Caveat Emptor:

This is a legacy document. All of the machines used for the experiments described in this paper have long since been retired, so the measured values and specifications for modern machines will be entirely different. Furthermore, the landscape has changed considerably since 2003 when this work was done. We are now in a multi-core, multi-processor world where the memory access time now depends not only on the machine's cache hierarchy and the data access patterns of your code, but also on the codes that might be running on nearby cores or peer-processors on the same node. As we move towards heterogeneous computing, with dissimilar computational elements on a single node or chip, the picture may become even fuzzier.

For the time being, however, the basic theory is still valid as are the recommendations. In addition, the code used to expose the hierarchy is still valid (although the results may vary depending on what other processes are running on sister cores or peer processors). Therefore, we have decided to keep the document on our server to provide a reasonable introduction to the cache hierarchy for the non-computer science and engineering folk who may use our supercomputers.

I'm an electrical engineer and feel that you cannot really appreciate knowing what time it is unless you also understand how the watch works! If you fall in that same category, or just want to know a bit more how cache behavior can effect your program, then this document might be just what you need. For what it's worth I use a very similar set of slides for my graduate computer architecture classes.

-- Dr. Gerald R. Morris, April 2011

The Effect of Cache on Memory Access Time

Gerald R. “Jerry” Morris

9 July 2003

Executive Summary

This set of slides empirically illustrates — for a variety of commercial processors — how cache hierarchy affects memory access time. The slides first provide a brief introduction to cache memory and the related topic of storage and access patterns. Then the experiment is presented in three distinct parts, 1) the code used to expose the memory hierarchy, 2) the published technical specifications for the various processors, and 3) plots showing the experimental results compared to the published specifications. The slides conclude with a few recommendations concerning programmer awareness of the cache hierarchy during the development of codes.

Overview

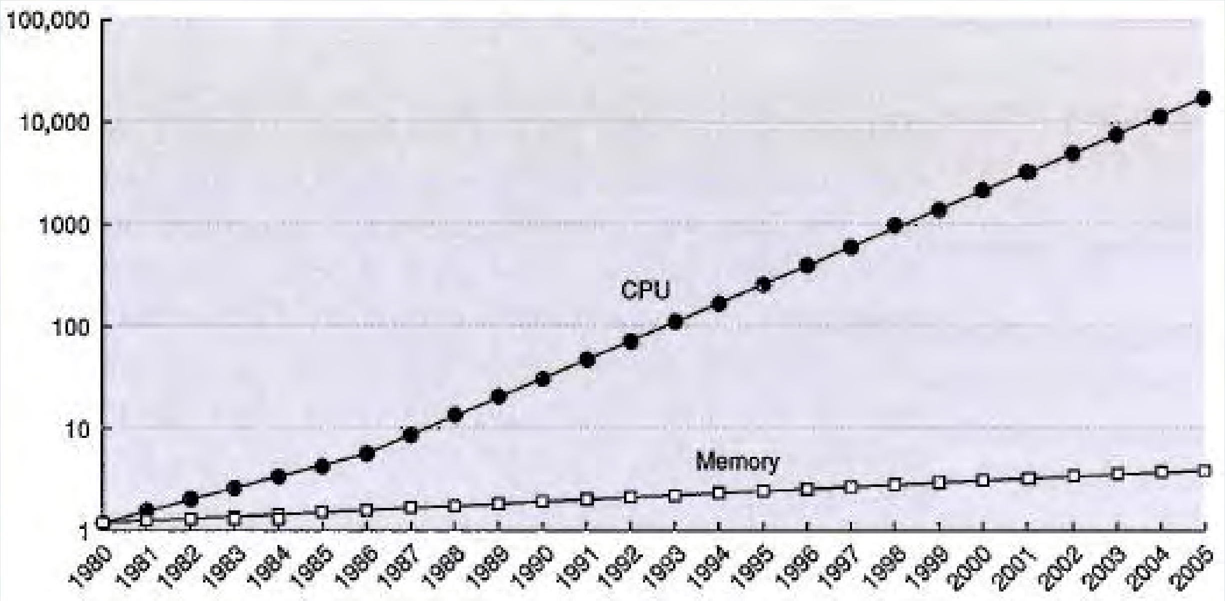
- Cache Memory Primer
- Storage and Access Patterns
- The Experiment
- Recommendations
- Q&A

Cache Memory Primer

- The Problem
- Mitigation via Cache Memory
- Cache Design Considerations

The Problem

- Processor-Memory Performance Gap
  - – EDO DRAM Access Time ≈ 60 ns
  - – SDRAM *Random* Access Time ≈ 40 ns
  - – SRAM Access Time ≈ 5 ns
  - – CPU Clock Cycle Time ≈ 0.3 ns
- It Is Getting Worse!



-- CA:AQA 3rd Ed, p 391, Hennessy & Patterson

Mitigation via Cache Memory

- Motivation: “Locality of Reference”
  - Temporal: Same Word Likely to Be Used Again
  - Spatial: Nearby Words Likely to Be Used
- Intuition: “Avoid Multiple Trips to the Store”
  - Get Entire Folder from Filing Cabinet, Then Go to Desk
  - Grab $^{1}/_{2}$", $^{9}/_{16}$", and $^{5}/_{8}$" Sockets, Then Crawl Under Truck
- Approach: “Memory Hierarchy”
  - Small, Expensive, Fast Memory Cache Close to CPU
  - Large, Inexpensive, Slow Memory Farther Away
  - Load Block (Several Words) of Data into Cache
    - Performance Closer to Fast Memory
    - Cost Closer to Slow Memory
- Reduces Average Memory Access Time

Average Memory Access Time

$$AMAT = Time_{Hit} + Rate_{Miss} \times Time_{Miss}$$

Example

$Time_{Hit}$ = 2 ns (Fetch Data from On-Chip Cache)

$Rate_{Miss}$ = 10%

$Time_{Miss}$ = 60 ns (Fetch Data from EDO DRAM)

$$AMAT = 2\text{ ns} + 0.1 \times 60\text{ ns} = 8\text{ ns}$$

Multilevel Cache

$$\text{AMAT} = \text{Time}_{\text{L1-Hit}} + \text{Rate}_{\text{L1-Miss}} \times \text{Time}_{\text{L1-Miss}}$$

$$\text{Time}_{\text{L1-Miss}} = \text{Time}_{\text{L2-Hit}} + \text{Rate}_{\text{L2-Miss}} \times \text{Time}_{\text{L2-Miss}}$$

Example

$\text{Time}_{\text{L1-Hit}}$ = 0.3 ns (Fetch Data from On-Chip L1 Cache)

$\text{Rate}_{\text{L1-Miss}}$ = 10%

$\text{Time}_{\text{L2-Hit}}$ = 5 ns (Fetch Data from SRAM L2 Cache)

$\text{Rate}_{\text{L2-Miss}}$ = 30%

$\text{Time}_{\text{L2-Miss}}$ = 40 ns (Fetch Data from SDRAM)

$$\text{AMAT} = 0.3\text{ ns} + 0.1 \times (5\text{ ns} + 0.3 \times 40\text{ ns}) = 2\text{ ns}$$

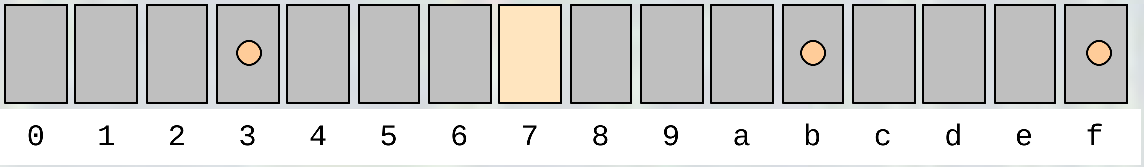
Cache Design Considerations

- Placement: Where Can a Block Be Placed?
- Identification: How Is a Block Found?
- Replacement: Which Block Is Evicted?
- Writes: What Happens on a Write?
- Miscellaneous
  - Split or Unified
  - Number of Levels
  - Sizes
  - Coherency Protocols
  - Etc.

Placement

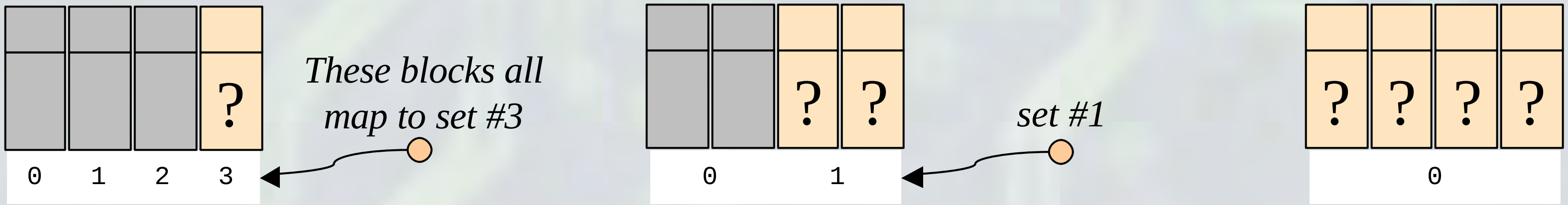
- Where Can a Memory Block Be Placed?
- Cache Set# = Block Address MOD #Sets
  - Direct-Mapped: Block Has Exactly 1 Mapping
  - N-Way Set-Associative: Block Has N Mappings
  - Fully-Associative: Block Can Map Anywhere
- Example: 16 Memory Blocks, Word in Block 7

Memory, 16 Blocks, Need a Word That Is in Block 7



  - Cache Has 4 Blocks, Consider 3 Possible Designs

Direct: 4 Sets of 1 Block | 2-Way Set-Associative: 2 Sets of 2 Blocks | Fully-Associative: 1 Set of 4 Blocks



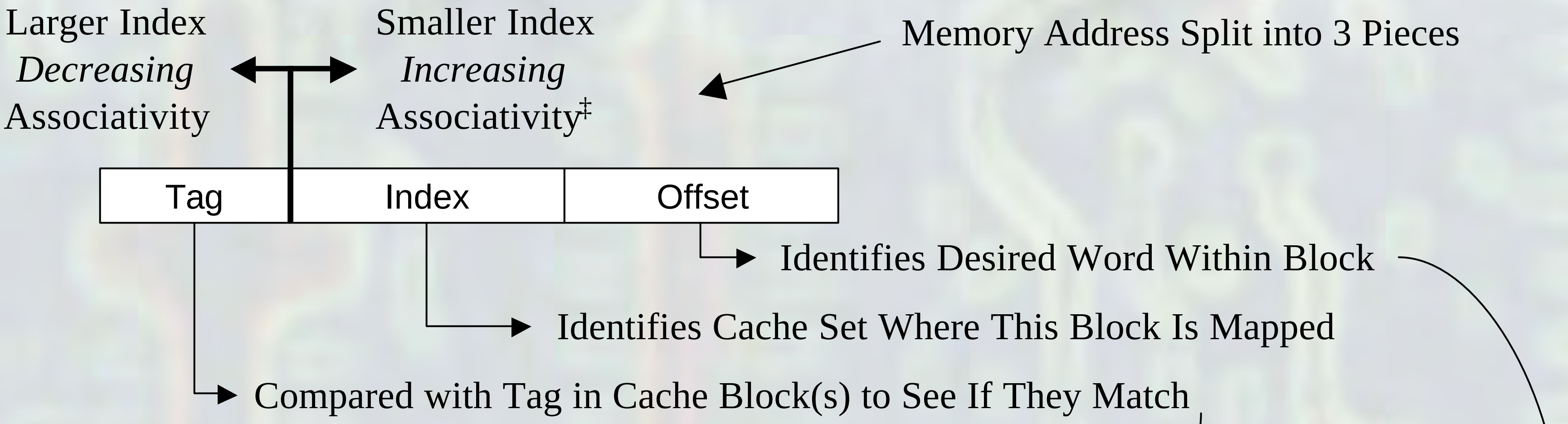
7 % 4 = 3
(one choice)

7 % 2 = 1
(two choices)

7 % 1 = 0
(anywhere)

Identification

- How Is a Block Found?

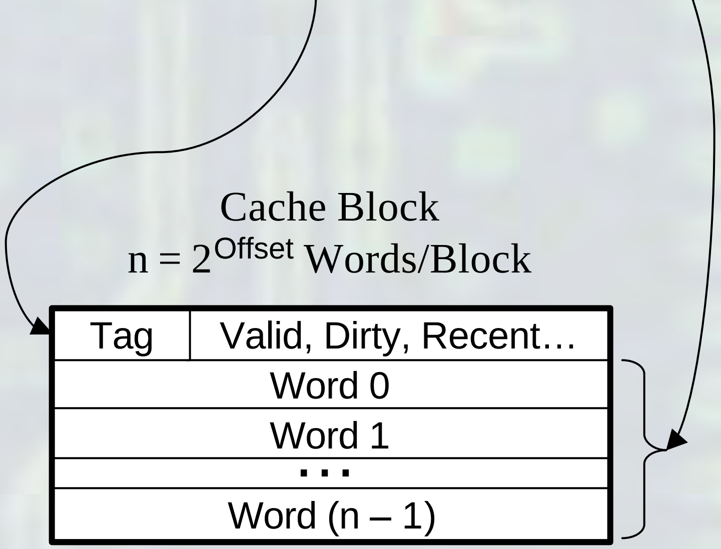


Identifies Desired Word Within Block

Identifies Cache Set Where This Block Is Mapped

Compared with Tag in Cache Block(s) to See If They Match

```
theSetOfBlocks ← Set[Index]
theBlock ←
  select Block from theSetOfBlocks
  where Block.Tag .EQ. Address.Tag
if FOUND then
  theWord ← theBlock[Offset]
else
  miss
fi
```



Cache Block
$n = 2^{\text{Offset}}$ Words/Block

Tag	Valid, Dirty, Recent…
Word 0	
Word 1	
…	
Word (n – 1)	

[‡] For Fully-Associative Cache There Are No Index Bits

Illustration

- Memory: 32 Blocks, 8 Words/Block

  lg (Memory Words) = lg (32 * 8) ⇒ 8 Address Bits[‡]

  lg (Block Words) = lg (8) ⇒ 3 Offset Bits (8 Words/Block)

  Address Bits - Offset Bits = 8 - 3 ⇒ 5 Bits for Index + Tag

- Cache: 8 Blocks
  - Direct-Mapped (8 Sets)

    lg (Sets) = lg (8) ⇒ 3 Index Bits ∴ 2 Tag Bits (3 + 2 = 5)

Tag <2>	Index <3>	Offset <3>

  - 2-Way Set-Associative (8 Blocks ÷ 2 Blocks/Set = 4 Sets)

    lg (Sets) = lg (4) ⇒ 2 Index Bits ∴ 3 Tag Bits (2 + 3 = 5)

Tag <3>	Index <2>	Offset <3>

  - Fully-Associative (1 Set)

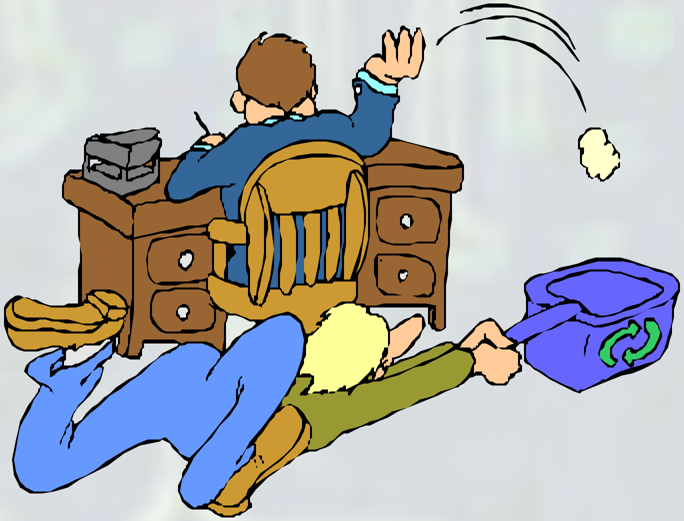
    lg (Sets) = lg (1) ⇒ 0 Index Bits ∴ 5 Tag Bits (0 + 5 = 5)

Tag <5>	Offset <3>

[‡] lg (x) = $\log_2$ (x)

Replacement

- Which Block Is Evicted from Cache?
  - Direct-Mapped Cache
    - No Choice: Must Evict Resident at Direct-Mapped Block
  - Set-Associative Cache
    - Least Recently Used (LRU)
      - Costly, Usually Approximated
    - Random, or Random But Not Last
      - Nearly as Good as LRU for Large Caches

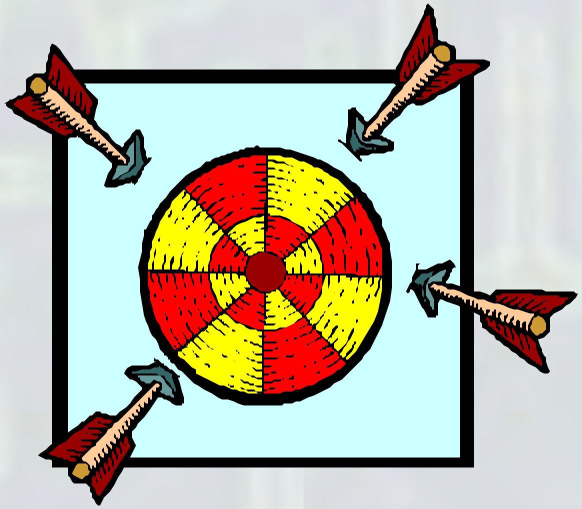


Writes

- What Happens on a Write?
  - Write-Back
    - Data Written Only to Cache
    - Dirty Bit Marks Blocks to Be Written Back upon Eviction
    - Writes Occur at Cache Memory Speeds
    - Multiple Writes to Block Only Need One Write to Lower Level
  - Write-Through
    - Data Written to Cache and Lower Level
    - Lower Level Matches Cache (Coherent)
    - Easier to Implement
    - Read Misses Never Require a Write

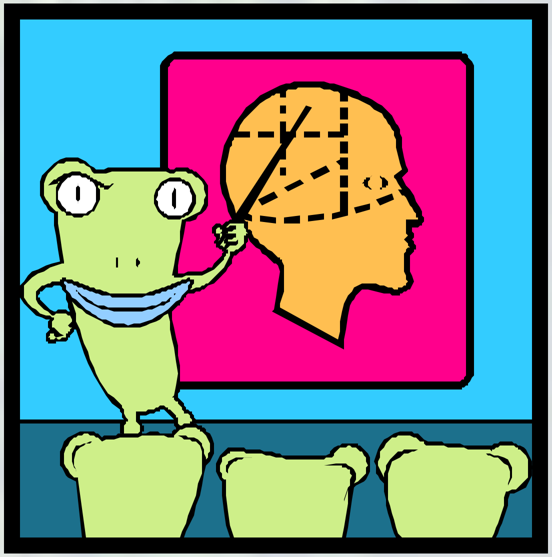
Write Misses

- What Happens on a Write Miss?
  - Write-Allocate (Fetch on Write)
    - Often Employed by Write-Back Caches
    - Subsequent Writes to Block Occur at Cache Level
  - No Write-Allocate (Write Around)
    - Often Employed by Write-Through Caches
    - Subsequent Writes to Block Must Write-Through Anyway



