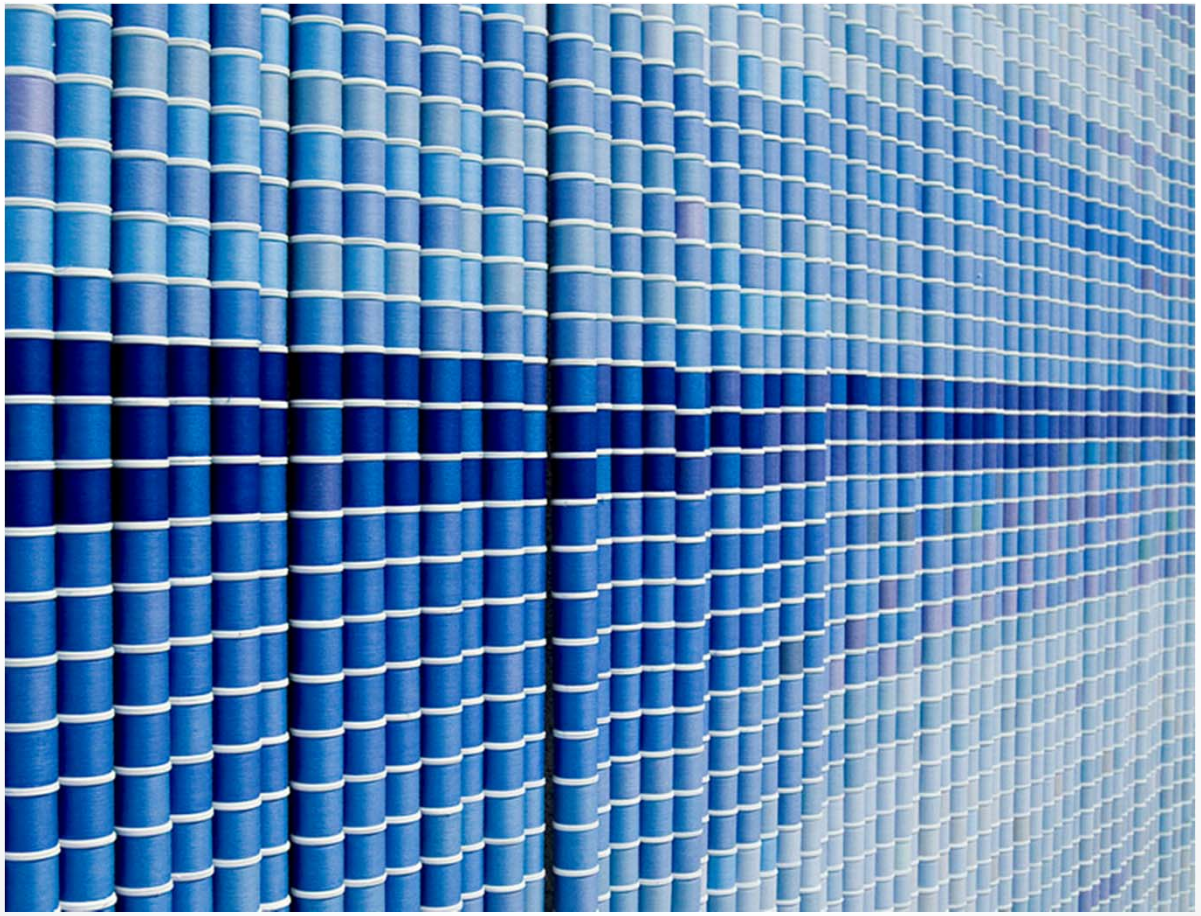
Many Cores, One Thread: The Search for Non-traditional Parallelism

Dean Tullsen
University of California, San Diego

There are some domains that feature nearly unlimited parallelism.

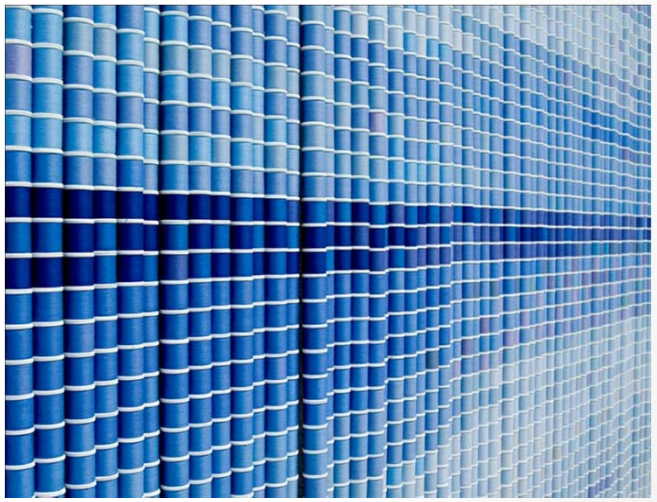


Others, not so much…

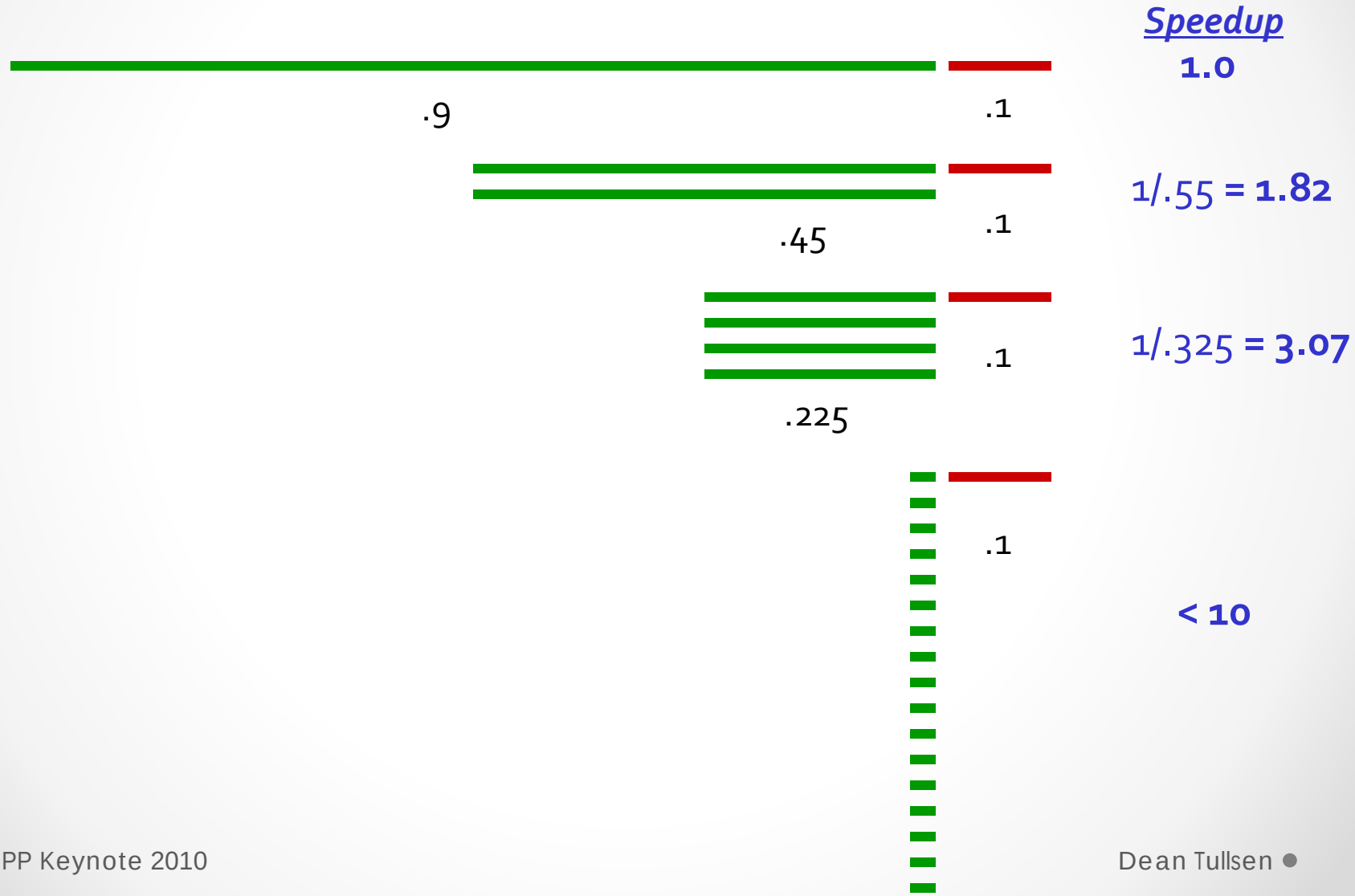
Moore's Law and Single-Thread Performance

