

VEE 2008 7th Mar 2008

Cell GC:
Using the Cell Synergistic Processor
as a Garbage Collection Coprocessor

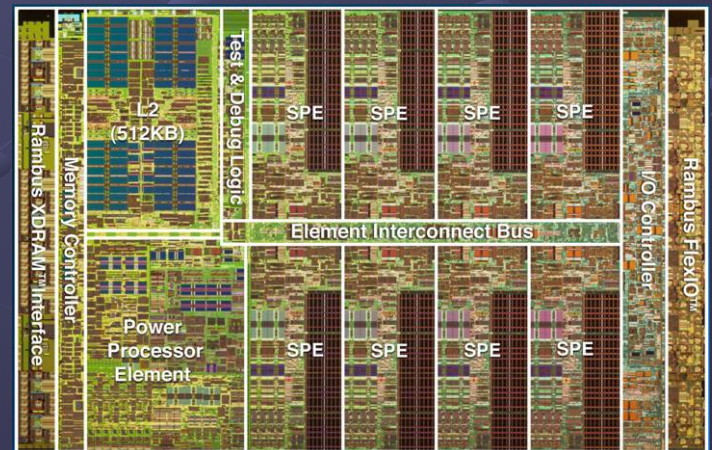
Chen-Yong Cher & Michael Gschwind
IBM T. J. Watson Research Center,
Yorktown Heights, NY 10598
{chenyong, mkg}@us.ibm.com

CELL Processor

- is a novel heterogeneous CMP architecture
- Game consoles, supercomputers & servers
- Designed to give programmers more control
- Best for dense, parallel computations
- How well can it run generic programs?

Key Idea:

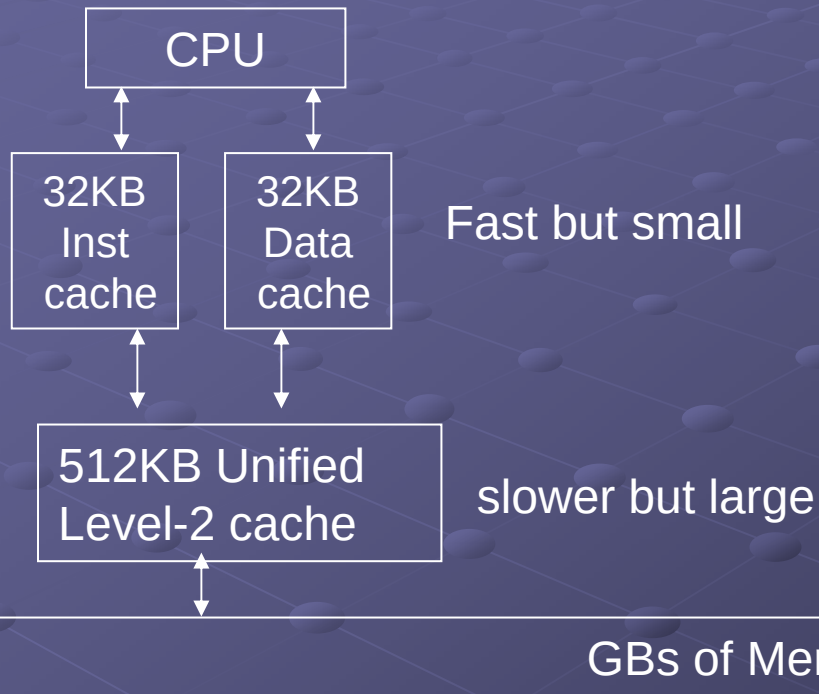
In this work, we explore offloading garbage collection from the host processor to the SPU processors, freeing host processors for other computations using this extra control.



3.2 Ghz, 1 PPU core + 8 SPU cores

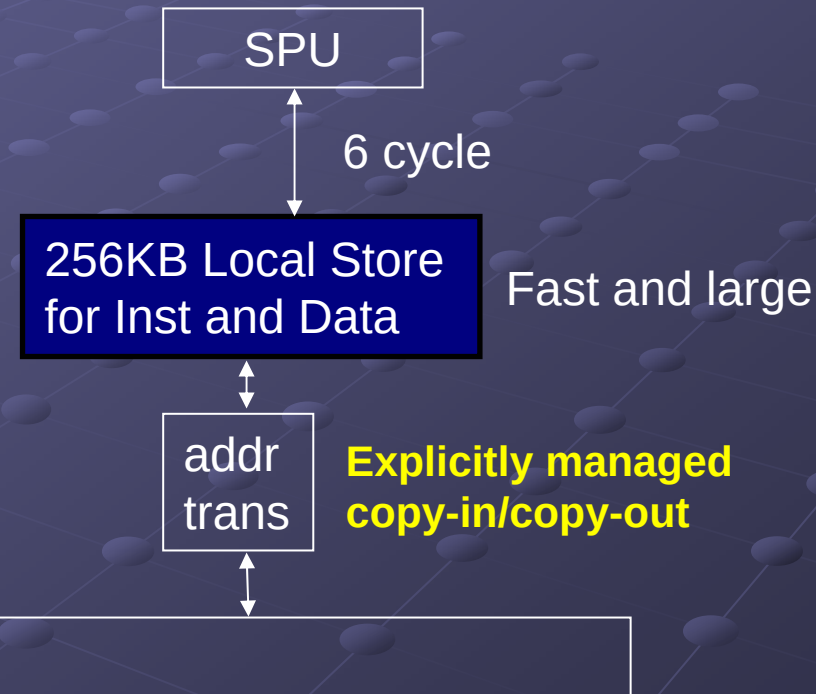
Illusion of Flat Memory

Conventional core



Coherency transparent to programmer

Cell synergistic processor unit (SPU)



No illusion of memory space
Explicitly managed by programmer
Non-blocking DMA in **large chunk**

1. **Explore SPU offload of GC**
2. **Identify GC-specific optimizations in Local Store**

Contributions

- Identify synchronization for offload GC to SPU
- Managing liveness coherency btw App / GC
- Quantify effects of software caches
- Identify best caching scheme
 - LS & DMA gives programmer control over caching
- Quantify previous GC prefetching schemes

Cell GC – what it does

- Implemented in GC library
 - Java programs benefit without reprogramming
- Offload GC from PPU to SPU
- Application pauses for SPU-GC (stop-the-world)
 - Allow other threads to run while GC offloaded
- Future work: parallel GC concurrent with mutator
 - note: mutator never pauses, therefore incremental or generational not needed.

