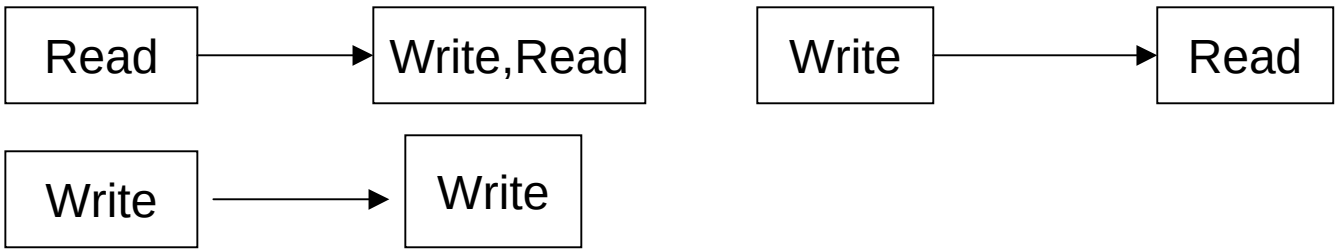
Evaluation of Memory Consistency Models in Titanium

Introduction

- What should be the correct ordering of memory operations?
- Uniprocessor: follow the program order; read and writes to different locations can be reordered safely
- Multiprocessor: correct semantics not clear; memory operations from different processors are not ordered
- Memory Consistency Models: Impose restrictions on the ordering of shared memory accesses
  - **strict: easy to program, but prevents many optimizations and generally thought to be slow**
  - **relaxed: better performance, but hard to code**

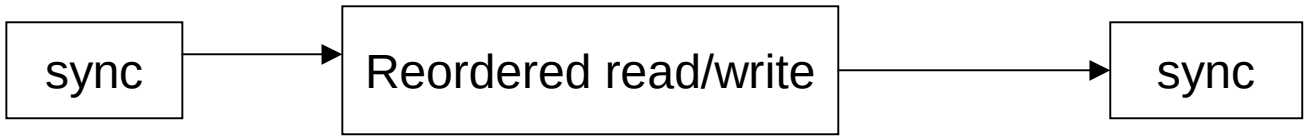
Sequential Consistency

- Definition [Lamport 79]:
  - A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program
- Enforce program ordering for single processor
- Memory accesses appear to execute atomically
- Few optimizations possible (no register allocation, write pipelining, etc)



Weak Ordering

- Relax all program order requirements
- Ordering is preserved only at synchronization points; read and write between barriers can be performed in any order
- Explicit sync instruction to maintain program order
- Eliminates almost all memory latencies, allows most reordering optimizations
- 2x over naïve SC implementation in some study
- Used by PowerPC



Effects of consistency models on program understanding


```
P1
//initially A, B = 0
A = 1
If (B == 0)
   foo();
```

```
P2

B = 1
If (A == 0)
   foo();
```

- SC: Either P1 or P2(or neither) will call foo(), but not both
- WC: foo() may be called by both P1 and P2, because of WR reordering

Enforce SC Using the Compiler

- Compiler can hide the weak consistency model of the underlying machine by inserting barriers
- Gives more flexibility to programmers, can choose between simplicity and performance
- SC can be violated at two levels
- Software level: register allocation, code motion, common subexpression elimination, etc.
- Hardware level: memory access reordering, read bypass pending write, non-blocking reads, etc.

Enforcing SC in Titanium

- Titanium compiler: disallow optimizations that could violate SC.
  - Only one: lifting of invariant exp out of loops
- GCC: One-at-a-time approach by adding fence instructions between accesses to non-stack variables
    - asm volatile ($fence : : : “memory”)
- Hardware:
  - Pentium III: $fence = locked instruction
  - Power3: $fence = sync instruction
- Can do better if apply delay set analysis