- In the 35+ years since the first microprocessor, Moore's law has been applied almost exclusively to increasing single-thread performance.
- Today, Moore's law is no longer being applied to single-CPU performance.
- Instead, additional transistors are being applied to increasing the number of on-chip contexts (multithreading and multi-core).
- The result is that our ability to maintain the performance corollary to Moore's law is critically dependent on our ability to create thread-level parallelism.

Do we still care about single thread performance?



Amdahl's Law and Massive Parallelism



The new economy:

Threads are free!

The big question is:

- How do we use the available hardware parallelism to ***add value*** to a system when thread-level parallelism is low?

The Parallelism Crisis

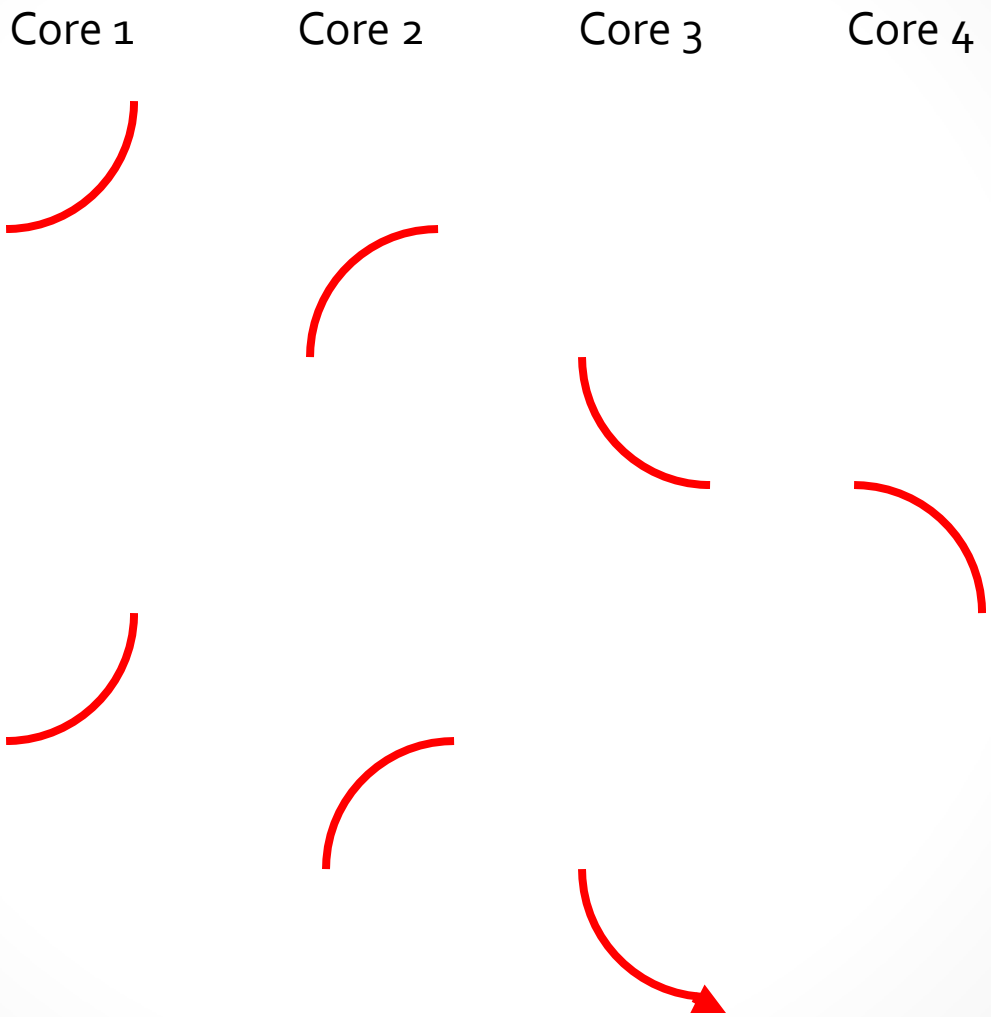
- Defined – The inevitable gap between the parallelism the hardware can exploit and the parallelism the software exhibits.

- We will attack the problem via traditional methods (parallelizing compilers, parallel languages and tools, etc.) and non-traditional methods.

