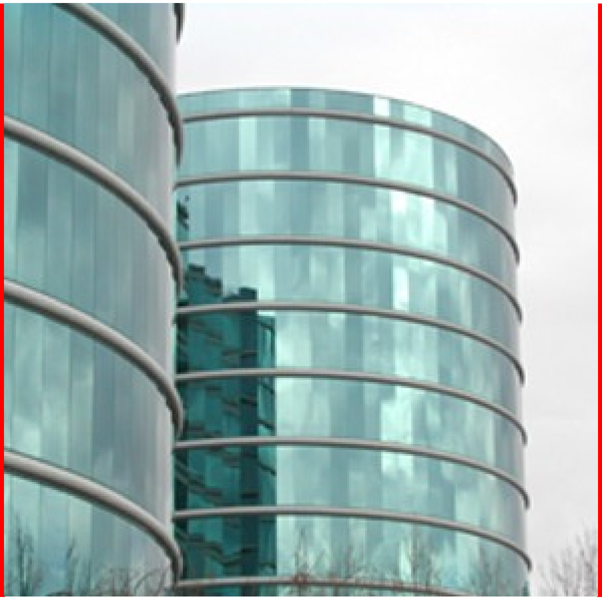
ORACLE®



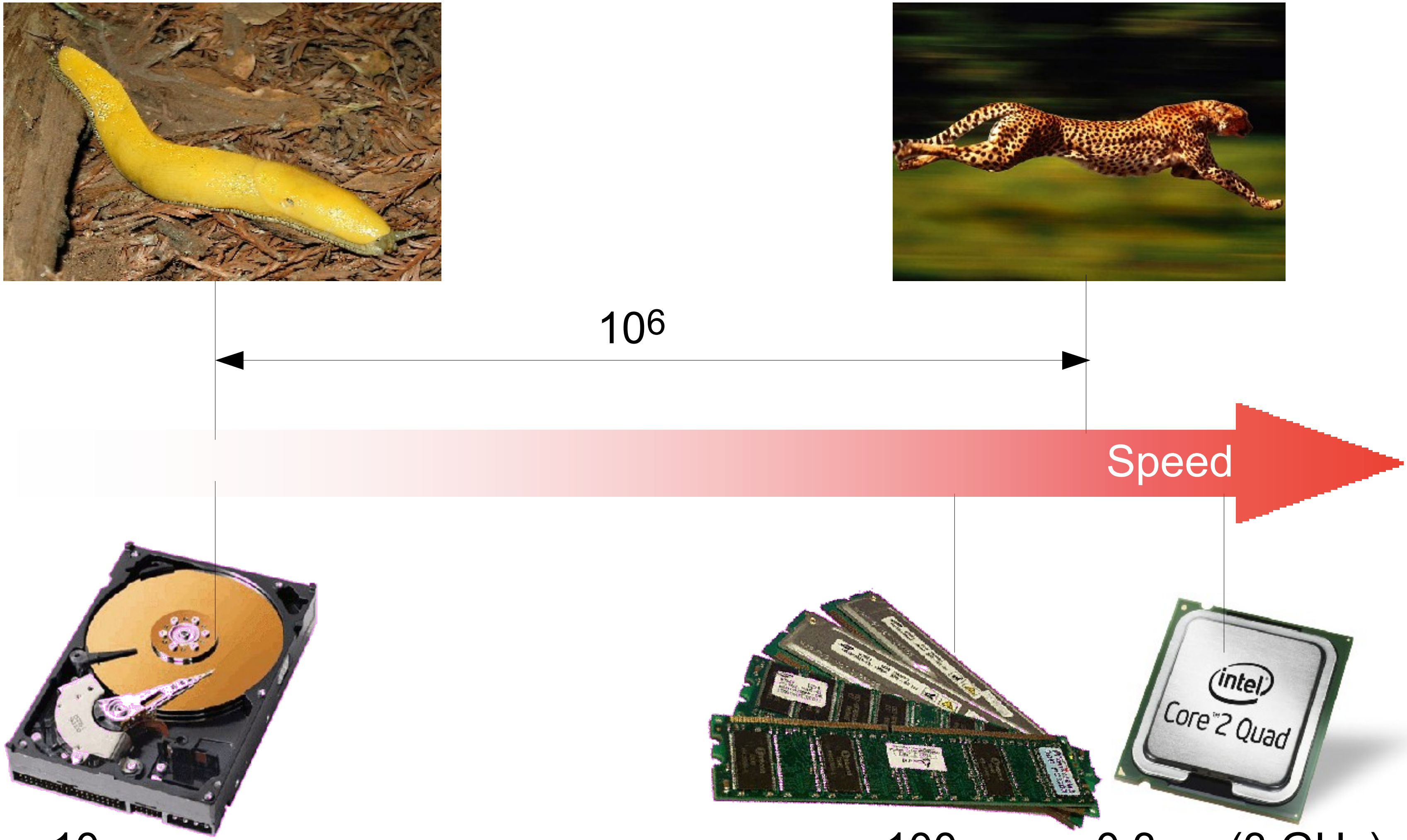
Memory as an I/O bottleneck - facts and consequences for high-performance data management

Tim Kaldewey[1,2], Andrea Di Blas[1,2], Scott Brandt[2], Eric Sedlar[1]

[1] Oracle Server Technologies – Special Projects
{tim.kaldewey, andrea.di.blas, eric.sedlar}@oracle.com
[2] University of California Santa Cruz - School of Engineering
{kalt, andrea, scott}@soe.ucsc.edu

Memory as an I/O bottleneck

High Performance Data Management



0.000047 mph

40-50 mph

10^6

Speed

10 ms

100 ns

0.3 ns (3 GHz)

High Performance Data Management

Agenda

- Introduction
  - Scope of high performance data management

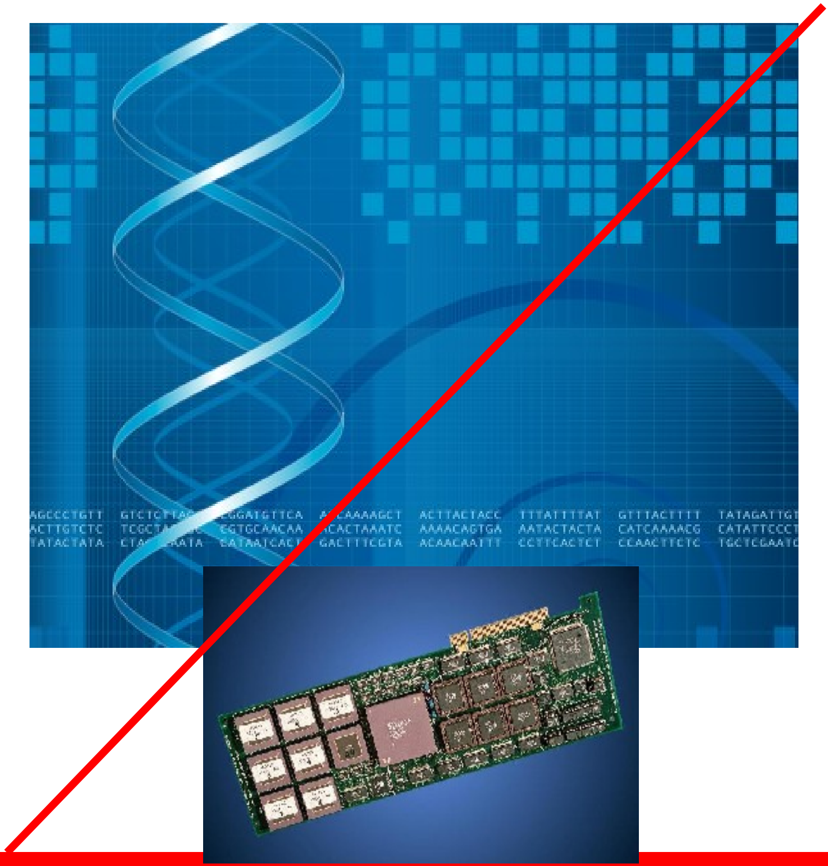
- Dissecting Memory performance
  - 4 key factors of memory performance