Storage and Access Patterns

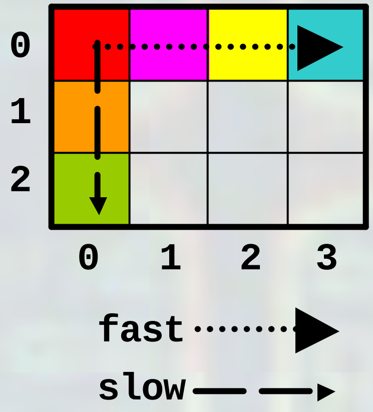
- Storage Pattern
  - Logical Data Structure
  - Mapping to Memory Locations
- Access Pattern
  - Given Storage Pattern
  - Optimal Way to Access Data
- Row-Major vs. Column-Major



Row-Major vs. Column-Major

"Row-Major Order"

Logical View



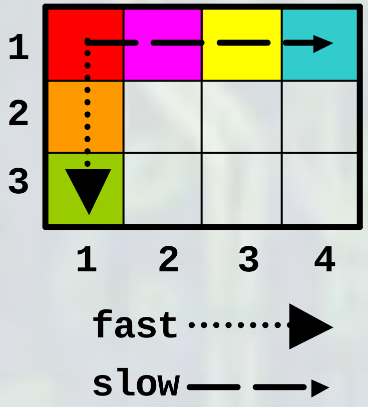
Storage Pattern
a[0][0]
a[0][1]
a[0][2]
a[0][3]
a[1][0]
a[1][1]
a[1][2]
a[1][3]
a[2][0]
a[2][1]
a[2][2]
a[2][3]

Optimal Access Pattern

```
int a[3][4];

/* contiguous
   memory access */

for(i=0;i<3;i++){
  for(j=0;j<4;j++){
    a(i,j)=0;
  }
}
```

"Column-Major Order"

Logical View



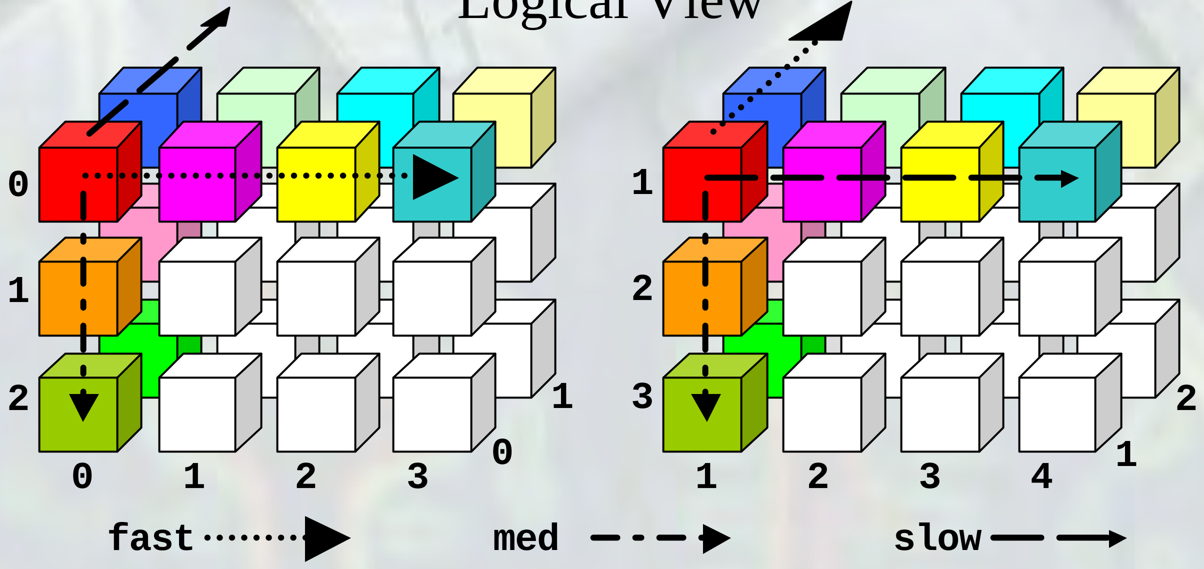
Storage Pattern
A(1,1)
A(2,1)
A(3,1)
A(1,2)
A(2,2)
A(3,2)
A(1,3)
A(2,3)
A(3,3)
A(1,4)
A(2,4)
A(3,4)

Optimal Access Pattern

```
      INTEGER A(3,4)

! CONTIGUOUS
! MEMORY ACCESS

      DO J=1,4
        DO I=1,3
          A(I,J)=0
        END DO
      END DO
```

3-D Array

Logical View



Optimal Access Pattern

```
int a[2][3][4];

/* contiguous
 * memory access */

for(i=0;i<2;i++){
   for(j=0;j<3;j++){
      for(k=0;k<4;k++){
         a[i][j][k]=0;
      }
   }
}
```

```
INTEGER A(2,3,4)

! CONTIGUOUS
! MEMORY ACCESS

DO K=1,4
   DO J=1,3
      DO I=1,2
         A(I,J,K)=0
      END DO
   END DO
END DO
```

Storage Pattern

a[0][0][0]	A(1,1,1)
a[0][0][1]	A(2,1,1)
a[0][0][2]	A(1,2,1)
a[0][0][3]	A(2,2,1)
a[0][1][0]	A(1,3,1)
a[0][1][1]	A(2,3,1)
a[0][1][2]	A(1,1,2)
a[0][1][3]	A(2,1,2)
a[0][2][0]	A(1,2,2)
a[0][2][1]	A(2,2,2)
a[0][2][2]	A(1,3,2)
a[0][2][3]	A(2,3,2)
a[1][0][0]	A(1,1,3)
a[1][0][1]	A(2,1,3)
a[1][0][2]	A(1,2,3)
a[1][0][3]	A(2,2,3)
a[1][1][0]	A(1,3,3)
a[1][1][1]	A(2,3,3)
a[1][1][2]	A(1,1,4)
a[1][1][3]	A(2,1,4)
a[1][2][0]	A(1,2,4)
a[1][2][1]	A(2,2,4)
a[1][2][2]	A(1,3,4)
a[1][2][3]	A(2,3,4)

N-D Array

Optimal Access Pattern

```
        →
int a[d_1][d_2]...[d_n];

/* contiguous memory access */

/* write loop indices
 * in same order as
 * array declaration */

for(i_1 = 0; i_1 < d_1; i_1++){
  for(i_2 = 0; i_2 < d_2; i_2++){

      ...

        for(i_n = 0; i_n < d_n; i_n++){
          a[i_1][i_2]...[i_n] = 0;
        }

      ...

    }
  }
}
```

```
             ←
INTEGER A(D_1,D_2,...,D_n)

! CONTIGUOUS MEMORY ACCESS

! WRITE LOOP INDICES
! IN REVERSE ORDER FROM
! ARRAY DECLARATION

DO I_n = 1, D_n

   ...


      DO I_2 = 1, D_2
        DO I_1 = 1, D_1
          A(I_1,I_2,...,I_n) = 0
        END DO
      END DO

   ...

END DO
```

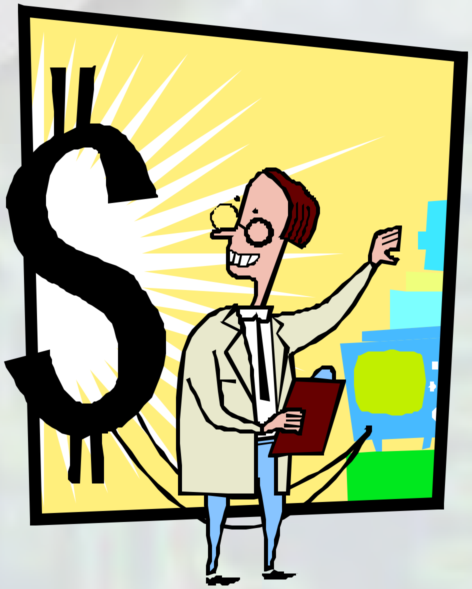
4-D Array Example

```c
#include <stdio.h>
int main() {

  int a4d[2][3][4][5];  /* our 4-D array and */
  int *a1d =            /* equivalent 1-D array */
    (int *)a4d;         /* occupy same memory */

  int i,i1,i2,i3,i4;
  char *format = "a4d[%d][%d][%d][%d] = %3d\n";

  /* seq of numbers in contiguous locs */
  for (i = 0; i < 120; i++) {
    a1d[i] = i;
  }

  /* same order as indices: contiguous access */
  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 3; i2++) {
      for (i3 = 0; i3 < 4; i3++) {
        for (i4 = 0; i4 < 5; i4++) {
          printf (format,
            i1,i2,i3,i4,a4d[i1][i2][i3][i4]);
        }
      }
    }
  }

  /* reverse indices: non-contiguous access */
  for (i4 = 0; i4 < 5; i4++) {
    for (i3 = 0; i3 < 4; i3++) {
      for (i2 = 0; i2 < 3; i2++) {
        for (i1 = 0; i1 < 2; i1++) {
          printf (format,
            i1,i2,i3,i4,a4d[i1][i2][i3][i4]);
        }
      }
    }
  }
} /* main */
```

```fortran
      PROGRAM MAIN
      IMPLICIT NONE

      INTEGER A4D(2,3,4,5)  ! OUR 4-D ARRAY AND
      INTEGER A1D(120)      ! EQUIVALENT 1-D ARRAY
      EQUIVALENCE (A4D,A1D) ! OCCUPY SAME MEMORY

      INTEGER I,I1,I2,I3,I4
10    FORMAT('A4D(',I1,',',I1,',',I1,',',I1,')=',I3)

      ! SEQ OF NUMBERS IN CONTIGUOUS LOCS
      DO I = 1,120
        A1D(I) = I
      END DO

      ! SAME ORDER AS INDICES: NON-CONTIGUOUS ACCESS
      DO I1 = 1,2
        DO I2 = 1,3
          DO I3 = 1,4
            DO I4 = 1,5
              PRINT 10,
     /          I1,I2,I3,I4,A4D(I1,I2,I3,I4)
            END DO
          END DO
        END DO
      END DO

      ! REVERSE INDICES: CONTIGUOUS ACCESS
      DO I4 = 1,5
        DO I3 = 1,4
          DO I2 = 1,3
            DO I1 = 1,2
              PRINT 10,
     /          I1,I2,I3,I4,A4D(I1,I2,I3,I4)
            END DO
          END DO
        END DO
      END DO
      END ! MAIN
```

The Experiment

- Code to Expose Memory Hierarchy
  - For Multiple Array Sizes and Strides
    - Measure Average Read+Write Access Time
- Executed on
  - HPCs as Single CPU Job
  - Desktops
- Results Plotted
  - Access Time Changes at or Near Cache Limits[‡]
  - Compared to Published CPU Specs



[‡] Can Be Abrupt or Gradual Depending on Cache Design

Code to Expose Hierarchy

```
/* cash.c
 * % cc cash.c -lm -o cash
 * Gerald R. "Jerry" Morris, ERDC MSRC, 16 May 2003
 * Based on Hennessy & Patterson, CA:AQA, 2nd Ed, page 477
 * Evaluates the behavior of memory system:
 *
 *    "The key is having accurate timing and then having the
 *     program stride through memory to invoke different
 *     levels of the hierarchy."  -- Hennessy & Patterson
 *
 * The essence of this program is that it measures time to read and write
 * memory at different cache sizes and strides.  The two outermost loops
 * vary cache size and stride.  The first inner while loop does multiple
 * read + write accesses of memory locations and accumulates elapsed time.
 * The code uses a second dummy while loop that does not access memory to
 * subtract the loop overhead.  The code then divides accumulated time by the
 * number of accesses to obtain average r+w time for given size and stride.
 * To avoid big numbers, size [B] & stride [B] reported in lg (log base 2).
 */
#include <stdio.h>
#include <sys/times.h>
#include <sys/types.h>
#include <time.h>
#include <math.h>
#include <string.h>
```

...Code...

```
/* smallest and largest cache (can vary for different boxes)*/
#define K (1024)
#define M (K*K)
#define CACHE_MIN (K)
#define CACHE_MAX (32*M)

/* time sample and clock tick */
#define SAMPLE (10)
#ifndef CLK_TCK
#define CLK_TCK (60)
#endif

double get_seconds() { /* to read time */
    struct tms rusage;
    times(&rusage);
    return ((double)(rusage.tms_utime) / CLK_TCK);
}

double lg(double x) { /* log base 2 */
   return (log(x)/log(2.0));
}

int cache[CACHE_MAX];  /* stride through various parts of this array */
```

...Code...

```
int main() {
    int register csize;    /* outer loop "cache" size */
    int register stride;   /* inner loop stride through the cache */
    int register lim;      /* upper limit cache reference for each stride */
    int register i;        /* loop indices */
    int register j;
    int register dummy;    /* for dummy loop equivalent of cache access */
    int register steps;    /* number of while loop iterations */
    int register dsteps;   /* ditto for dummy loop */
    double sec0;           /* start time */
    double sec;            /* time accumulator */

    /* want a comma separated variable file, so first dump the headers */
    printf
      ("\"lg(size [B]/[B])\",\"lg(stride [B]/[B])\",\"r+w time [ns]\"\n");

    /* vary "cache" from min to max in powers of 2 */
    for (csize = CACHE_MIN; csize <= CACHE_MAX; csize *= 2) {

        /* vary stride from 1 thru csize/2 in powers of 2 */
        for (stride = 1; stride < csize ; stride *= 2) {

            sec = 0; /* initialize time accumulator */
            lim = csize - stride + 1; /* upper cache ref limit this stride */
```

...Code...

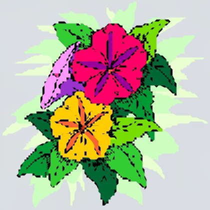
```
memset (cache, 0, sizeof(cache));  /* initialize the array */
steps = 0; /* total number of while iterations */

while (sec < 1.0) { /* repeat till we've run for 1 second */

    sec0 = get_seconds(); /* start timer */

    /* outer for loop allows eviction from cache (if needed) */
    /* amortizes the eviction cost (if any) and */
    /* gives a better picture of average memory access time */
    for (i = SAMPLE * stride; i != 0; i--) {

        /* inner loop does the actual read and write of memory */
        for (j = 0; j < lim; j += stride) {
            cache[j]++;  /* r+w one location in memory */
        } /* for j */

    } /* for i */

    steps++; /* number of while loop iterations */
    sec += (get_seconds() - sec0); /* add to time so far */

} /* while sec */
```

...Code

```
            /* subtract loop overhead, run dummy (non-memory) ver of loop */
            dummy = 0;  /* initialize dummy cache (a register variable) */
            dsteps = 0;  /* want same number of steps as above */

            while (dsteps < steps) {  /* repeat for same number of steps */
                sec0 = get_seconds(); /* start timer */
                for (i = SAMPLE * stride; i != 0; i--) {
                    for (j = 0; j < lim; j += stride) {
                        dummy++; /* r+w non-memory loc(register variable) */
                    }
                }
                dsteps++; /* record iterations */
                sec -= (get_seconds() - sec0); /* subtract loop OH time */
            } /* while dsteps */

            /* for each size and stride, print time per access [ns] */
            /* NOTE: # accesses = steps*SAMPLE*stride*((lim-1)/stride+1)) */
            printf("\"%6.0f\",\"%6.0f\",\"%14.0f\"\n",
              lg(csize * sizeof(int)),
              lg(stride * sizeof(int)),
              sec * 1e9 / (steps*SAMPLE*stride * ((lim - 1) / stride + 1)));

        } /* for stride */
    } /* for csize */

} /* main */
```

CPU Tech Specs

- HPCs
  - HP AlphaServer SC45: 1000 MHz Alpha EV68
  - HP AlphaServer SC40: 833 MHz Alpha EV68
  - SGI Origin 3900: 700 MHz MIPS R16000
  - SGI Origin 3800: 400 MHz MIPS R12000
  - Cray T3E LC-1350/1200: 675/600 MHz Alpha 21164
  - IBM SP-Power4: 1300 MHz POWER4
  - IBM SP-Power3: 375 MHz POWER3
- Desktops
  - Dell Precision 340: 2000 MHz Pentium 4 ♂
  - Dell OptiPlex GX200: 933 MHz Pentium III
  - Apple PowerBook G4: 800 MHz PowerPC G4 ♀



♂ PCs ♀ Macintosh

AlphaServer SC45

Hyper Links

SC45 Chart 1

SC45 Chart 2

- 1000 MHz Alpha 21264/EV68
- L1: 64 KB On-Chip Data Cache†
  - 2-Way Set-Associative, 64-Byte Blocks
  - Write-Back, Write-Allocate
- L2: 8 MB Off-Chip Unified Cache‡
  - 1 MB to 16 MB (SC45 Has 8 MB), 64-Byte Blocks
  - Configurable Cache Coherency Protocols
- References
  - EV68 Hardware Reference Manual
    - ftp://ftp.compaq.com/pub/products/alphaCPUdocs/index.txt
  - SC45 Facts and Figures
    - http://www.hp.com/techservers/systems/sys_sc45_features.html

† EV68 Has 64 KB On-Chip Instruction Cache, 2-Way Set-Predict, 64-Byte Blocks

‡ EV68 Has On-Chip Duplicate Tag Array to Maintain L2 Cache Coherency

AlphaServer SC40

Hyper Links

SC40 Chart 1

SC40 Chart 2

- 833 MHz Alpha 21264/EV68
- L1: 64 KB On-Chip Data Cache†
  - 2-Way Set-Associative, 64-Byte Blocks
  - Write-Back, Write-Allocate
- L2: 8 MB Off-Chip Unified Cache‡
  - 1 MB to 16 MB (SC40 Has 8 MB), 64-Byte Blocks
  - Configurable Cache Coherency Protocols
- References
  - EV68 Hardware Reference Manual
    - ftp://ftp.compaq.com/pub/products/alphaCPUdocs/index.txt
  - SC40 Facts and Figures
    - http://www.hp.com/techservers/systems/sys_sc40_features.html