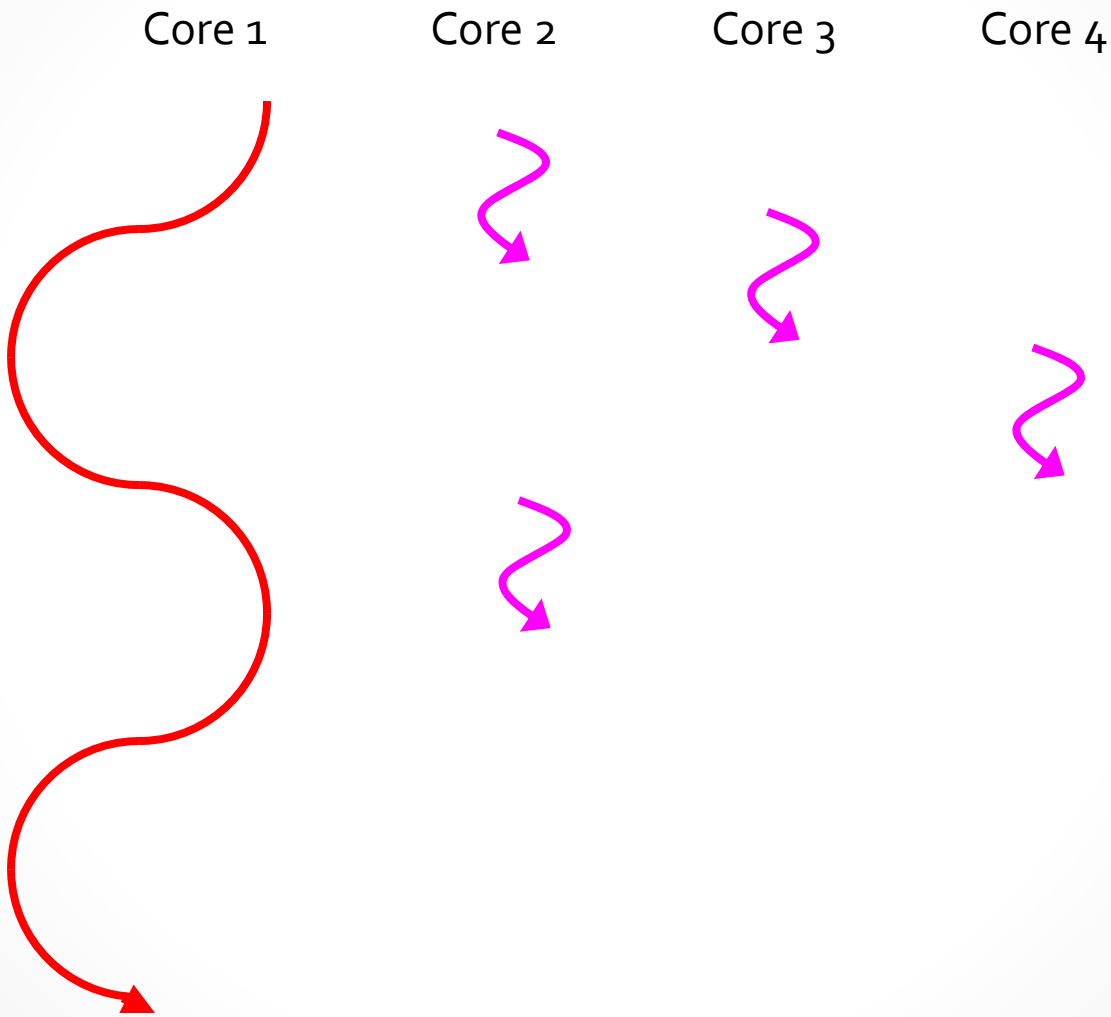
Non-traditional Parallelism

- Parallelism – Use multiple contexts to achieve better performance than possible on a single context.
- Traditional Parallelism – We use extra threads/processors to offload computation. Threads divide up the execution stream.
- Non-traditional parallelism – Extra contexts are used to speed up computation without necessarily off-loading any of the original computation
  - Primary advantage → nearly any code, no matter how inherently serial, can benefit from parallelization.
  - Another advantage – threads can be added or subtracted without significant disruption.

Traditional Parallelism



Non-Traditional Parallelism I



Traditional Parallelism

Non-Traditional Parallelism

Two Techniques for Non-Traditional Parallelism

- Helper Threads
- Migration as a first-class compiler primitive
- (not going to talk about speculative multithreading, aka thread level speculation)

So What's going on in all those extra threads?

- Precomputing memory addresses (or branch directions) and moving data closer to the processor.
- Generating and optimizing code
- Aggregating cache space

Four Approaches to Non-Traditional Parallelism

- **Helper thread prefetching**
- Event-Driven Simultaneous Compilation
- Software Data Spreading via thread migration
- (removed)

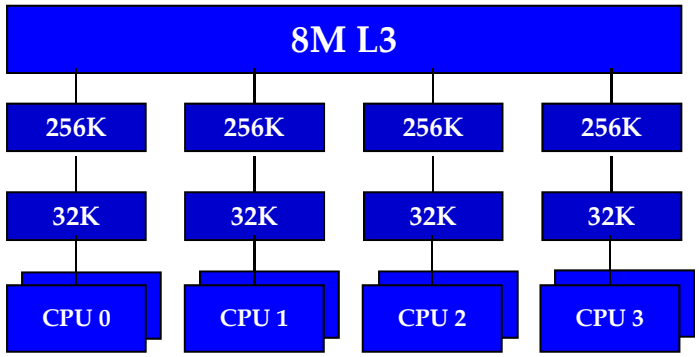
Helper Threads precompute architectural state that may be used by the main thread

- Chappell 1999, Zilles 2001, Collins 2001, Luk 2001

- Speculative Precomputation uses slices derived from the main thread to precalculate load addresses and prefetch data into shared cache.
- Branch directions also possible but require significant additional support.

Speculative Precomputation Exploits Shared Caches

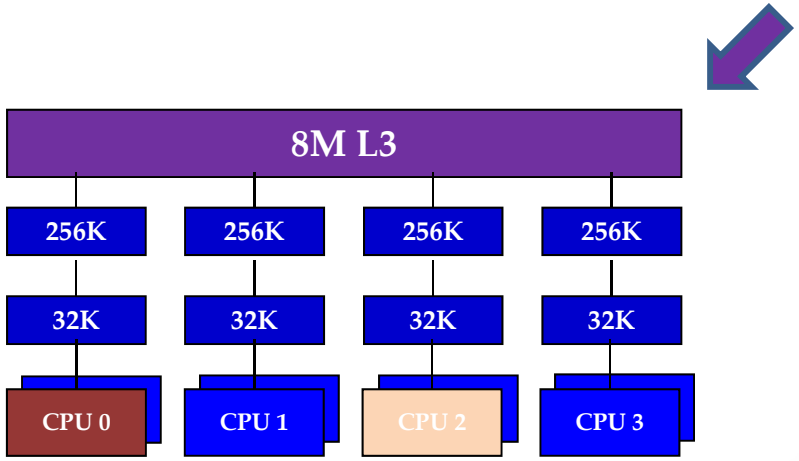
- Easiest and most useful using multitheading contexts.
- Cannot prefetch into private caches of multicores w/o multithreading???



4-core (8 SMT context) Nehalem

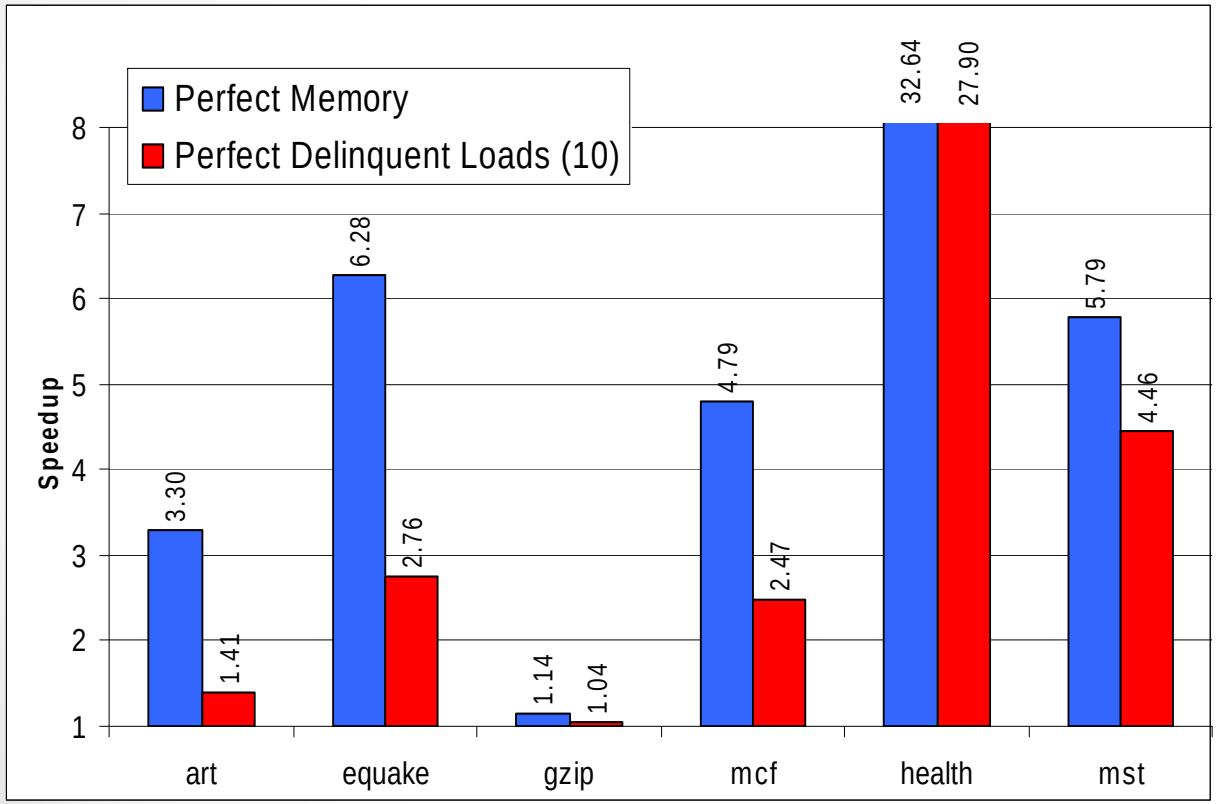
Speculative Precomputation Exploits Shared Caches