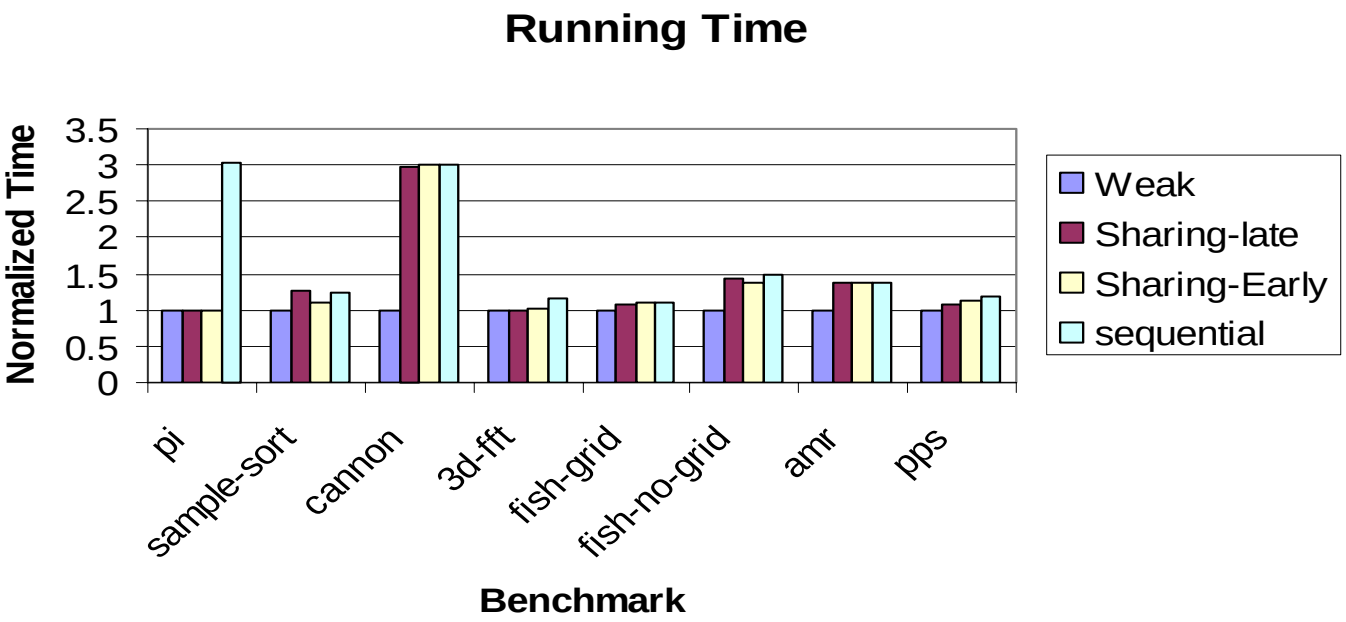
Data Sharing Analysis

- If data is private, no need to enforce sequential consistency on memory operations
- Naïve: fence on every memory access (e.g., C)
- Titanium SC: distinguishes between stack (private) and heap (shared) data
- Titanium sharing inference:
  - **Late : no deference/assignment of global pointers to private data**
  - **Early: Shared data cannot transitively reach private data**
  - **Late qualifies more variables to be private**

Experimental Setup

- Machine Description
  - Millennium Node: 4-way Pentium III (PC model)
  - Seaborg: 16-way IBM Power3 chips (WO model)
- Benchmarks: Titanium Programs
  - Pi: Monte Carlo simulation
  - Sample-sort: distributed sorting algorithm
  - Cannon: matrix multiplication
  - 3d-fft: 3D fast-fourier transform
  - Fish-grid: n-body simulation
  - Pps: poisson solver for uniform grid
  - Amr: poisson solver on an adaptive mesh

Running Time on Millennium Node



- 10% to 3x performance gap
- sharing analysis does not help except for small benchmarks