† EV68 Has 64 KB On-Chip Instruction Cache, 2-Way Set-Predict, 64-Byte Blocks

‡ EV68 Has On-Chip Duplicate Tag Array to Maintain L2 Cache Coherency

Origin 3900

Hyper Links

3900 Chart 1

3900 Chart 2

3900 Chart 3

- 700 MHz MIPS R16000
- L1: 32 KB On-Chip Data Cache†
- L2: 8 MB Unified Cache
- References
  - *hinv*

```
512 700 MHZ IP35 Processors
CPU: MIPS R16000 Processor Chip Revision: 2.1
FPU: MIPS R16010 Floating Point Chip Revision: 2.1
Main memory size: 524288 Mbytes
Instruction cache size: 32 Kbytes
Data cache size: 32 Kbytes
Secondary unified instruction/data cache size: 8 Mbytes
```

† R16000 Has 32 KB On-Chip Instruction Cache

Origin 3800

Hyper Links

3800 Chart 1

3800 Chart 2

3800 Chart 3

- 400 MHz MIPS R12000
- L1: 32 KB On-Chip Data Cache†
- L2: 8 MB Unified Cache
- References
  - *hinv*

```
512 400 MHZ IP35 Processors
CPU: MIPS R12000 Processor Chip Revision: 3.5
FPU: MIPS R12010 Floating Point Chip Revision: 3.5
Main memory size: 524288 Mbytes
Instruction cache size: 32 Kbytes
Data cache size: 32 Kbytes
Secondary unified instruction/data cache size: 8 Mbytes
```

† R12000 Has 32 KB On-Chip Instruction Cache

T3E LC-1350/1200

Hyper Links

T3E Chart 1

T3E Chart 2

T3E Chart 3

- 675/600 MHz Alpha 21164
- L1: 8 KB On-Chip Data Cache†
  - Direct-Mapped, 32-Byte Blocks
  - Write-Through, No Write-Allocate
- L2: 96 KB On-Chip Unified Cache‡
  - 3-Way Set-Associative, 32-Byte or 64-Byte Blocks
  - Write-Back, Write-Allocate
- References
  - 21164 Hardware Reference Manual
    - ftp://ftp.compaq.com/pub/products/alphaCPUdocs/archives/index.txt
  - T3E User Survey
    - http://www.nersc.gov/magazine/link_archive/jun97/t3eresponse.html

† 21164 Has 8 KB On-Chip Instruction Cache, Direct-Mapped, 32-Byte Blocks

‡ 21164 Has Optional Off-Chip L3, Not Used on T3E (Uses Streams Instead)

SP-Power4

Hyper Links

[Power4 Chart 1]

[Power4 Chart 2]

[Power4 Chart 3]

- 1300 MHz PowerPC 4
- L1: 32 KB On-Chip Data Cache[†]
  - 2-Way Set-Associative, 128-Byte Blocks
- L2: 1.4 MB Unified Cache
  - 4-Way Set-Associative
- L3: 128 MB Unified Cache
- References
  - NAVO Web Site
    - http://www.navo.hpc.mil/usersupport/MARC/overview.html
  - */site/bin/sysinfo*

[†] POWER4 Has 64 KB On-Chip Instruction Cache, Direct-Mapped, 128-Byte Blocks

SP-Power3

Hyper Links

Power3 Chart 1

Power3 Chart 2

- 375 MHz PowerPC 630 (Power3)
- L1: 64 KB On-Chip Data Cache†
  - 128-Way Set-Associative, 128-Byte Blocks
- L2: 8 MB Unified Cache
  - 4-Way Set-Associative
- References
  - */site/bin/sysinfo*

† POWER3 Has 32 KB On-Chip Instruction Cache, 128-Way Set-Associative, 128-Byte Blocks

Precision 340

Hyper Links

340 Chart 1

340 Chart 2

- 2000 MHz Pentium 4
- L1: 8 KB On-Chip Data Cache[†]
  - 4-Way Set-Associative, 64-Byte Blocks
  - Write-Through
- L2: 512 KB On-Chip Unified Cache
  - 8-Way Set-Associative, 64-Byte Blocks
- References
  - **S**ystem **AN**alyzer, **D**iagnostic and **R**eporting **A**ssistant
    - SiSoftware Sandra
    - http://www.sisoftware.net/?location=pinformation
  - GEEK.com
    - http://www.geek.com/procspec/intel/northwood.htm

[†] P4 Has 12 K µOp On-Chip Instruction Cache, 8-Way Set-Associative, 64-Byte Blocks

OptiPlex GX200

Hyper Links

GX200 Chart 1

GX200 Chart 2

- 933 MHz Pentium III
- L1: 16 KB On-Chip Data Cache†
  - 4-Way Set-Associative, 32-Byte Blocks
  - Write-Through
- L2: 256 KB On-Chip Unified Cache
  - 8-Way Set-Associative, 32-Byte Blocks
- References
  - **S**ystem **AN**alyzer, **D**iagnostic and **R**eporting **A**ssistant
    - SiSoftware Sandra
    - http://www.sisoftware.net/?location=pinformation
  - GEEK.com
    - http://www.geek.com/procspec/intel/pentium3consumer.htm

† PIII Has 16 KB On-Chip Instruction Cache, 4-Way Set-Associative, 32-Byte Blocks

PowerBook G4

Hyper Links

G4 Chart 1

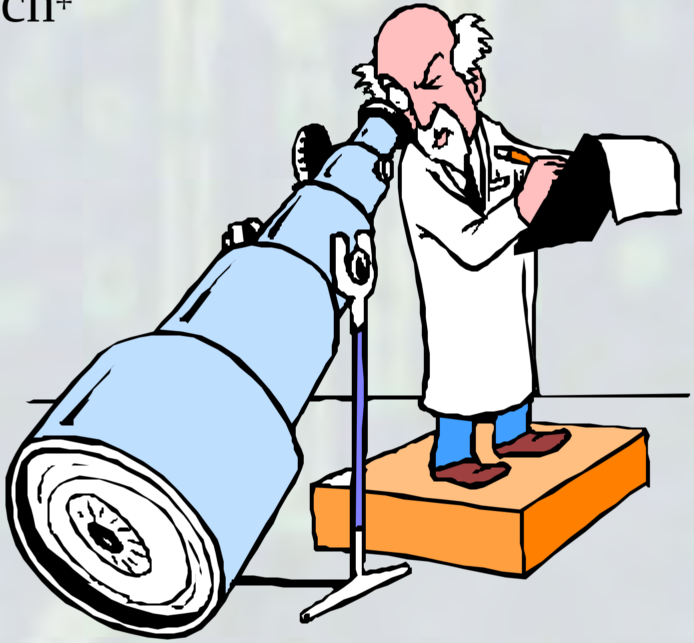
G4 Chart 2

- 800 MHz PowerPC G4
- L1: 32 KB On-Chip Data Cache†
- L2: 256 KB On-Chip Unified Cache
- L3: 1 MB Off-Chip Unified Cache
- References
  - PowerBook G4 Technical Specs
    - http://www.apple.com/powerbook/specs.html
  - GEEK.com
    - http://www.geek.com/procspec/apple/g4.htm

† G4 Has 32 KB On-Chip Instruction Cache

Results vs. CPU Tech Specs

- Results Plotted as 3-D Column Charts
  - X-Axis Shows Stride Size
  - Y-Axis Shows Data Structure Size
  - Z-Axis Shows Memory Access Read+Write Time
- Highlight Cache Breakpoints
  - Experimental and CPU Specs Match‡



‡ Mostly!

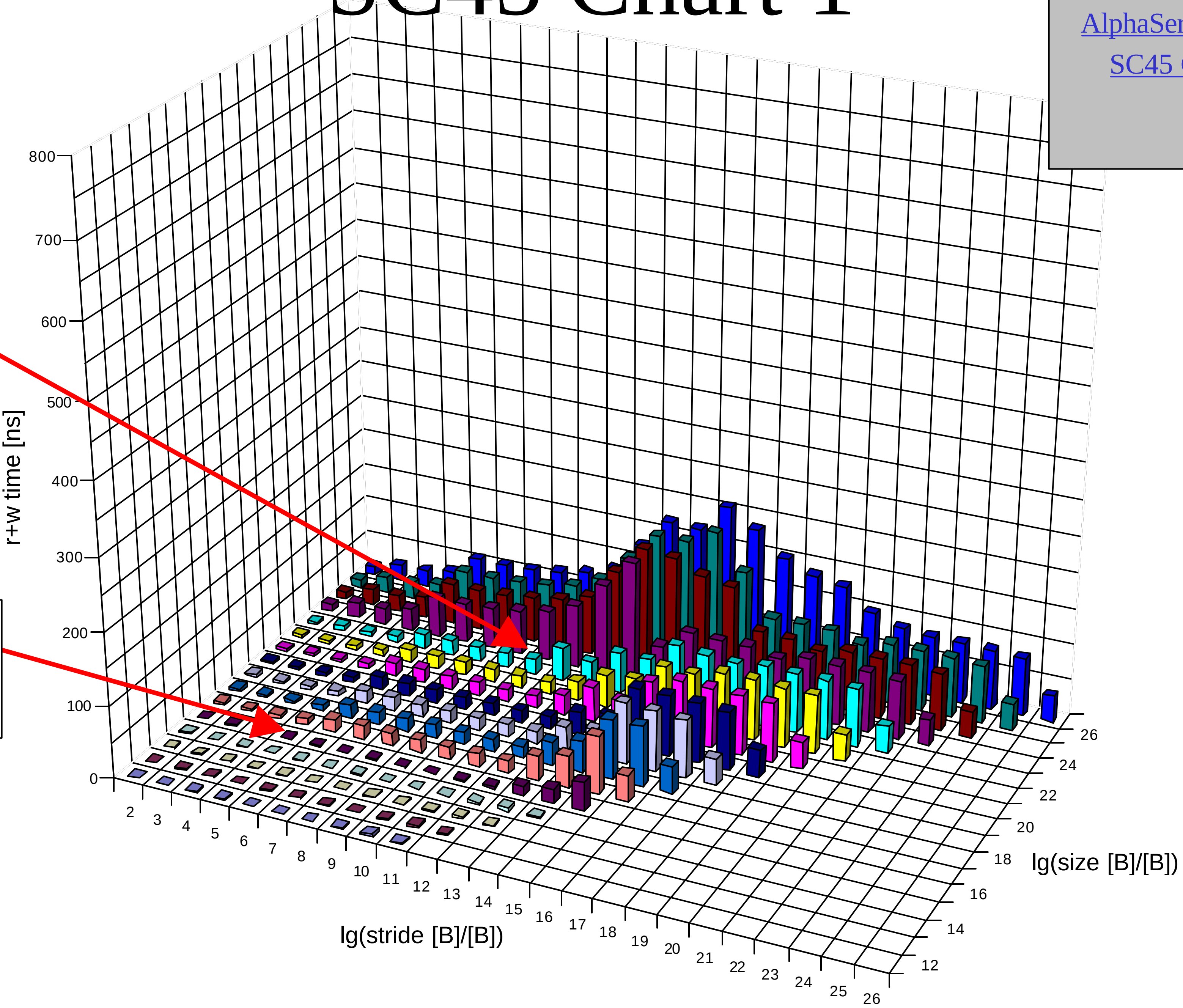
AlphaServer SC45 (emerald)

SC45 Chart 1

Hyper Links

AlphaServer SC45

SC45 Chart 2



L2
8 MB
23 Bits

L1
64 KB
16 Bits

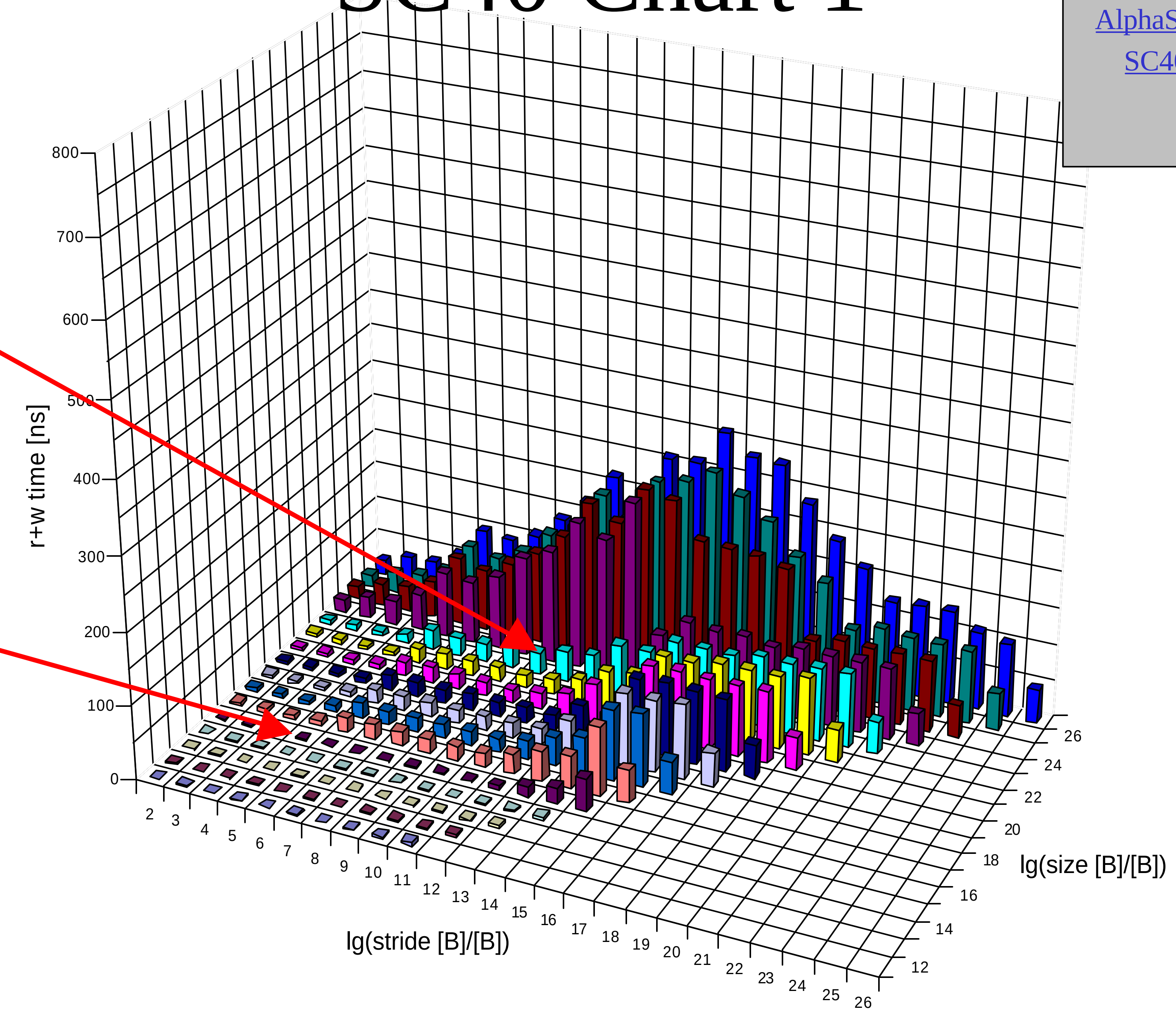
AlphaServer SC40 (opal)

SC40 Chart 1

Hyper Links

AlphaServer SC40

SC40 Chart 2



L2
8 MB
23 Bits

L1
64 KB
16 Bits

Origin 3900 (sand)

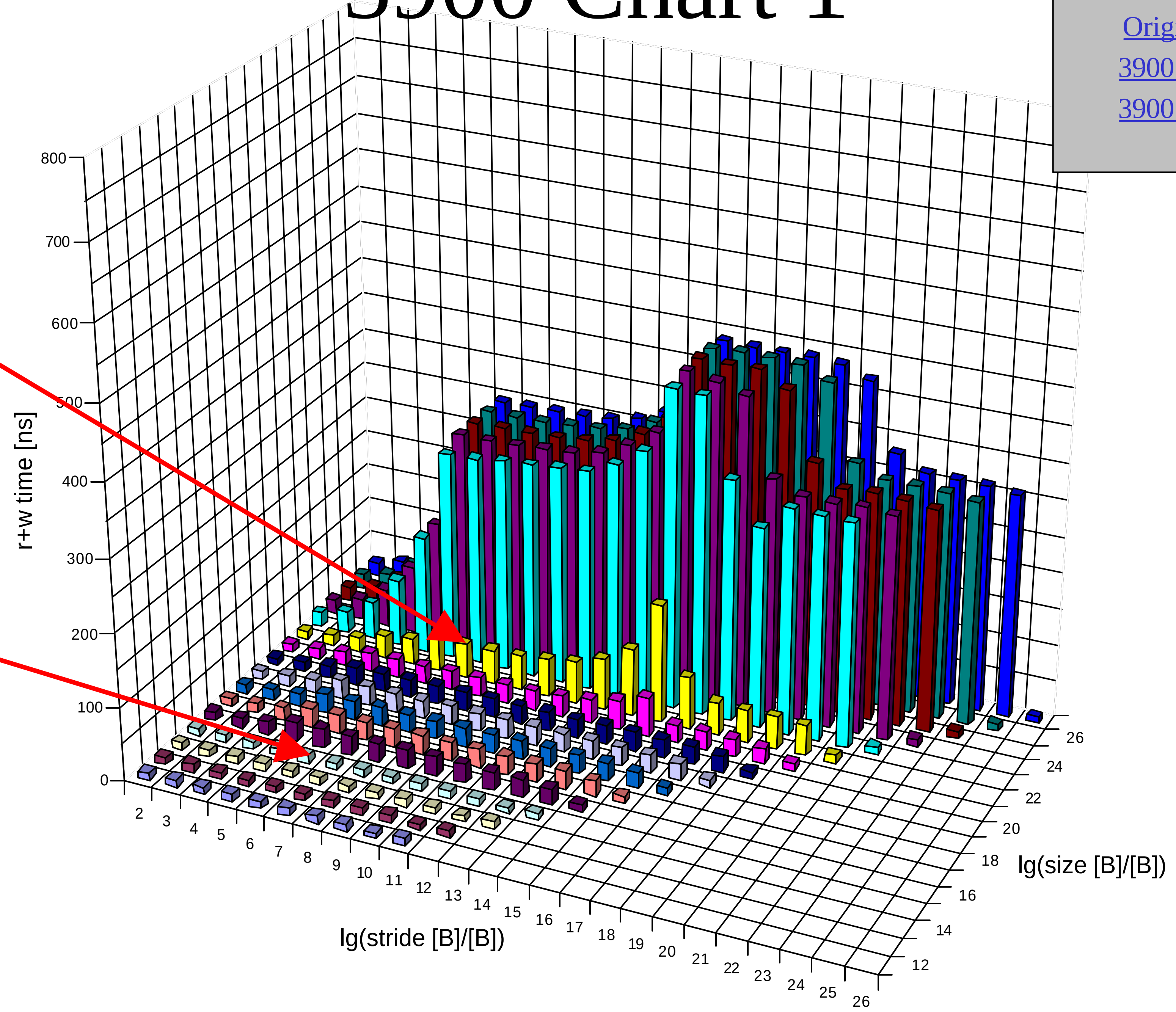
3900 Chart 1

Hyper Links

Origin 3900

3900 Chart 2

3900 Chart 3



L2?
4 MB?
22 Bits?

