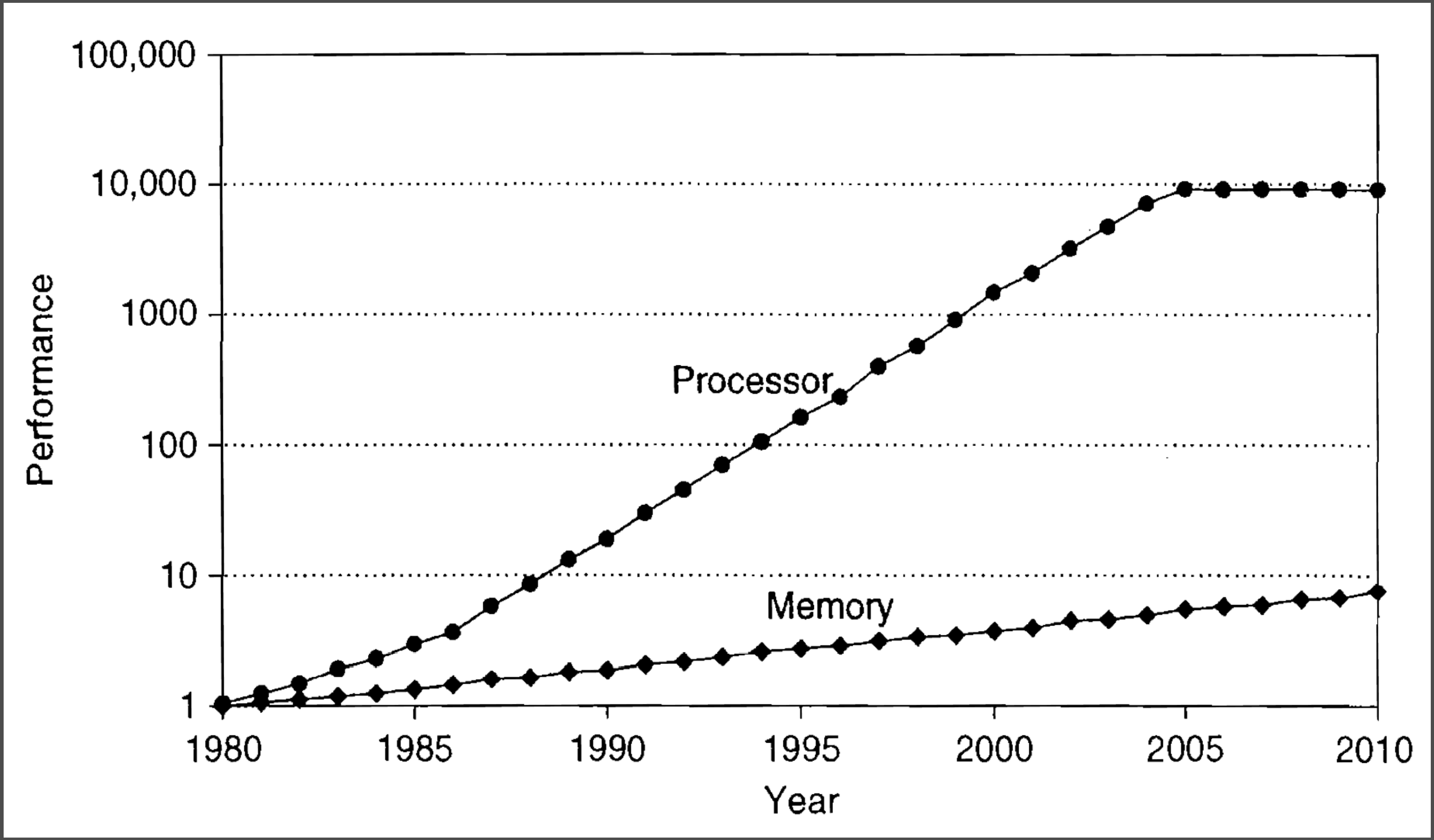
Dissecting Memory Problems – A Semantic Approach

Alfredo Gimenez

Motivation

Historical trends in memory performance and energy efficiency show that memory access is becoming one of the most significant bottlenecks to increasing performance and energy efficiency

Motivation - **Performance**

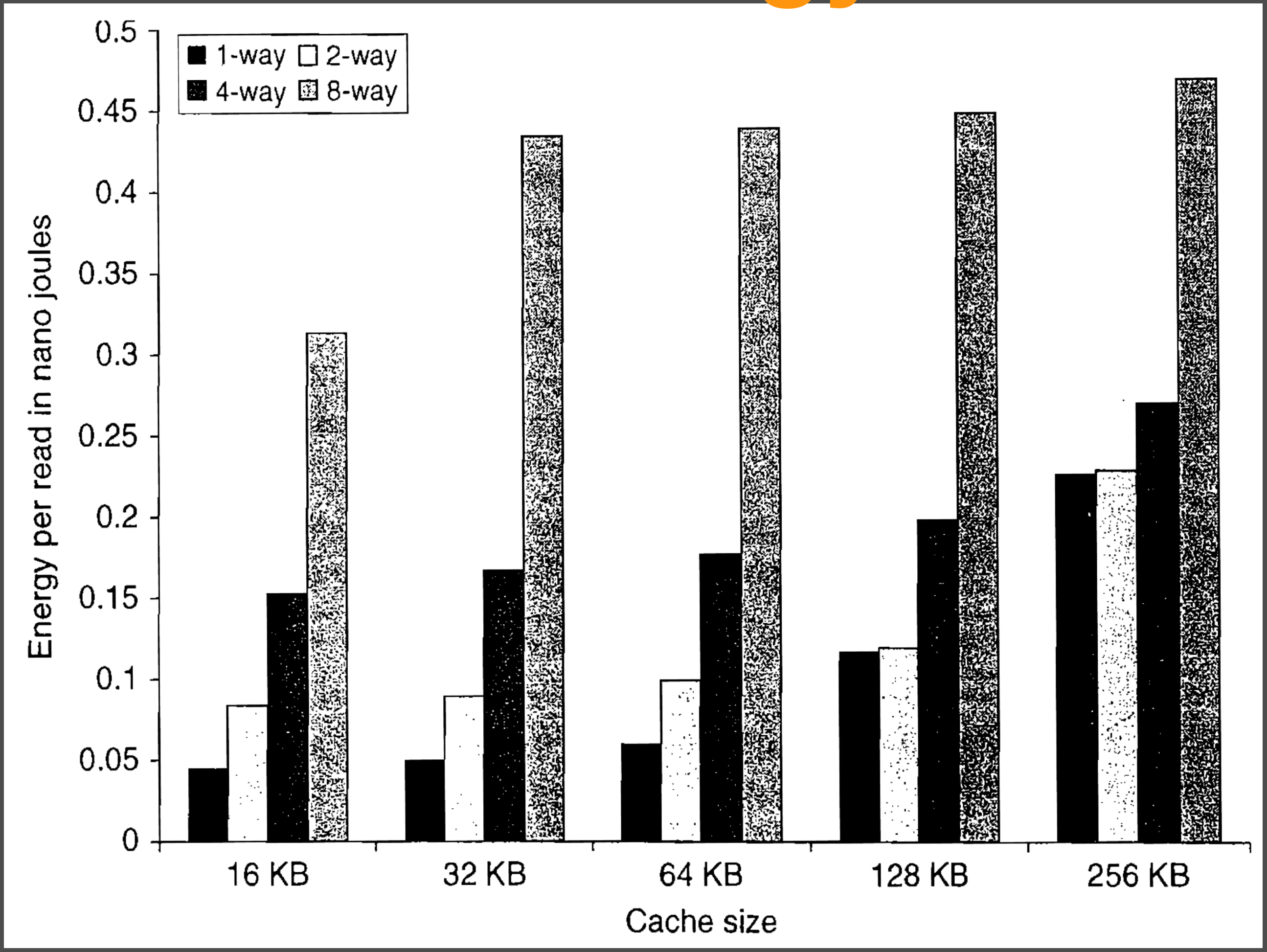


Single core performance and memory performance gains relative to 1980*

Memory is becoming a more frequent and larger bottleneck

*Hennessy and Patterson, Computer Architecture, a Quantitative Approach, 5th ed.

Motivation – **Energy Efficiency**



As cache size and associativity increases, power consumption also increases*

Cache-efficiency → Energy efficiency

*Hennessy and Patterson, Computer Architecture, a Quantitative Approach, 5th ed.