Outline

1. Intro: offloading of GC to SPUs
2. Coherency btw local store and memory
3. Optimizations in the local store
4. Results of Cell GC
5. Comparisons of PPU and SPU

Mark-Sweep Garbage Collector

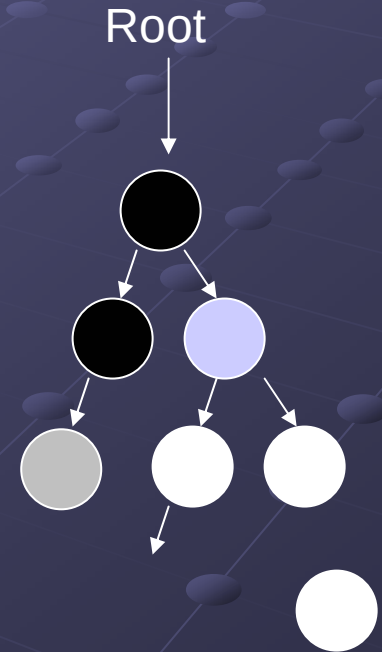
● Good properties

- In-place GC
- Handles cyclic structures
- Lazy-sweeping
- Traverses only live objects

Porting GC to SPU with local store

Three data structures needed for GC: **Mark Stack**, **Heap Blocks** and **Headers**.

```
while (ptr = pop_mark_stack()) <- read Mark Stack  
  length = alloc_size(ptr);  
  for (i=0...length)  
    p = ptr[i]; <- read Heap Block  
    if (legal_ptr(p))  
      get_hdr(p); <- read Header pointer (hash table)  
      if (not_marked(p)) <- read Header  
        mark(p); <- write Header  
        push_mark_stack(p); <- write Mark Stack
```



Coherency model

● Heap Blocks

- Read-only from MEM in size of 1KB

● Mark Stack

- Master copy in MEM
- Written back at end of SPU-GC or when spill
- Spill infrequent

● Header Blocks/Mark Bits

- Master copy in MEM
- Written back at end of SPU-GC or when spill

Heap Blocks dominate misses.

Cell GC: four ways to read heap blocks

- Pointer-Chasing (base case)
 - Most bandwidth efficient
- Software Cache
 - Emulate a hardware cache in local store
 - Brings in nearby data that are hopefully useful
 - Adds instruction overhead to accesses
- Operand Buffer
 - Fetches a large block into local store
 - No additional overhead to accesses
- Hybrid of software cache and operand buffer
 - Selects which method based on locality patterns
 - Read-only property avoids issues of data coherency

Software Caches

Remember hardware cache:
Data Address split into three:



Used to select the right data in data cache, example:



- Can be implemented as software cache in local store!
- Fast tag checks using the four-way SIMD instructions reduces overhead to accesses [IBM Systems Journal 2006]

Flexible use of Local Store

- Things you can do with local store/SW\$
 1. Split I and D storage based on demand
 - Executable 20KB, the rest used by data
 2. Further split into operand buffer and SW\$
 3. Configure line size of software cache
 4. Programming prefetching & capitulative load