- Memory analysis & model – application(s)
  - FAST: Fast Architecture Sensitive Tree search
  - P-ary Search: Scalable parallel search algorithm
  - Optimizing database row formats
  - Fast string comparison
  - Fast memory management

- Conclusions

High Performance Data Management - Workloads

- Data-intensive
- Processor performance is not a problem
- Sifting through large quantities of data fast enough is



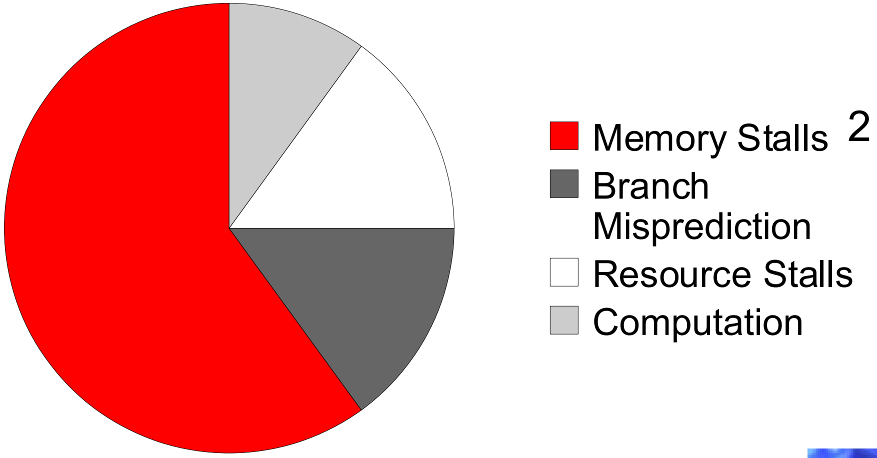
In-Memory Storage

- One application
  - Handling millions of (similar) jobs simultaneously, e.g. search engine
- Predictable performance?
  - Average response time
  - For Individual query?
  - How come this works so well?

Web Results **1** - **10** of about **12,500,000** for **latency** [definition]. (**0.07** seconds)

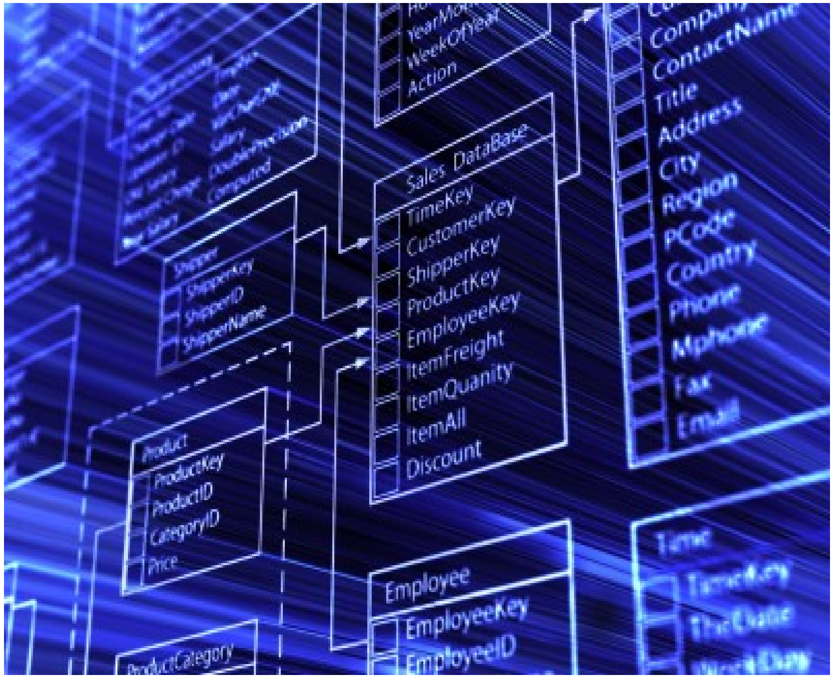
Ping	~40 ms
Disk accesses	~15 ms
Memory access	~100 ns

High Performance Data Management – “It's the memory stupid!”[1]



- Performance ~100 ns
- Predictability – multi-level caches
- Rapidly growing sizes [3]