Mitigating the Memory Access Bottleneck

The software solution: write code which makes use of the fastest and most efficient cache

Figuring out how to optimize code for cache efficiency is not trivial, and often not portable

We need a way to collect and interpret memory performance data to help make software cache optimization easier

Gathering Memory Performance Data

- Up until recently, could only gather process-wide data
  - e.g. # of cache misses over time
- Recent hardware additions allow us to sample load events precisely
  - Sampling based on events/instructions
  - Intel PEBS, AMD IBS

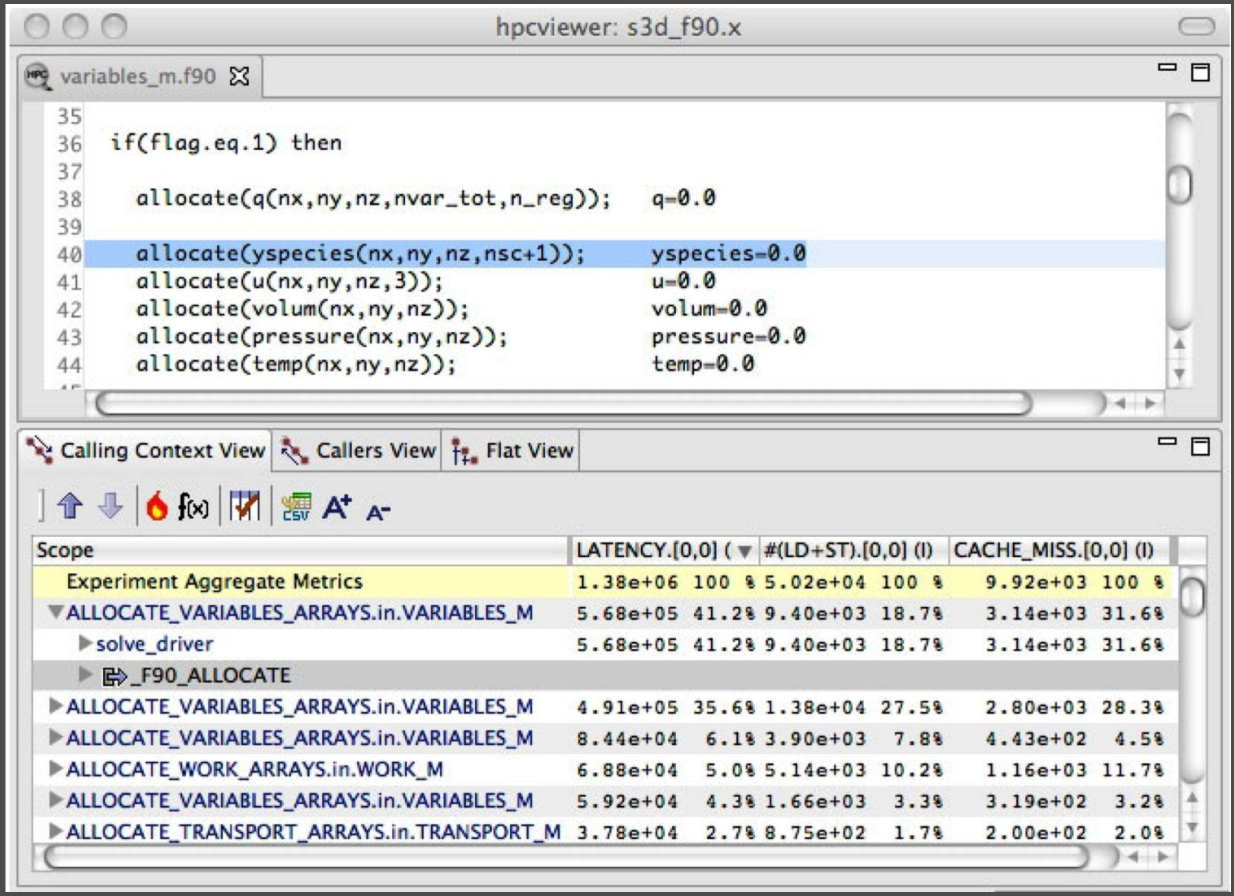
Gathering Memory Performance Data

- Load Event Samples contain:
  - The raw address operand of the load instruction
  - How many cycles the load took
  - Where in the memory hierarchy the address was resolved (e.g. L1 cache, RAM)

- **Still, we need a way to effectively interpret these samples**

Interpreting Memory Data

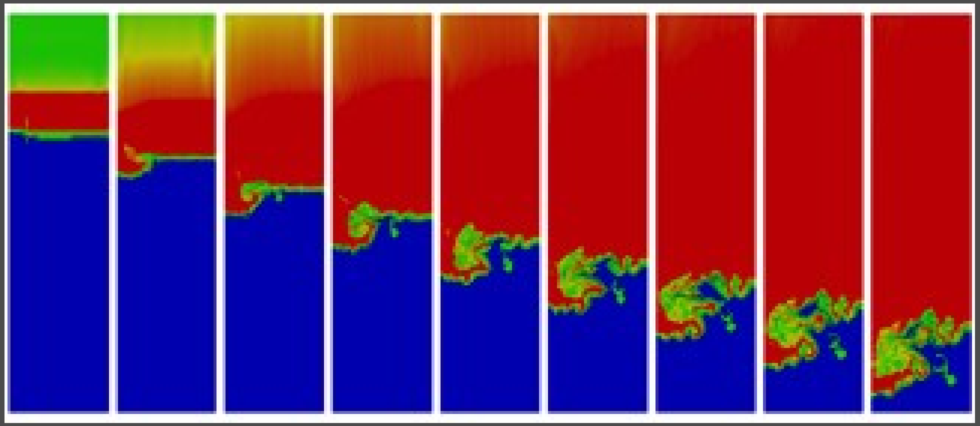
- “Data-centric”: accumulate the samples in terms of data symbols, i.e. variables [Liu]
- Store allocated buffer addresses in a data structure, correlate samples post-mortem



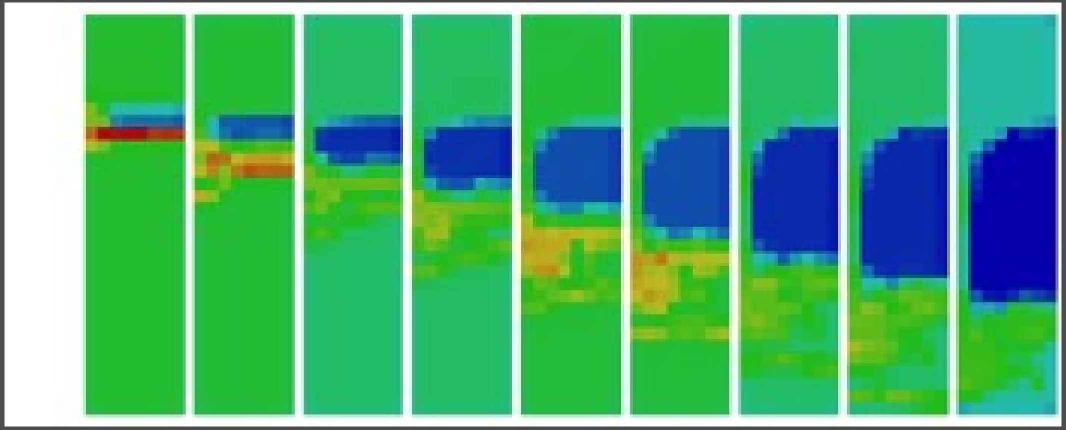
Xu Liu and John Mellor-Crummey, "Pinpointing Data Locality Problems Using Data-Centric Analysis" 2011 International Symposium on Code Generation and Optimization (CGO11) April 2-6, Chamonix, France.

Interpreting Hardware Data

- Hardware Domain → Natural Domain [PAVE]
- Per-process flops overlaid onto the natural domain
- Hardware counter data interpreted in terms of the problem being solved



Hydrodynamics simulation results



FLOP/s per MPI process, mapped onto the natural domain – the physical space of the problem

Bringing Higher-Level Semantics to Memory Performance Data

- We'd like to answer questions like:
  - Where, within this buffer, are RAM hits occurring?
  - How does memory performance correlate with the physical space of a simulation? (edge cases?)
  - What part of the algorithm (not the code) results in most inefficient memory accesses?
  - At what exact point are we exhausting L1 cache? L2?