- Easiest and most useful using multithreading contexts.
- Cannot prefetch into private caches of multicores w/o multithreading???



4-core (8 SMT context) Nehalem

Speculative Precomputation – Motivation

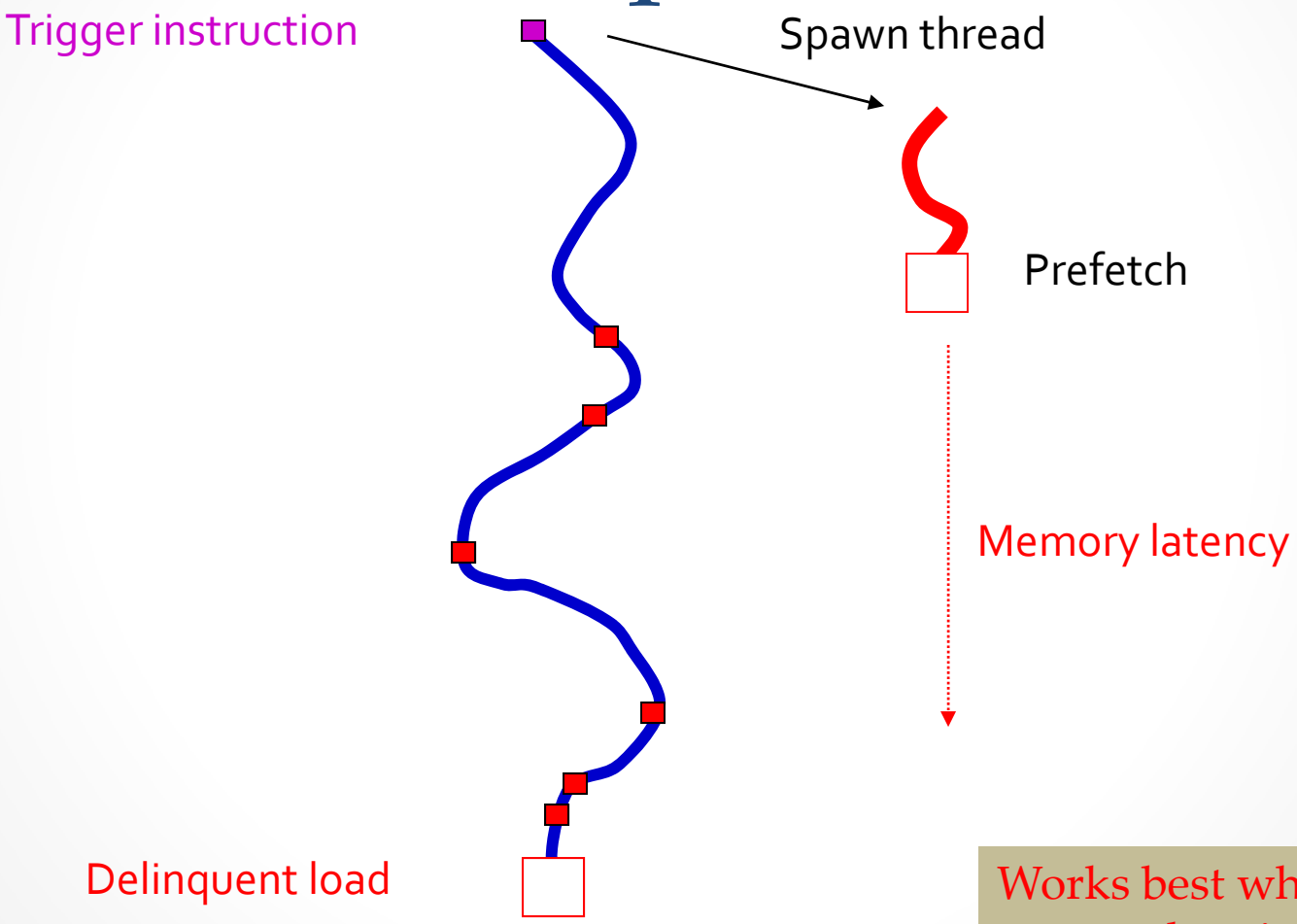


- Achieve most of the gains of eliminating all load delays, by only eliminating the delays of 10 static loads

Therefore…

- It is worthwhile devoting very heavyweight mechanisms to those static loads – even an entire thread context.
- And remember, threads are free.