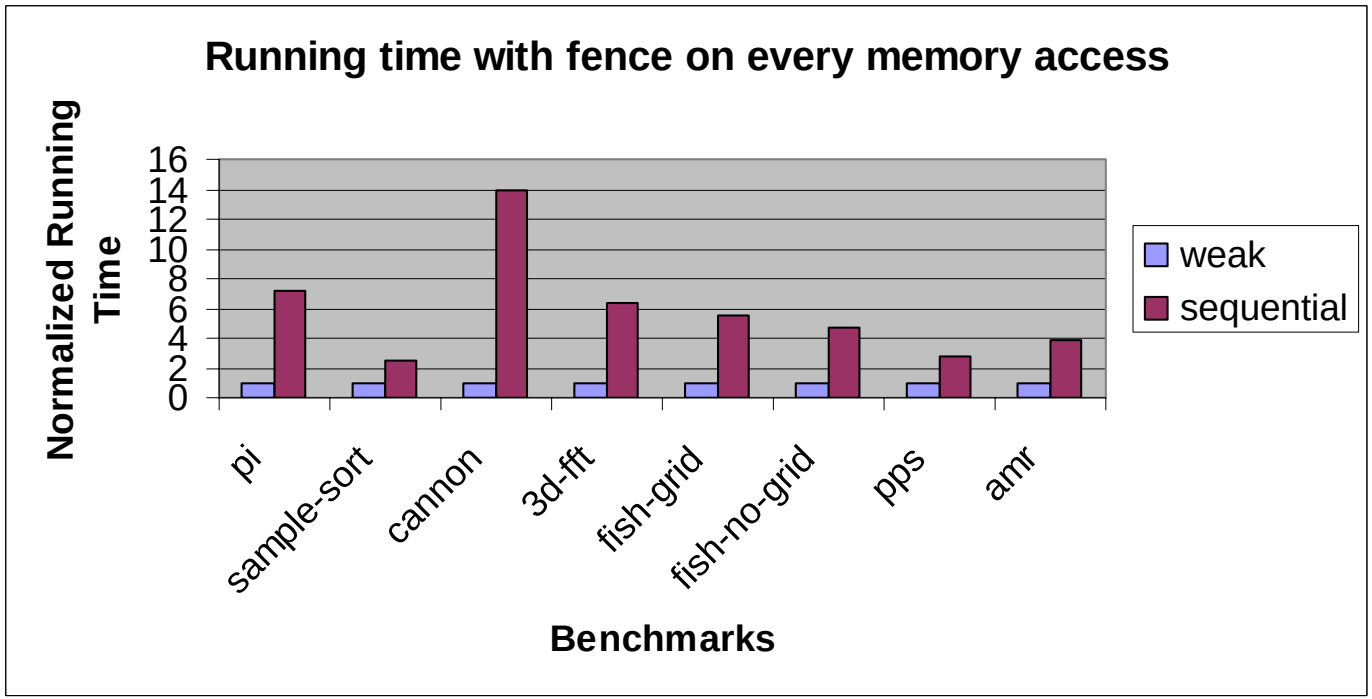
Performance Analysis

- Effect of Titanium compiler optimizations
- Effectiveness of sharing inference
- Effects of GCC optimizations
- Effects of architecture reordering
- Hardware performance data

Effect of Titanium compiler optimizations

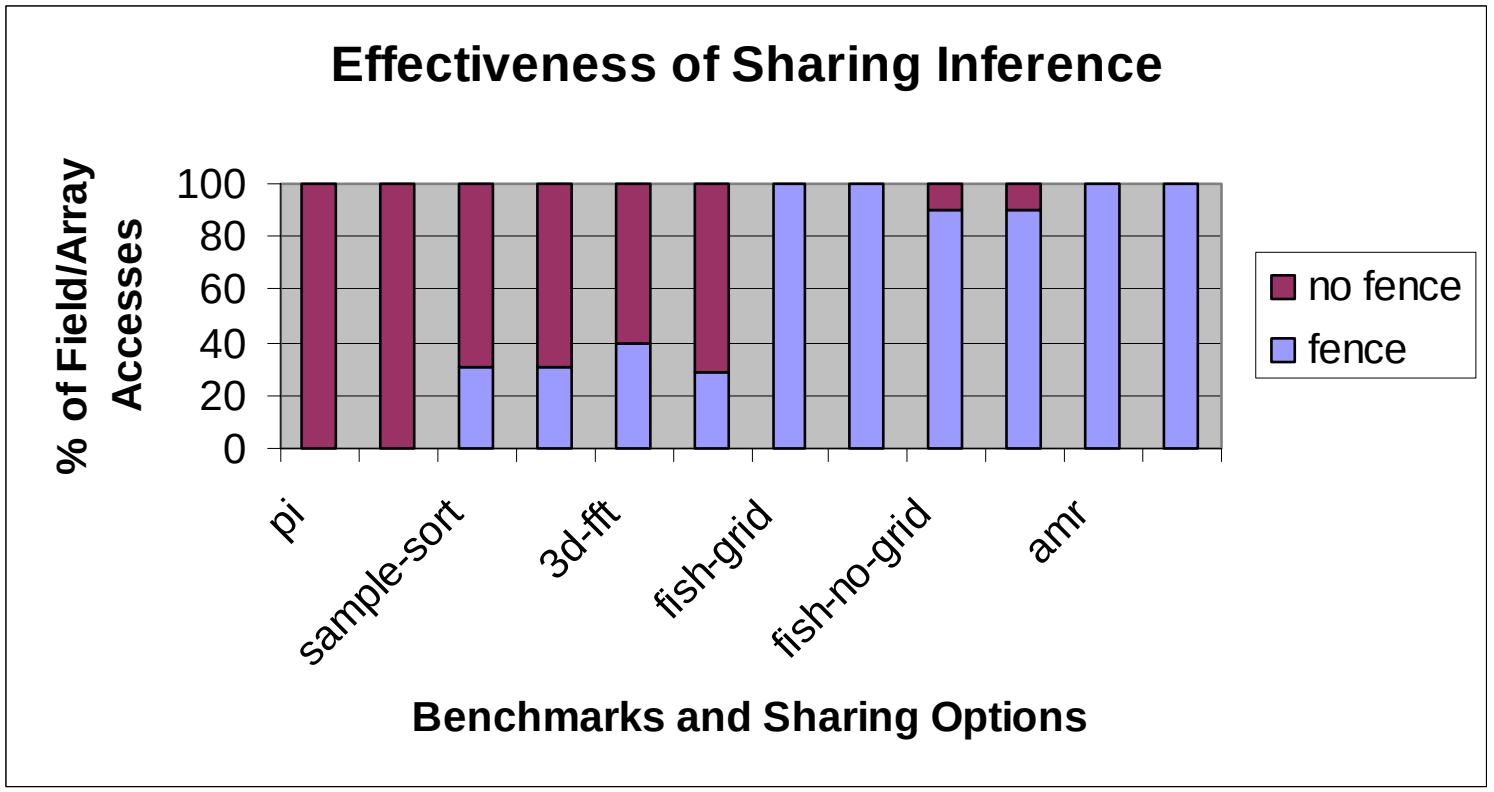
- Collect how many lifting operations were prevented by SC
- Turns out the restriction makes virtually no difference; only in pps, amr do we see expressions not lifted because of SC restriction (5, 6%)
- Running pps,amr with loop lifting constraint turned off yields similar performance ratio
- Makes sense intuitively since rarely do programmers leave an invariant assignment to shared variables in the loop