Semantic Memory

- To answer these, we need to know:
  - Which buffers are relevant and what do they represent?
  - How are they accessed?
  - How do they map to the Natural Domain of an application?
- We store this information in a

Semantic Memory Tree

Semantic Memory

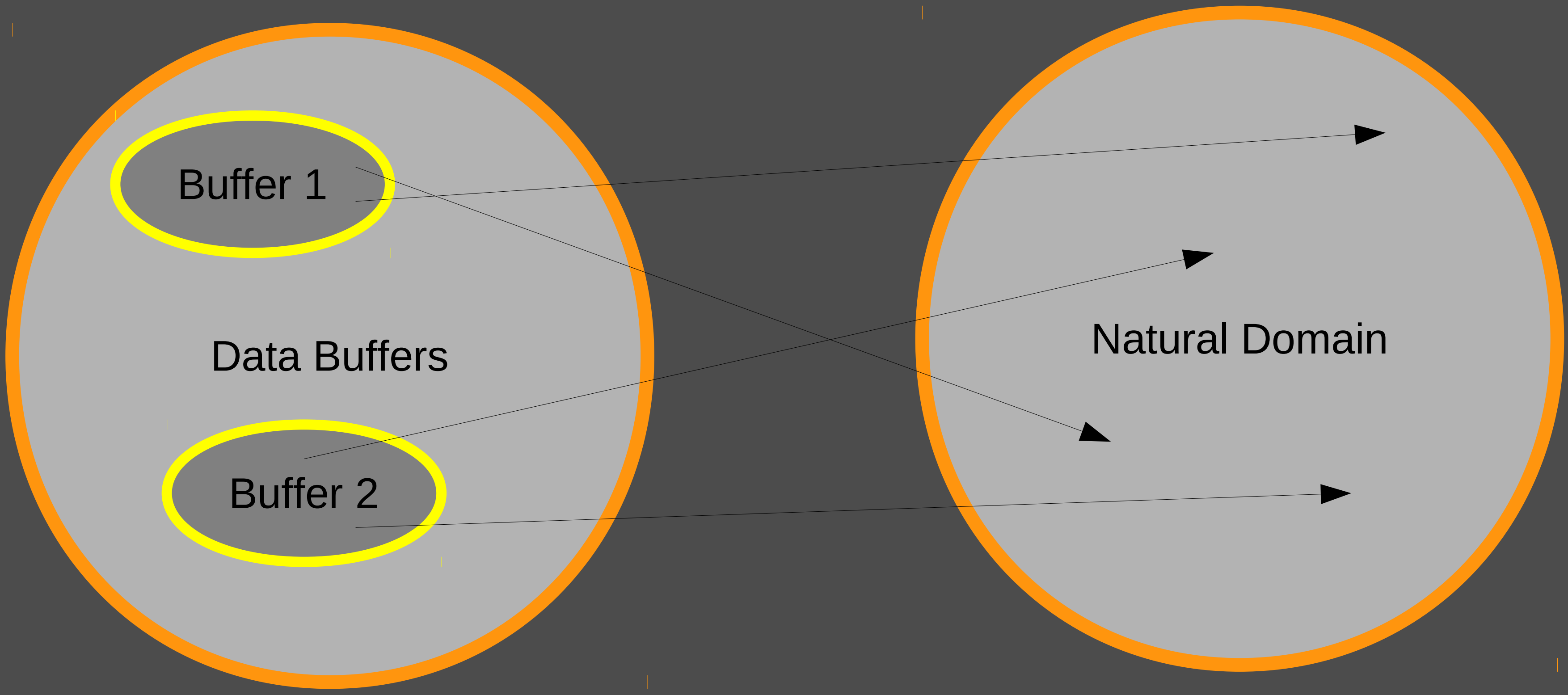
- **Semantic Memory Range**
  - Label, e.g. “mesh elements”
  - Size of a single element, e.g. sizeof(double)
  - Length of vector, e.g. 3 elements/vector
  - Address of first element
  - Address of last element

Semantic Memory

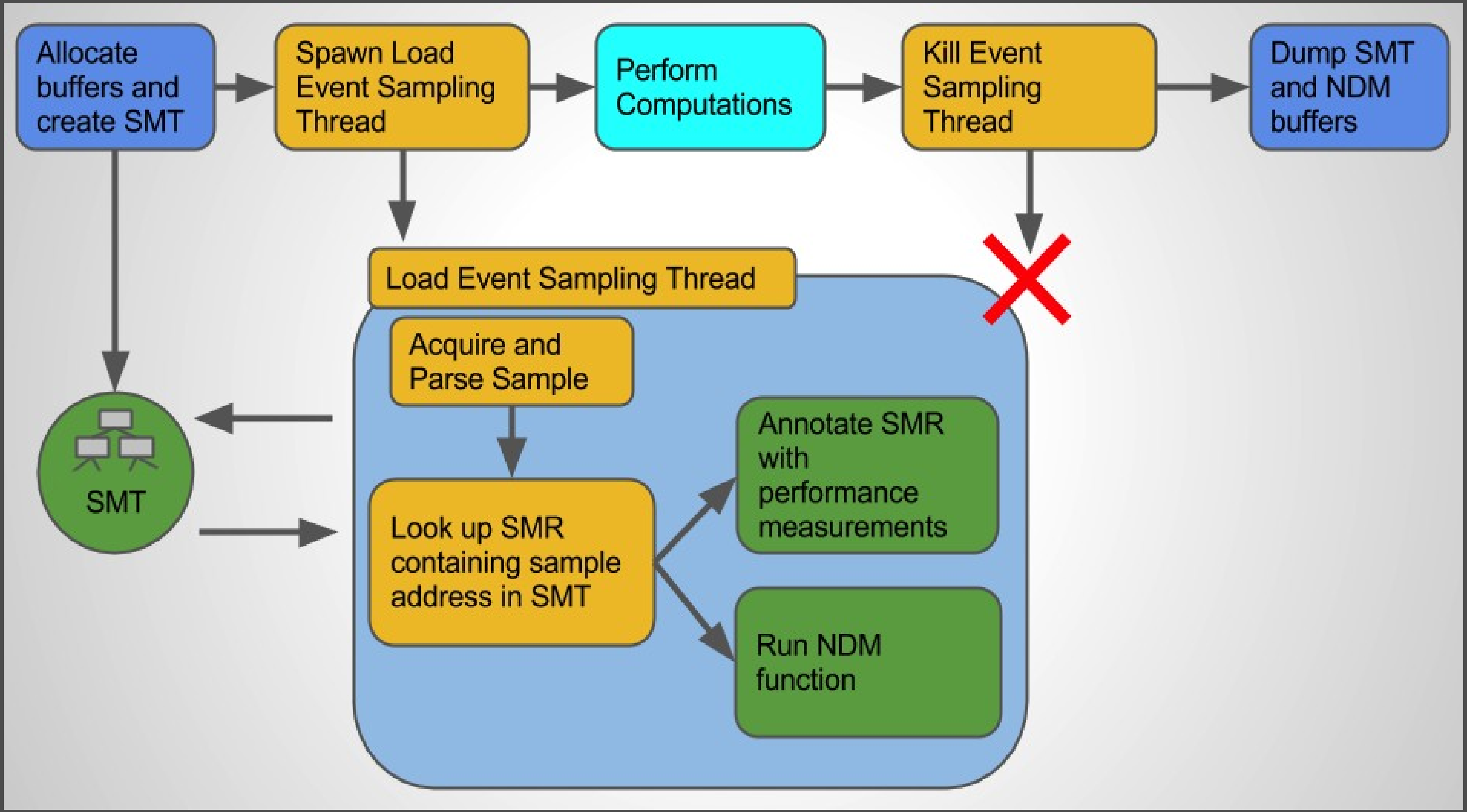
- **Semantic Memory Tree**
  - A tree of Semantic Memory Ranges (SMRs)
  - Self-balancing (AVL) **lookup** tree
  - Semantically-organized **visualization** tree

Semantic Memory

- **Natural Domain Mapping**
  - A programmer-defined function to map indices from a buffer to a location in the Natural Domain



Instrumentation Overview



Instrumentation Syntax

```
#include "SMRTree.h"
```

Creating SMRs

```
...

    SMRTree *smrt = new SMRTree();

    int N = 1024;
    int D = 3;
    double scalar[N];
    double vector[N*D];
    std::vector<CustomType> custom[N];

    smrt->addSMR("Scalar Data",// Label
                 scalar,       // Start address
                 scalar+N,     // End address
                 sizeof(double)); // Element size
    smrt->addSMR("Vector Data",
                 vector,
                 vector+N*D,
                 sizeof(double),
                 D);           // Dimensions
    smrt->addSMR("Custom Data",
                 custom); // Type and addresses
                          // inferred from std::vector
...
```

Instrumentation Syntax

```
smrt = new SMRTree();
SMRNode *A_SMR = smrt->addSMR("A",sizeof(double),A,&A[N*N]);
SMRNode *B_SMR = smrt->addSMR("B",sizeof(double),B,&B[N*N]);
SMRNode *C_SMR = smrt->addSMR("C",sizeof(double),C,&C[N*N]);

SMRNode *input_group = smrt->createSMRGroup("Input");
SMRNode *output_group = smrt->createSMRGroup("Output");

input_group->addGroupMember(A_SMR);
input_group->addGroupMember(B_SMR);
output_group->addGroupMember(C_SMR);
```