Speculative Precomputation

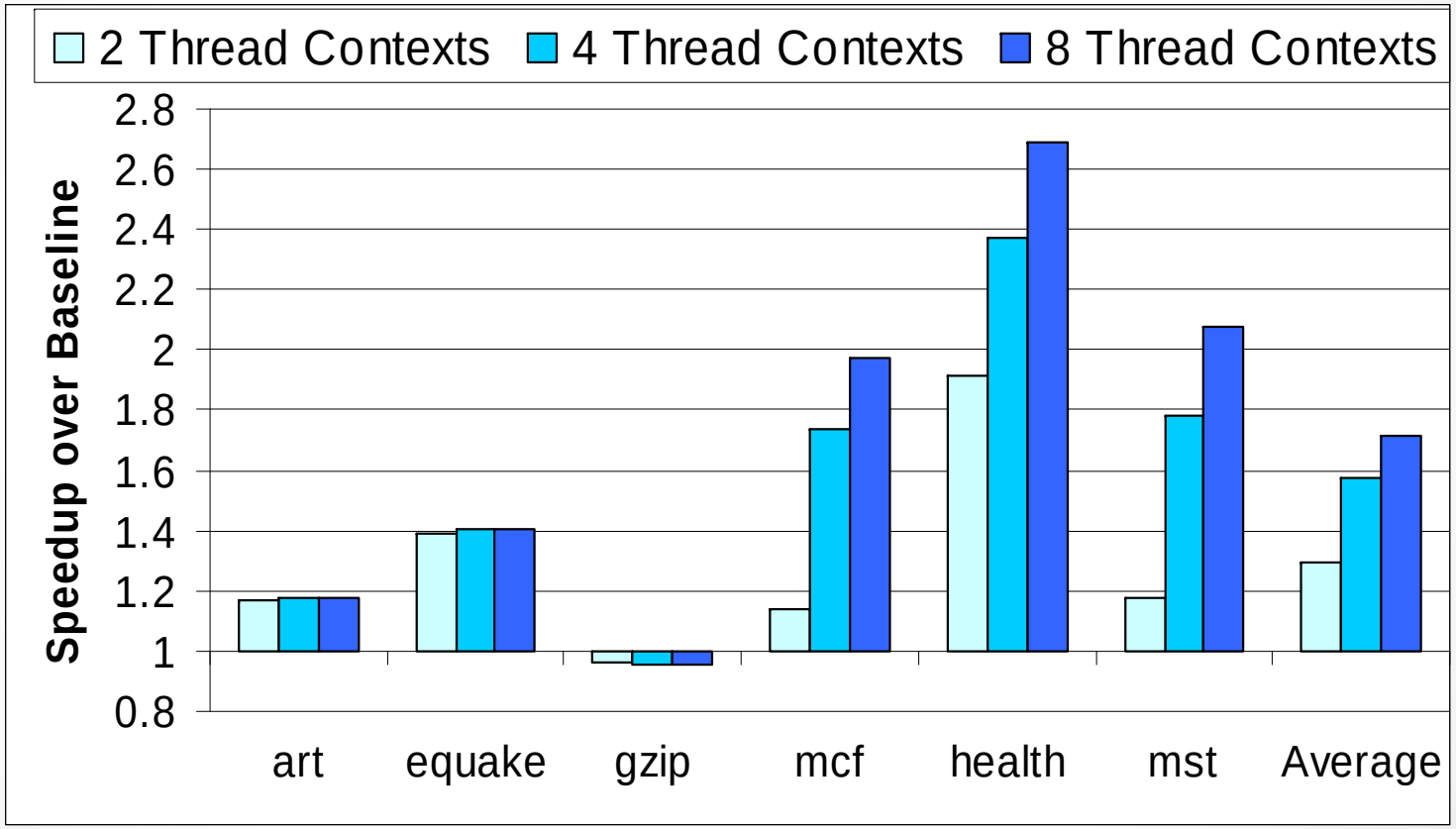


Works best when we can create a looping thread, amortizing the cost of spawning.

Advantages over Traditional (HW or SW) Prefetching

- Because SP uses actual program code, can precompute addresses that fit no predictable pattern.
- Because SP runs in a separate thread, it can interfere with the main thread much less than software prefetching. When it isn't working, it can be killed.
- Because it is decoupled from the main thread, the prefetcher is not constrained by the speed of the main thread.

SP Performance



Generating Helper Threads

- This technique relies heavily on our ability to generate p-slices (helper thread code)
- This is non-trivial, since we are typically targeting irregular code (code that the hw and sw prefetcher misses)
- We typically want to "distill" the p-slice from the original code, retaining the original access patterns.

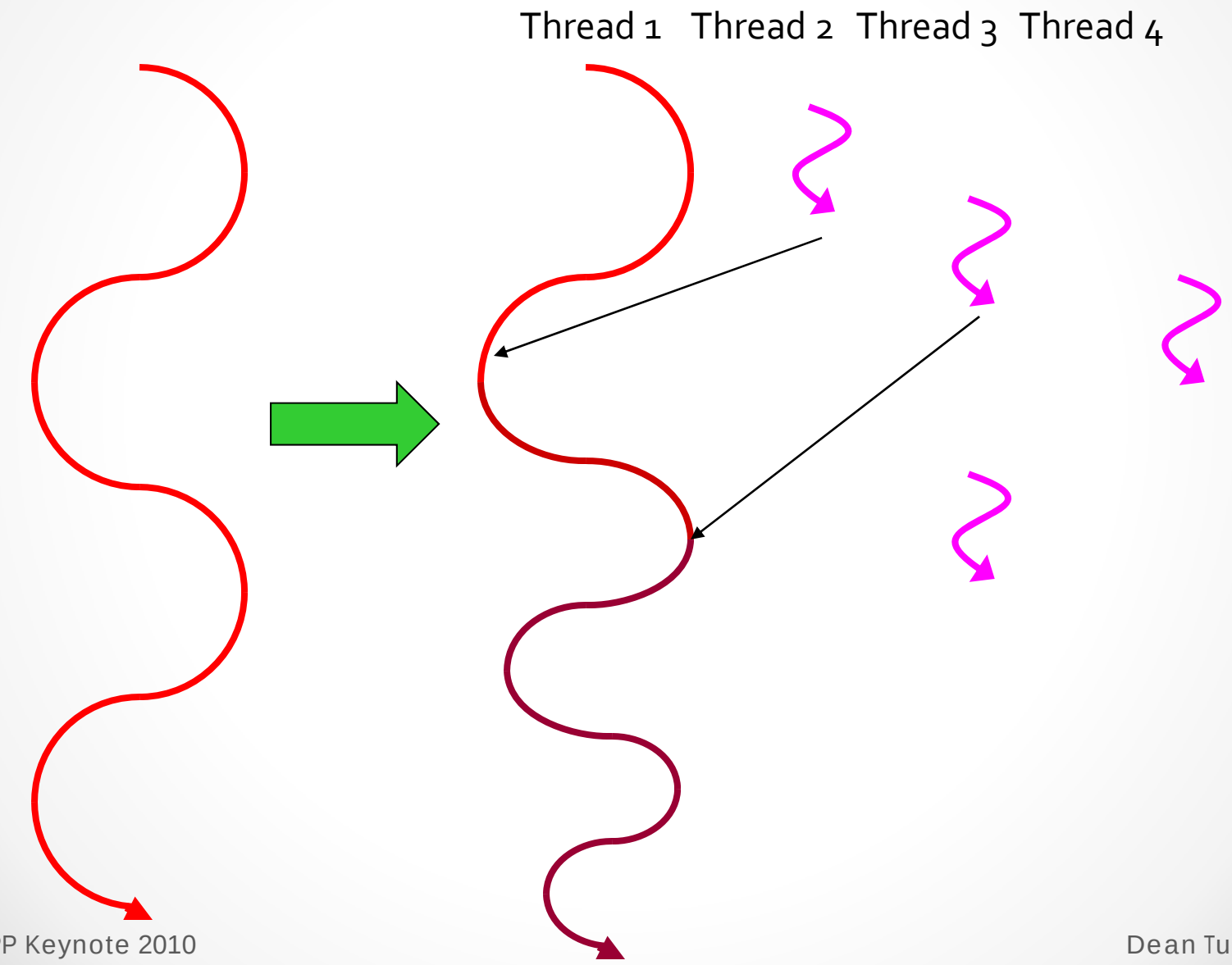
Generating Helper Threads

- By hand (most of the early work)
- Via static compiler [Kim and Yeung]
- Via hardware (dynamic speculative precomputation [Collins, et al.])
- Dynamically via *helper threads*!!! (stay tuned)