Data Sharing Analysis



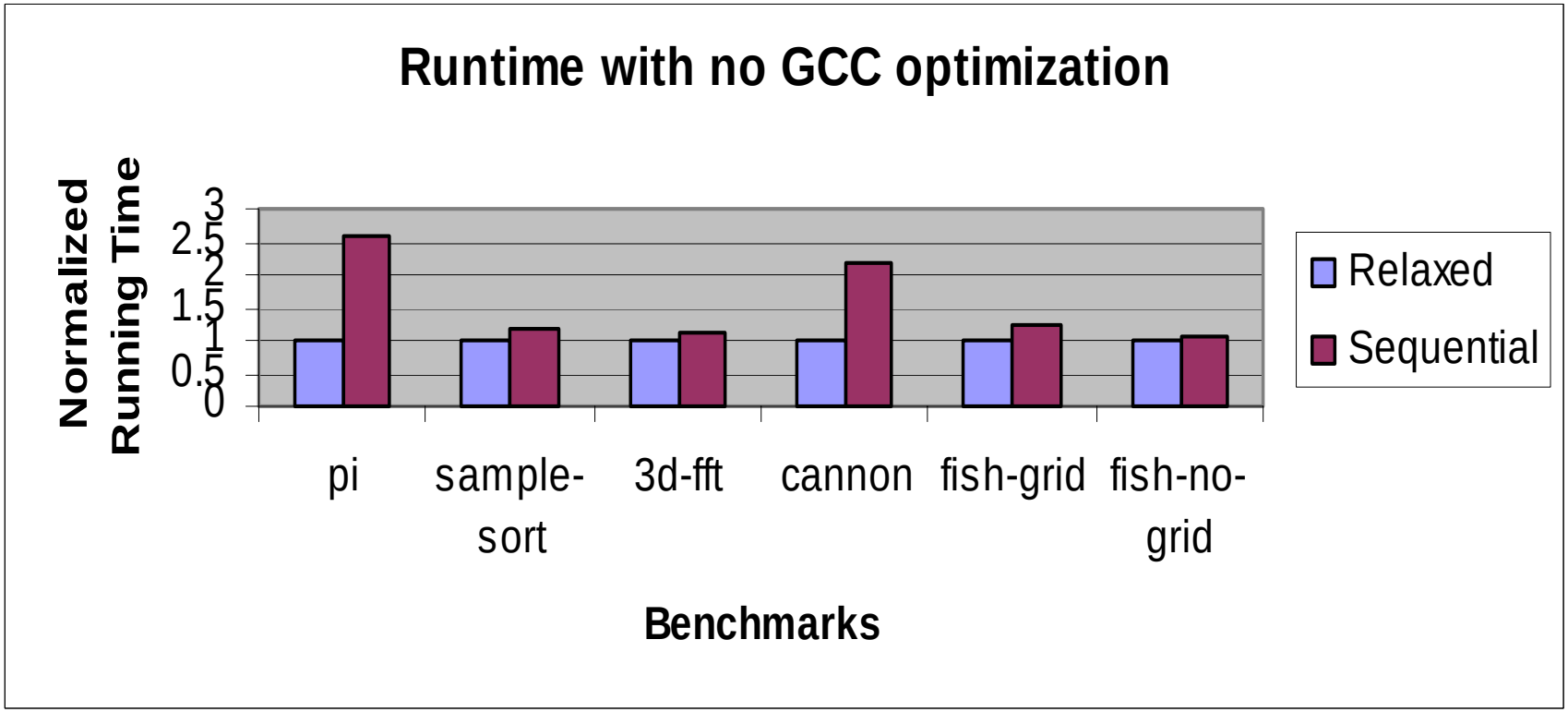
- 2.5 to 14x performance gap between SC and weak
- Identifying stack data as private is very important in lowering the performance penalty of SC