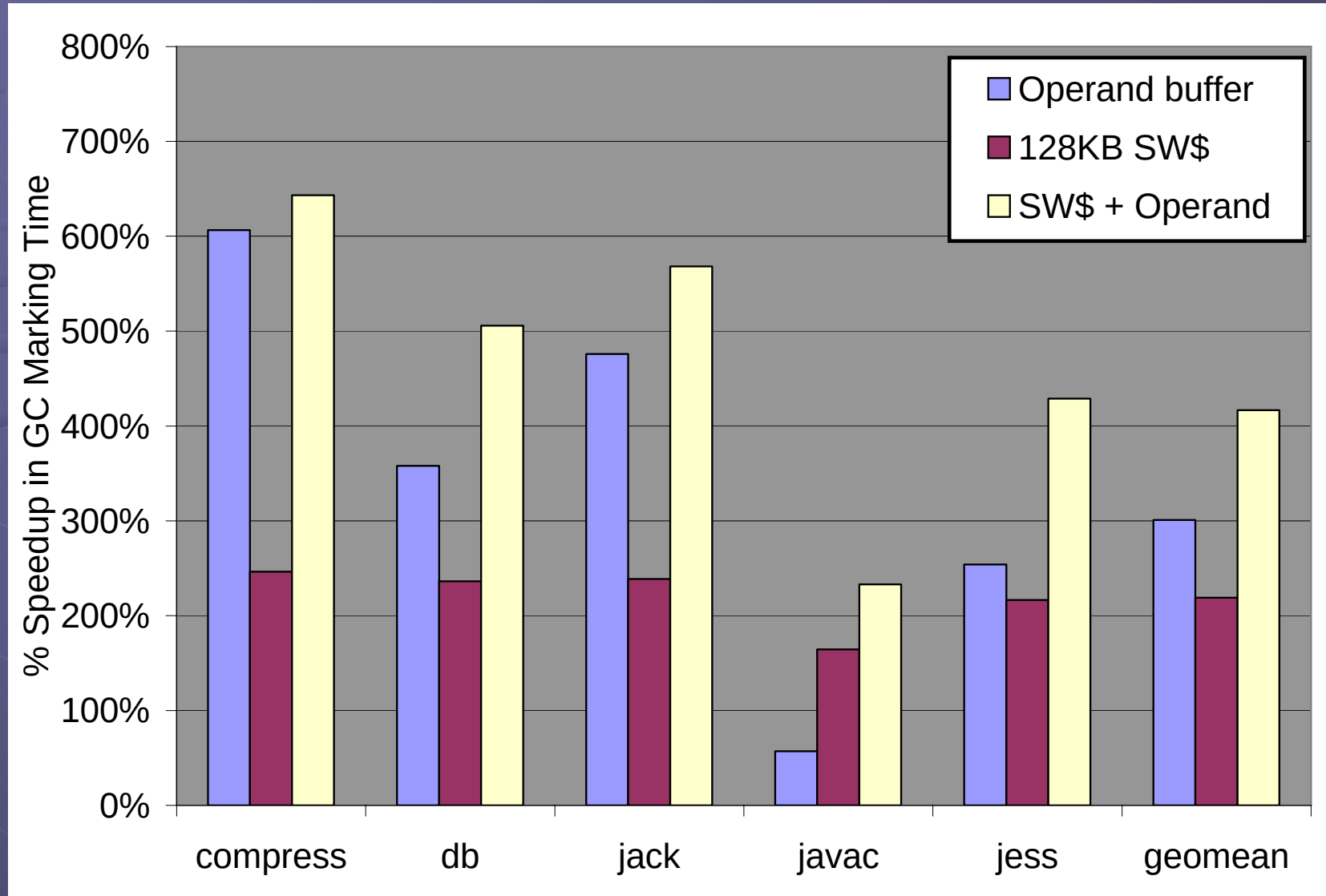
Outline

1. Intro: offloading of GC to SPU
2. Coherency btw local store and memory
3. Optimizations in the local store
4. Results of Cell GC
5. Comparisons of PPU and SPU

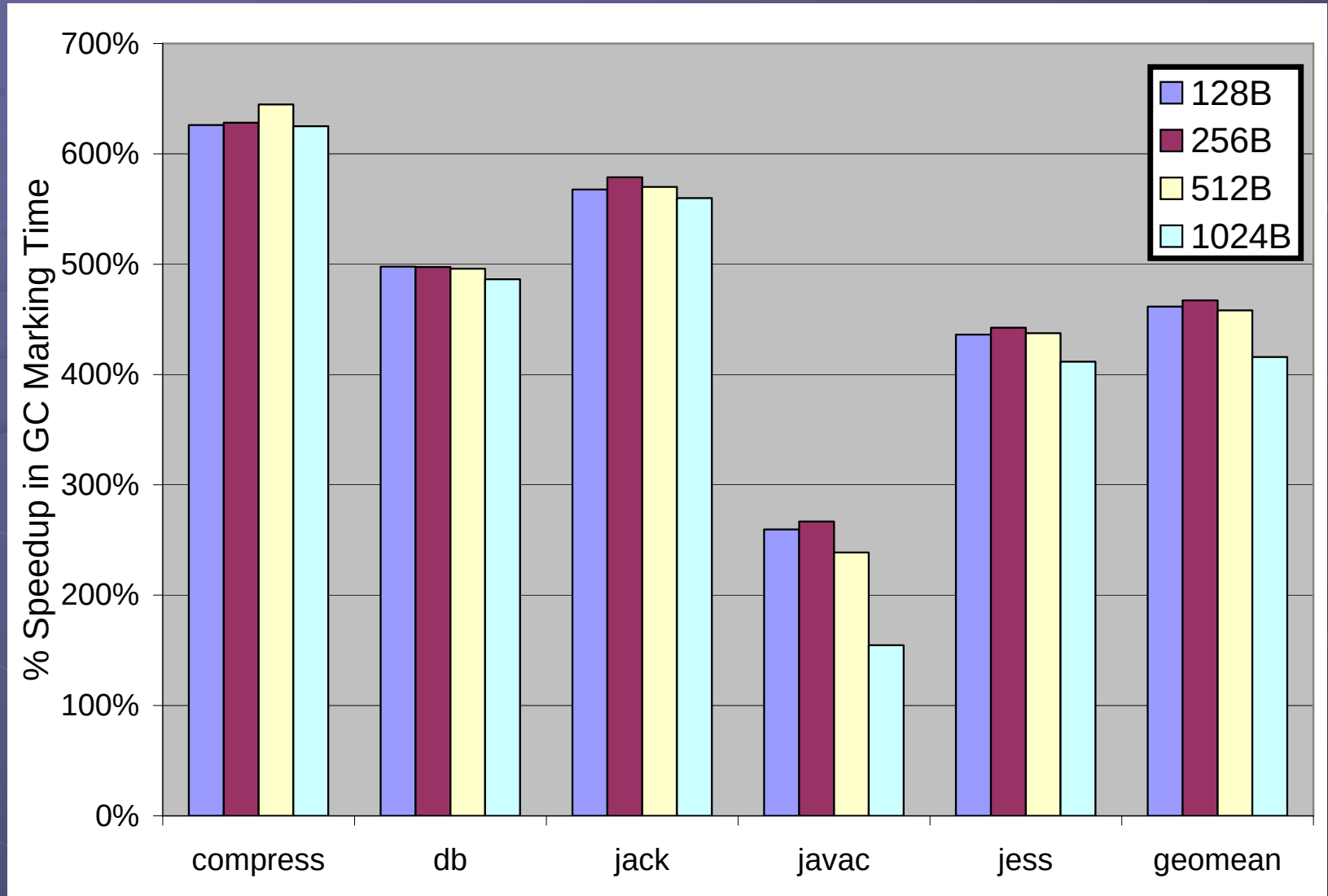
Methodology

- Boehm-Demers-Weiser (BDW) Mark-sweep GC
 - Shipped with GCC 4.1.1
- Cell Blade of two 3.2GHz Cell BE chips, 1GB memory
- Heap sizes/GC config: out-of-the-box
- Redo GC on PPU to cross check with SPU GC results
- Local Store Usage
 - 20KB executable binary
 - 32KB LS mark stack
 - 40KB Header cache
 - 128KB software cache for heap blocks (512B line)
 - 4KB Operand Buffer