Group ranges by semantics, i.e. “input” and “output”

Instrumentation Syntax

Mapping to the Natural Domain via a custom function

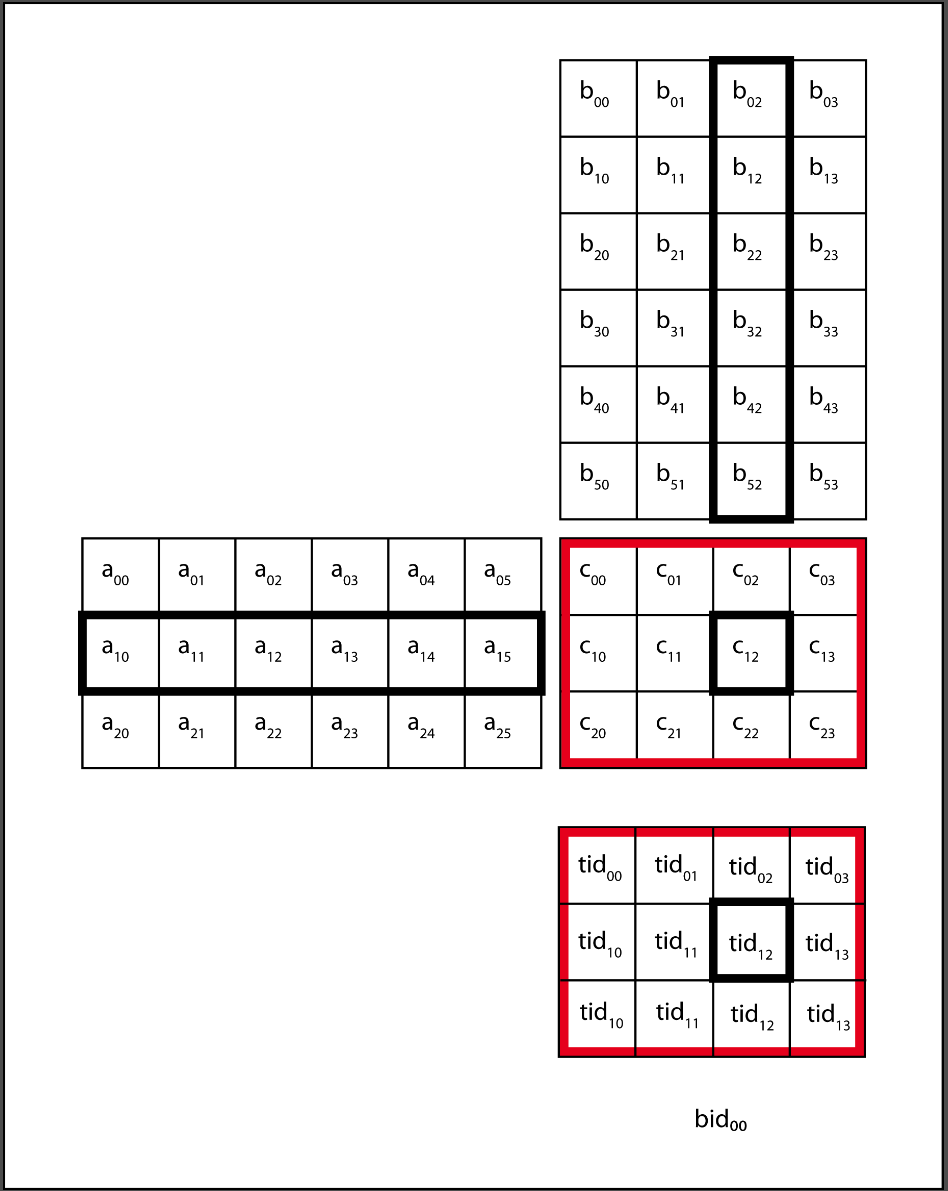
```
void* matrixNDMfunc(SMRNode *smr,
                    struct mem_sample *sample)
{
    // Obtain the index of the address
    int bufferIndex =
        smr->elementalIndexOf(sample->daddr);

    // Calculate the x and y indices (row-major)
    int xIndex = bufferIndex % ROW_SIZE;
    int yIndex = bufferIndex / ROW_SIZE;

    // Record into global cost buffer
    globalCost[xIndex][yIndex] += sample->cost;
}
...
```

Visualizing the data!

1) Visualize the Semantic Memory Tree
2) Visualize the data overlaid onto the Natural Domain

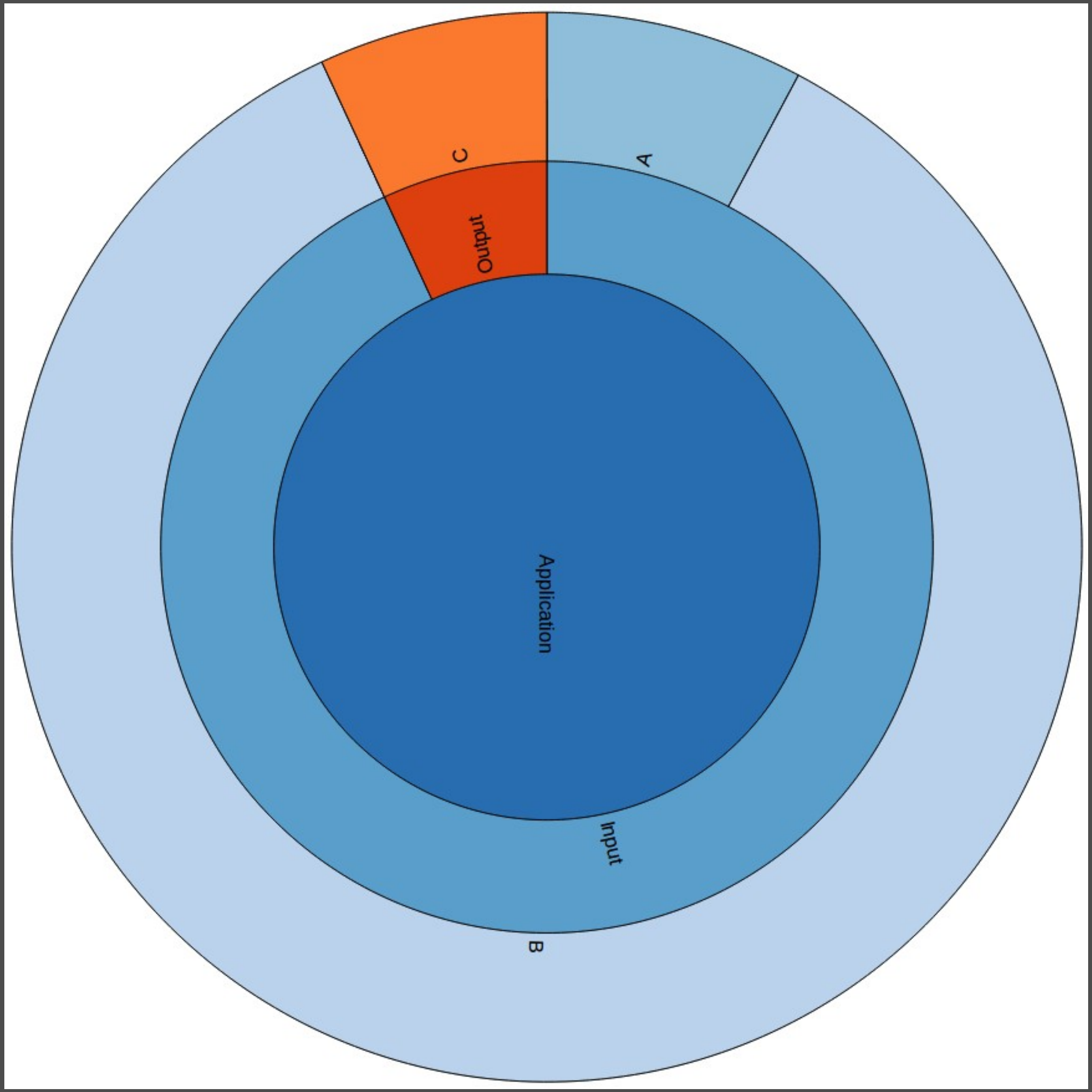
A Canonical Case-Study: Matrix Multiplication

- Naive matrix multiplication exhausts cache limits, causes poor memory access performance
- Blocked matrix multiplication allows elements to be reused, blocks can fit in cache