L1
32 KB
15 Bits

Origin 3800 (sard)

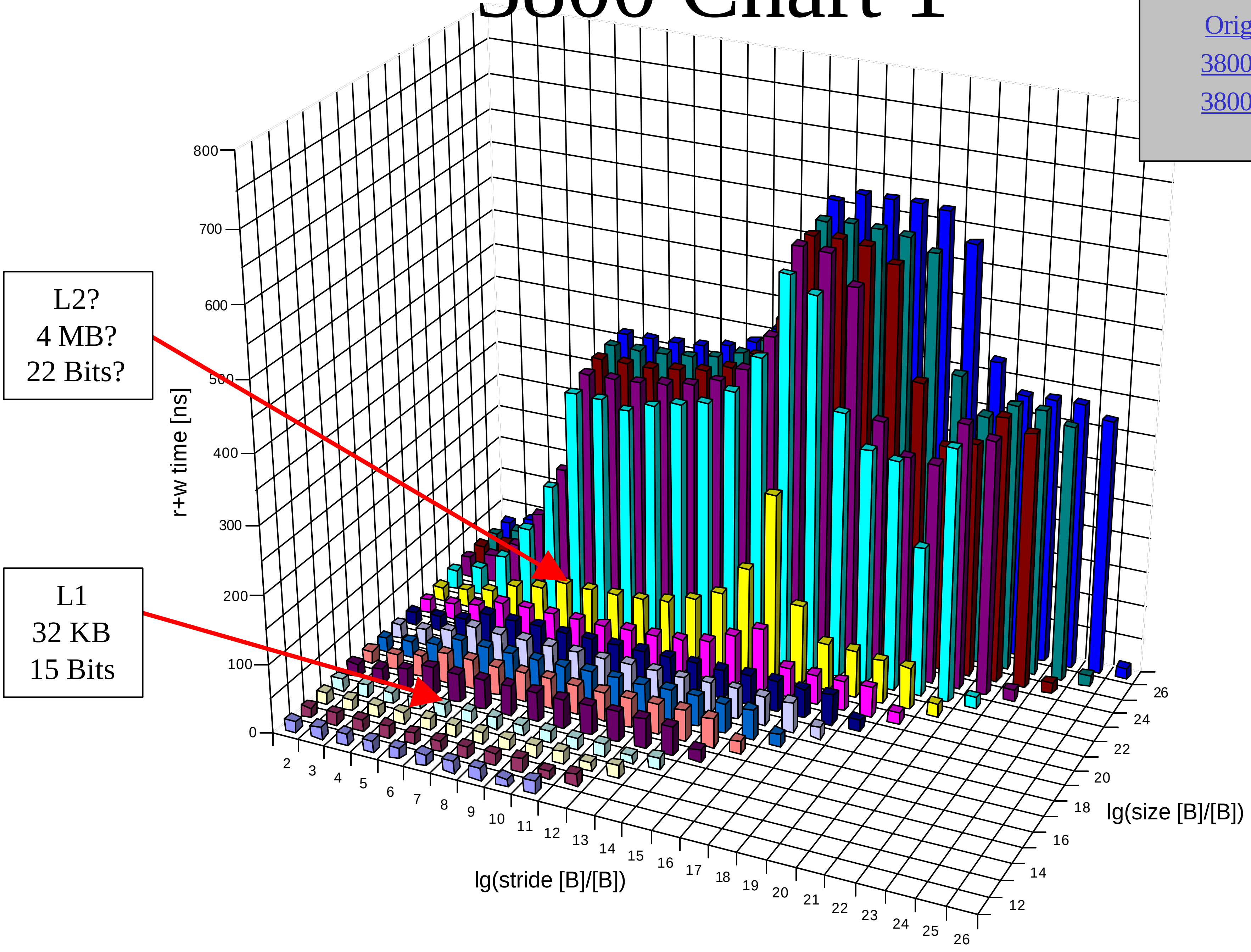
3800 Chart 1

Hyper Links

Origin 3800

3800 Chart 2

3800 Chart 3



L2?
4 MB?
22 Bits?

L1
32 KB
15 Bits

T3E LC-1350/1200 (jim)

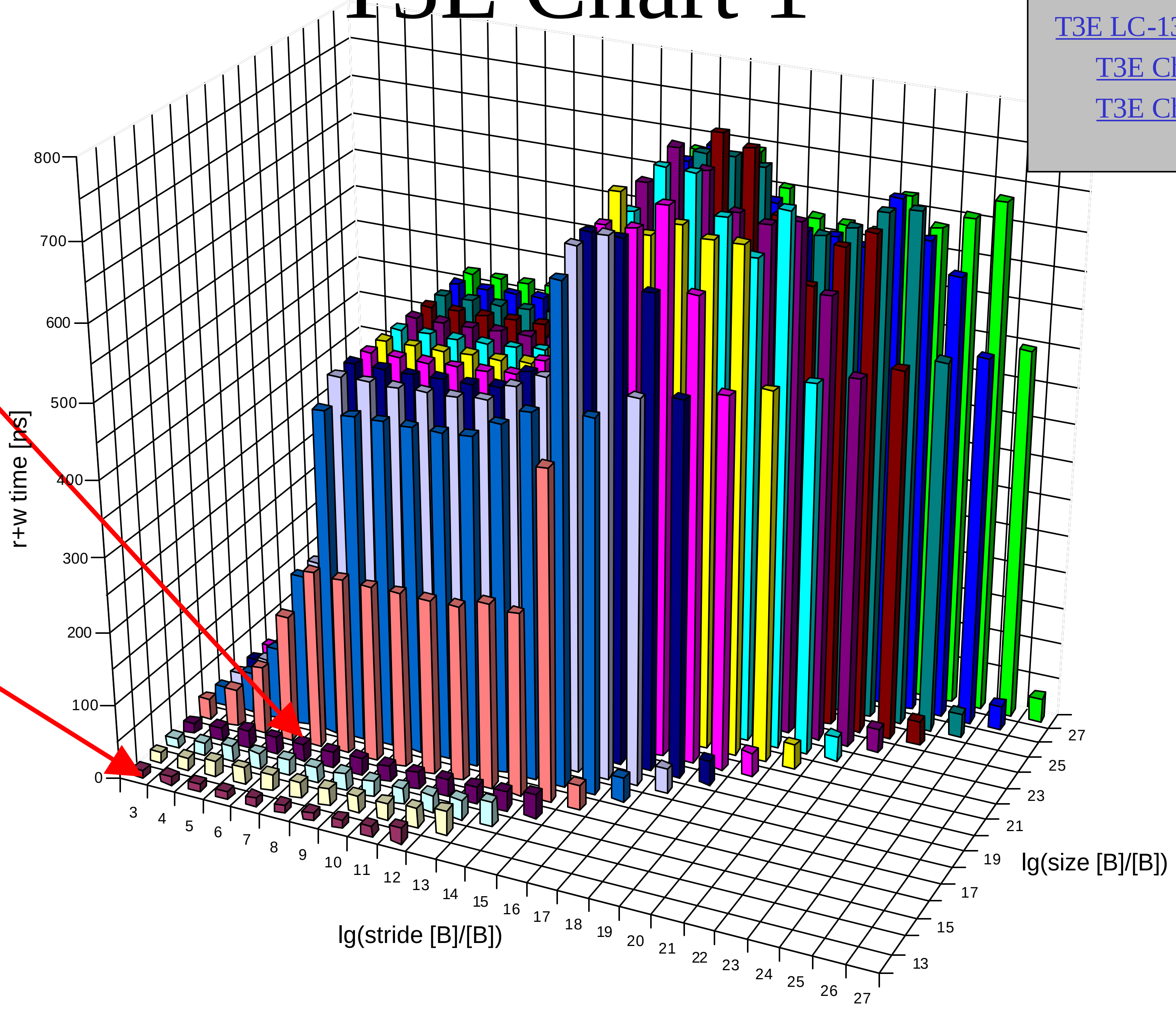
T3E Chart 1

Hyper Links

T3E LC-1350/1200

T3E Chart 2

T3E Chart 3



L2?
64 KB?
16 Bits?

L1
8 KB
13 Bits

SP-Power4 (marcellus)

Power4 Chart 1

Hyper Links

SP-Power4

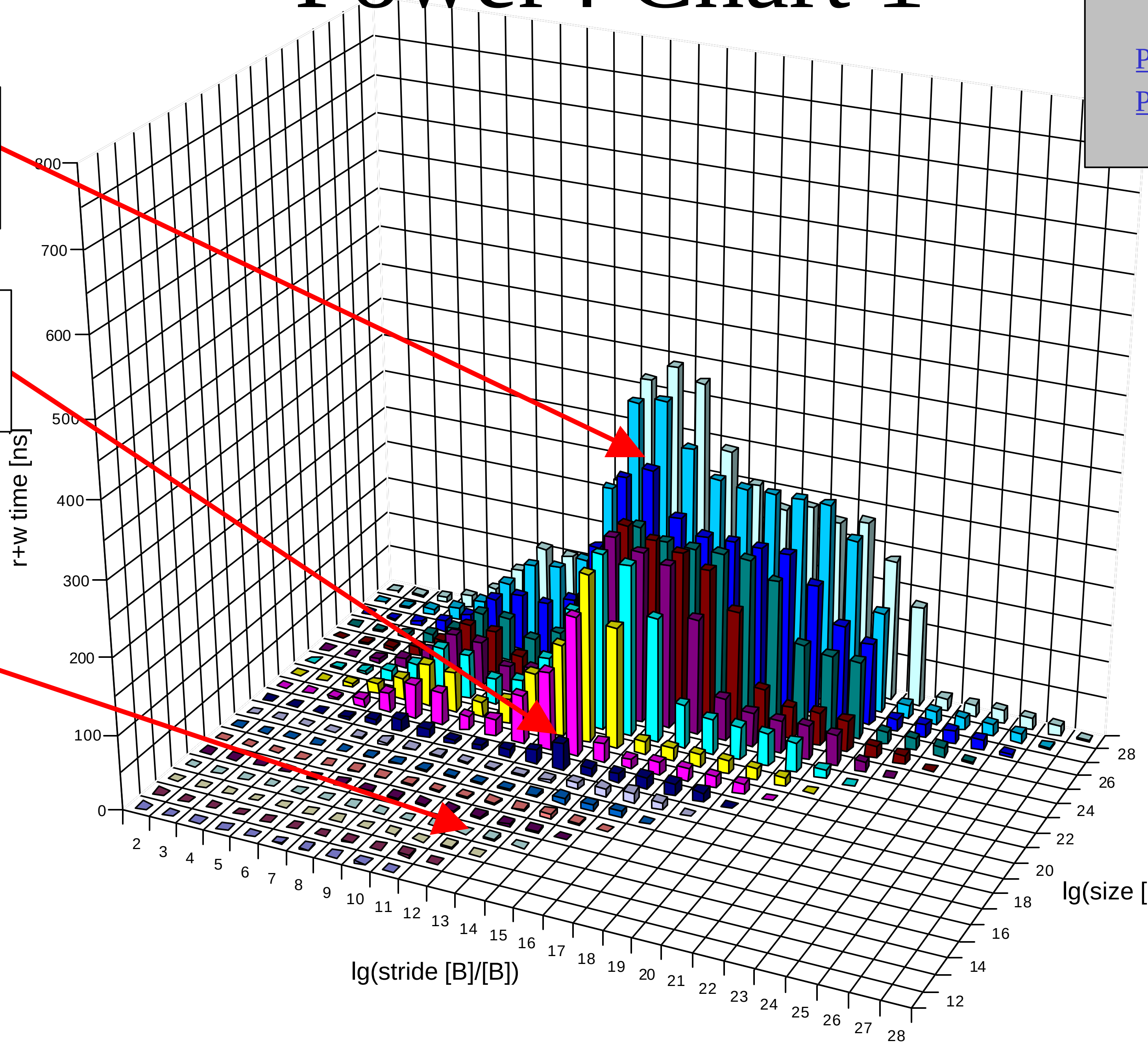
Power4 Chart 2

Power4 Chart 3

L3
128 MB
27 Bits

L2?
1 MB?
20 Bits?

L1



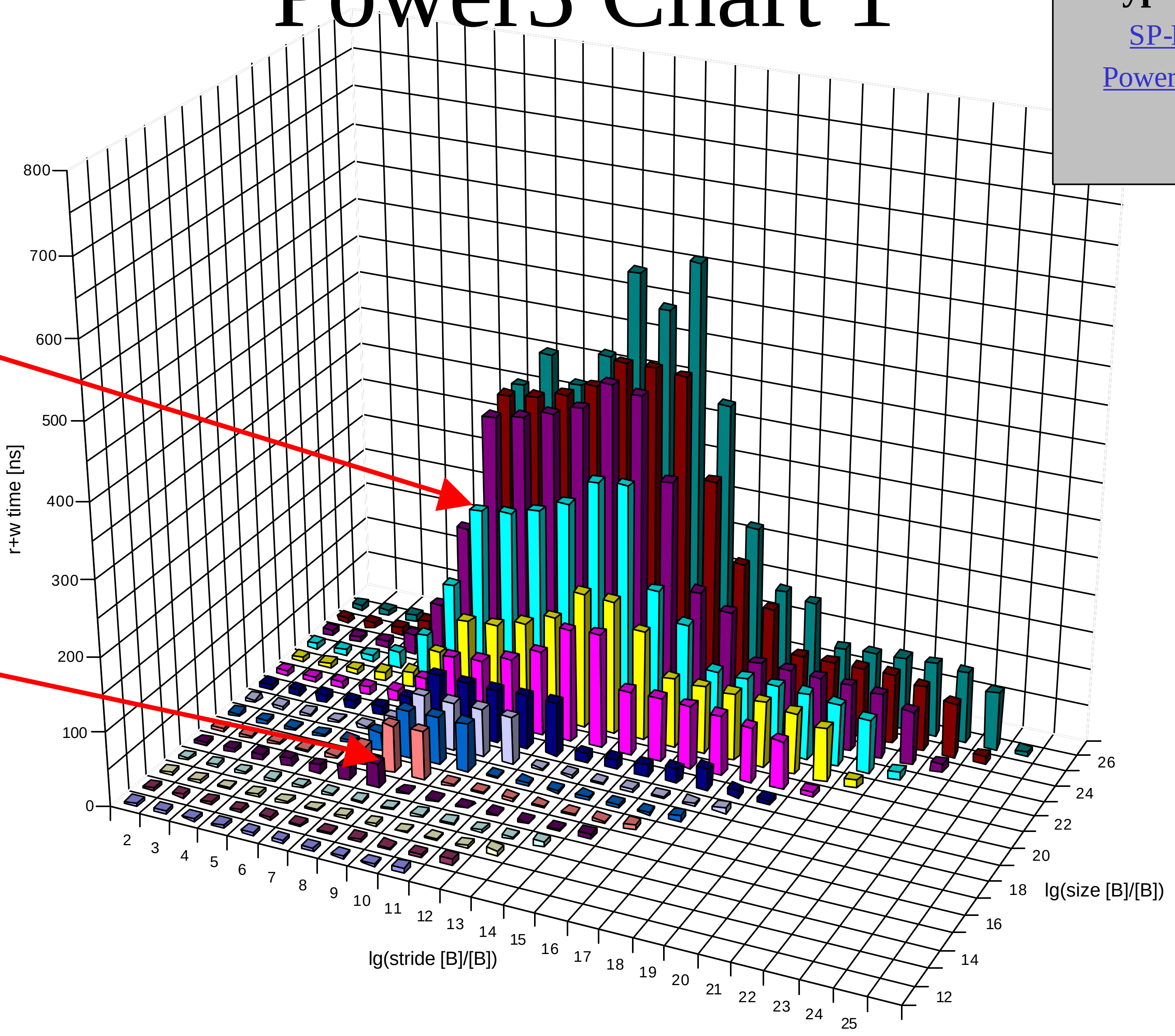
SP-Power3 (habu)

Power3 Chart 1

Hyper Links

SP-Power3

Power3 Chart 2



L2
8 MB
23 Bits

L1
64 KB
16 Bits

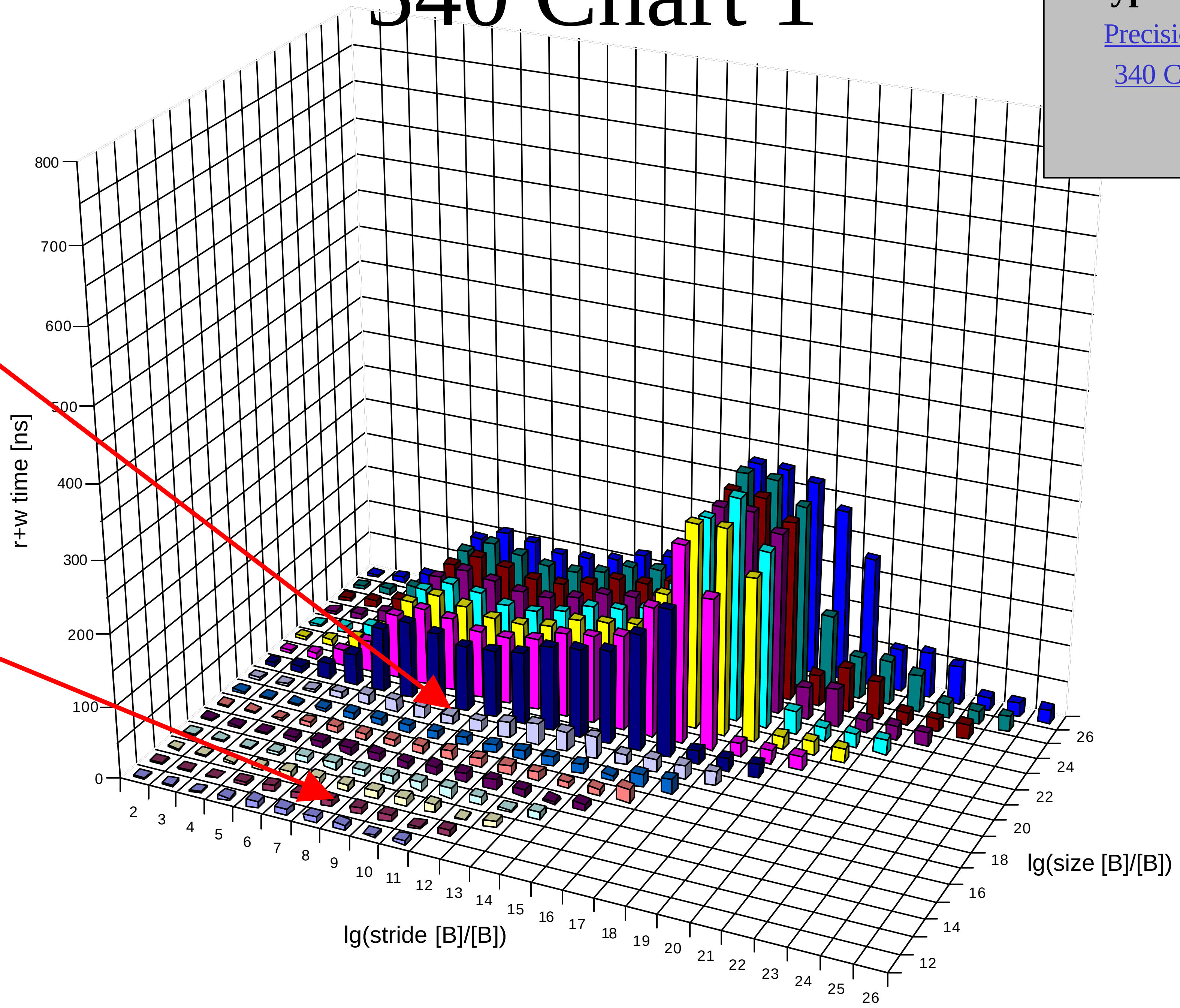
Precision 340 (gmorris-1)

340 Chart 1

Hyper Links

Precision 340

340 Chart 2



L2
512 KB
19 Bits

L1?