Effectiveness of Sharing Inference



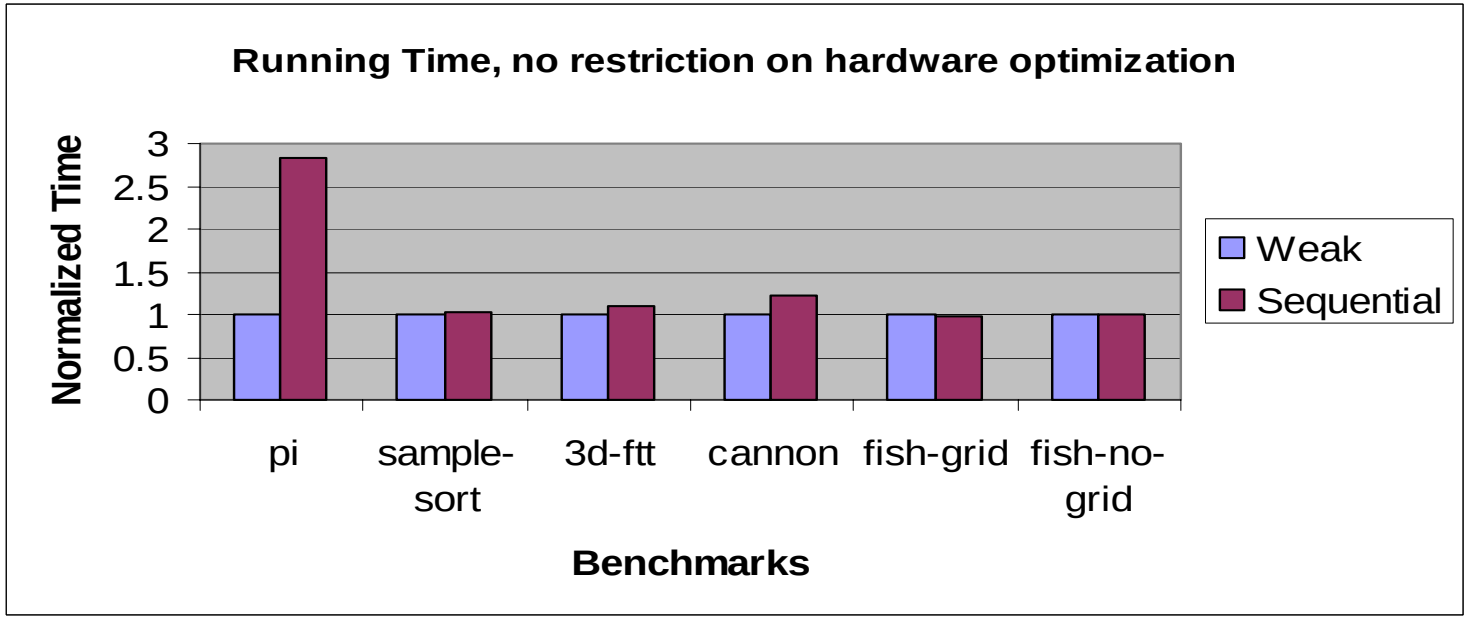
- First bar is early, second is late enforcement
- Effective for small benchmarks, not so for large ones