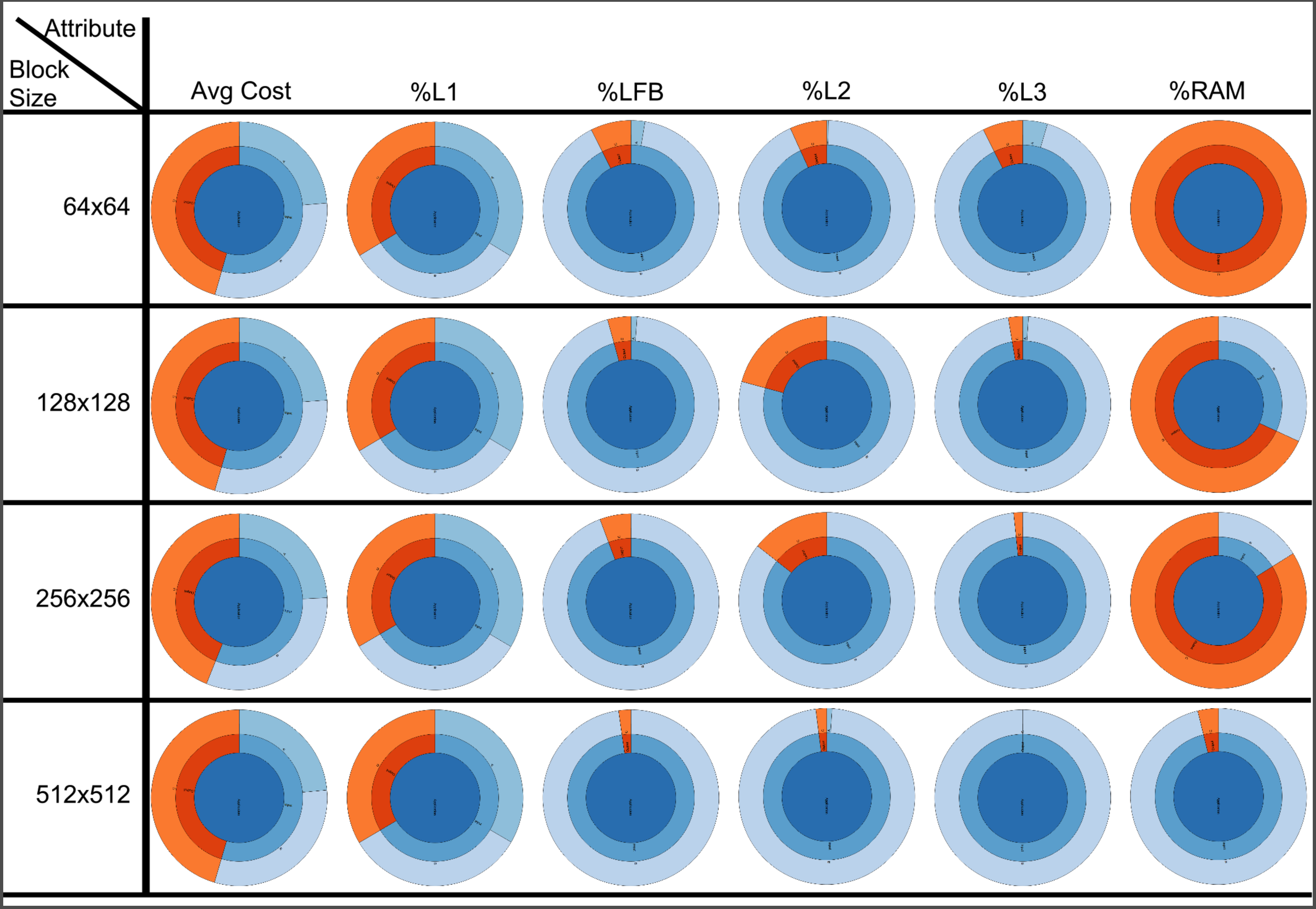


Semantic Memory Tree View

Example: % of Samples Resolved in L2 Cache

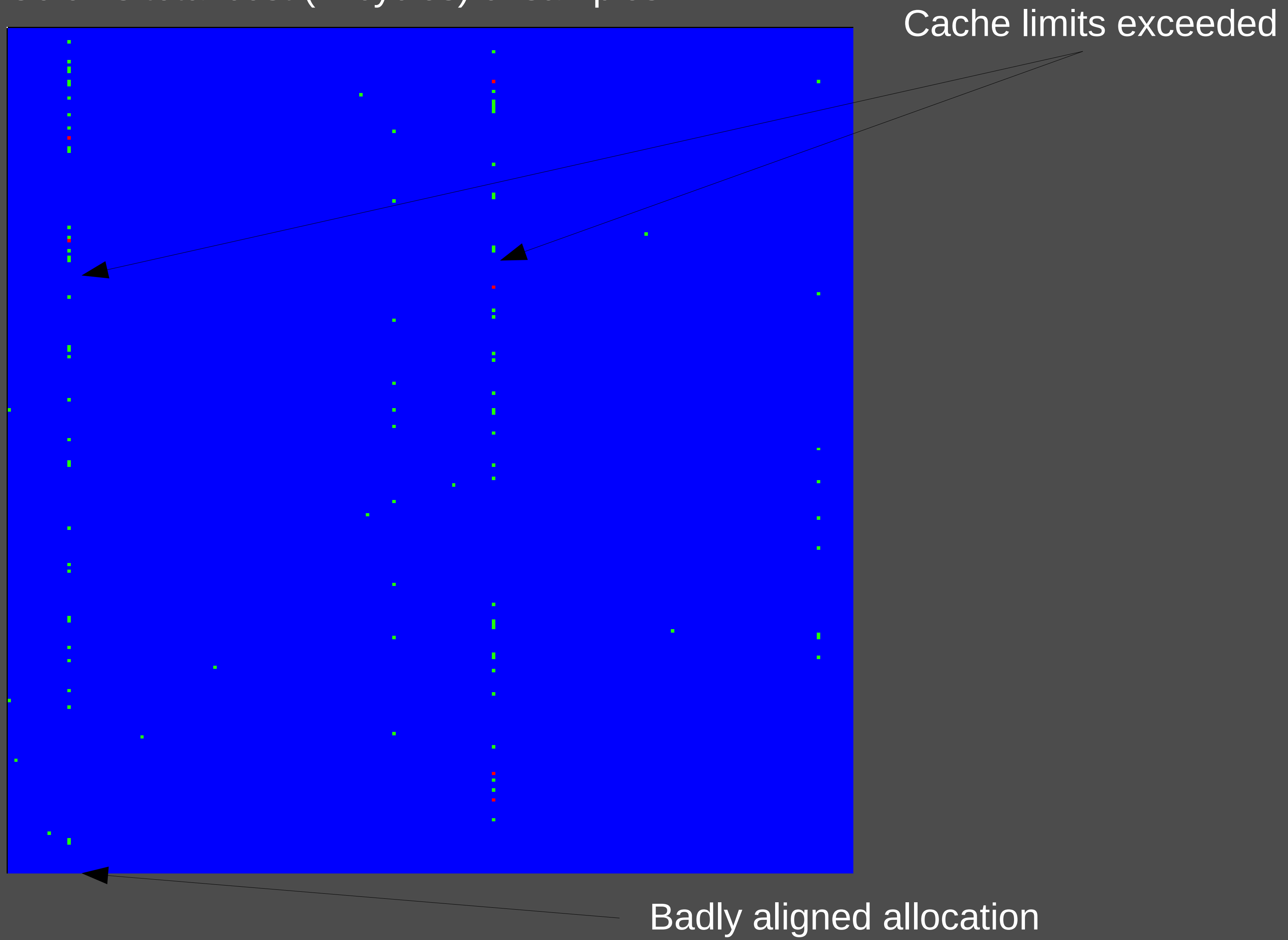


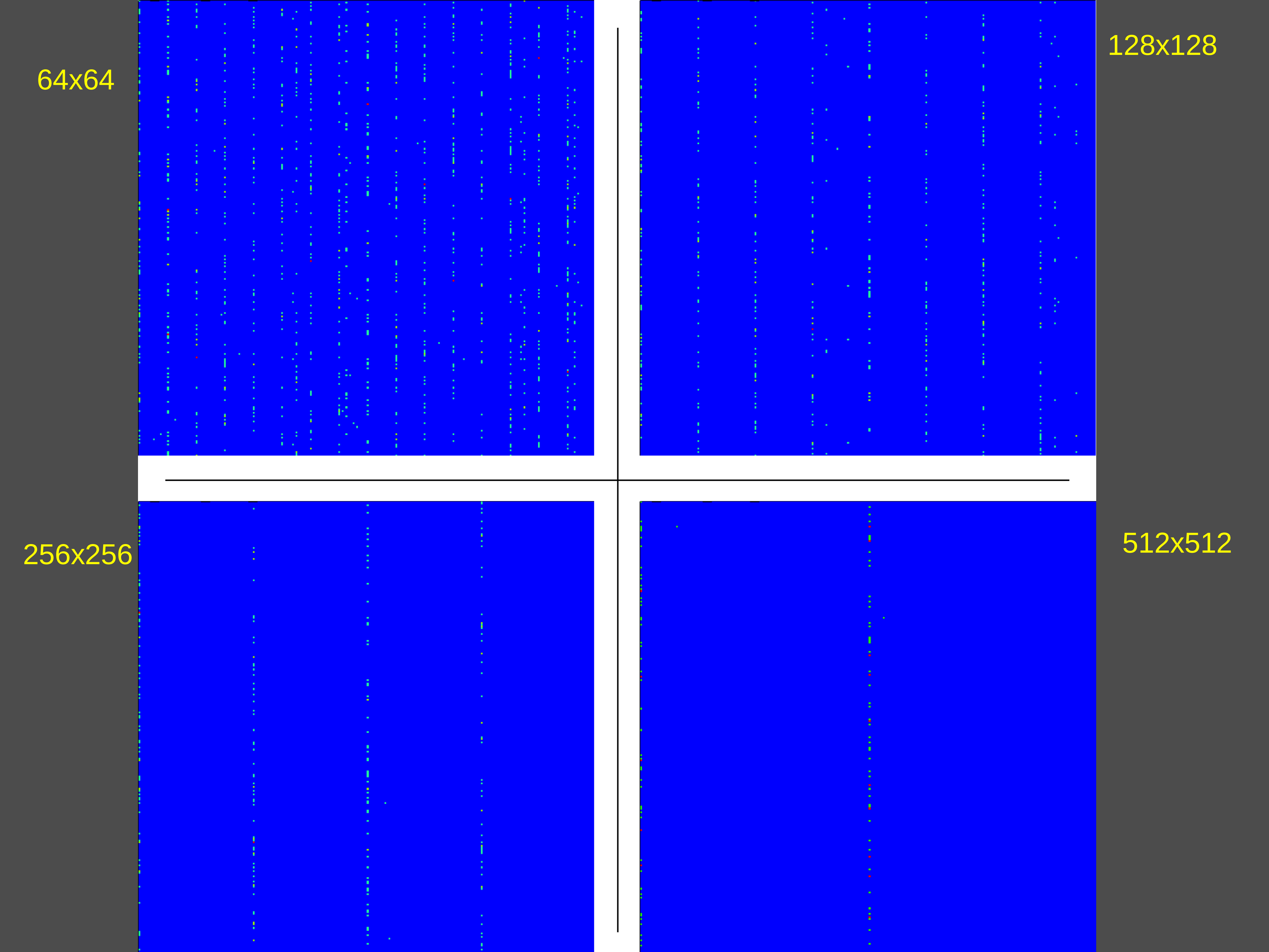
Semantic Memory Tree View



Natural Domain Overlay

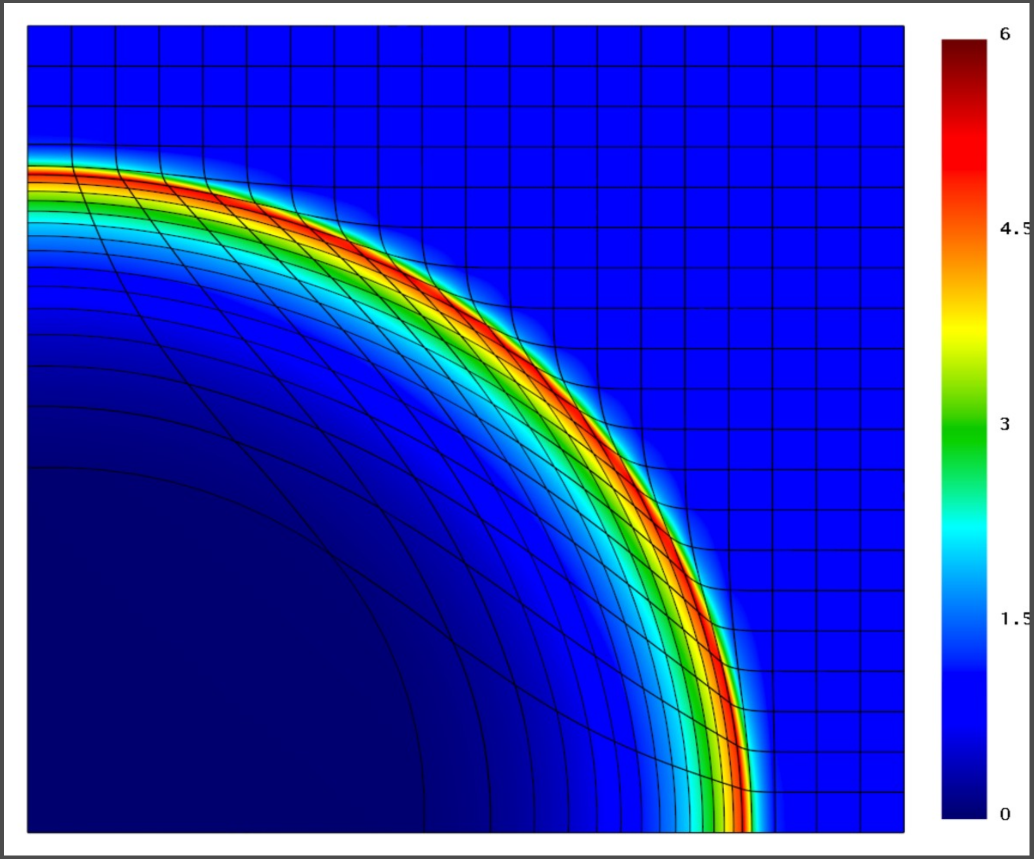
X, Y are matrix indices