OptiPlex GX200 (gmorris-3)

GX200 Chart 1

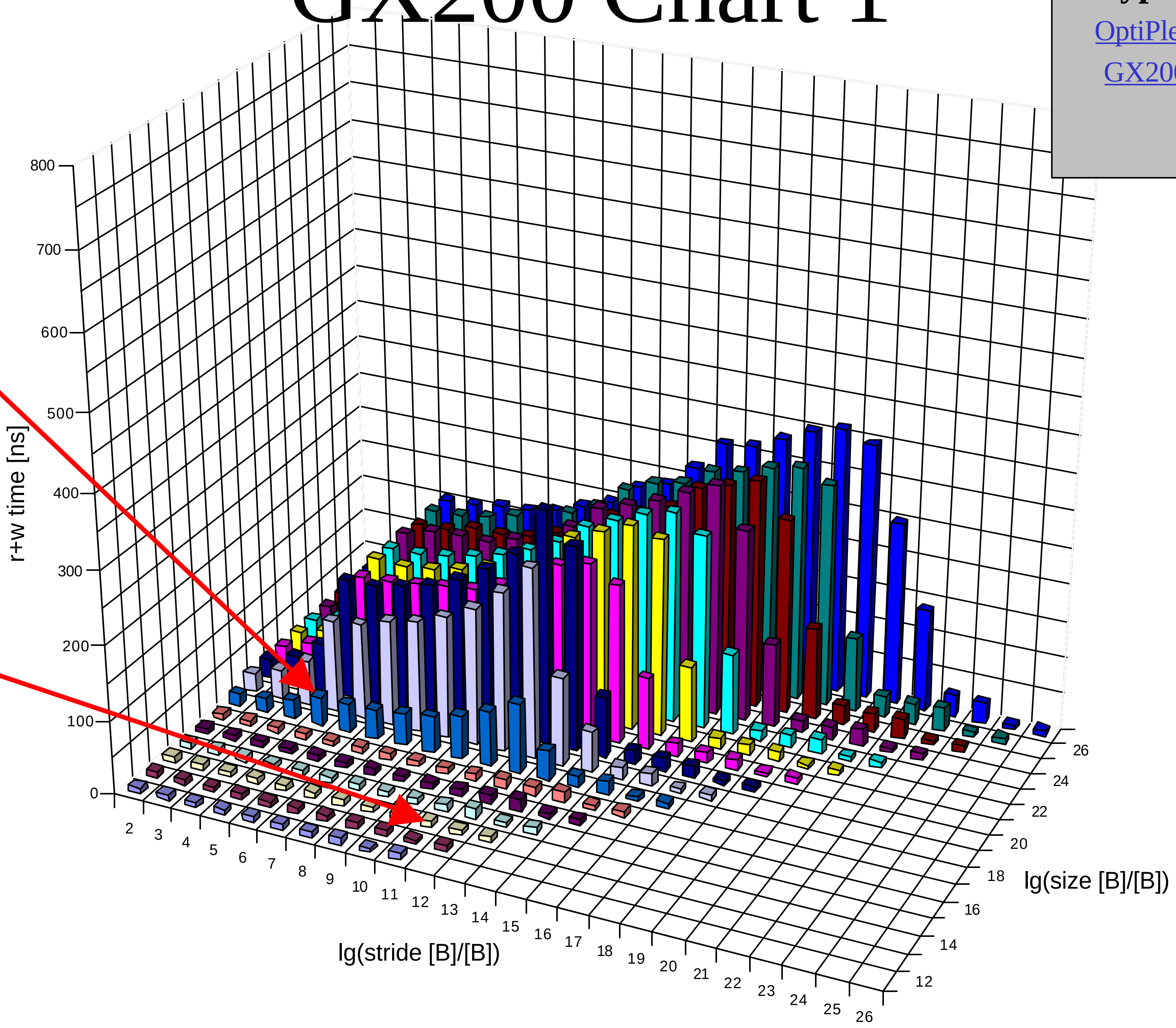
Hyper Links

OptiPlex GX200

GX200 Chart 2

L2
256 KB
18 Bits

L1?



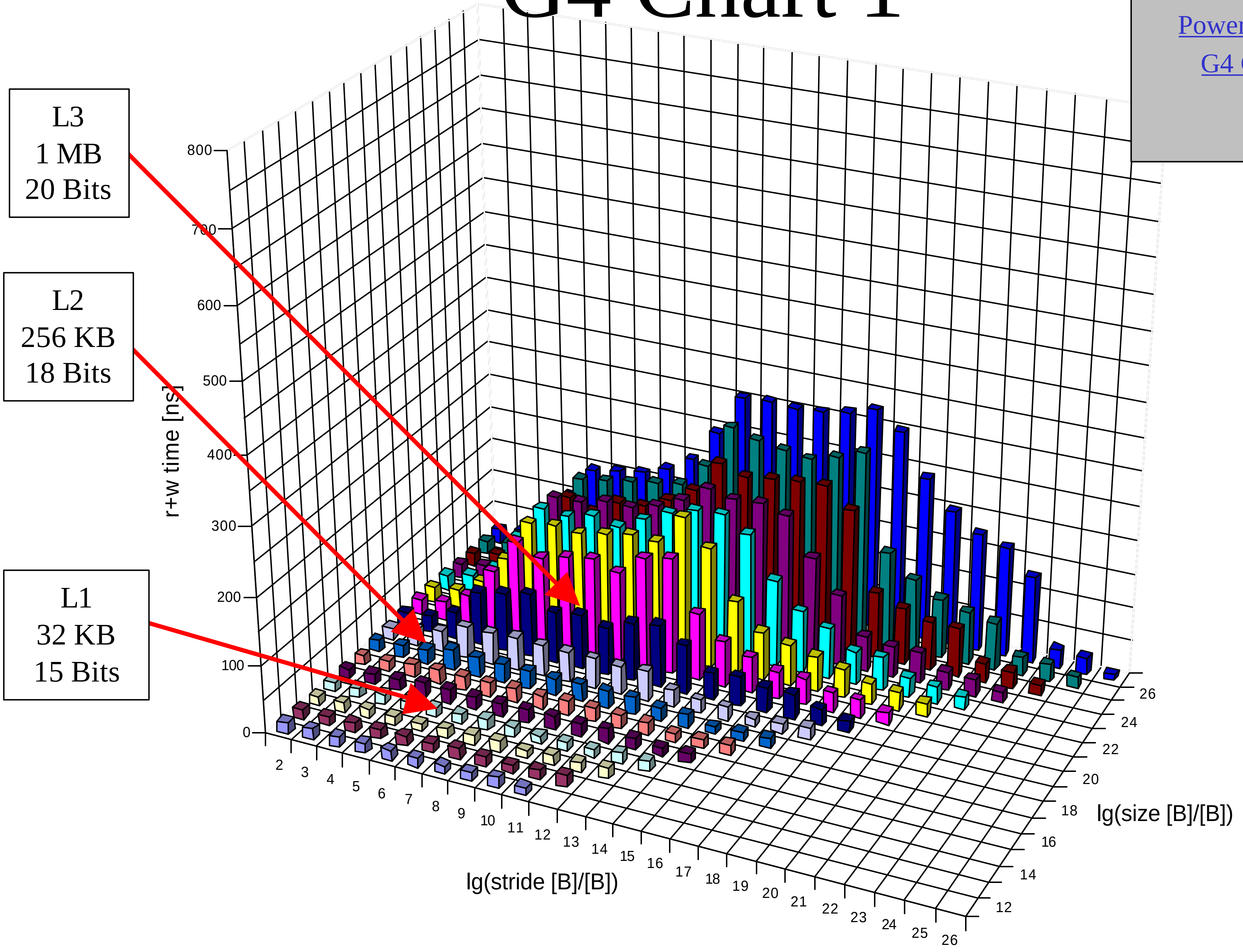
PowerBook G4 (octavo)

G4 Chart 1

Hyper Links

PowerBook G4

G4 Chart 2



L3
1 MB
20 Bits

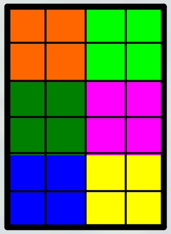
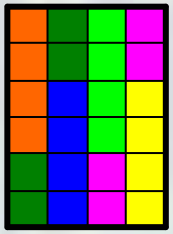
L2
256 KB
18 Bits

L1
32 KB
15 Bits

Recommendations

- Be Acquainted with Memory Hierarchy
  - Memory Size, Cache Size, Block Size, Word Size
  - Associativity
  - Replacement Policy
  - Write Policy, Write Miss Policy
  - Etc.
- General Rule: “If It Is Already in Cache, Use It”
- Prime Directive: “Use Existing Library Routines”
  - Already Tuned by Very Clever Folks
  - LAPACK
  - SCALAPACK
  - Etc.
- Understand Data Storage and Access Patterns
- Employ “Cache-Aware” Coding Techniques

Cache-Aware Coding

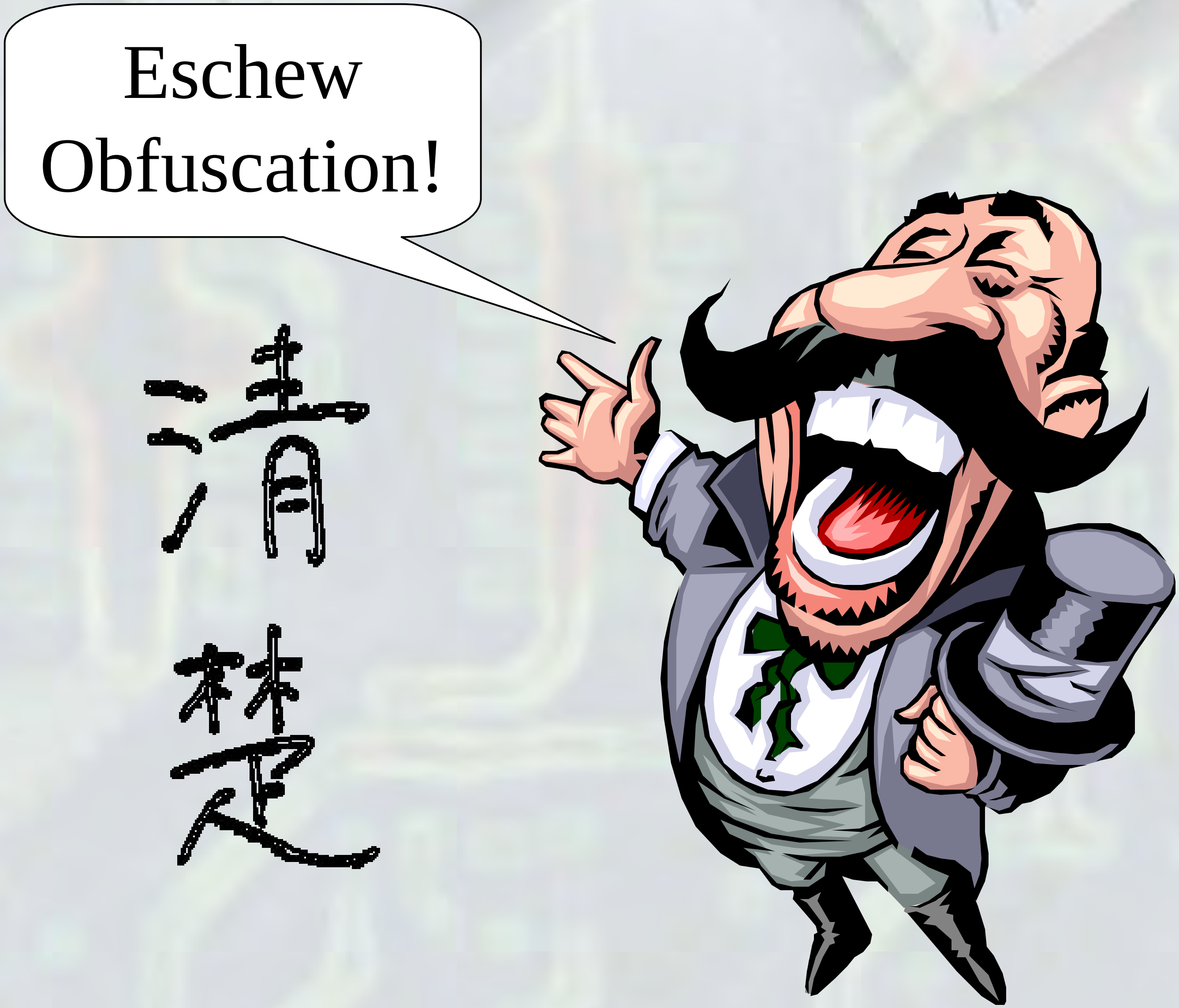
- More CPUs and Smaller Arrays[†]
- Array of Records Rather Than Multiple Arrays
  - Record Elements Might Be Contiguous
  - Separate Arrays Will Not Be Contiguous
- Blocking
  - Use Sub Matrices, e.g.,  Rather Than 
- Loop Fusion
  - Combine Loops Referencing Nearby Elements
- Loop Interchange (Rearrange Loop Indices)
  - Based on Storage Pattern and Algorithm
  - Select Best Access Pattern
- Etc.

[†] Sometimes!

Summary

- Cache Memory Primer
- Storage and Access Patterns
- The Experiment
- Recommendations

Questions



清楚

Backup Slides

- These backup slides can be viewed via the hyper links on previous slides.

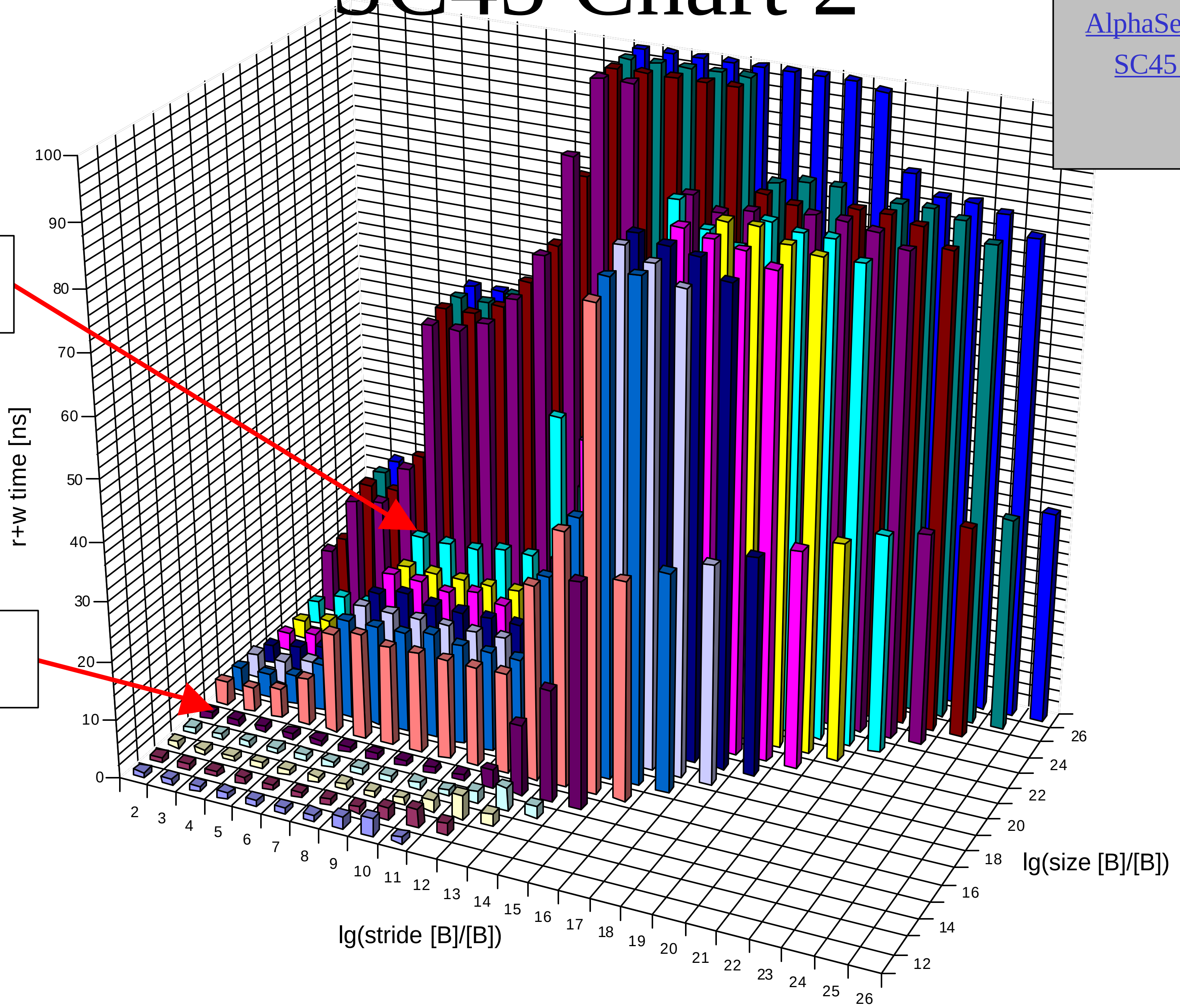
AlphaServer SC45 (emerald)