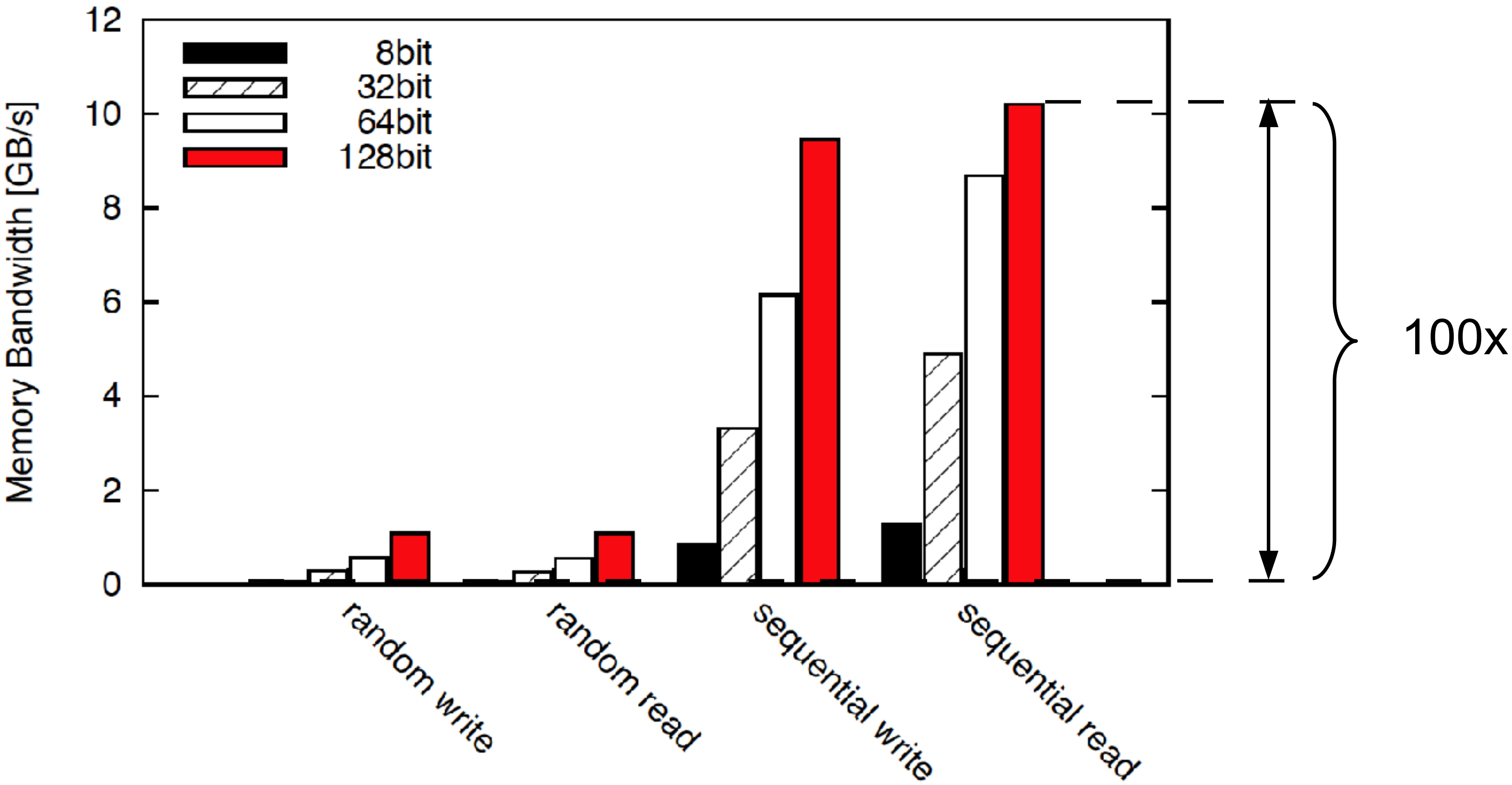
>



---

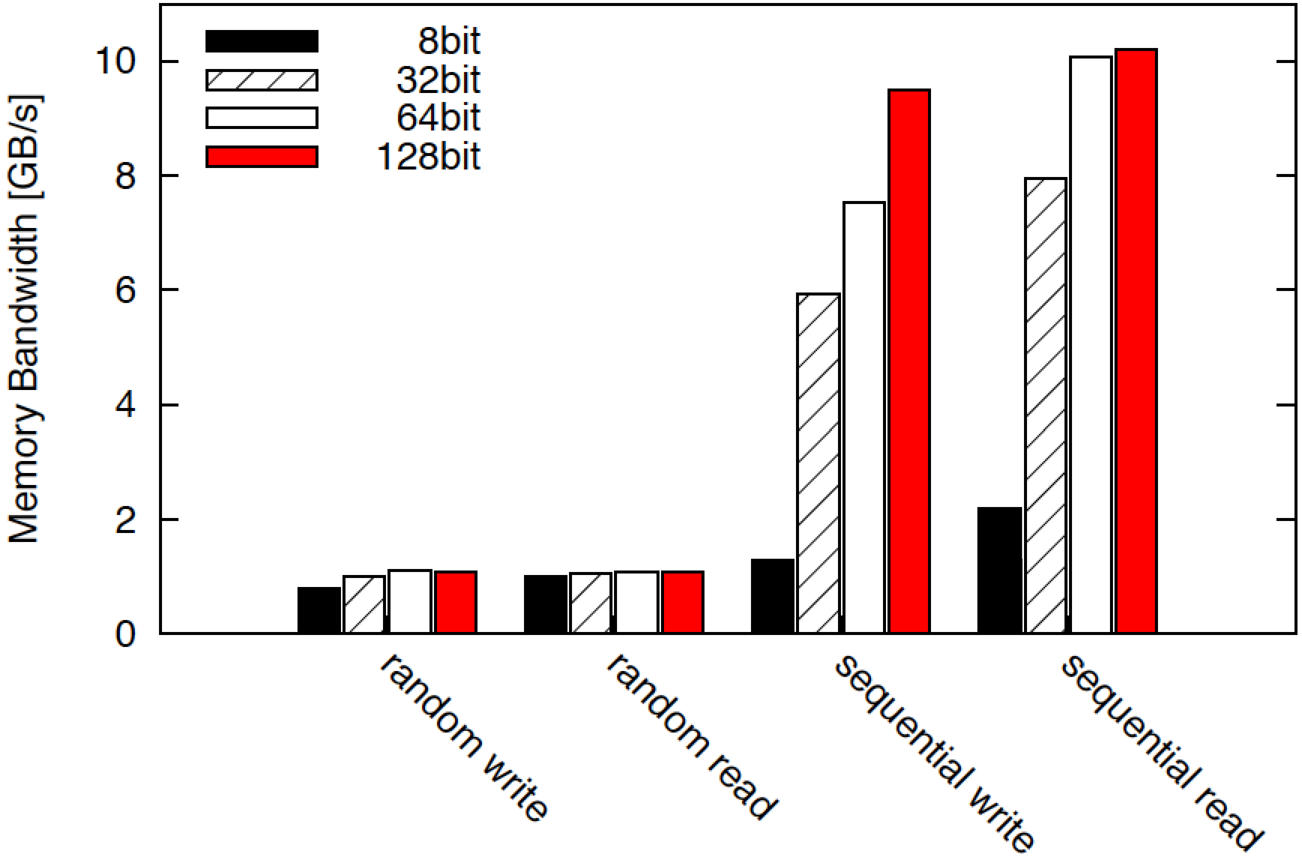
[1] R. Sites. It’s the memory, stupid! MicroprocessorReport, 10(10),1996
[2] A. Ailamaki et al. DBMSs on a modern processor: Where does time go? VLDB’99
[3] K. Schlegel. Emerging Technologies Will Drive Self-Service Business Intelligence. Garter Report 2/08

Memory Performance – Characterization



- Memory performance depends on:
  - Access pattern, method & word size

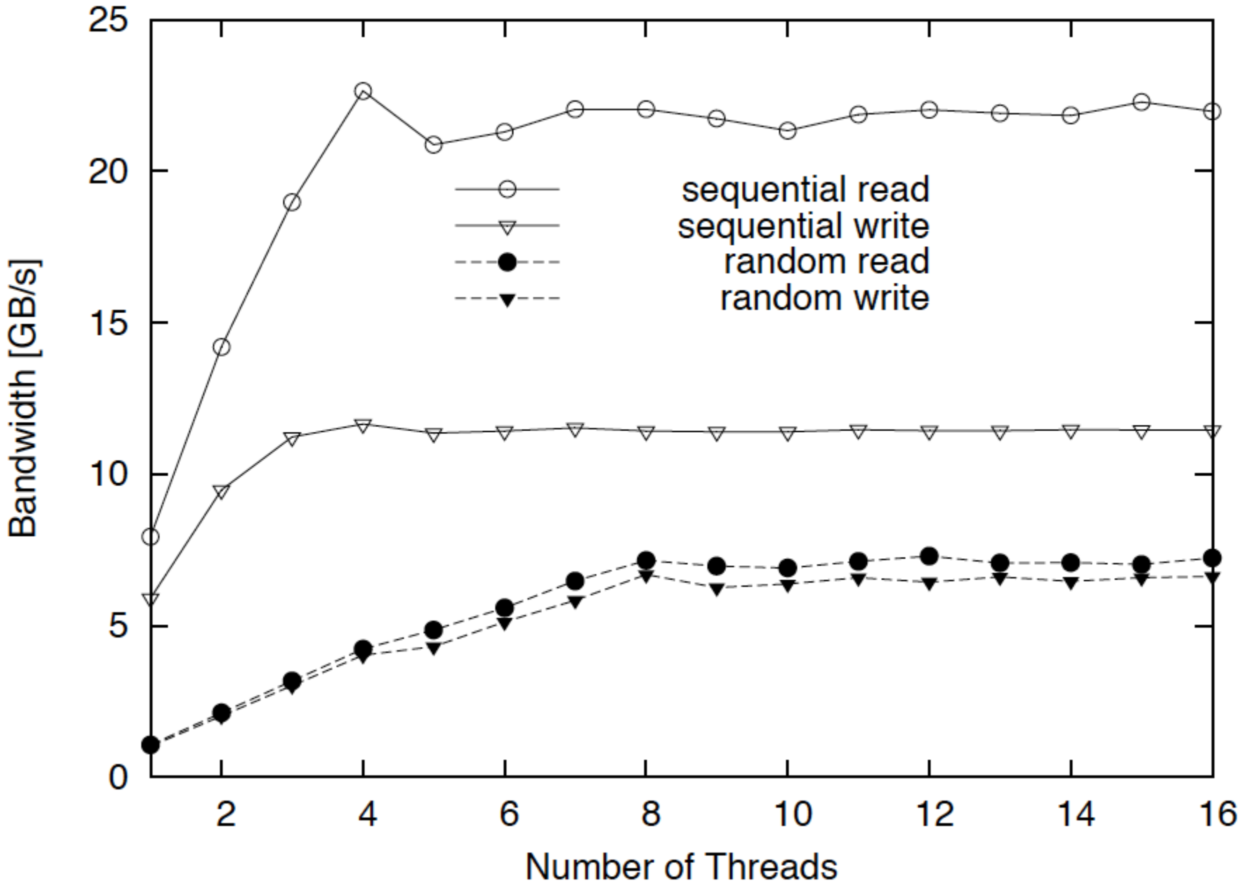
Memory Bandwidth ≠ function of #Bytes accessed



- Even if we normalize accesses to 128bit (16x8-bit, 4x32bit, 2x64bit)
- Small data types clog the memory controller !