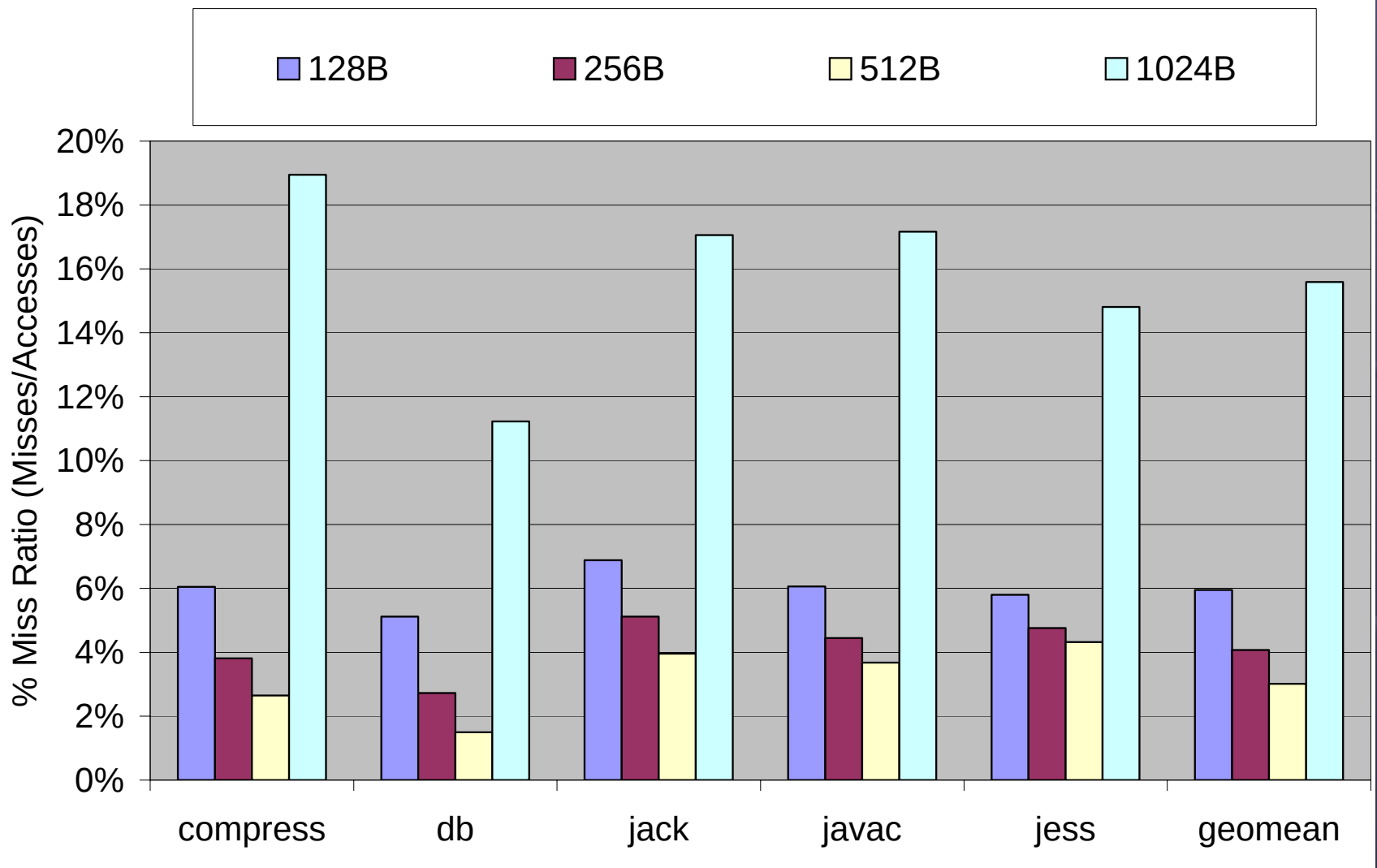
Speedups for various caching schemes



Varying Line Size of Software Cache



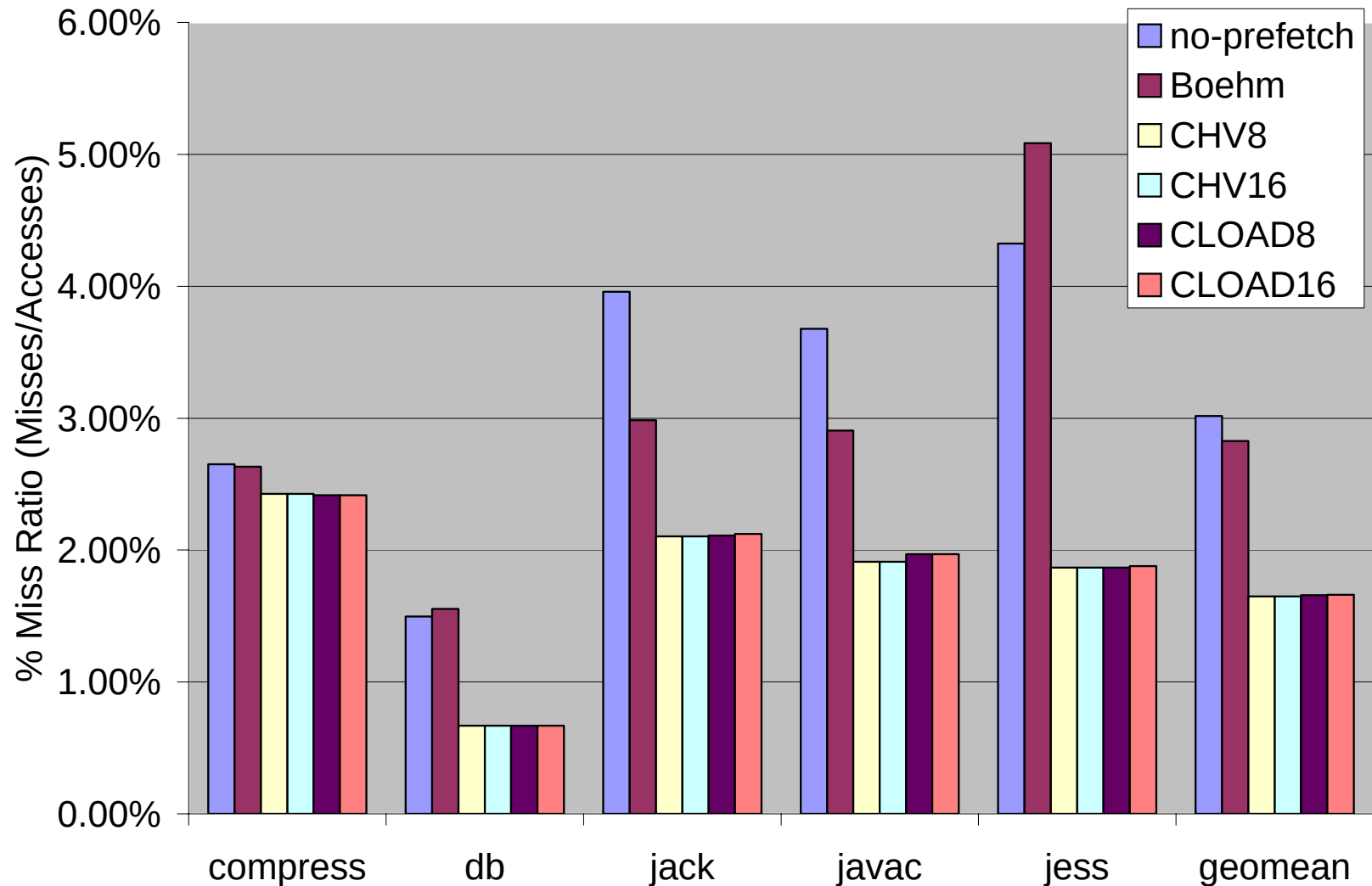
Miss rates of Varying Line Size



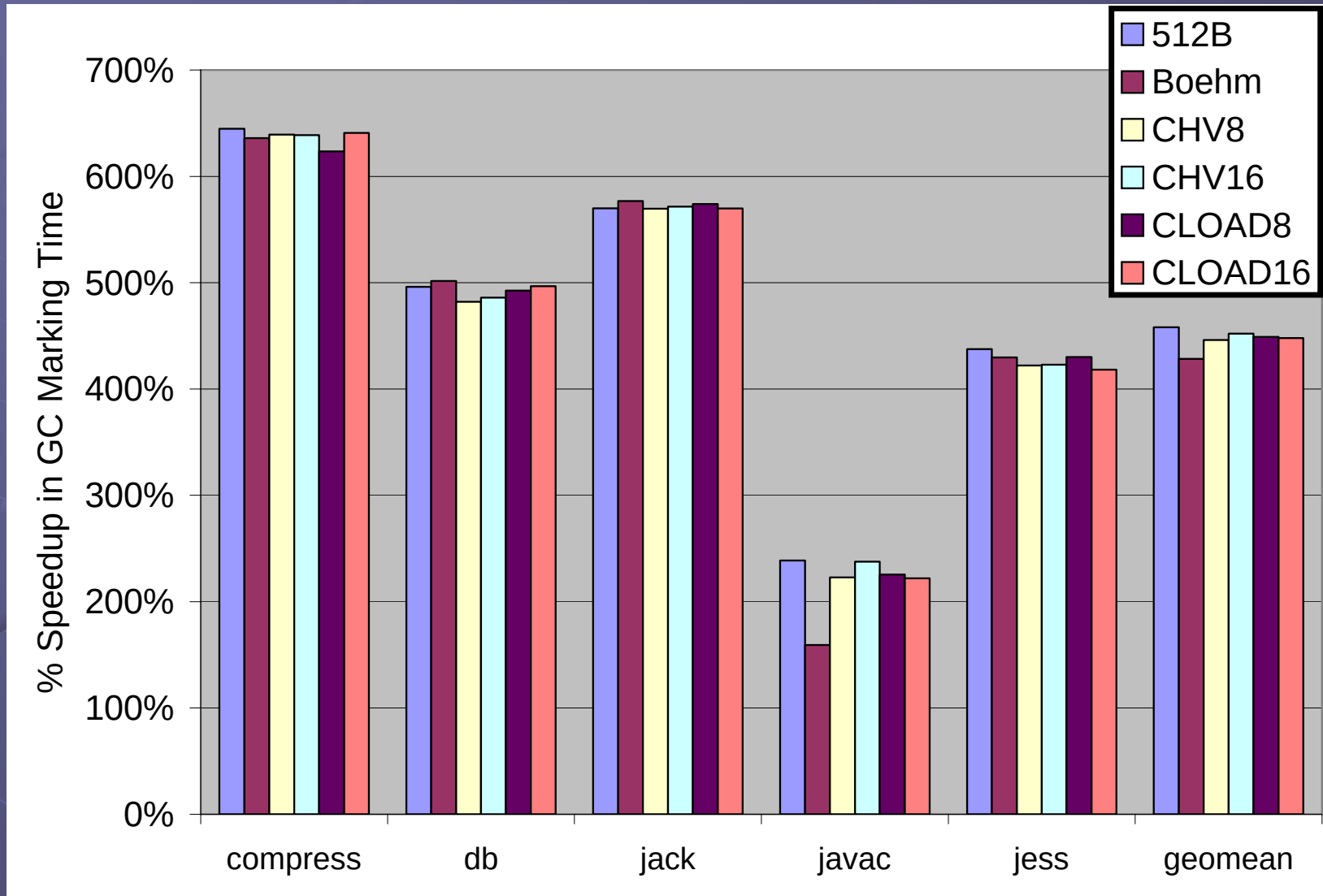
Prefetching and Capitulative Load

- Prefetch possible in local store
- On illegal prefetch target address
 - D-cache prefetching is safe and efficient
 - LS needs to check addr before prefetching
- We experiment with previous schemes
 - Boehm's Prefetching (Prefetch-on-grey)
 - CHV Prefetching (Buffered Prefetching)
 - Capitulative Loads (expose D\$ misses to users)