SC45 Chart 2

Hyper Links

AlphaServer SC45

SC45 Chart 1



23 Bits
8 MB L2

16 Bits
64 KB L1

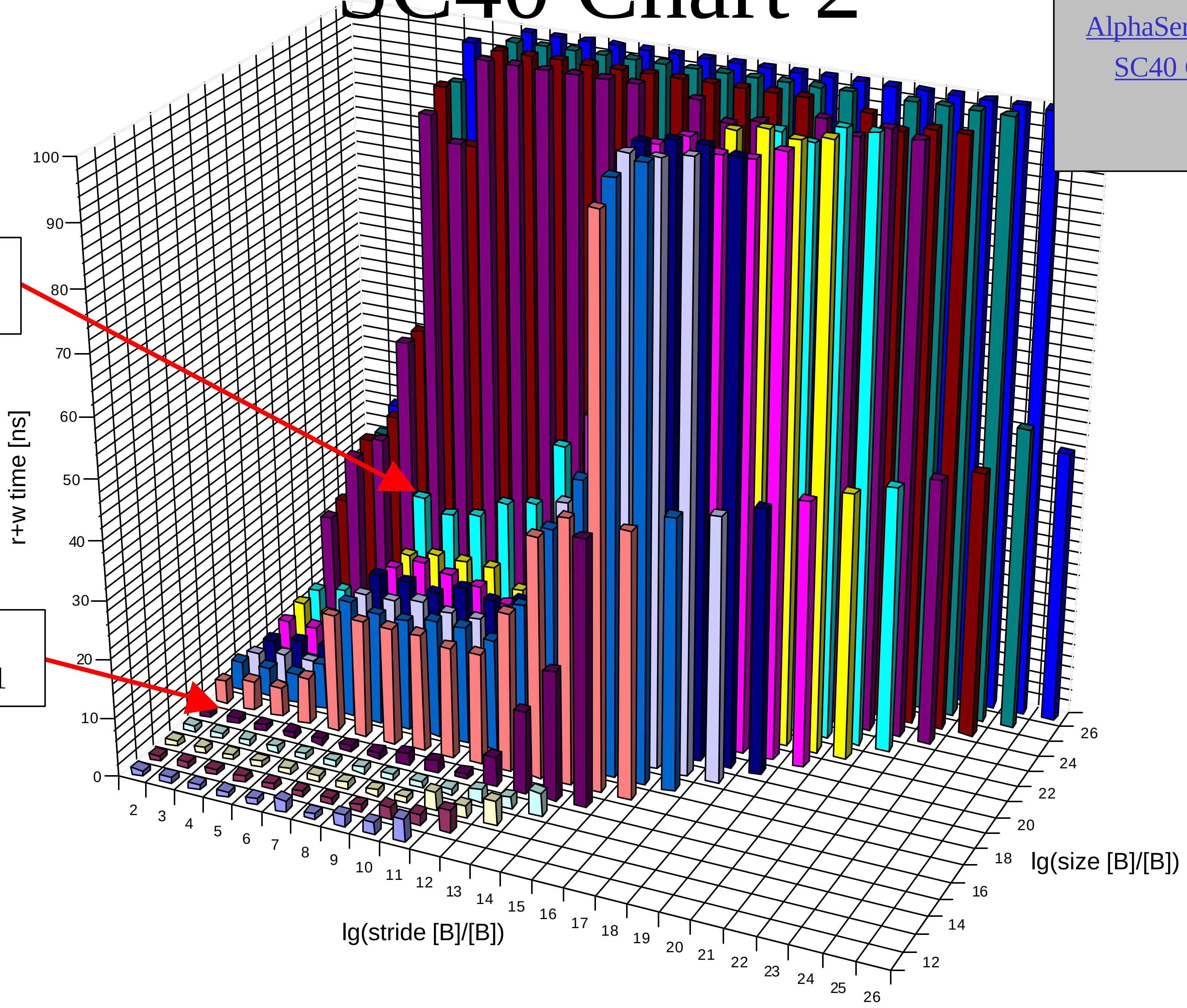
AlphaServer SC40 (opal)

SC40 Chart 2

Hyper Links

AlphaServer SC40

SC40 Chart 1



23 Bits
8 MB L2

16 Bits
64 KB L1

Origin 3900 (sand)

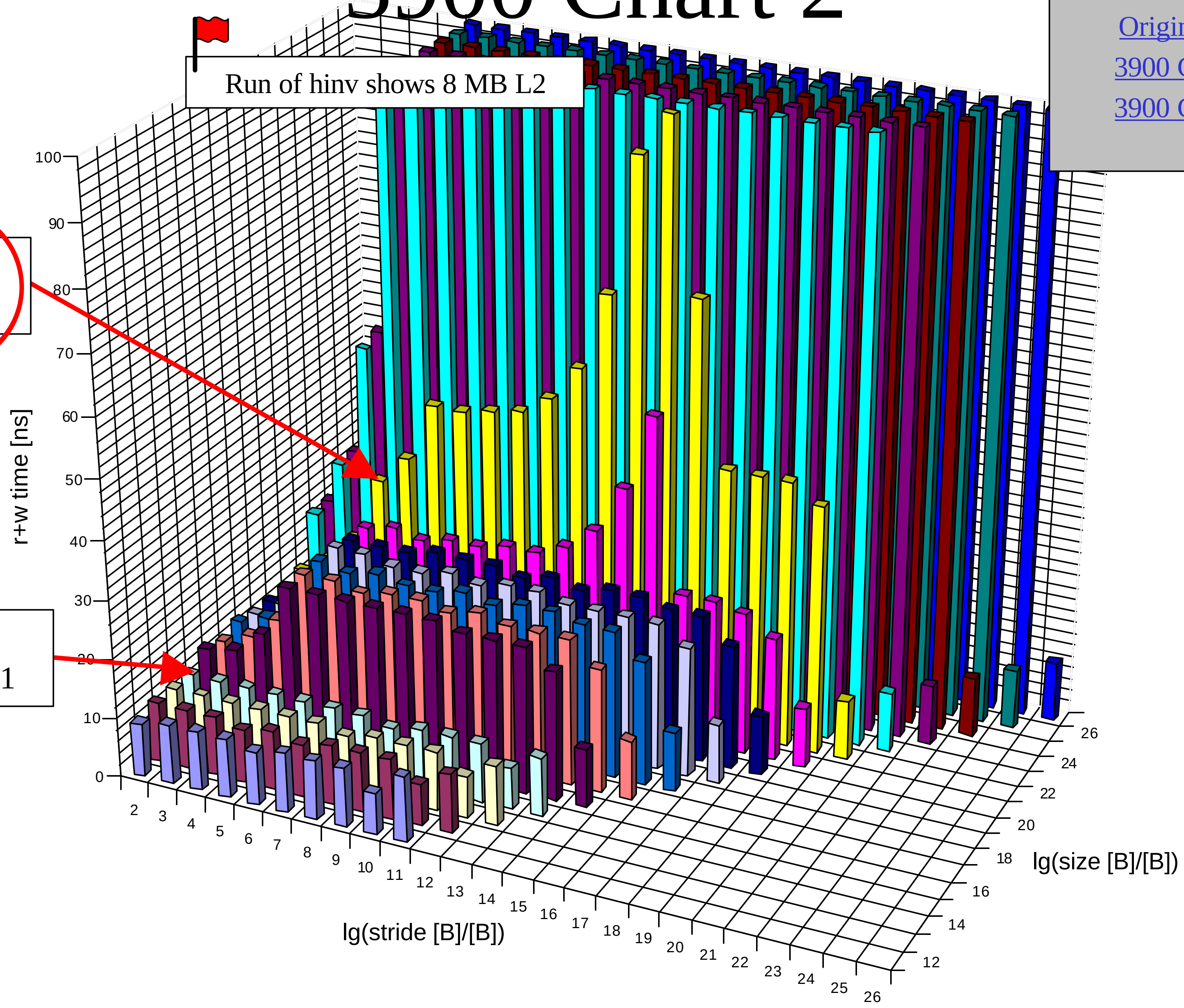
3900 Chart 2

Hyper Links

Origin 3900

3900 Chart 1

3900 Chart 3



Run of hinv shows 8 MB L2

~~22 Bits~~
~~4 MB L2~~

15 Bits
32 KB L1

Origin 3800 (sard)

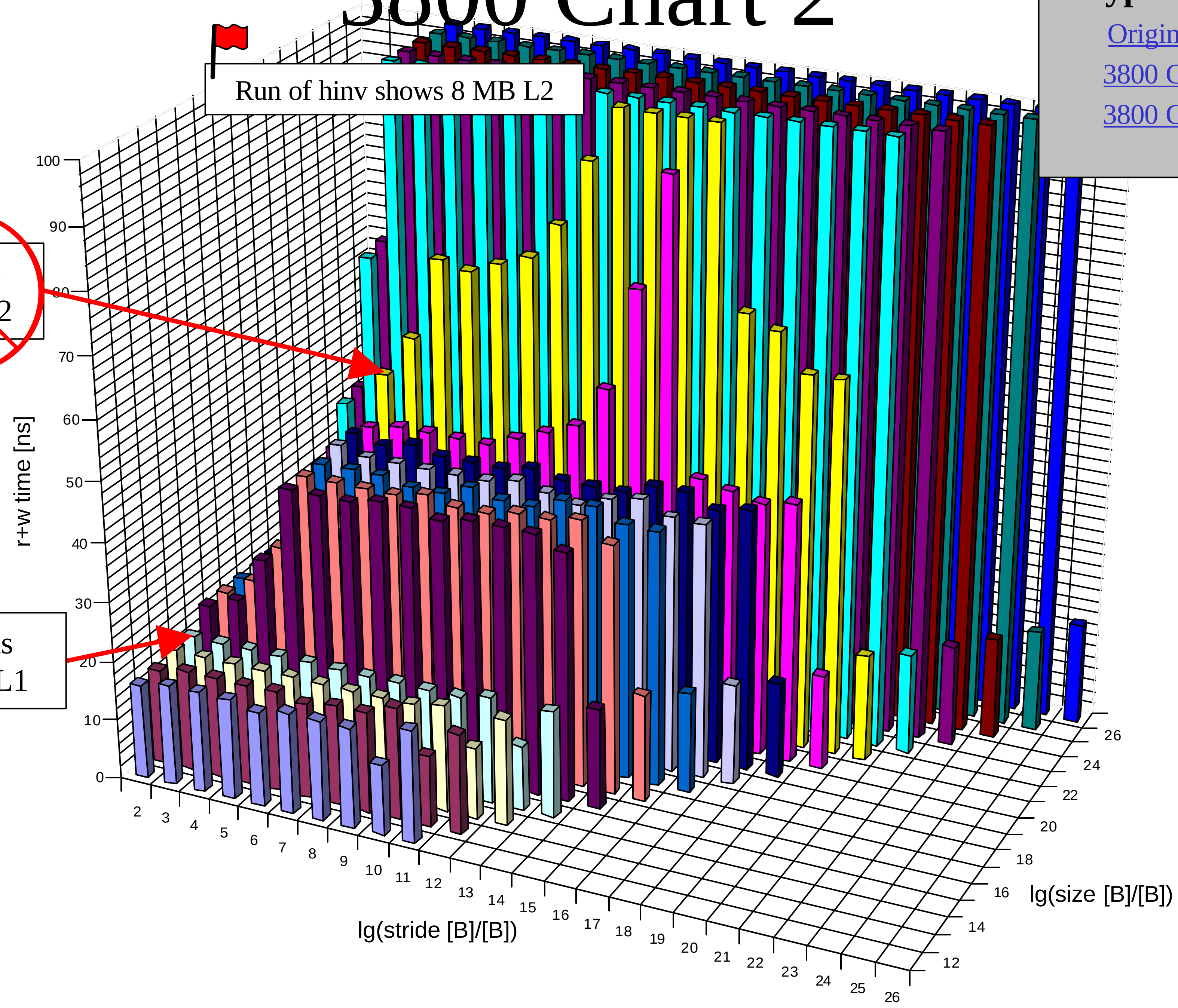
3800 Chart 2

Hyper Links

[Origin 3800]

[3800 Chart 1]

[3800 Chart 3]



Run of hinv shows 8 MB L2

~~22 Bits~~
~~4 MB L2~~

15 Bits
32 KB L1

T3E LC-1350/1200 (jim)

T3E Chart 2

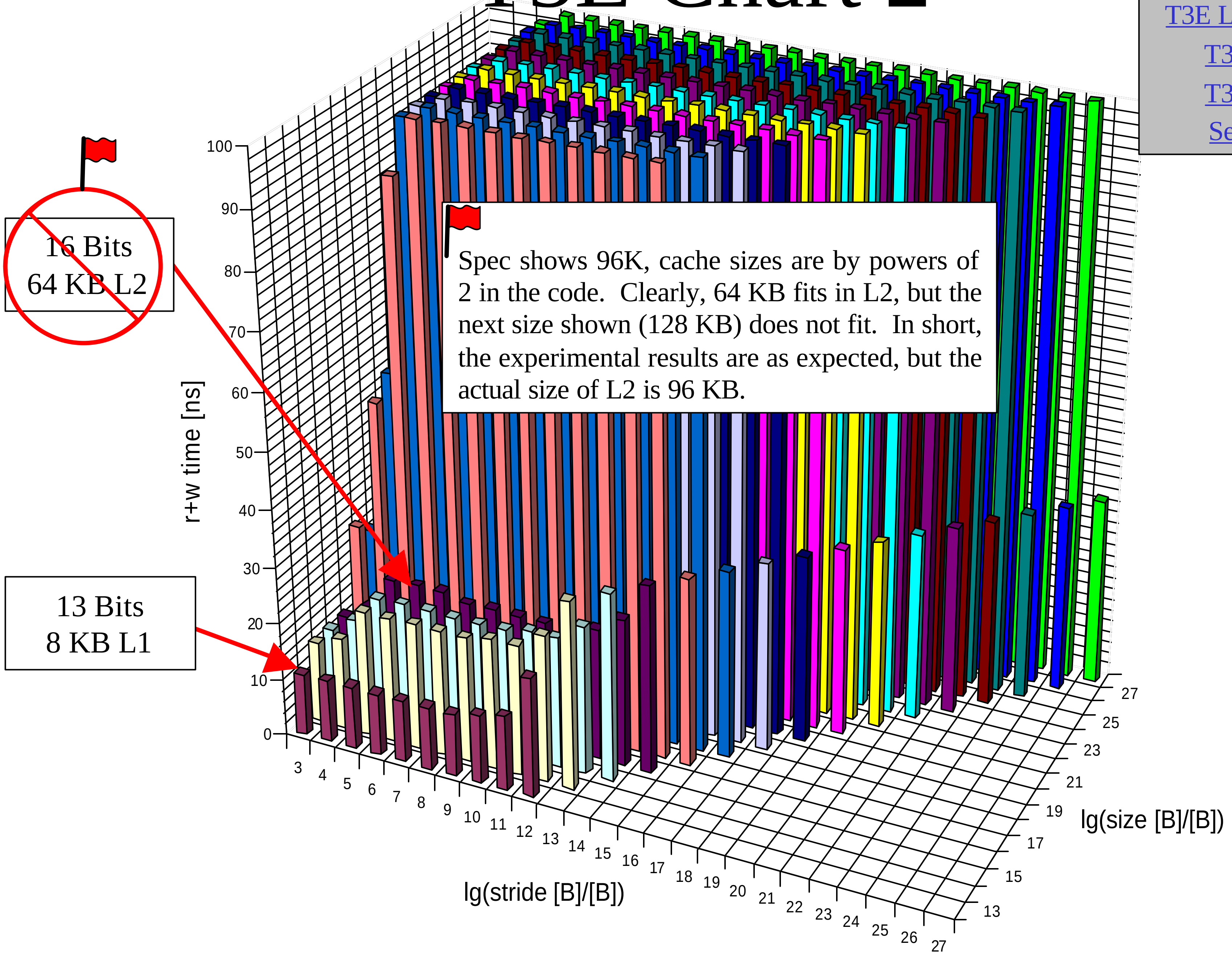
Hyper Links

T3E LC-1350/1200

T3E Chart 1

T3E Chart 3

Serendipity



~~16 Bits 64 KB L2~~

13 Bits 8 KB L1

Spec shows 96K, cache sizes are by powers of 2 in the code. Clearly, 64 KB fits in L2, but the next size shown (128 KB) does not fit. In short, the experimental results are as expected, but the actual size of L2 is 96 KB.

SP-Power4 (marcellus)

Power4 Chart 2

Hyper Links

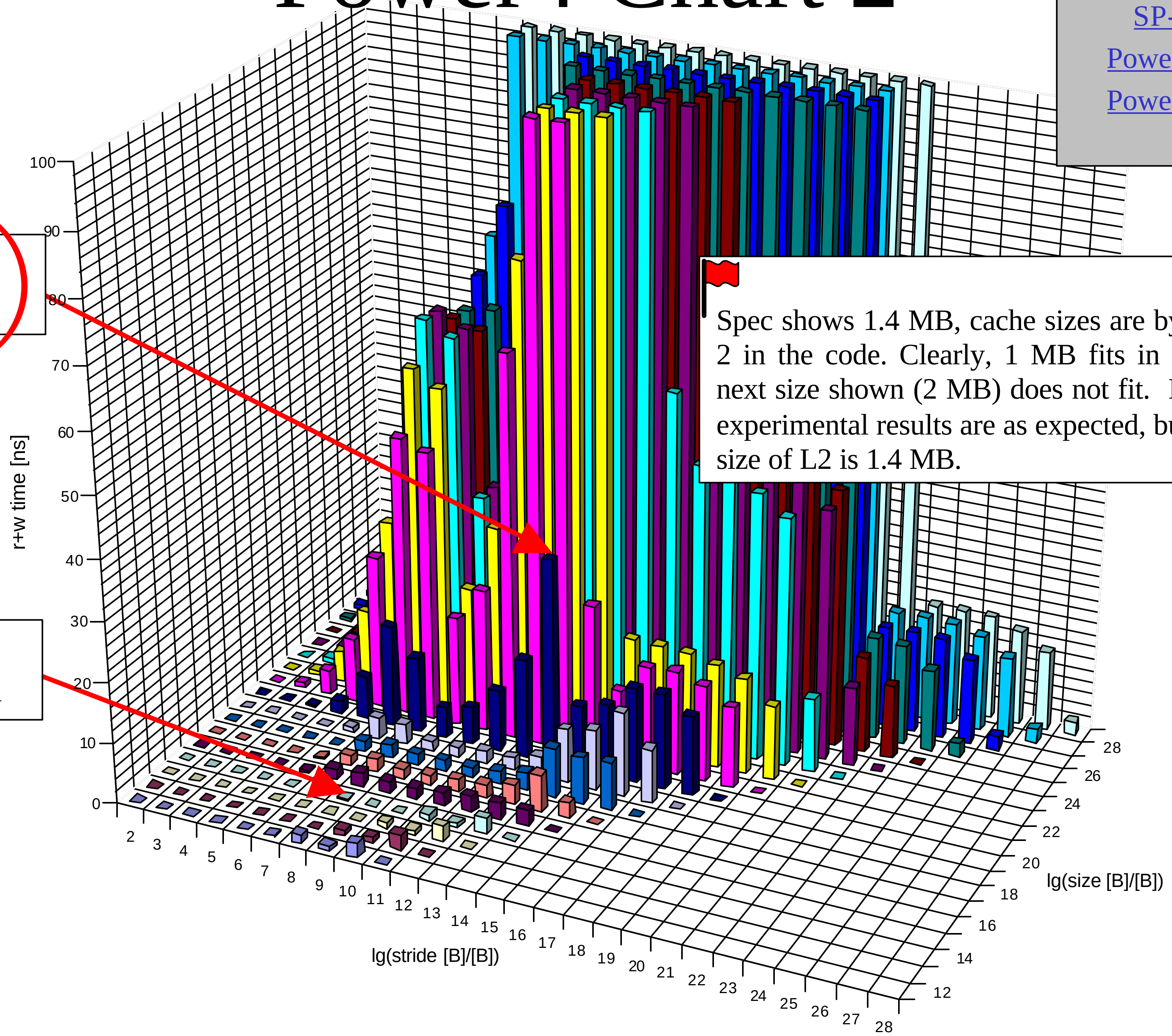
SP-Power4

Power4 Chart 1

Power4 Chart 3

~~20 Bits~~
~~1 MB L2~~

15 Bits
32 KB L1



Spec shows 1.4 MB, cache sizes are by powers of 2 in the code. Clearly, 1 MB fits in L2, but the next size shown (2 MB) does not fit. In short, the experimental results are as expected, but the actual size of L2 is 1.4 MB.