Memory Bandwidth – Multithreading

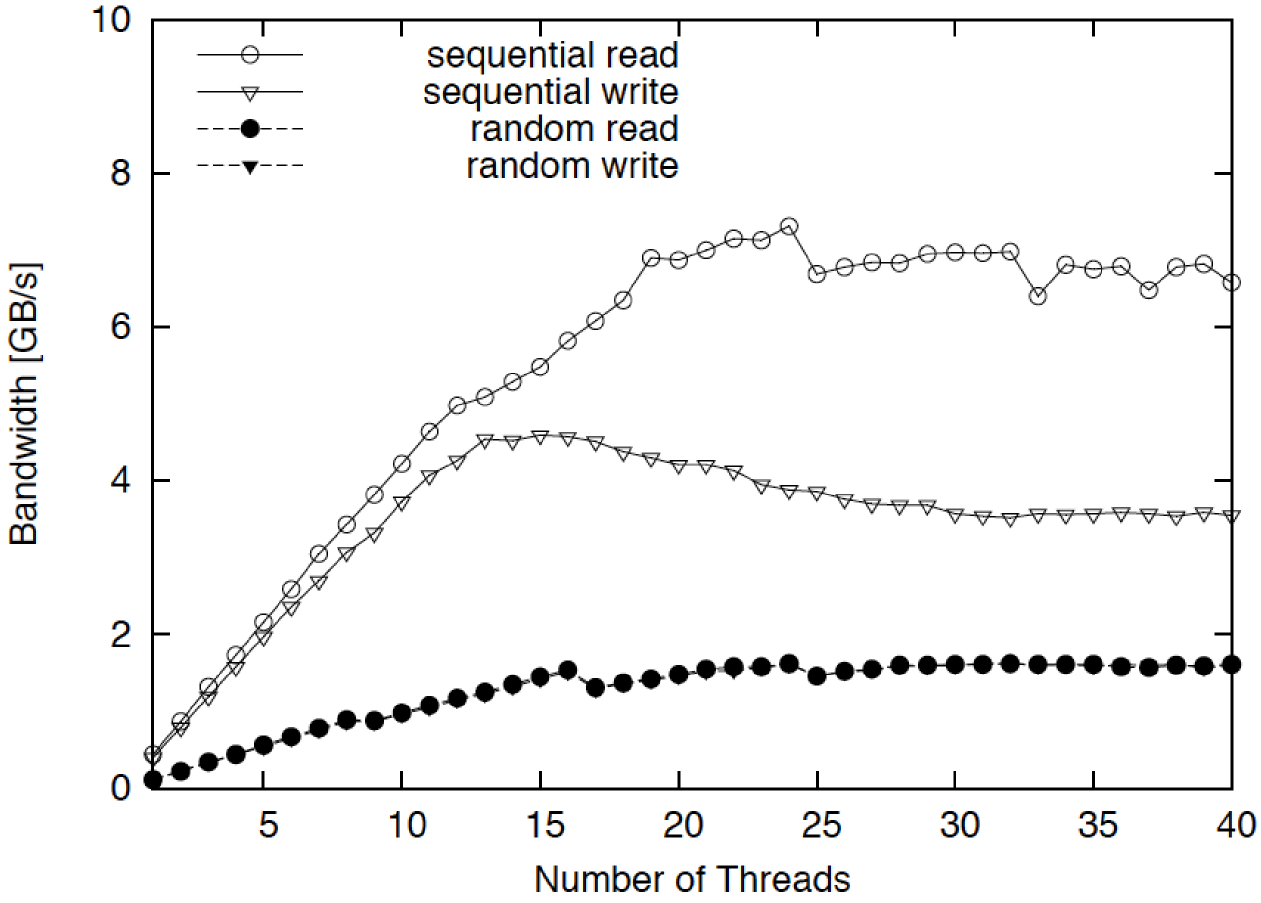
- Peak performance requires parallel memory access
- Hardware threads further accelerate random workloads



Results for a quad-core i7 2.66GHz, DDR3 1666. 32GB data accessed as 64-bit words.

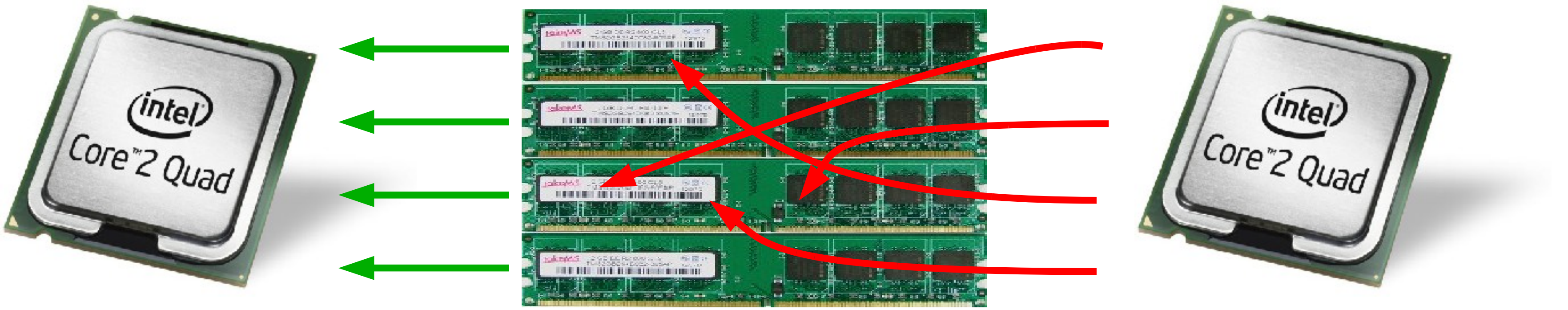
Peak Memory Performance

- Required level of concurrency depends on the target architecture



Results for an 8-core UltraSPARC T1 1.2 GHz, DDR2 533. 32GB data accessed as 64-bit words.

RAM = Random Access memory ?



Aspect	Performance impact	Reason
Access pattern	18x sequential vs. random 2x read vs. write	MTU = Cache Line (64B)
Word Size	16x 128-bit vs. 8-bit words	Memory Controller
Parallelism	32x multithreaded vs. serial	

- How can we exploit that knowledge?
  - To predict application performance .... *a paper in the works*
  - Accelerate data intensive apps .... e.g. search

Why Search ?

Honestly, how many times a day do you visit

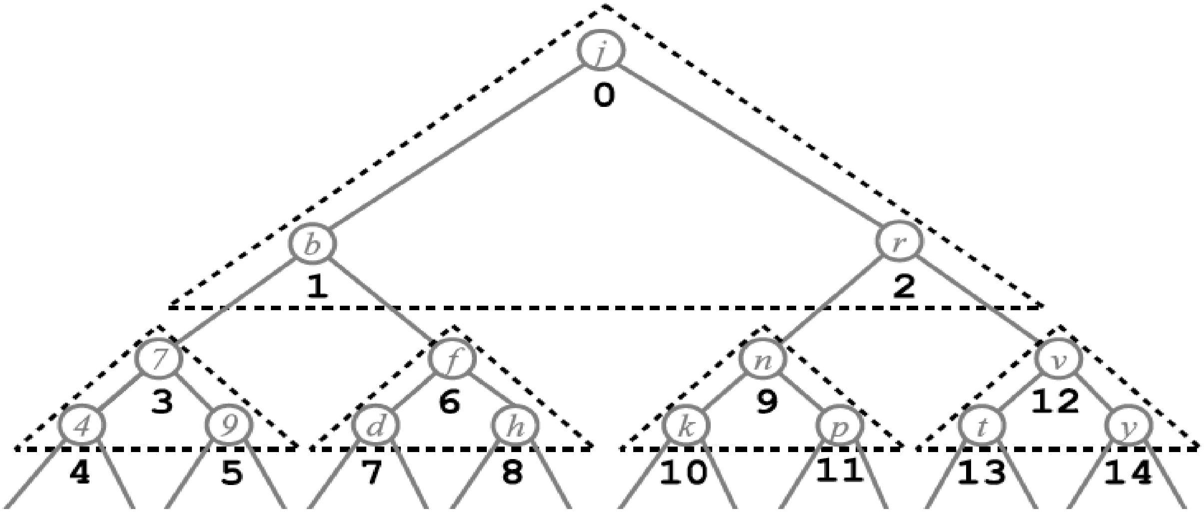
Google™

YAHOO!®

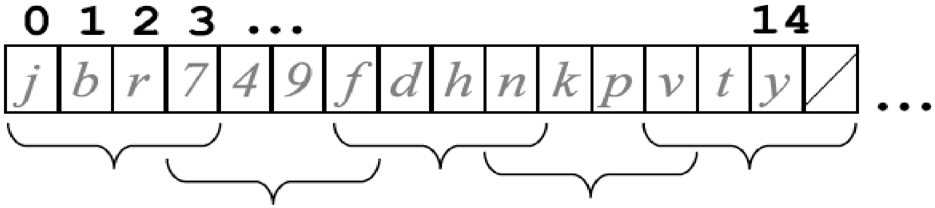
?