Prefetching & CLoad Miss Ratio



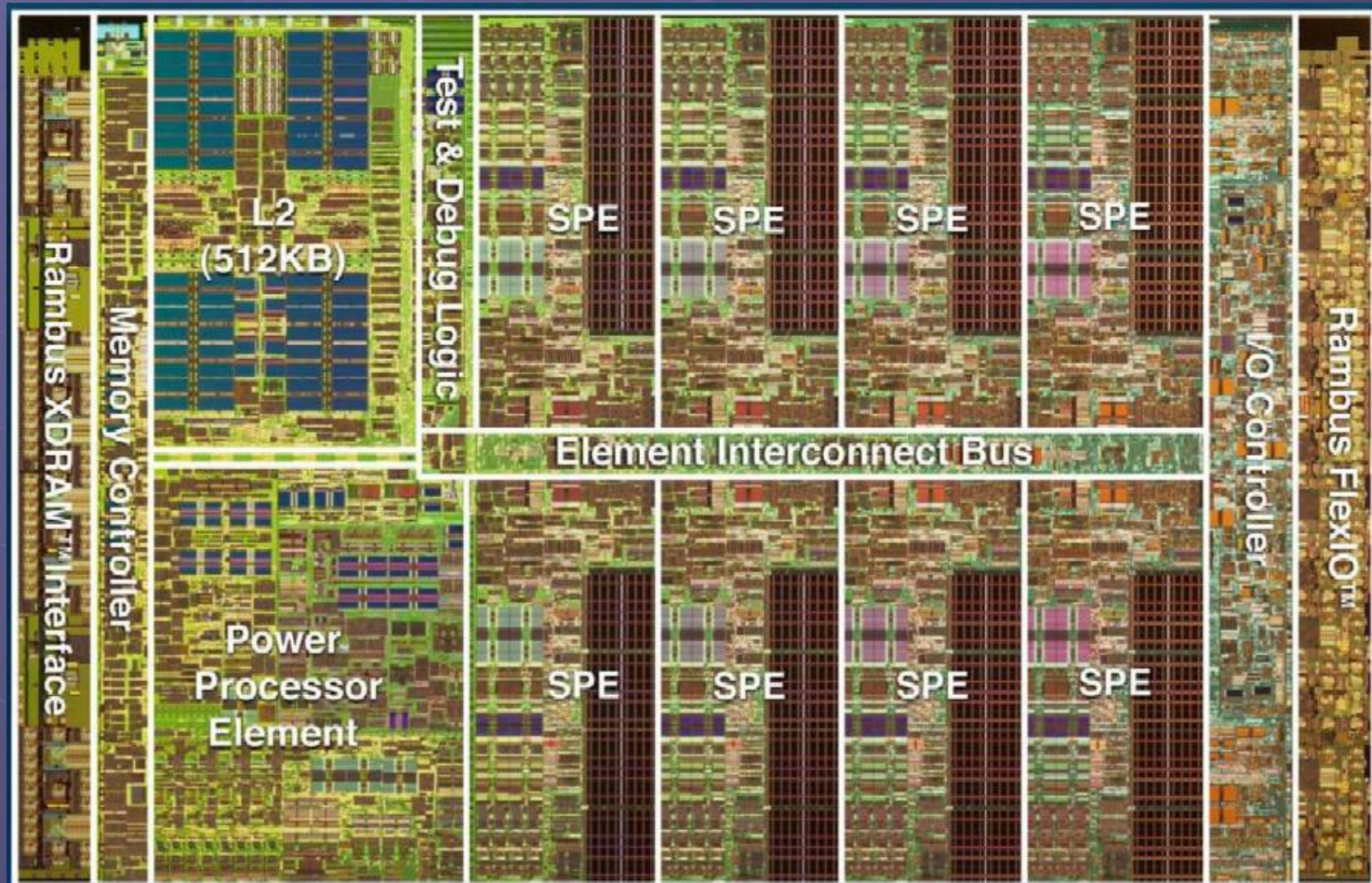
Prefetching & CLoad Speedup



Outline

1. Intro: offloading of GC to SPUs
2. Coherency btw local store and memory
3. Optimizations in the local store
4. Results of Cell GC
5. Comparisons of PPU and SPU