Color is total cost (in cycles) of samples




64x64
128x128
256x256
512x512

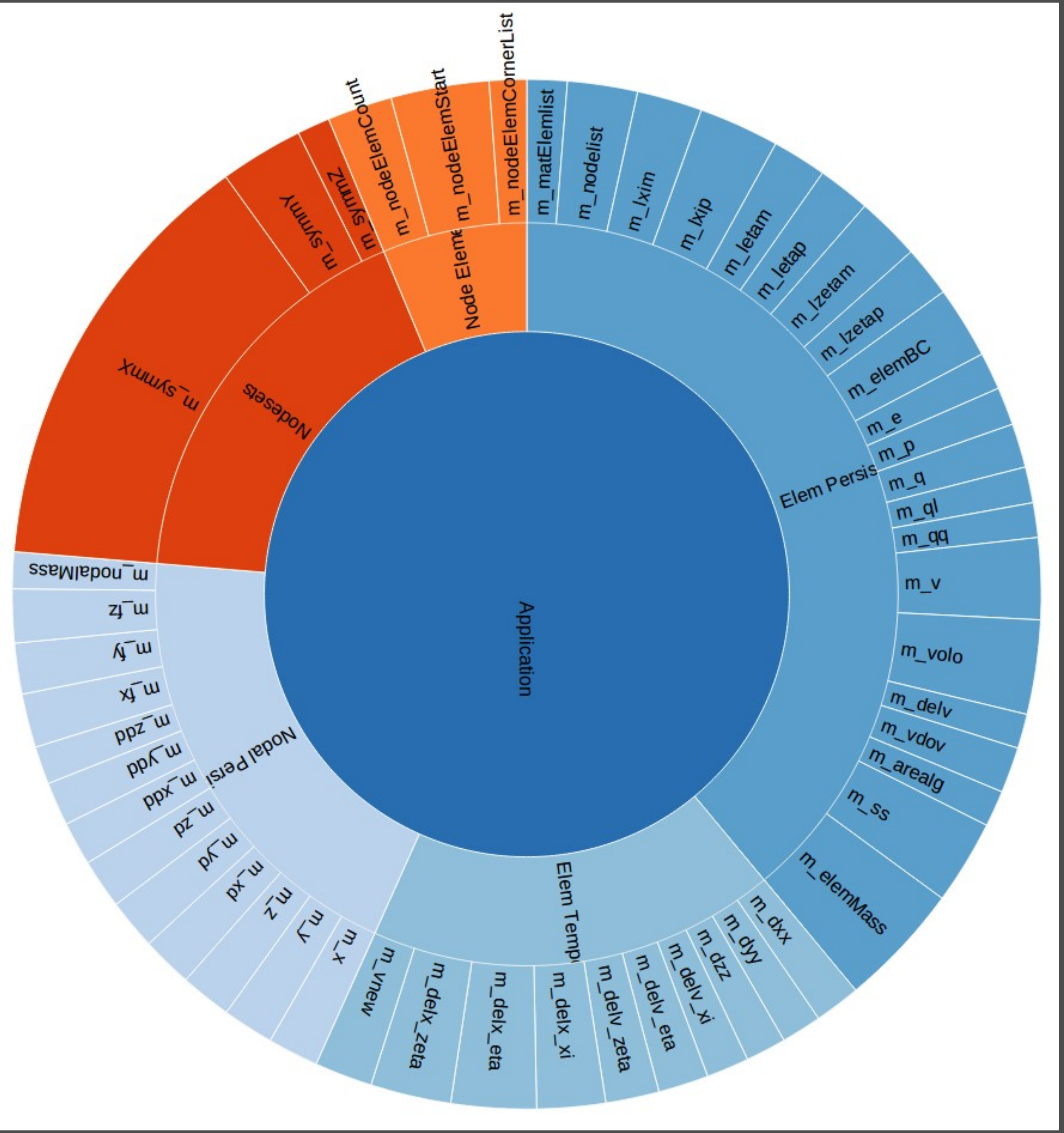
A Real-World Example: LULESH

- Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
- Unstructured mesh means a more complex NDM function (have to calculate indirection)