Four Approaches to Non-Traditional Parallelism

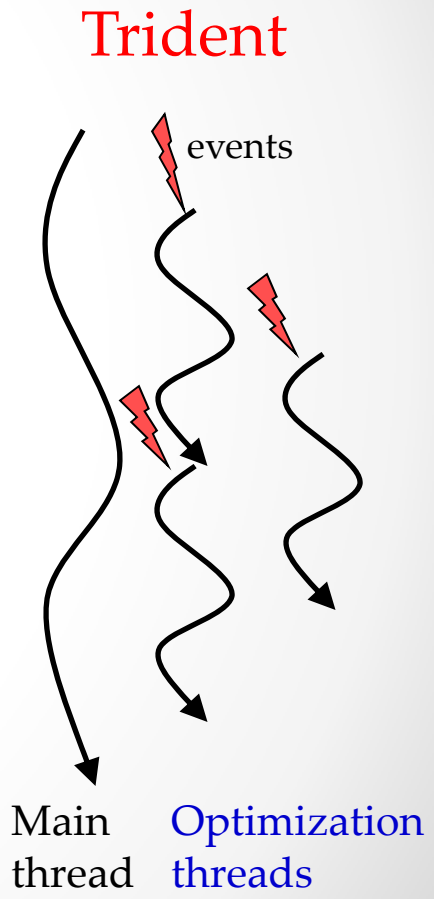
- Helper thread prefetching
- Event-Driven Simultaneous Compilation
- Software Data Spreading via thread migration
- (removed)

Event-Driven Optimization



Event-Driven Optimization: the big idea

- Use "helper threads" to recompile/optimize the main thread (in a code cache).
- Optimization is triggered by interesting events that are identified in hardware (event-driven).
- In this way, the application binary becomes more and more specialized to the runtime behavior of the thread.



A new model of Compilation

- Execution and compiler optimization occur *in parallel*

Advantages of Event-Driven Optimization

- Low overhead profiling of runtime behavior (never need to stop profiling)
- Low overhead optimization by exploiting alternate hardware context/core.
- Quick response to the program's changing behavior
- Enables aggressive optimizations

What kind of events can you trigger on?

- Frequent branches
- Poorly performing branches
- Highly biased branches (easily optimized)
- Frequent loads
- Frequently missed loads
- Loads with high value locality
- Regions of low ILP
- …

What kind of optimizations can you do?

- Well, just about anything. But here are some things we have demonstrated:
  - **Dynamic Value Specialization**
  - Inline software prefetching
  - Helper thread prefetching

Why dynamic value specialization?

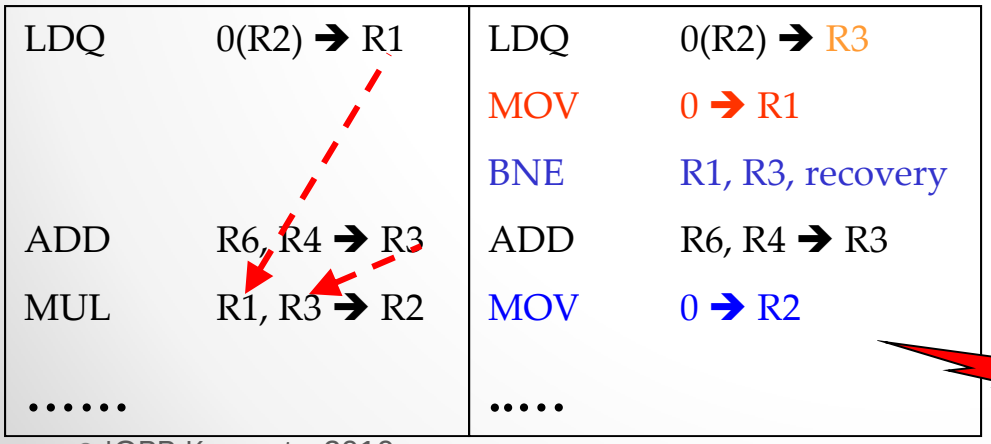
- Value specialization
  - Make a special version of the code corresponding to likely live-in values
- Advantages over hardware value prediction
  - Value predictions are made in the background and less frequently
  - No limits on how many predictions can be made
  - Allow more sophisticated prediction techniques
  - *Propagate predicted values* along the trace
  - Trigger other optimizations such as strength reduction

Why dynamic value specialization?

- Value specialization
  - Make a special version of the code corresponding to likely live-in values
- Advantages over software value specialization
  - Can adapt to semi-invariant runtime values (eg, values that change, but slowly)
  - Adapts to actual dynamic runtime values.
  - Detects optimizations that are no longer working.

Dynamic value specialization

- Specialize on
  - Semi-invariant "constants"
  - Strided values (details omitted)
- Verify dynamically in recovery code



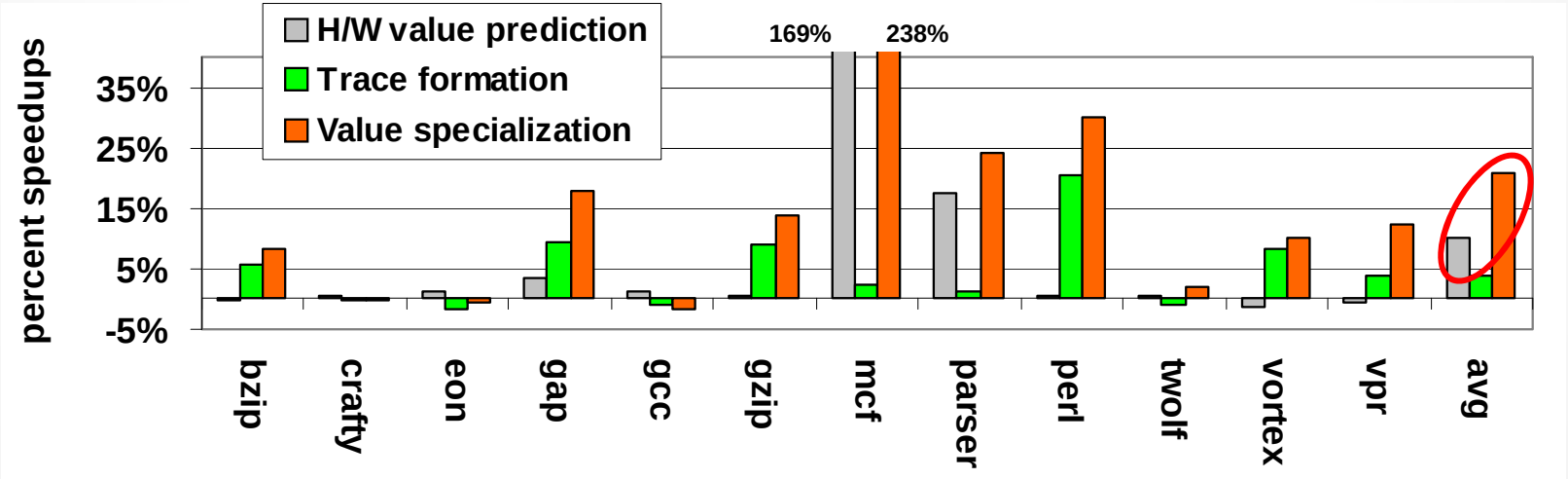
- Perform the original load into a scratch register
- Move predicted value into the load destination
- Check the predicted value, branch to recovery if not equal
- Perform constant propagation and strength reduction

compensation block

- Copy the scratch into load destination
- Jump to next instruction after load in the original binary

No dependency!