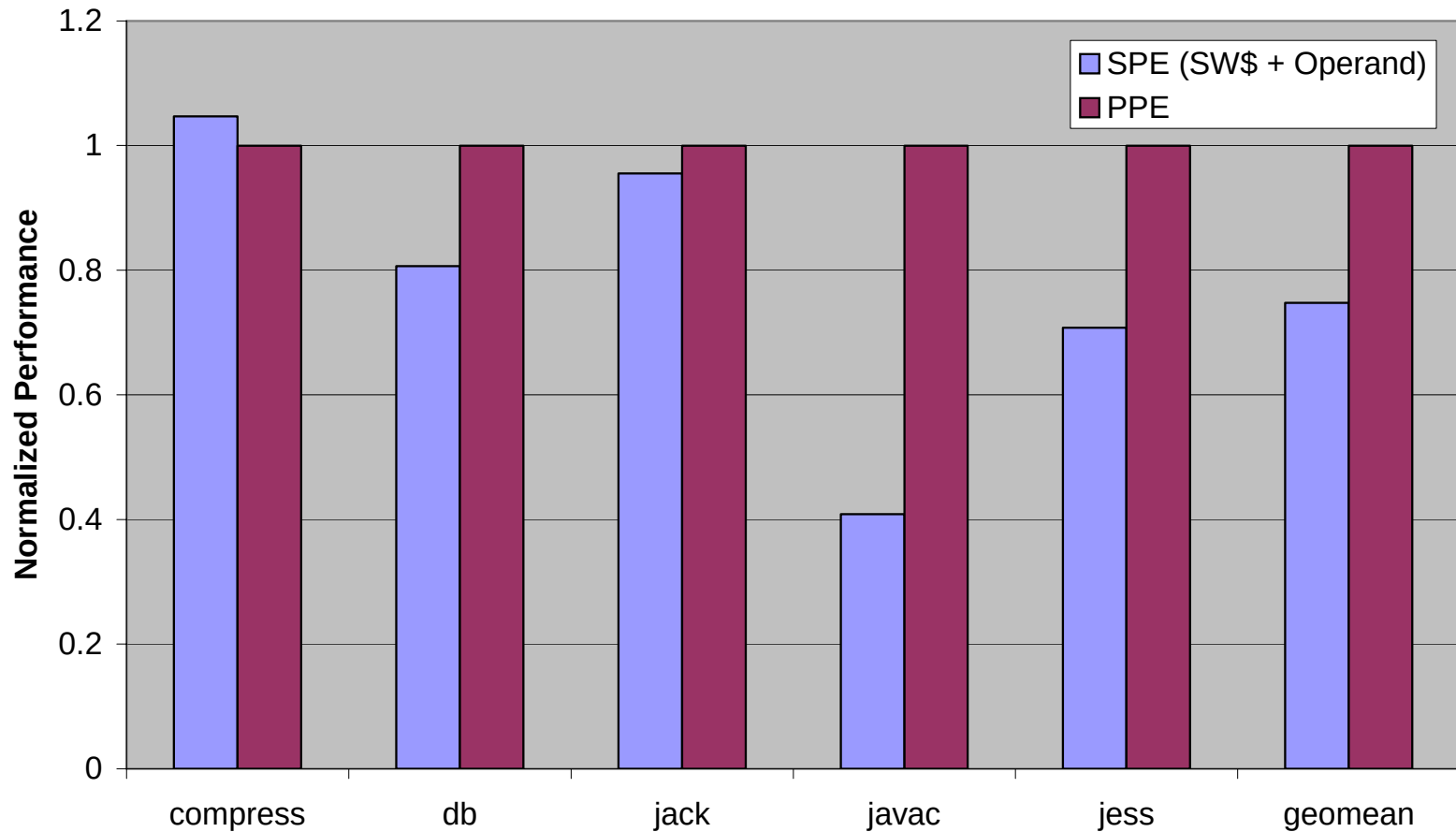
Cell Broadband Engine



SPU is much more area-efficient than PPU

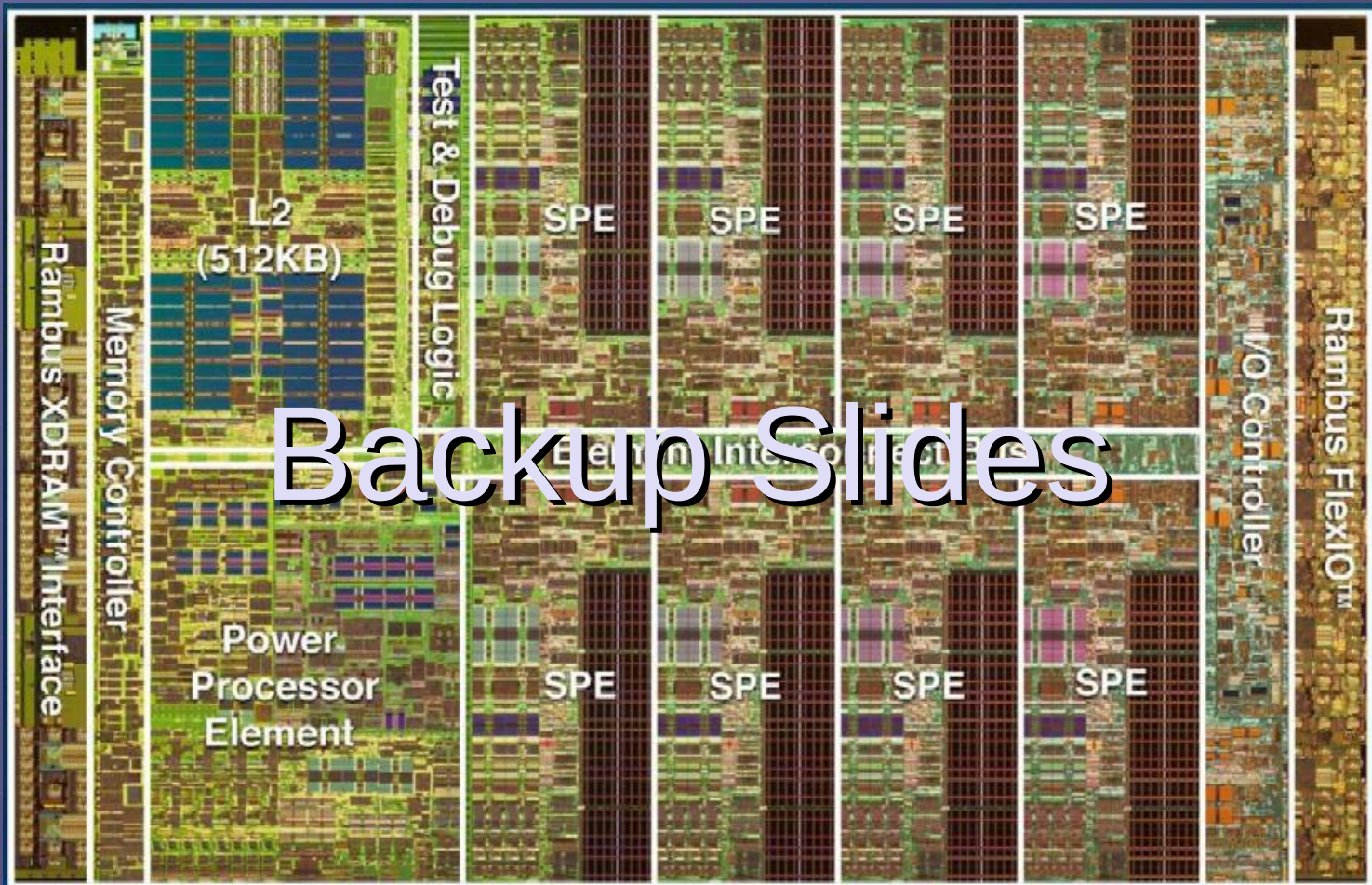
Comparing PPU and SPU, both 3.2Ghz

Normalized Mark Performance
SPE versus PPE



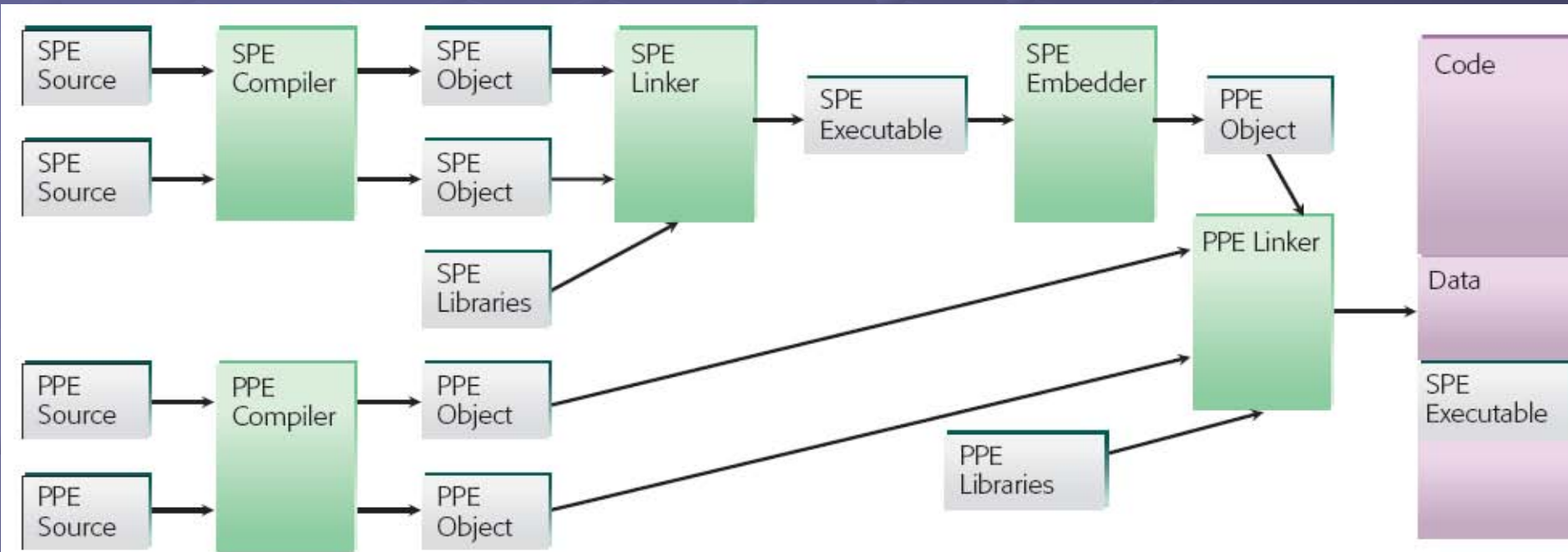
Conclusions

- Offloading GC to SPU is viable
- Reasonable performance for single-SPU
- Explicitly managed local store improves SPU GC performance
- Quantify effects of prefetching
- Future work
 - concurrent, parallel SPU-GC is promising



Backup Slides