FAST: Fast Architecture Sensitive Tree Search [4]

- Starting from a binary tree produce a memory/access optimal layout



- Parent and children stored linearly: use SSE vectors for load & compare



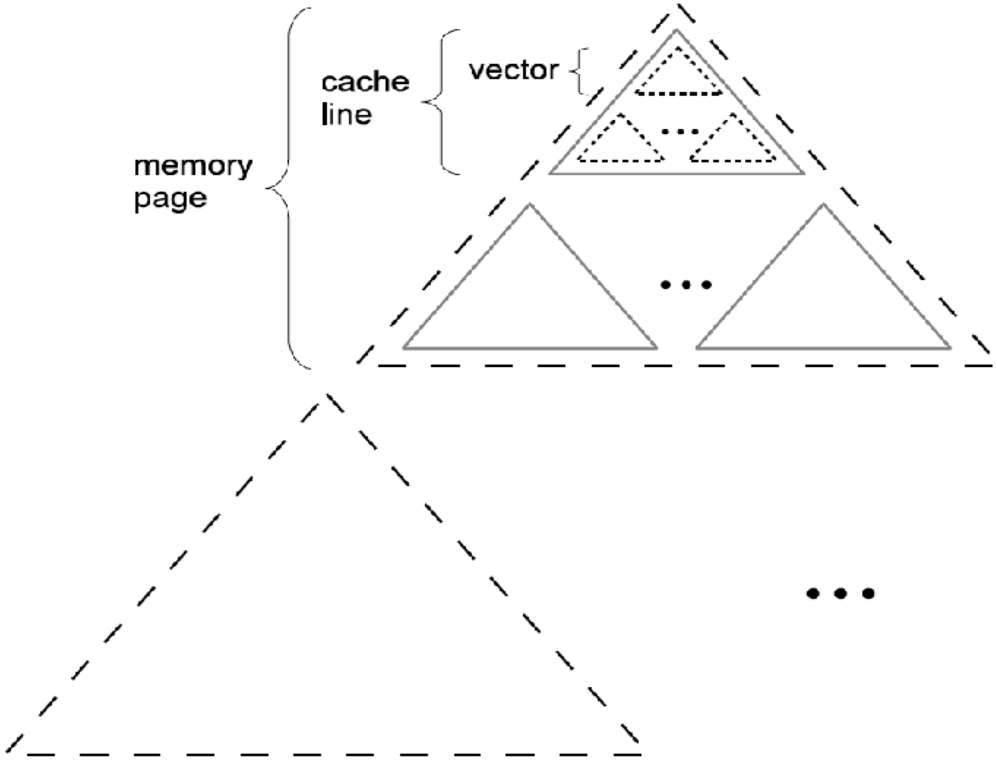
- Can place 4 tree levels in one cache line
- Require only 2 memory requests to access for 4 tree levels

---

[4] C. Kim, Jatin C., N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD'10. *Best paper award*

FAST: Fast Architecture Sensitive Tree Search

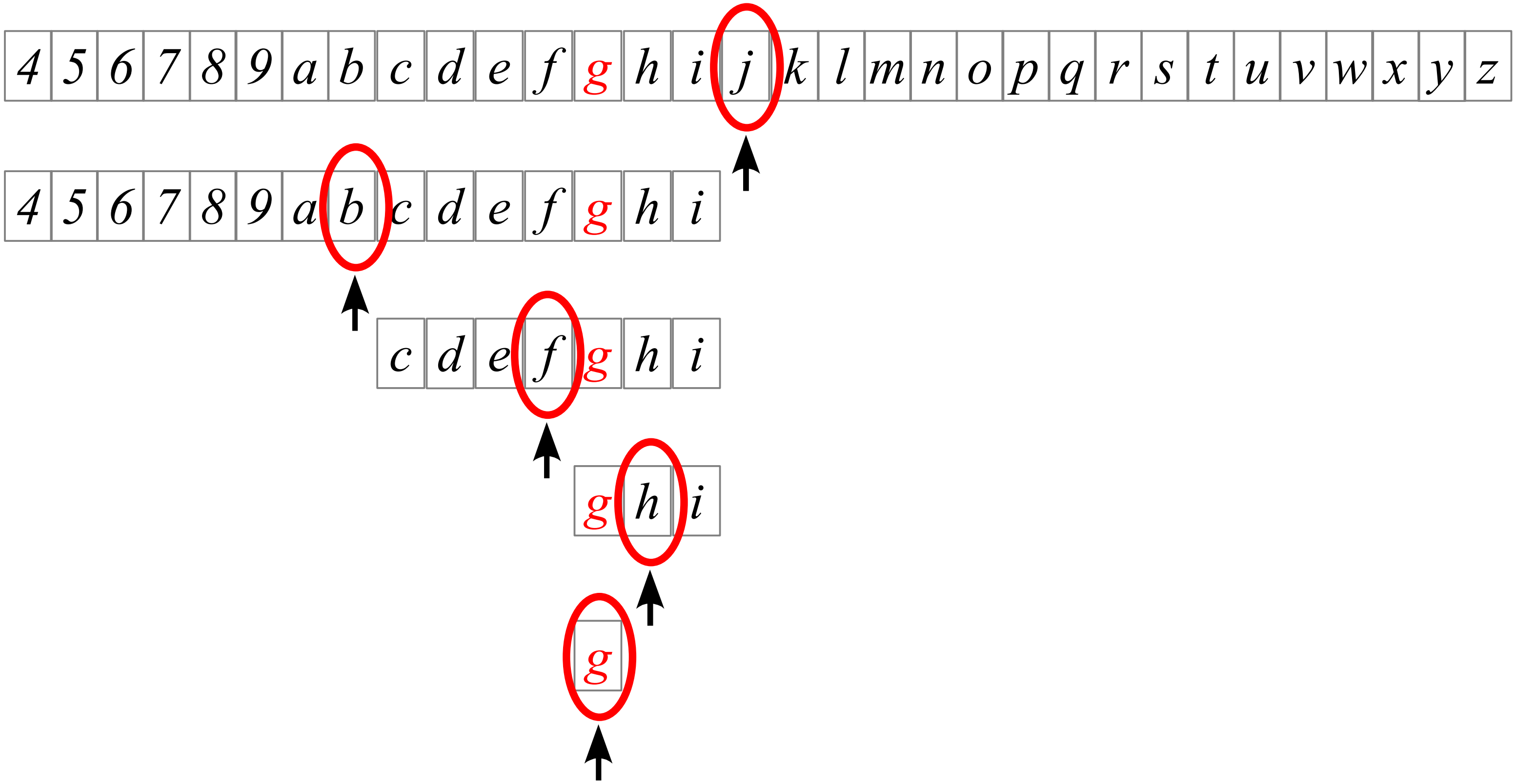
- At a larger scale consider page size to minimize TLB misses



- Can outperform a conventional implementation by 2.5x
- The downside:
  - Accesses get very complicated (15 nodes per cache line)
  - Updates are even worse (rebalancing)