Effects of GCC optimization (on Mill Node)



- Done by disabling GCC optimizations
- no major decrease in performance gap

Effects of Architecture Reordering (on Mill Node)

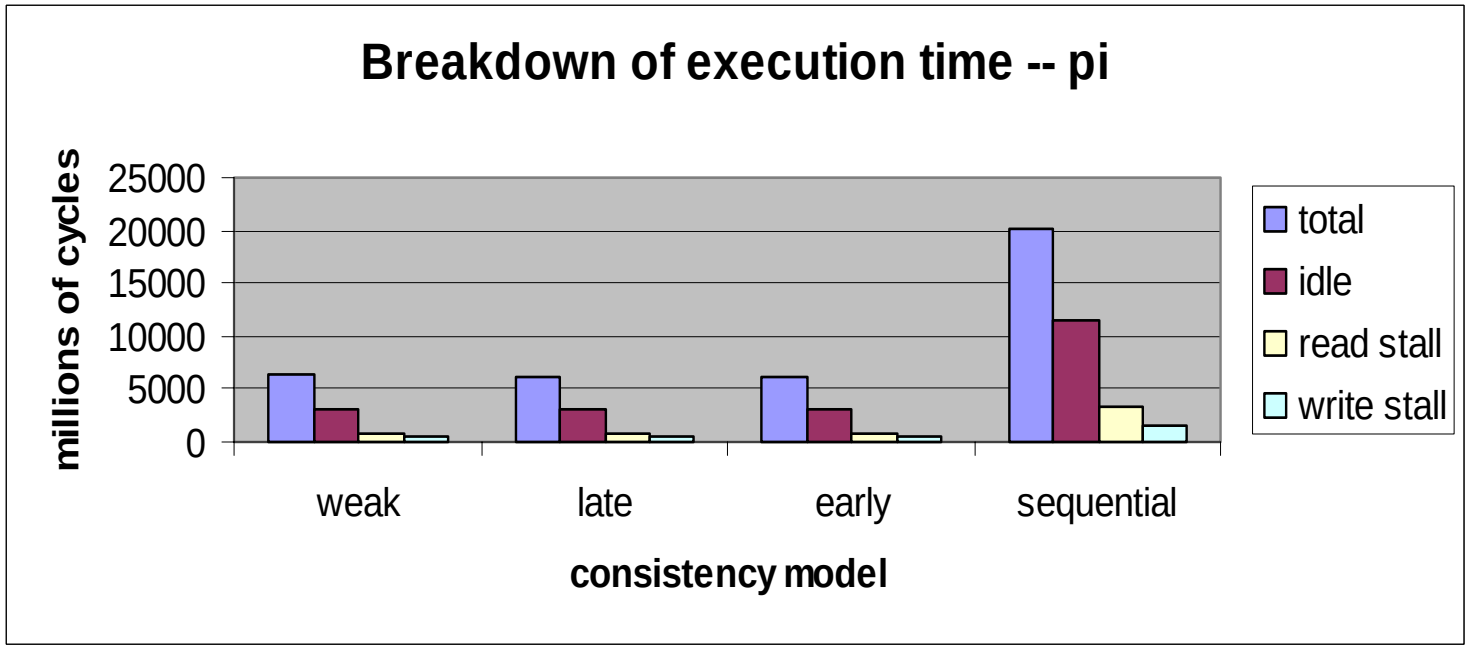


- Compiler does not prevent processor from reordering memory operations
- Decrease in performance gap observed
- Means Performance penalty for SC is higher at hardware level

Number of Sync Instructions

Benchmark	weak	early	late	sequential
pi	52	164	164	48986808
3d-fft	37	6094953	6094953	6094953
fish-grid	501567	11307479	11574263	11598128
amr	109288	36560155	36306984	36308439

Breakdown of Execution Time: Pi (millions of cycles)



• SC's performance penalty mainly caused by increase in idle cycles(waiting for sync instructions to complete)

Conclusion

- Performance gaps exists between SC and relaxed models
  - — Size of gap is Highly dependent on the benchmarks
  - — In general, not very expensive
- Architecture reordering accounts for most of the performance difference; Titanium and gcc optimization's effect appear limited
- Sharing inference is not effective in reducing cost of SC
- At the hardware level
  - — SC causes significant higher number of idle cycles, because of sync instructions