Can apparent superluminal neutrino speeds be explained as a quantum weak measurement?

M V Berry¹, N Brunner¹, S Popescu¹ & P Shukla²

¹H H Wills Physics Laboratory, Tyndall Avenue, Bristol BS8 1TL, UK
²Department of Physics, Indian Institute of Technology, Kharagpur, India

Abstract

Probably not.
Memory Hierarchy

- Registers
- Cache
- Main memory
- Magnetic disk
- Tape
- Optical disk
Multi-Level Caches ($’s circa 2007)

- Options: separate **data** and **instruction caches**, or a **unified cache**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Regs</th>
<th>L1 d-cache</th>
<th>Unified L2 Cache</th>
<th>Memory</th>
<th>disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:</td>
<td>200 B</td>
<td>8-64 KB</td>
<td>1-4MB SRAM</td>
<td>128 MB DRAM</td>
<td>30 GB</td>
</tr>
<tr>
<td>speed:</td>
<td>3 ns</td>
<td>3 ns</td>
<td>6 ns</td>
<td>60 ns</td>
<td>8 ms</td>
</tr>
<tr>
<td>$/Mbyte:</td>
<td>$100/MB</td>
<td>$1.50/MB</td>
<td>$100/MB</td>
<td>$1.50/MB</td>
<td>$0.05/MB</td>
</tr>
<tr>
<td>line size:</td>
<td>8 B</td>
<td>32 B</td>
<td>32 B</td>
<td>8 KB</td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper
Intel Pentium Cache Hierarchy

- **L1 Data**
  - 1 cycle latency
  - 16 KB
  - 4-way assoc
  - Write-through
  - 32B lines

- **L1 Instruction**
  - 16 KB, 4-way
  - 32B lines

- **L2 Unified**
  - 128KB--2 MB
  - 4-way assoc
  - Write-back
  - Write allocate
  - 32B lines

- **Main Memory**
Locality

- **Principle of Locality** states within a given period of time, programs tend to reference a relatively confined area of memory repeated.
- This Principle is a measurable phenomena and produces a *working set*.
- The working set is the set of memory locations referenced over a fixed period of time, or window, extending from the current time into the past.
Spatial Locality, or the fact that when a given address has been referenced, it is likely that addresses near it will be referenced within a short period of time.

Temporal Locality, or the fact that once a particular memory item has been referenced, it is likely that it will be referenced again within a short period of time.
Consider a “block” in a cache

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time
2-Level Hierarchy

• A two level hierarchy suggests that memory can be structured in two layers, or two interacting layers.
  – **Primary level** -- small and fast
  – **Secondary level** -- large and slow
• This relationship indicates a trade-off in memory technologies between processing time expense and hardware purchasing expense.
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);      /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
Memory Mountain Test Function

/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

read throughput (MB/s)

Slopes of Spatial Locality

Ridges of Temporal Locality

stride (words)

working set size (bytes)
We can use *math* to describe
Hits vs misses

\[ t_a = h t_p + (1 - h) t_s \]

- \( t_a \): actual access time
- \( t_p \): primary memory access time
- \( t_s \): secondary memory access time
- \( h \): hit ratio, or hits per memory reference
Hits vs misses

\[ t_a = h t_p + (1 - h) t_s \]

- **Assume:**
  - \( t_p \) -- primary memory access time is 10 ns.
  - \( t_s \) -- secondary memory access time is 100 ns.
Hits vs misses

\[ t_a = h t_p + (1 - h) t_s \]

<table>
<thead>
<tr>
<th>h</th>
<th>( h t_p )</th>
<th>((1-h)t_s)</th>
<th>( t_a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.2</td>
<td>2</td>
<td>80</td>
<td>82</td>
</tr>
<tr>
<td>0.4</td>
<td>4</td>
<td>60</td>
<td>64</td>
</tr>
<tr>
<td>0.5</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>0.6</td>
<td>6</td>
<td>40</td>
<td>46</td>
</tr>
<tr>
<td>0.8</td>
<td>8</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>1.0</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Matrix Multiplication Example

- **Major Cache Effects to Consider**
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

- **Description:**
  - Multiply $N \times N$ matrices
  - $O(N^3)$ total operations
  - Accesses
    - $N$ reads per source element
    - $N$ values summed per destination
    - but may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable `sum` held in register!
Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Cache Line (Block) size = 16 bytes (fits 4 32-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows

- **Analysis Method:**
  - Look at access pattern of inner loop
Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations

- **Stepping through columns in one row:**
  ```c
  for (i = 0; i < N; i++)
      sum += a[0][i];
  ```
  - accesses successive elements (doubles, 8 bytes)
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B (for now, assume B=32)

- **Stepping through rows in one column:**
  ```c
  for (i = 0; i < n; i++)
      sum += a[i][0];
  ```
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)
Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kij)