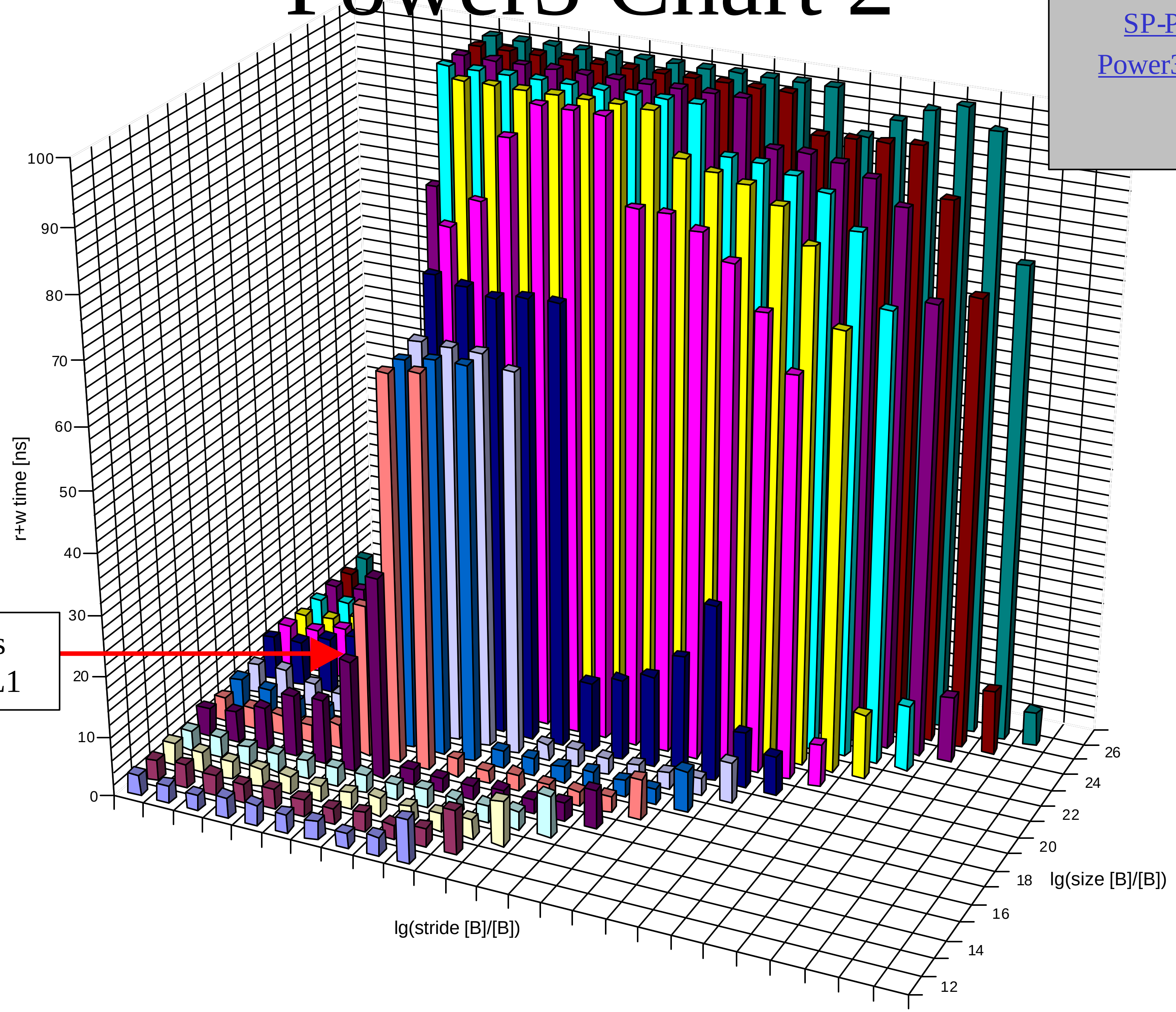
SP-Power3 (habu)

Power3 Chart 2

Hyper Links

SP-Power3

Power3 Chart 1



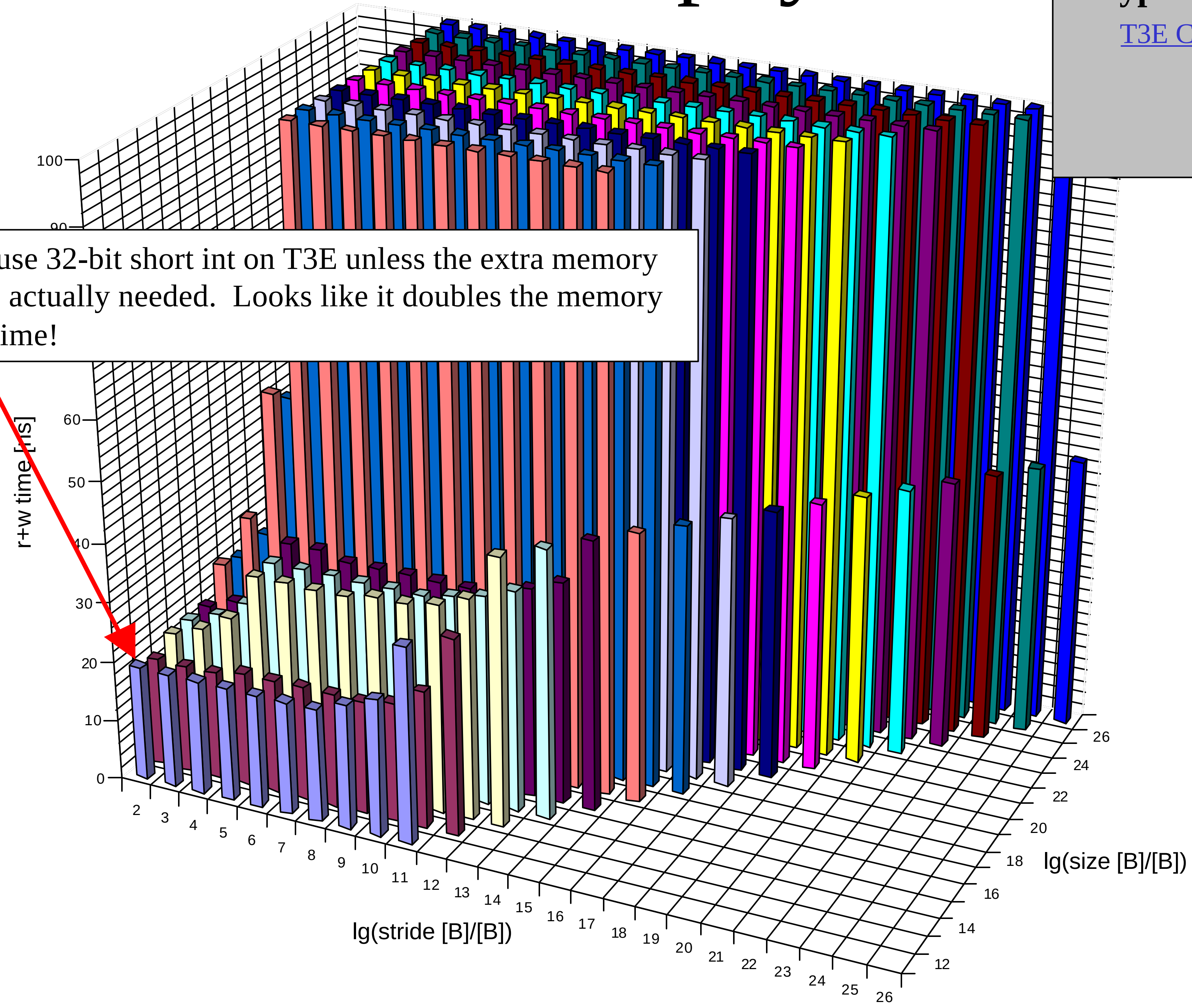
16 Bits
64 KB L1

jim (32-bit)

Serendipity

Hyper Links

T3E Chart 2



Do not use 32-bit short int on T3E unless the extra memory space is actually needed. Looks like it doubles the memory access time!

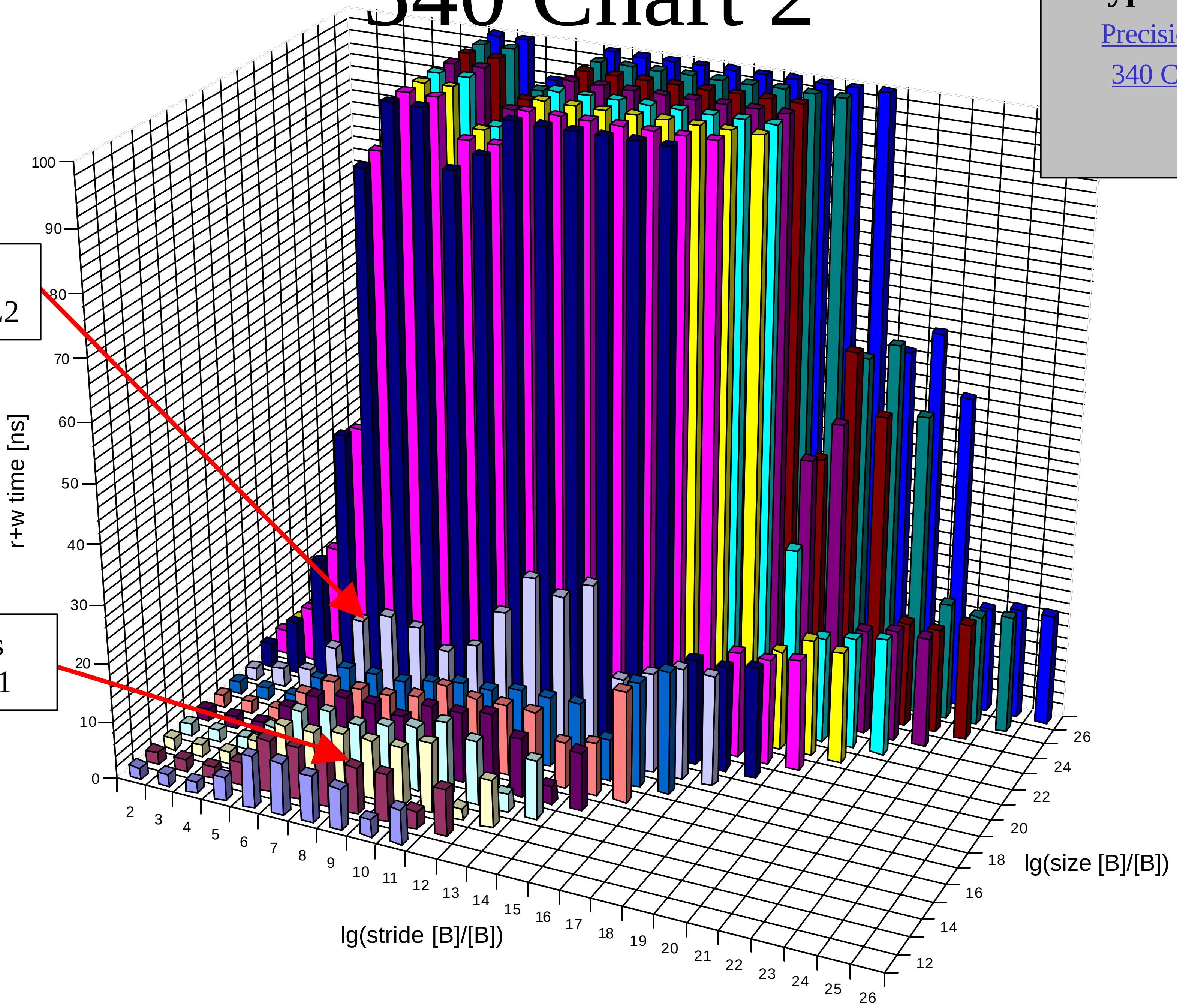
Precision 340 (gmorris-1)

340 Chart 2

Hyper Links

Precision 340

340 Chart 1



19 Bits
512 KB L2

13 Bits
8 KB L1

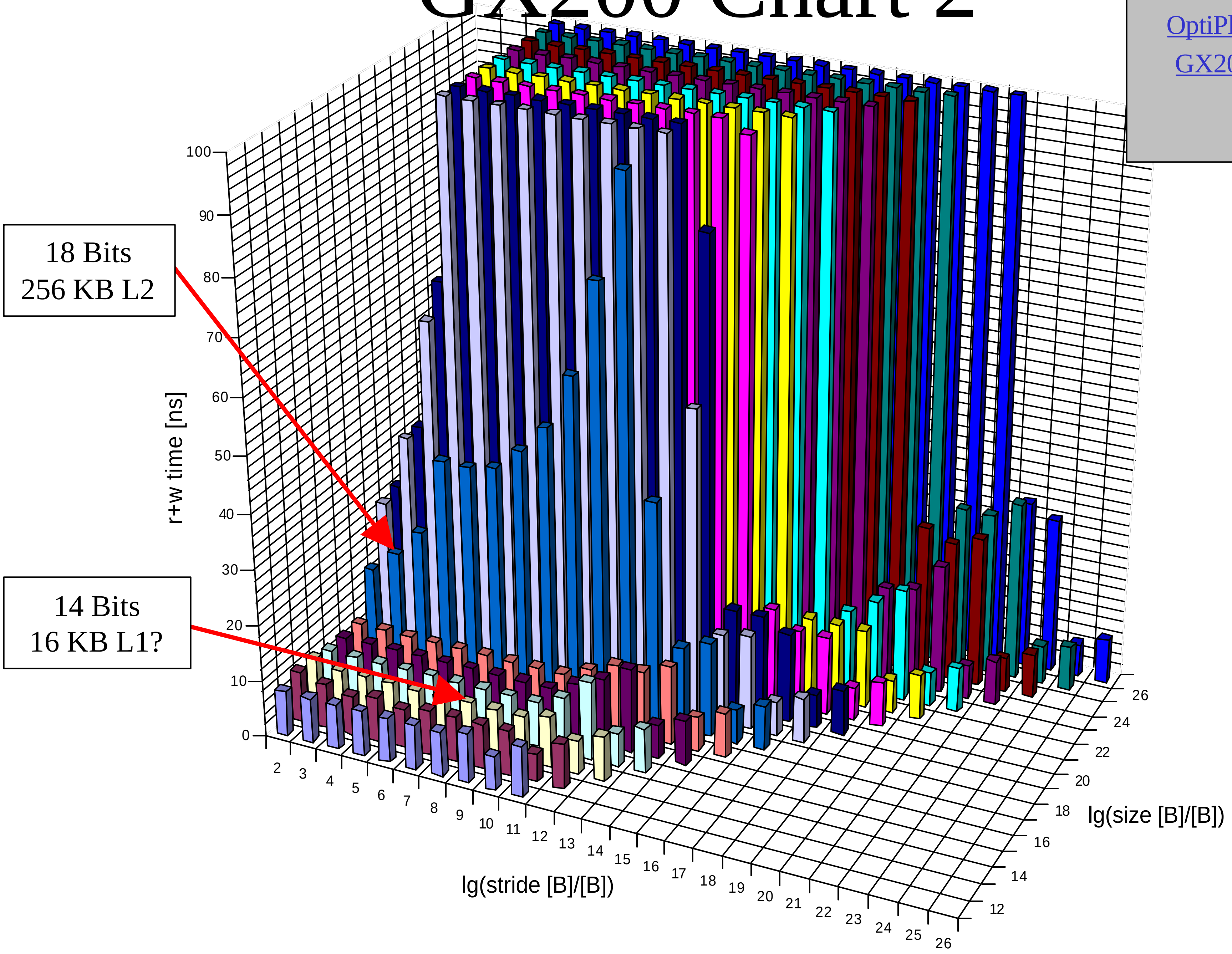
OptiPlex GX200 (gmorris-3)