Databases usually use inverted lists or B-trees

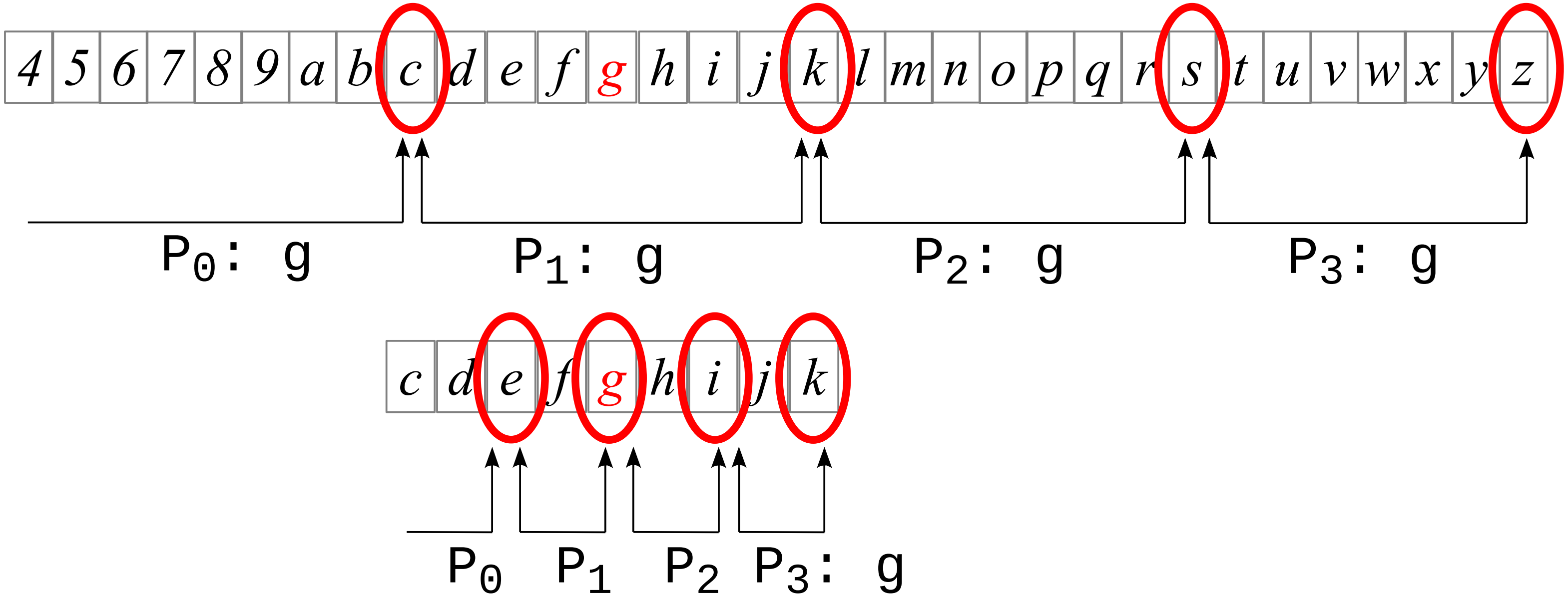
- Accelerate search on existing data structures?
- Binary search has random access pattern and is inherently serial



- Complexity $\log_2(n)$ iterations = # random memory accesses
  - If we could parallelize the memory accesses ....

Parallel Search

- Divide an conquer using multiple threads/processors (4)
  - Still random but parallel memory access
  - Pivot elements can be shared and lower bound set to -∞



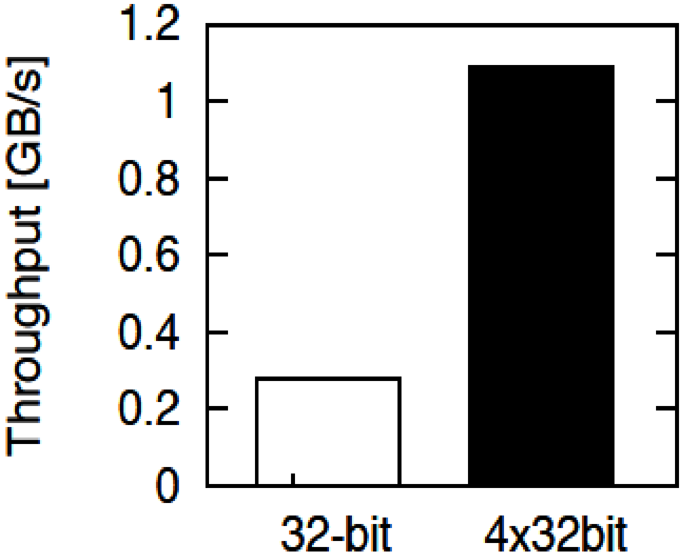
- We called this approach P-ary search[5] – P for the # of processors
- Complexity $\log_p(n)$ = # concurrent random memory accesses
- B-trees do nothing else but co-locate pivot elements
  - memory accesses are now sequential =)

---

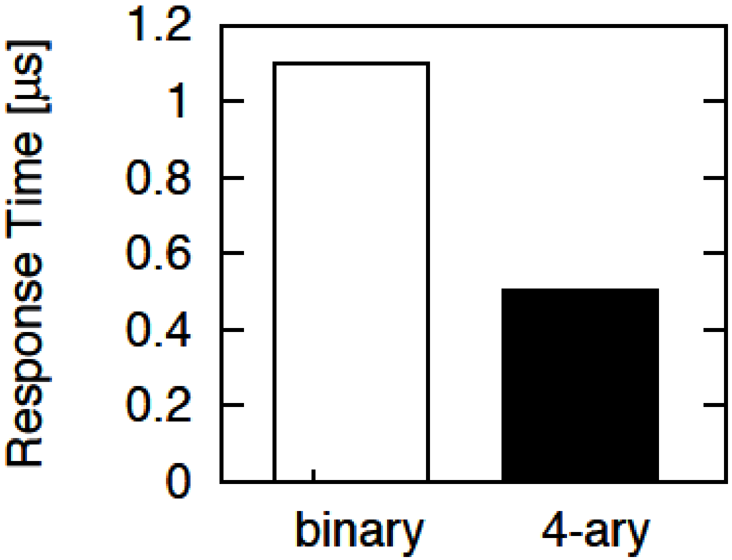
[5] T. Kaldewey, A. Di Blas, Jeff Hagen, Eric Sedlar Parallel Search onVideo Cards. USENIX HOTPAR'09

P-ary Search – Response time

- CPU Implementation using x86 SSE vectors
- Binary vs. P-ary search



Random access memory bandwidth,
Core i7 quad 2.66GHz, DDR3 1666.

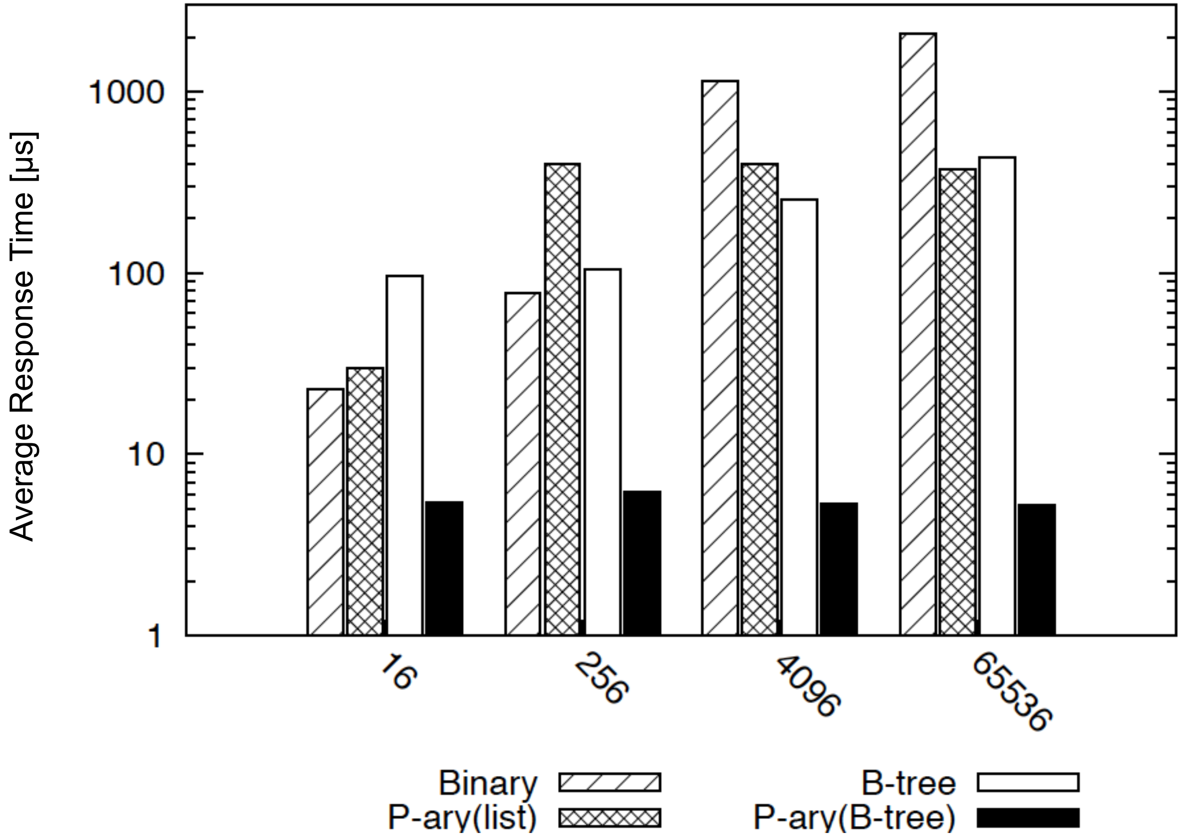


Searching a 512MB data set with
134mill. 4-byte integer entries

- Strong correlation of memory and application performance
  - Exploiting parallel memory access to improve response time

A Glimplse into the parallel future

- 32-ary Search on a GPU
  - Response time is workload independent



Searching a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b (240 cores), 1.5GHz, GDDR3 1.2GHz.