Performance of dynamic value specialization



What kind of optimizations can you do?

- Well, just about anything. But here are some things we have demonstrated:
  - Dynamic Value Specialization
  - **Inline software prefetching**
  - Helper thread prefetching

Software prefetching

Limitations of existing static prefetching techniques:

- Address / aliasing resolution
- Timeliness
- Hard to identify delinquent loads
- Variation due to data input or architecture

Why prefetch distance is so hard

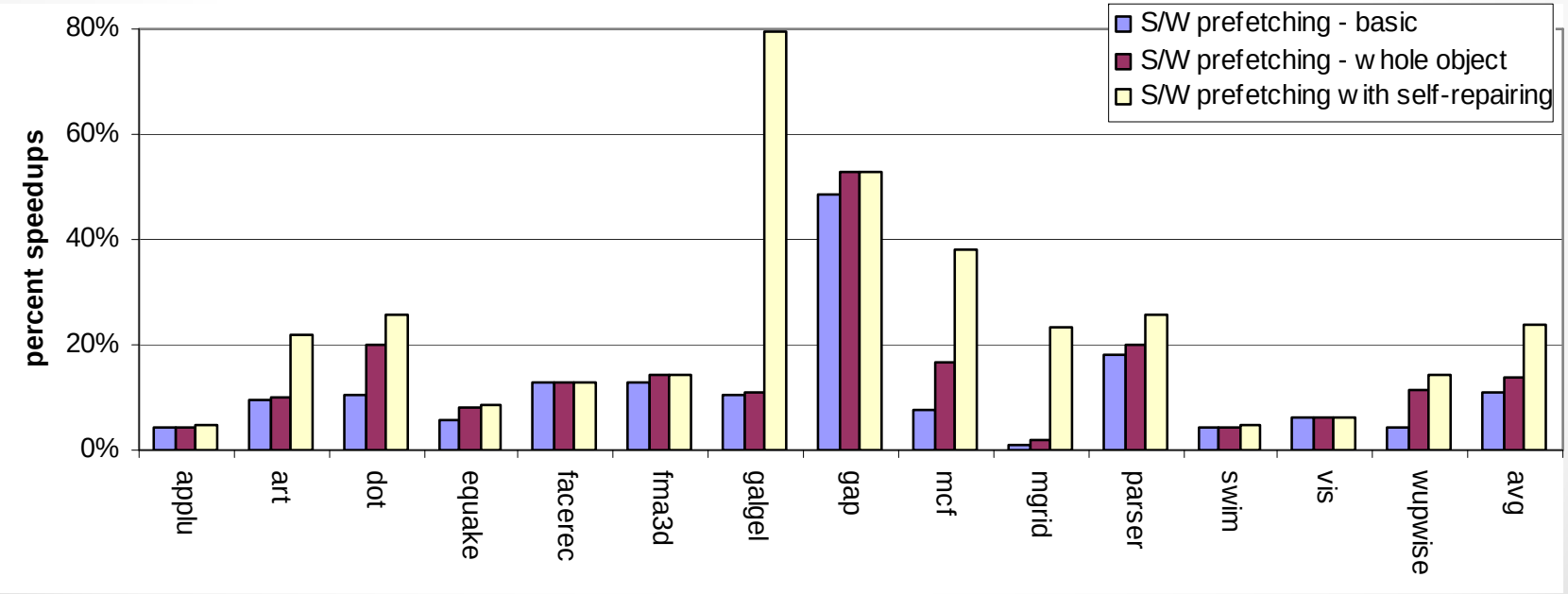
(iter 1)
load A
load B
load C
(iter 2)
load A
load B
load C
(iter 3)
load A
load B
load C
(iter 4)
load A
load B
load C
(iter 5)
load A
load B
load C

How event-driven compilation solves the problem

- Re-optimization is *extremely* cheap, and still triggered when loads miss (ie, prefetch distance is not right).
- This makes it easy, then, to dynamically discover the right prefetch distance, and rediscover when conditions change
- That is, we simply use a *trial-and-error* approach to discover a value (the correct prefetch distance) that is nearly impossible to precompute statically.

Performance of self-repairing prefetching

- Baseline: H/W stride-based prefetching stream buffers
- Self-repairing based prefetching achieves 23% speedup
- 12% better than software prefetching without repairing



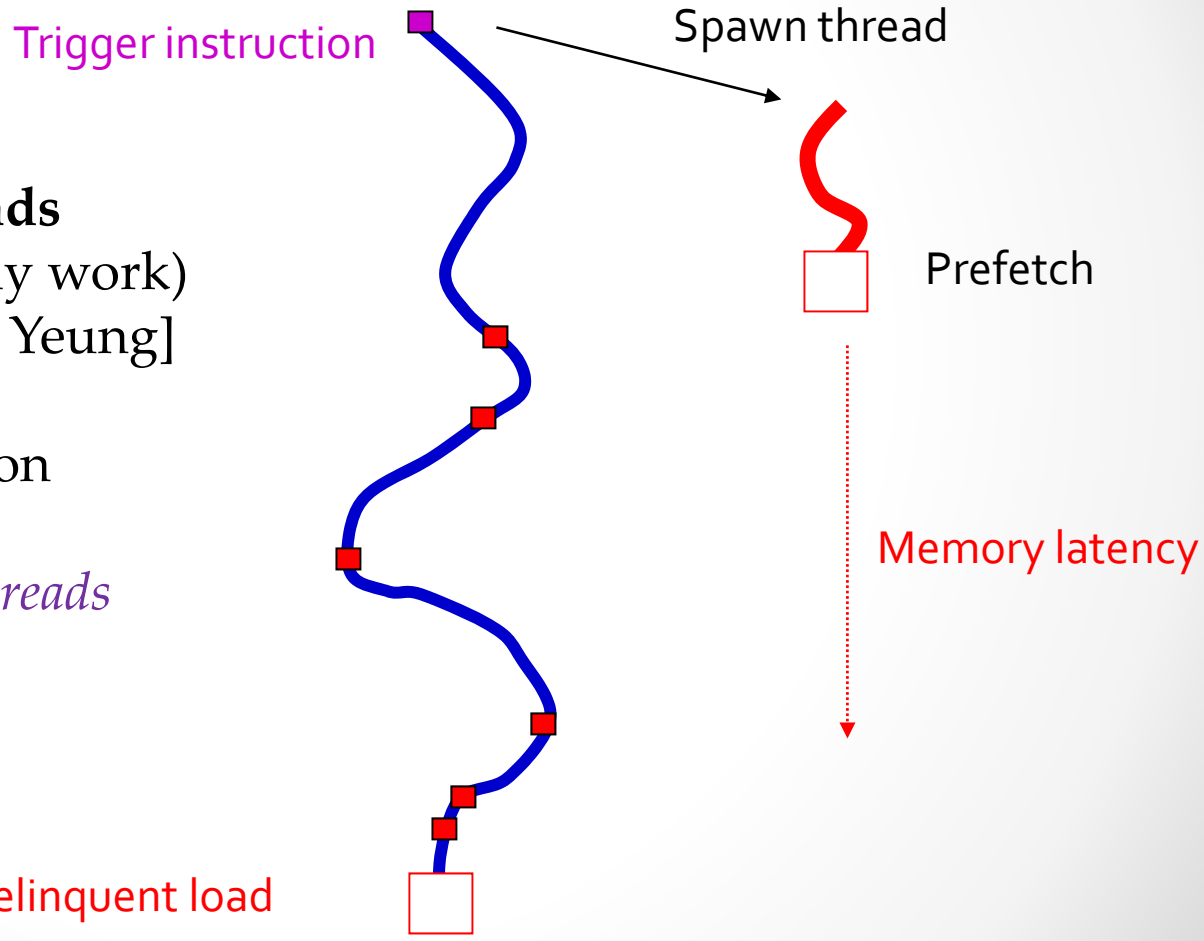
What kind of optimizations can you do?