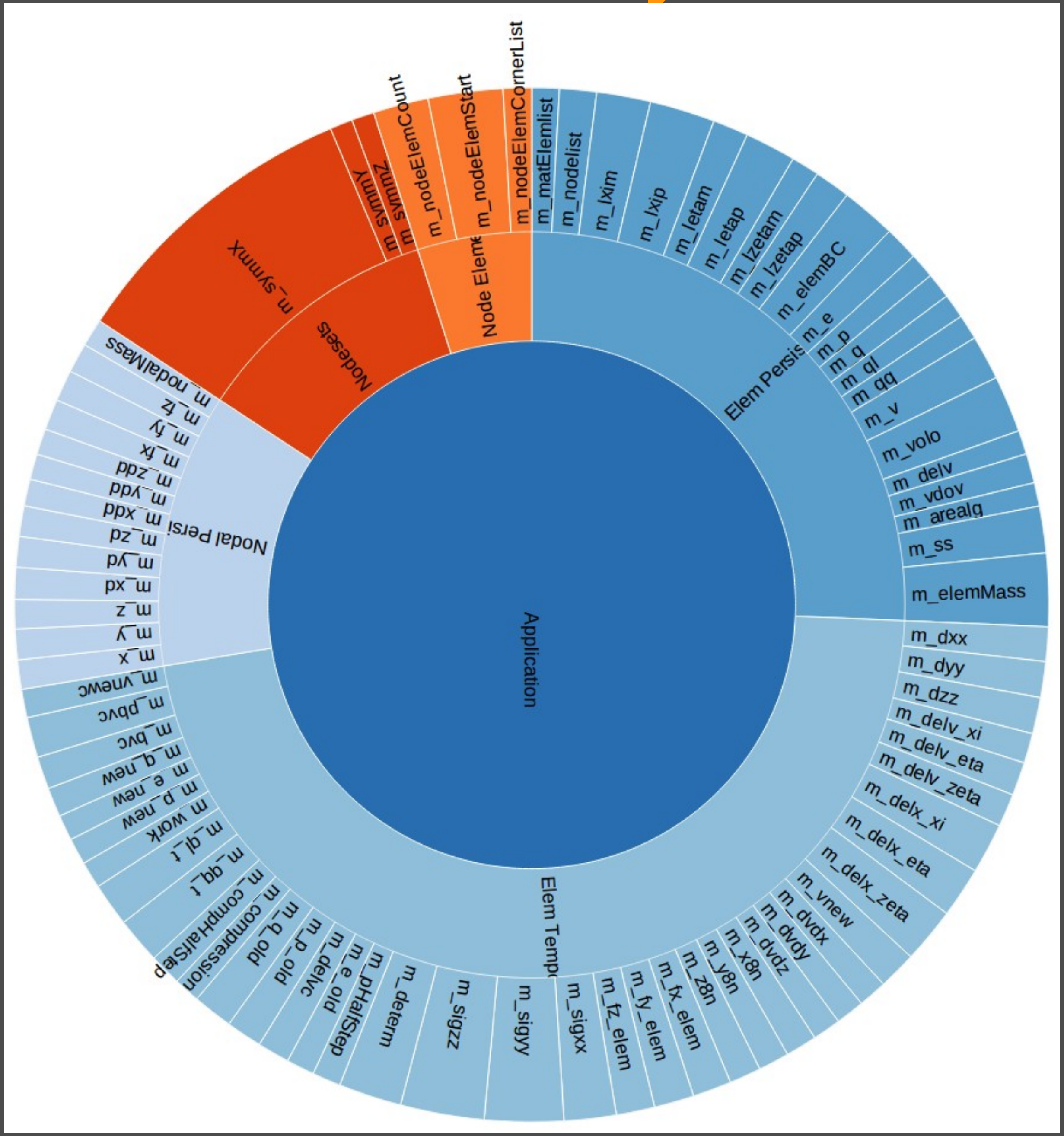
Semantic Memory Tree View

Avg Cost



Semantic Memory Tree View

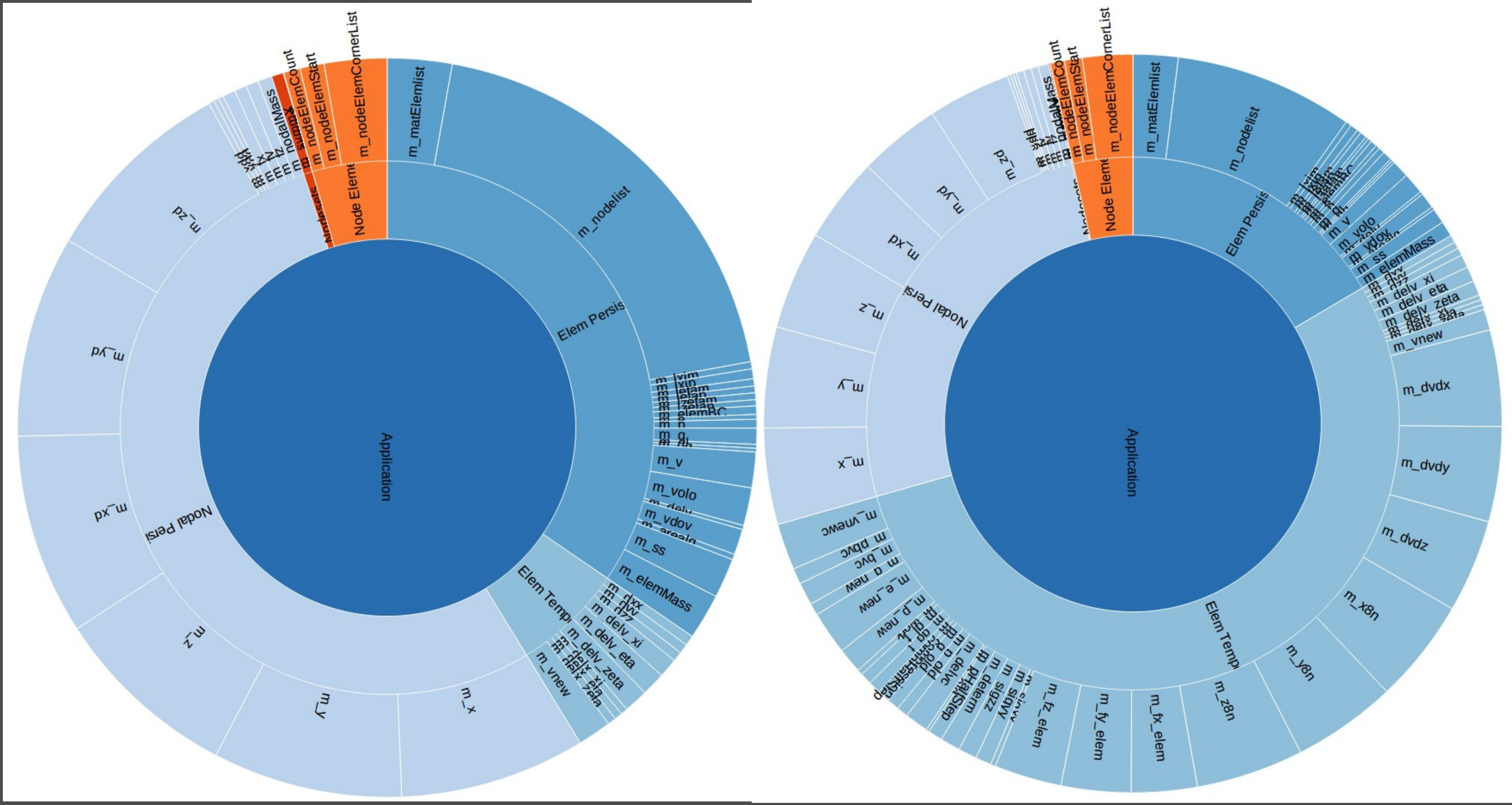
Avg Cost



Optimization: using more temporary variables

Persistent variables less of a factor

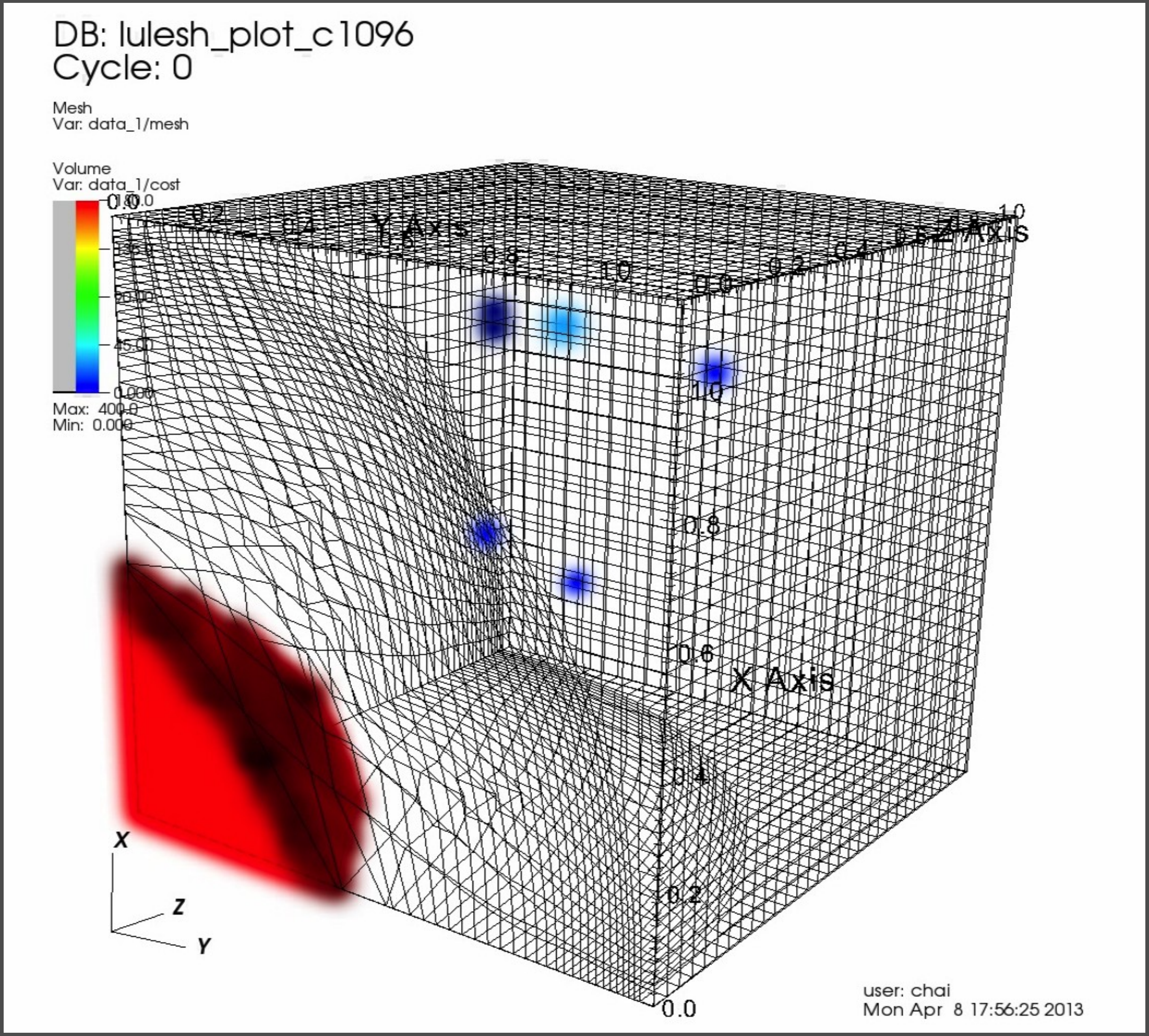
Semantic Memory Tree View



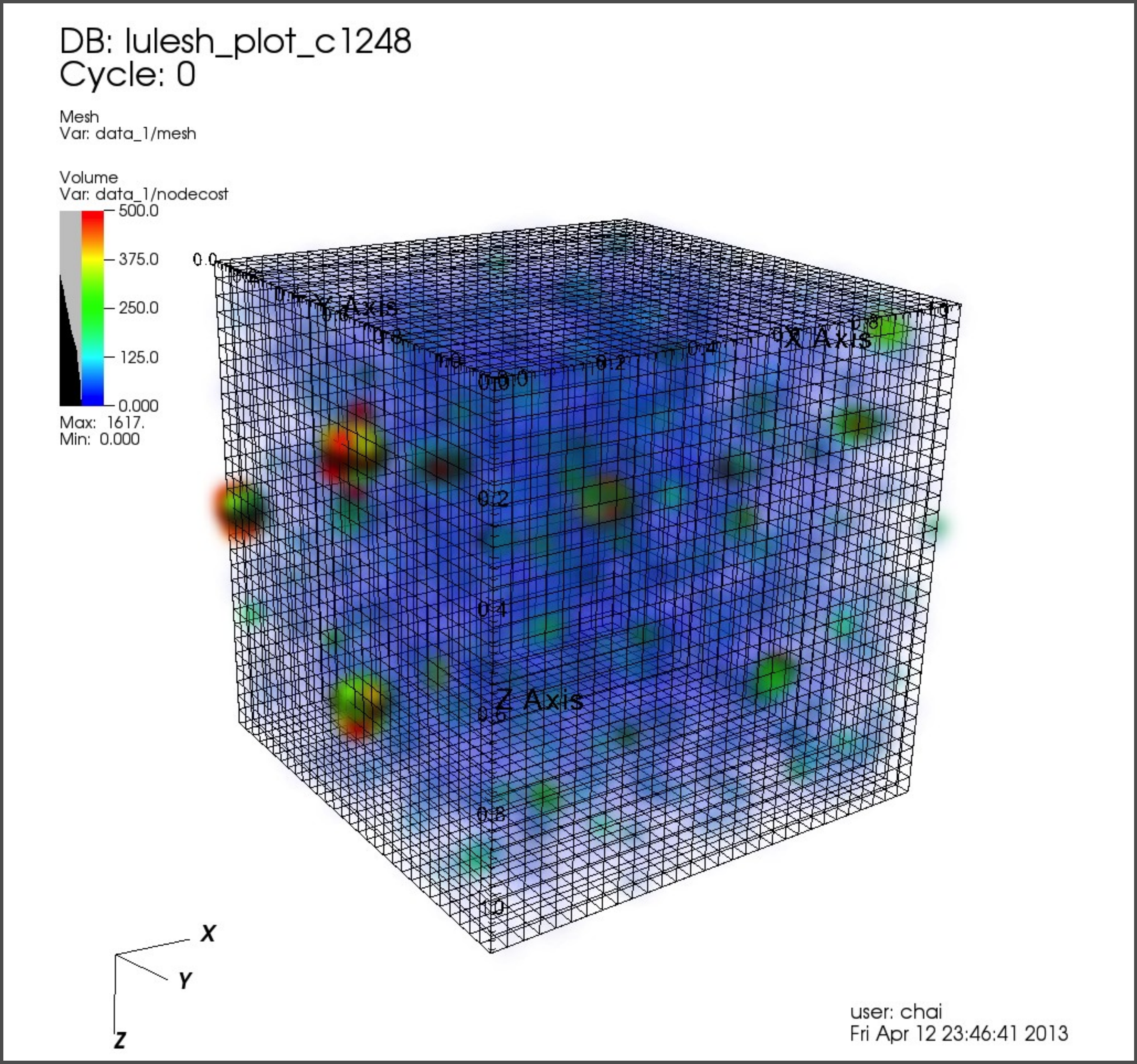
Unoptimized

Optimized

Natural Domain Overlay



Natural Domain Overlay



???

Conclusions

- Semantic Memory Tree Visualizations provide
  - Some higher-level semantics to the data-centric view
  - A general outline to find problems
  - Relative bottlenecks (X is accessed slower than Y)
- Natural Domain Overlay Visualizations provide
  - Fine-grained information about where problems are happening
  - Possibly difficult to interpret, best in conjunction with SMT visualization

Next Steps

- Better way to see many variables
  - L1 %, average cost, total cost, etc
  - Absolute data analysis (currently relative information)
- Correlate data with other metrics
  - Hardware information
  - Access patterns (time-stamping samples)
- Automatic problem detection
  - Process the output to pinpoint problems