Database Row Format(s)

- Consider a table with 500 columns stored by rows
  - We would like to get the value(s) in column #250 (= an average access)
- Consider the following data structures:
  - Option 1: 1 Byte storing column width/length followed by data, default in many (old) databases

3	a	b	c	2	e	f	4	h	i	j	k	9	m

…

  - Option 2: Grouping column width' and data

0	3	2	4	9

…

a	b	c	e	f	h	i	j	k	m

…

  - Option 3: An array of 2 Byte column offsets followed by data

0	3	5	9	18

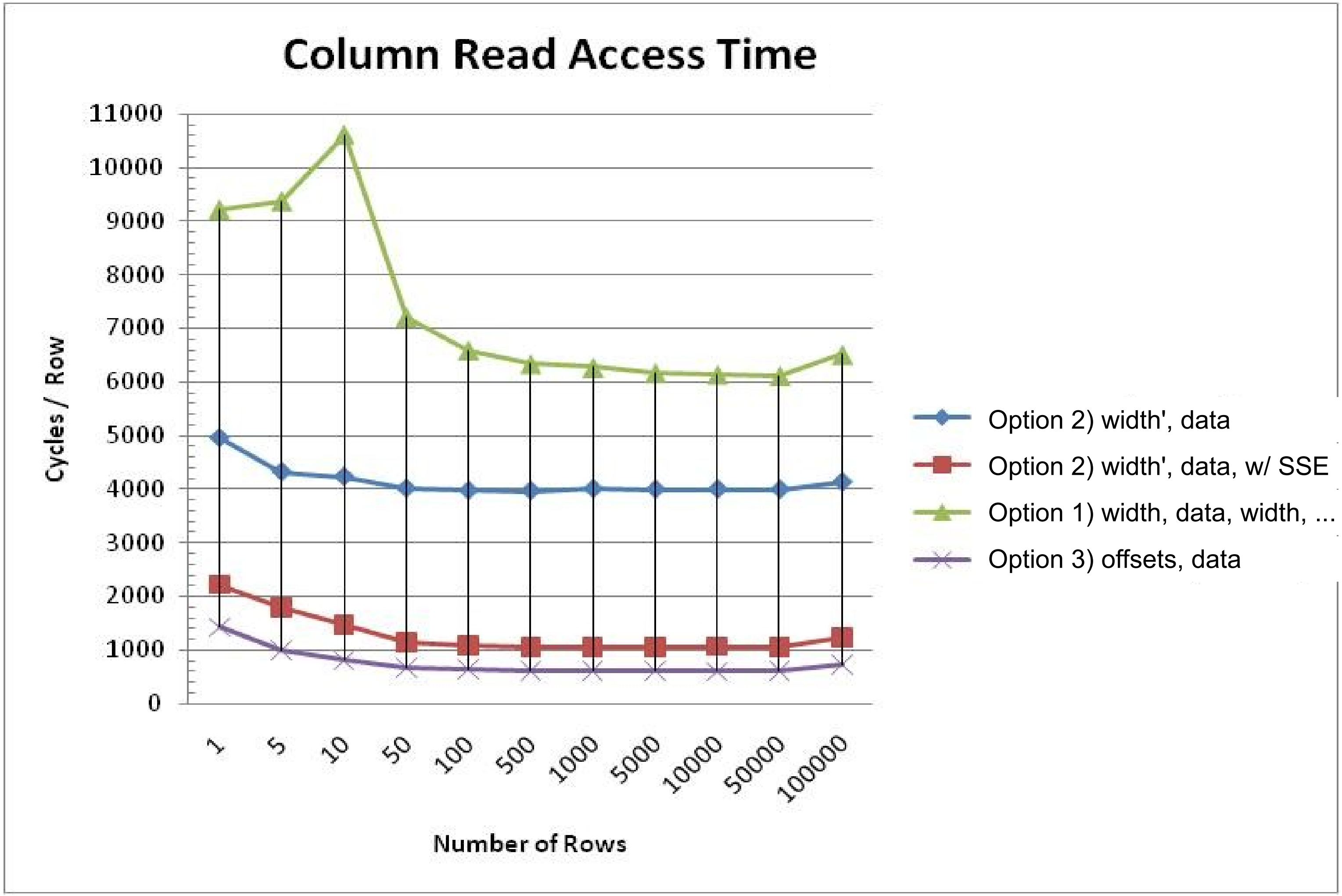
…

a	b	c	e	f	h	i	j	k	m

…

Database Row Format(s)

- 500 column table – rowstore, tablescan – accessing column 250

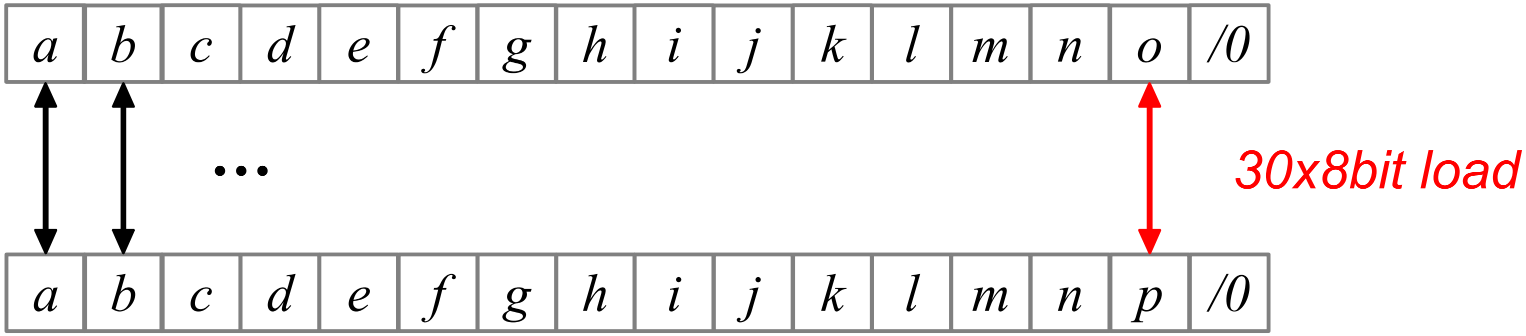


ORACLE

String Comparison

- Search naturally requires MANY comparisons
- The strcmp() library function:

```
int strcmp(const char* s1, const char* s2){
      while (*s1 == *s2++)
            if (*s1++ == 0)return 0;
      return (*s1 - *(s2 - 1));
}
```



a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	/0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	p	/0