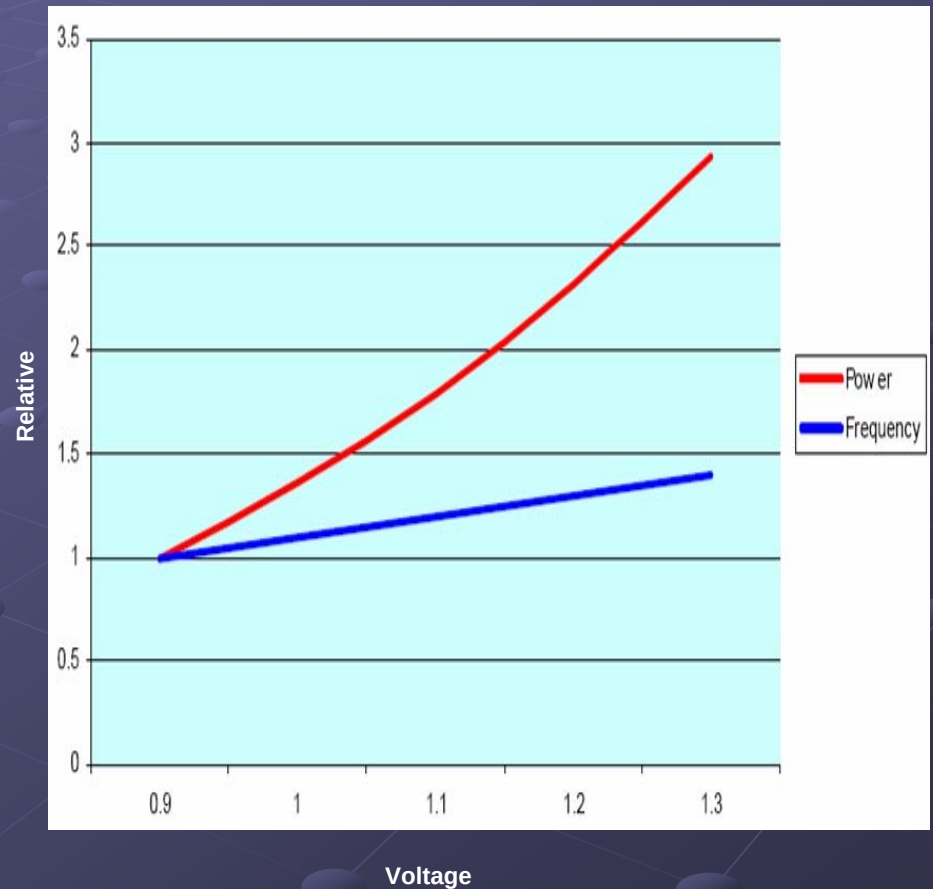
Compiling and linking an Integrated Executable



Cell BE 90nm Implementation Characteristics

- 241M transistors
- 235mm²
- Design operates across wide frequency range
 - Optimize for power & yield
- > 200 GFlops (SP) @3.2GHz
- > 20 GFlops (DP) @3.2GHz
- Up to 25.6 GB/s memory bandwidth
- Up to 75 GB/s I/O bandwidth
- 100+ simultaneous bus transactions
 - 16+8 entry DMA queue per SPE

Frequency Increase vs. Power Consumption



Source: Kahle, Spring Processor Forum 2005