/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

**Misses per Inner Loop Iteration:**
- A
- B
- C

Approx speedup relative to 1.25?
Cache = 10ms and memory = 100ms
n is BIG

\[ t_a = h t_p + (1 - h) t_s \]
Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ikj)

/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kji)

Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Summary of Matrix Multiplication

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = **1.25**

```plaintext
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = **0.5**

```plaintext
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = **2.0**

```plaintext
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```
Pentium Matrix Multiply Performance

• Miss rates are helpful but not perfect predictors.
SO, RON AND GINNY ARE LOVERS?

WHAT? NO, THEY'RE BROTHER AND SISTER.

OH...

NOT EVEN A LITTLE BIT?

THE REST OF LUNCH BETWEEN GEORGE R.R. MARTIN AND J.K. ROWLING WAS PRETTY AWKWARD.
#define N 1000

typedef struct {
    int vel[3];
    int acc[3];
} point;

point p[N];

void clear1 (point *p, int n){
    int i, j;
    for (i=0; i<n; i++){
        for (j=0; j<3; j++)
            p[i].vel[j] = 0;
        for (j=0; j<3; j++)
            p[i].acc[j]=0;
    }
}

void clear3 (point *p, int n){
    int i, j;
    for (j=0; j<3; j++){
        for (i=0; i<n; i++)
            p[i].vel[j] = 0;
        for (i=0; i<n; i++)
            p[i].acc[j]=0;
    }
}
General Cache Organization \((S, E, B)\)

- \(E = 2^e\) lines per set
- \(S = 2^s\) sets
- \(B = 2^b\) bytes per cache block (the data)

Cache size: \(S \times E \times B\) data bytes
Cache

• Direct-mapped cache (E=1):
  – Primary cache memory has $S$ blocks of memory of $B$ size.
  – Secondary memory has $S'$ blocks of memory of $B$ size.
  – $S'$ is a multiple of $S$.
  – Each block will map into a specific cache location based on the block number modulus $S$.
  – If a block exists in a specific location, then it will be evicted.
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

\[ S = 2^s \text{ sets} \]

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\hline
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\hline
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\hline
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array}
\]

Address of int:
\[
\begin{array}{|c|c|c|}
\hline
\text{t bits} & 0...01 & 100 \\
\hline
\end{array}
\]

find set
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
Size of this Cache?
Size of this Main Memory?
2nd (Colorful) Example

Main Memory contains:

\[ 2^5 \times 2^8 \times 2^3 = 2^{16} \text{Bytes} = 64\text{KB} \]

Cache Memory contains:

\[ 2^8 \times 2^3 = 2^{11} \text{Bytes} = 2\text{KB} \]
### 3rd Example

4 sets, one line per block, 2 bytes per block, 4-bit addresses

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Tag bit ((t=1))</th>
<th>Index bits ((s=2))</th>
<th>Offset bits ((b=1))</th>
<th>Block number (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>00</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

Read at address 0
Read 1
Read 13
Read 8
Read 0
Why index via the middle bits?

4-set cache

High-order bit indexing

Middle-order bit indexing

Set index bits
Cache

- Associative Cache --
  - Primary cache memory has $S$ blocks of memory of $B$ size.
  - Secondary memory as $S'$ blocks of memory of $B$ size.
  - $S$ and $S'$ are not related.
  - Each block in memory will be hashed into the cache and may not necessarily displace an existing block.
Associative-Mapped Caches

- Tag memory
- Valid bits
- Cache memory
- Main memory

Tag field, 13 bits
Valid, 1 bit
One cache line, 8 bytes
Main memory address: Tag 13, Byte 3
Set-Associative Cache

- Primary cache memory has $S$ regions of $E$ blocks memory of $B$ size.
- Secondary memory has $S'$ blocks of memory of $B$ size.
- $S'$ is typically a multiple of $S$, and each region can store $E$ blocks of memory.
- Each block will map into a specific cache region based on the block number modulus $S$.
- If there is an empty block in a region then existing blocks will not be overwritten; otherwise some existing data must be evicted.
Set-Associative Cache (2-way)

<table>
<thead>
<tr>
<th>Tag Memory</th>
<th>Valid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory block numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group #:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Tag field, 5 bits
One Cache line, 8 bytes
Set-Associative Cache (2-way)

Main Memory : ?
Cache Memory : ?
Set-Associative Cache (2-way)

Main Memory: \(2^5 \times 2^8 \times 2^3 = 2^{16}\) bytes 
\[= 2^6 \text{ KB} = 64 \text{ KB}\]

Cache Memory: \(2^8 \times 2^3 \times 2 = 2^{12}\) bytes 
\[= 2^2 \text{ KB} = 4 \text{ KB}\]
6.30

A) CT: [11-4], CI: [3-2], CO:[1-0]

B)

- No, Unknown
- Yes, N/A
- Yes, C0