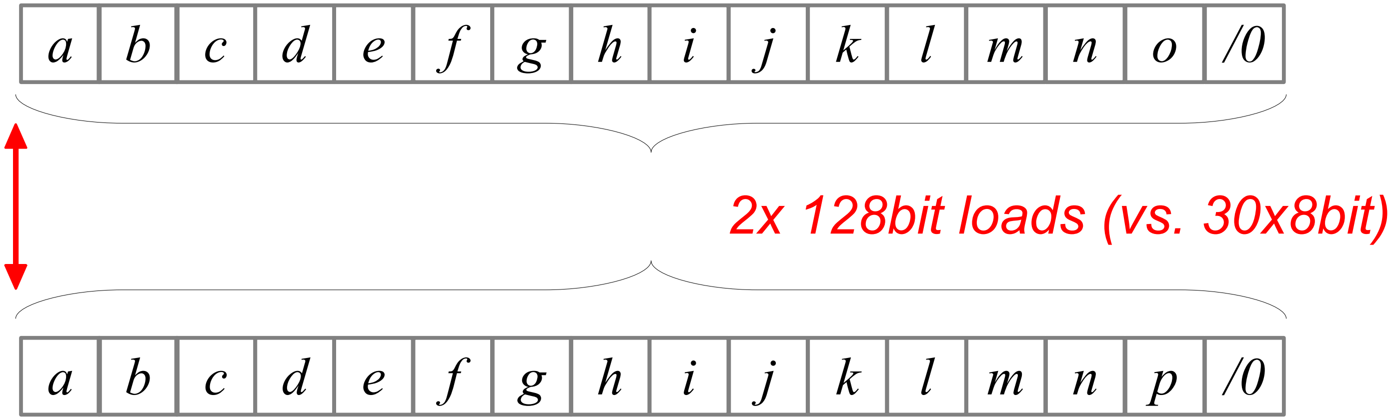
…

30x8bit load

- Byte-wise memory access is slow

String Comparison

- Strcmp as an example for efficient Vector computing



a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*/0*

2x 128bit loads (vs. 30x8bit)

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*p*	*/0*

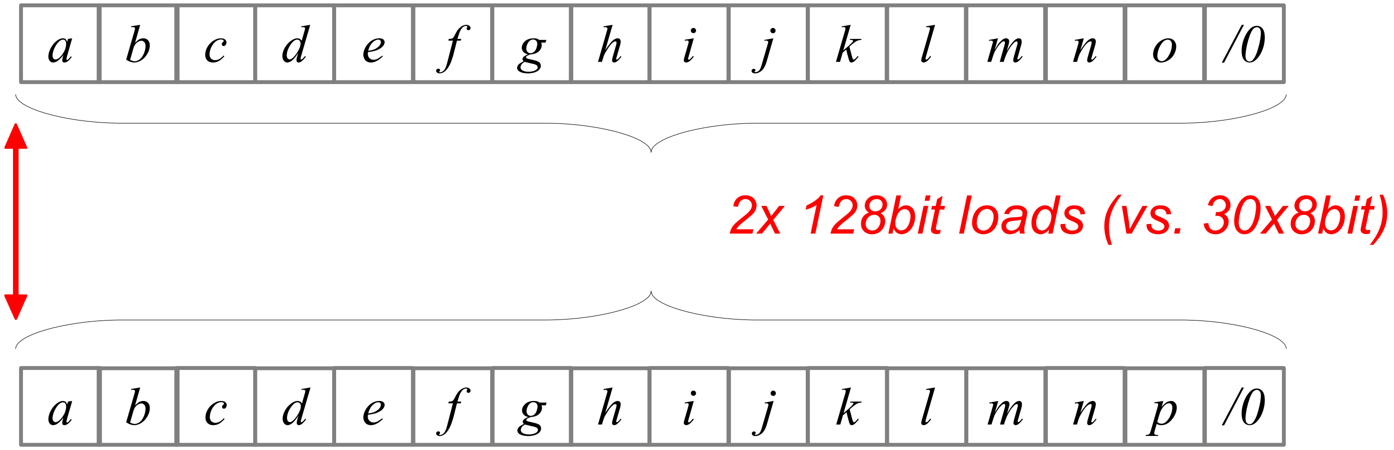
```
int SSEstrcmp (const void* s1, const void* s2) {
   int r=0;
   int gt=0;
   __asm__ volatile(
                // load strings s1,s2 into 128bit xmm regs
                " movdqa (%[s1]), %%xmm0\n\t"
                " movdqa (%[s2]), %%xmm1\n\t"
                // save the second for later
                " movdqa %%xmm1, %%xmm2\n\t"
```

Vectorized strcmp()

```
// s1==s2 ?
" pcmpeqb %%xmm0, %%xmm1\n\t"
// bitmask reduction to 16 bit
" pmovmskb %%xmm1, %[r]\n\t"
// We now have one bit set for each matching byte in s1,s2
// Now bit mask a match to 0(=found)
" xor $0xFFFF, %[r]\n\t"
// if all 0 then s1=s2 and we are DONE
" je end\n\t"
//s1>s2
" pcmpgtb %%xmm2, %%xmm0\n\t"
// reduce result to 16 bit
" pmovmskb %%xmm0, %[gt]\n\t"
// shift left, because there is a 0th byte
" shl $1, %[gt]\n\t"
" shl $1, %[r]\n\t"
// scan for the first > bit set
" bsf %[gt], %[gt]\n\t"
// scan for the first = bit set
" bsf %[r], %[r]\n\t"
// if they are the same s1>s2 [r>0 already]
" cmp %[r] ,%[gt]\n\t"
" je end\n\t"
// s1<s2, hence return -1
" mov $-1, %[r]\n\t"
"end:\n\t"
```

String Comparison

- Strcmp as an example for efficient Vector computing



a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*/0*
a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*p*	*/0*

2x 128bit loads (vs. 30x8bit)

- Larger word-size reduce # memory requests
- Measured speedup of up to 2x
- Latest CPU generation (Core i7) has a builtin strcmp instruction **`pcmpstri`**
  - Marginal performance gain as #memory access does not change

Fast memory mangement

- Our research suggests that larger word sizes yield better performance
  - Rewrite **`memcpy`**, **`memcmp`**, **`memset`**, ... ?
  - … or wait for a new compiler

```
typedef struct filepath{
   unsigned char  file_1[64];
   Unsigned char path_1[512];
}filepath_t;

int foo(){
   char str1[]="Sample string";
   char str2[40];
   filepath_t sth;
   memset((void*)&(sth),0,
             sizeof(sth));
   memcpy(str2,str1,40);
```

```
foo:
...
 movaps    %xmm0, -16(%esp,%eax)
 movaps    %xmm0, -32(%esp,%eax)
 movaps    %xmm0, -48(%esp,%eax)
 movaps    %xmm0, -64(%esp,%eax)
 subl      $64, %eax
 jne       ..B1.7
...
 movaps    576(%esp), %xmm0
 movaps    %xmm0, (%esp)
 movaps    592(%esp), %xmm1
 movaps    %xmm1, 16(%esp)
 movsd     608(%esp), %xmm2
 movsd     %xmm2, 32(%esp)
```

- Parallel memory accesses yield even better performance
  - Parallelize **`memxxx`**?

Conclusions & Ongoing Work

- We have shown the keys determinants of memory performance:
  - Access pattern: sequential vs. random
  - Access method: read vs. write
  - Word size
  - Concurrency

- This research inspired
  - Fast Architecture Sensitive Tree Search [4]
  - P-ary (K-ary) search, scalable parallel search algorithm(s) [5,6]
  - Parallel hash and sort-merge join to reduce query response time [7]
  - Optimizing database row formats
  - Several product optimizations: fast memcpy, SSE strcmp, ...

- Ongoing work
  - Further exploring the concept of parallel memory access to reduce query response time on parallel architectures [8]
  - Technology Transfer(s)

---

[4] C. Kim, Jatin C., N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. ACM SIGMOD'10. Best paper award

[5] T. Kaldewey, A. Di Blas, Jeff Hagen, Eric Sedlar Parallel Search onVideo Cards. USENIX HOTPAR'09

[6] B. Schlegel, R. Gemulla, and W. Lehner. K-ary Search on Modern Processors. In DaMoN'09

[7] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. Nguyen, A. Di Blas, V. Lee, N. Satish, P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. VLDB'09

[8] T. Kaldewey, A. Di Blas. Large Scale GPU Search. GPU (Computing) Gems, Volume 2. to appear

Questions ?

Disclaimer

The author's views expressed in this presentation do not necessarily reflect the views of Oracle.

ORACLE IS THE INFORMATION COMPANY

ORACLE®