GX200 Chart 2

Hyper Links

OptiPlex GX200

GX200 Chart 1



18 Bits
256 KB L2

14 Bits
16 KB L1?

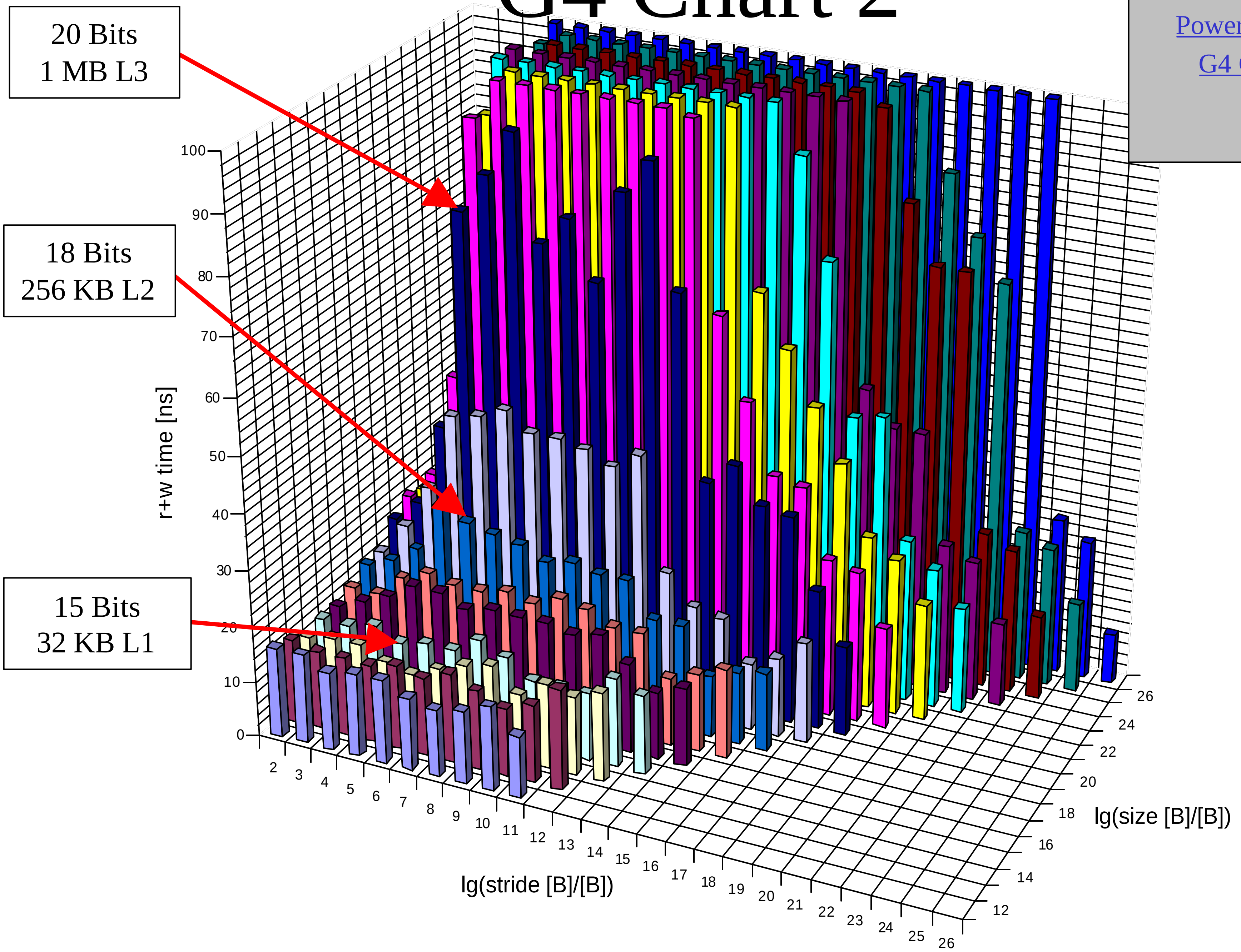
PowerBook G4 (octavo)

G4 Chart 2

Hyper Links

PowerBook G4

G4 Chart 1


20 Bits
1 MB L3

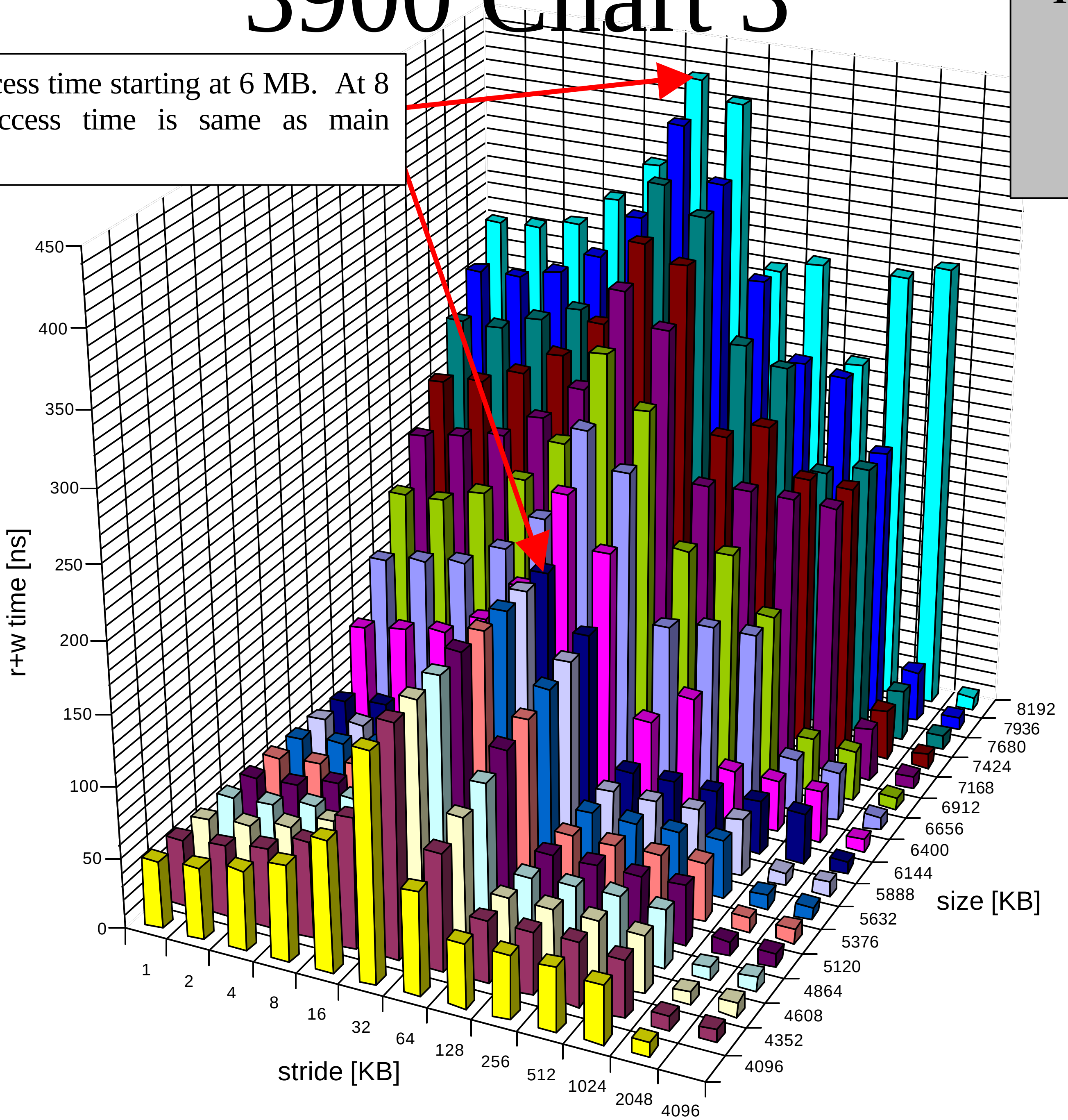
18 Bits
256 KB L2

15 Bits
32 KB L1

Origin 3900 (sand) L2 Cache Boundary

3900 Chart 3

Gradual increase in access time starting at 6 MB. At 8 MB L2 boundary, access time is same as main memory access time.



Hyper Links

Origin 3900

3900 Chart 1

3900 Chart 2

Origin 3800 (sard) L2 Cache Boundary

3800 Chart 3

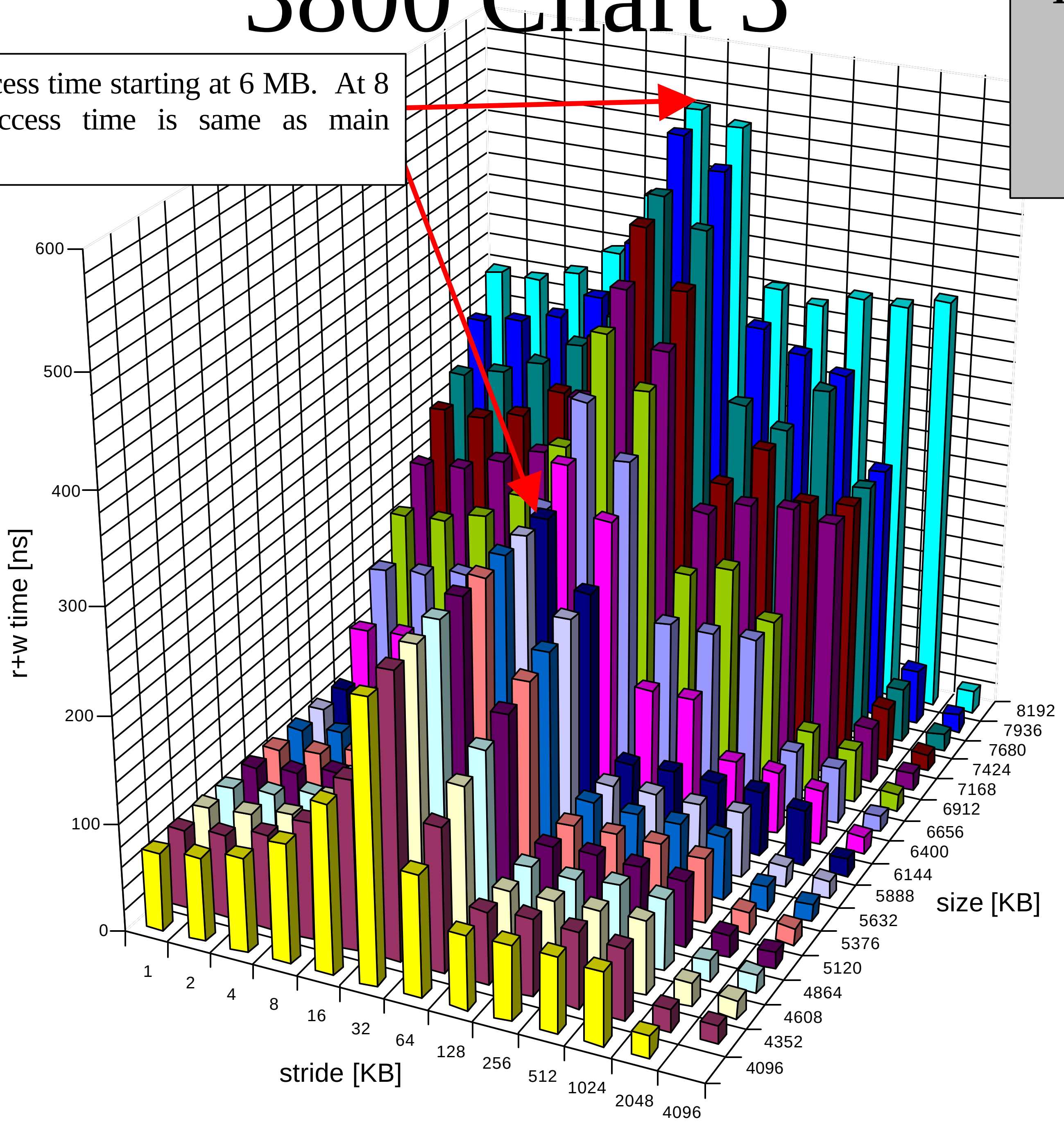
Gradual increase in access time starting at 6 MB. At 8 MB L2 boundary, access time is same as main memory access time.

Hyper Links

Origin 3800

3800 Chart 1

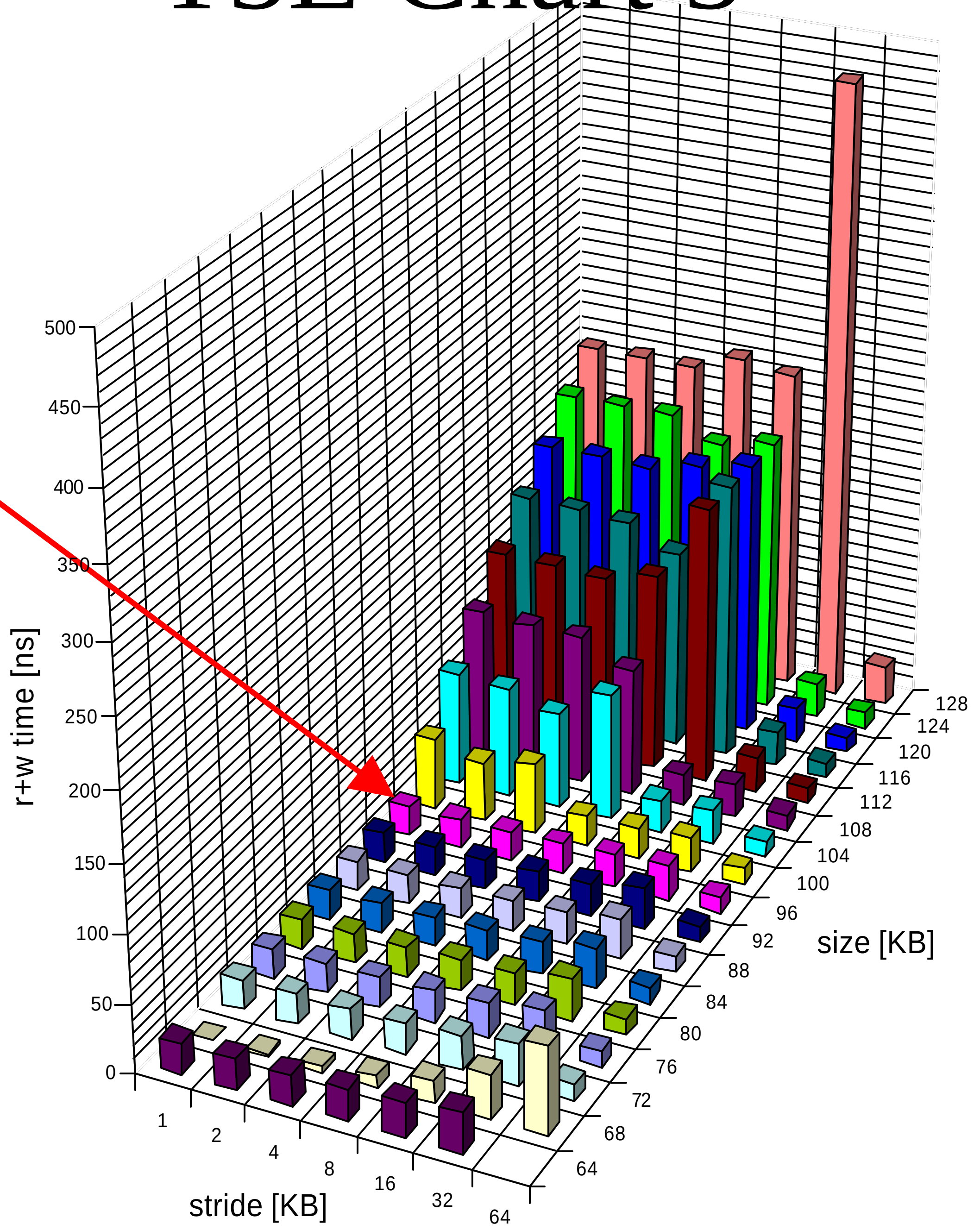
3800 Chart 2



T3E LC-1350/1200 (jim) L2 Cache Boundary

T3E Chart 3

96 KB L2



Hyper Links

T3E LC-1350/1200

T3E Chart 1

T3E Chart 2

SP-Power4 (marcellus) L2 Cache Boundary

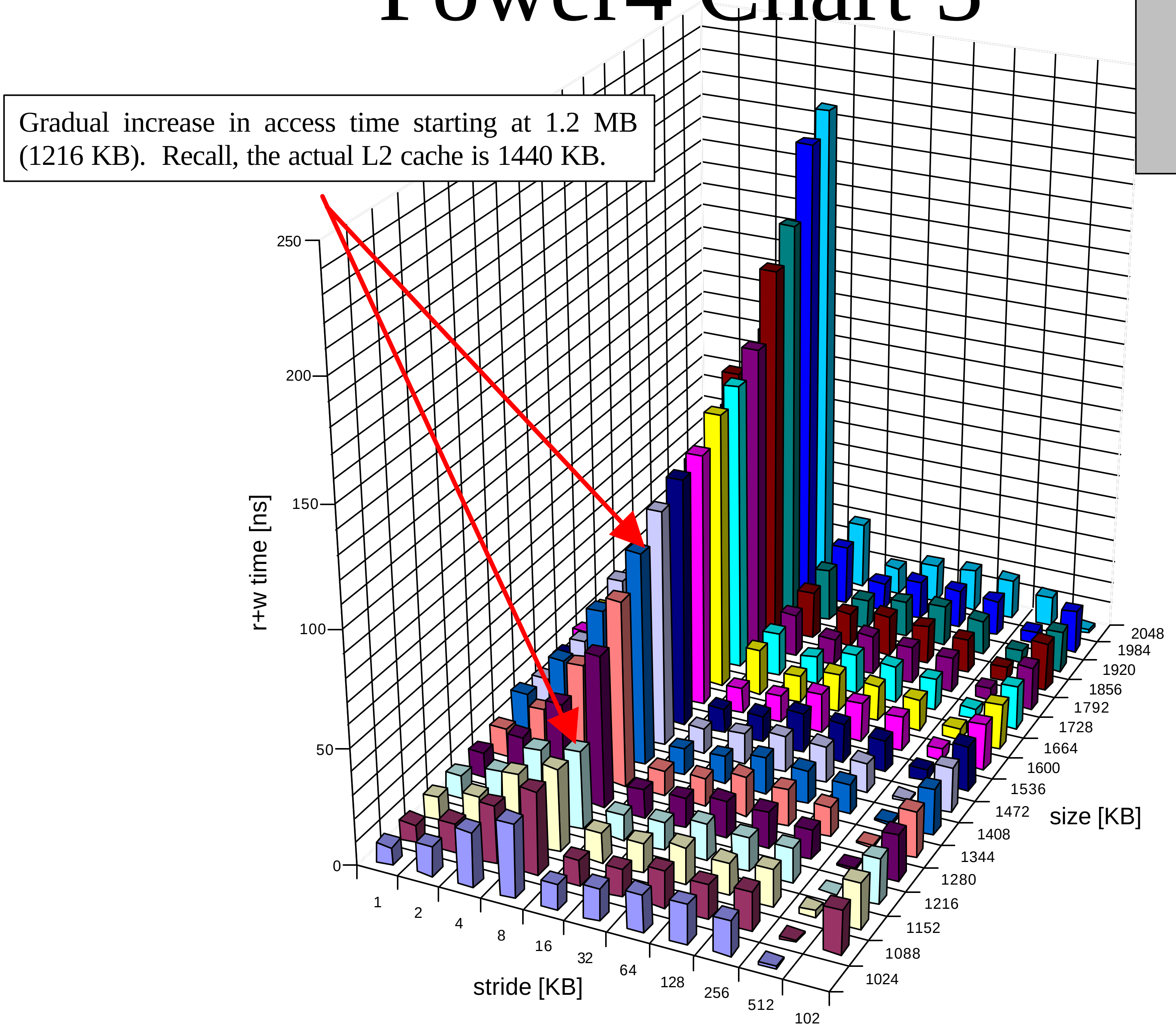
Power4 Chart 3

Hyper Links

SP-Power4

Power4 Chart 1

Power4 Chart 2



Gradual increase in access time starting at 1.2 MB (1216 KB). Recall, the actual L2 cache is 1440 KB.