- Well, just about anything. But here are some things we have demonstrated:
  - Dynamic Value Specialization
  - Inline software prefetching
  - Helper thread prefetching

Remember – Speculative Precomputation

Generating Helper Threads

- By hand (most of the early work)
- Via static compiler [Kim, Yeung]
- Via hardware (dynamic speculative precomputation [Collins, et al.])
- Dynamically via *helper threads*



Helper thread prefetching

- Can potentially be more effective than inline prefetching.
- However, more complex, with more things to get right/wrong
  - How far ahead to trigger
  - When to terminate (end of loop)
  - When to terminate (prefetching off-track or ineffective)
  - Synchronization between helper and main thread – degree of decoupling
- These vary not just with load latencies, but also control flow, etc.
- Again, our ability to continuously adapt is key.

Event-driven simultaneous compilation

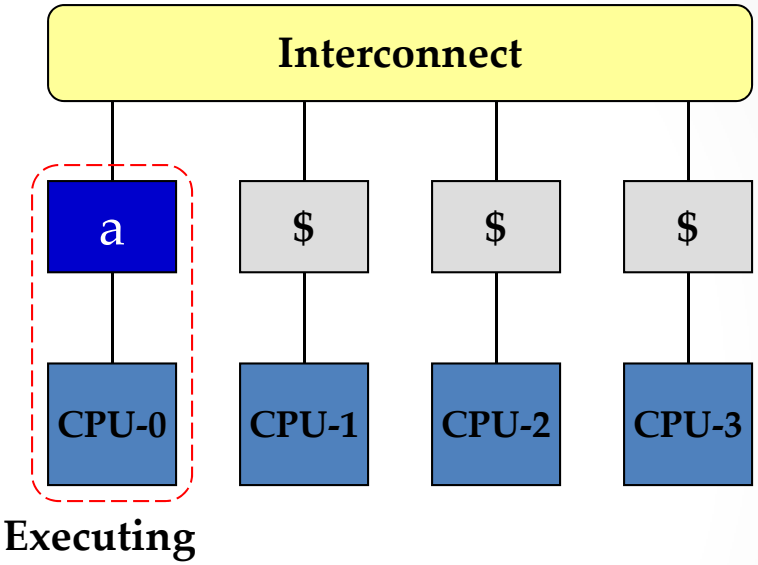
- Is a powerful way to use hardware parallelism to accelerate performance
- Works on completely serial code
- Incurs almost no runtime overhead (in performance or power)

Four Approaches to Non-Traditional Parallelism

- Helper thread prefetching
- Event-Driven Simultaneous Optimization
- Software Data Spreading via thread migration
- (removed)

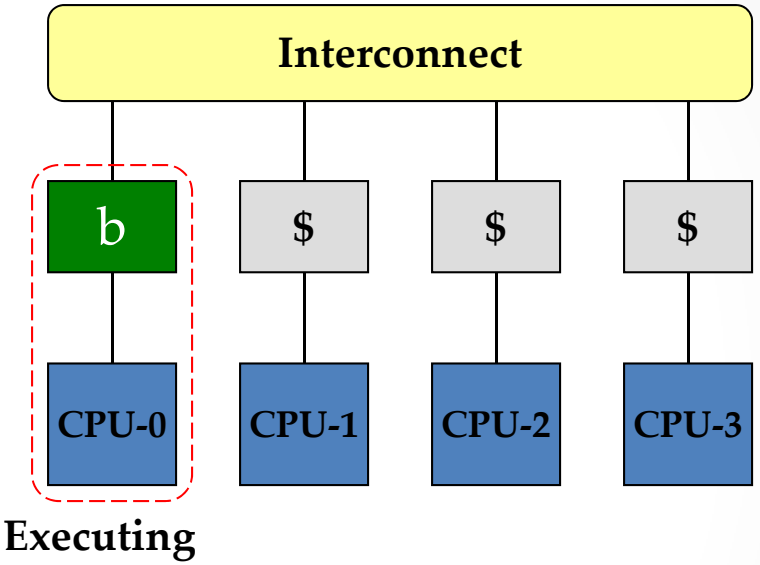
Memory Intensive Single Thread Execution

```
for i=1 to 100  {
      for j=1 to 1000      Loop1
            a[j] = a[j-1] + a[j+1]
       ……….
      for j=1 to 2000      Loop2
            b[j] = b[j-1] + b[j+1]
}
```



Memory Intensive Single Thread Execution

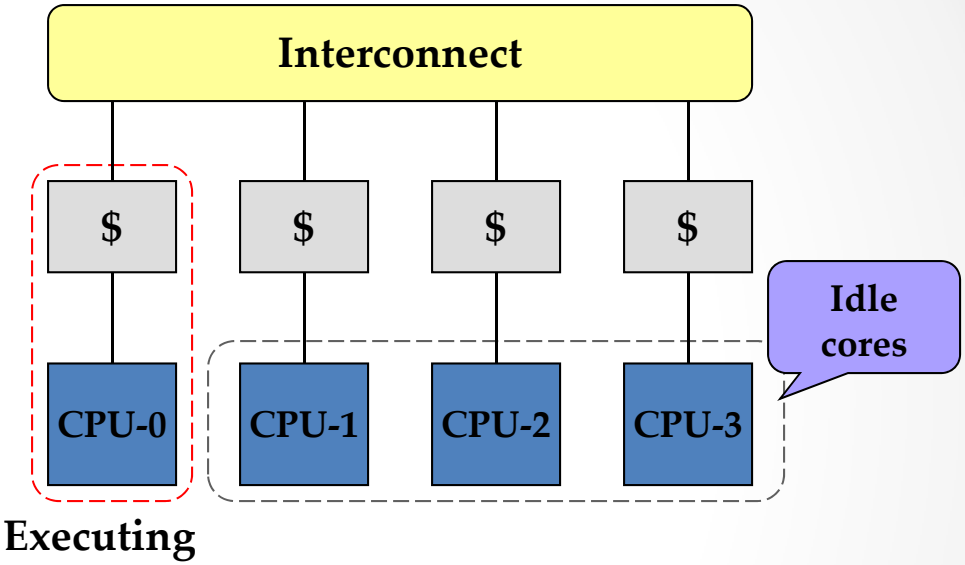
```
for i=1 to 100  {
      for j=1 to 1000      Loop1
            a[j] = a[j-1] + a[j+1]
       ……….
      for j=1 to 2000      Loop2
            b[j] = b[j-1] + b[j+1]
}
```



- One Cache is not large enough to hold both ‘a’ and ‘b’
  - Capacity misses

Can We Exploit Idle Resources ?

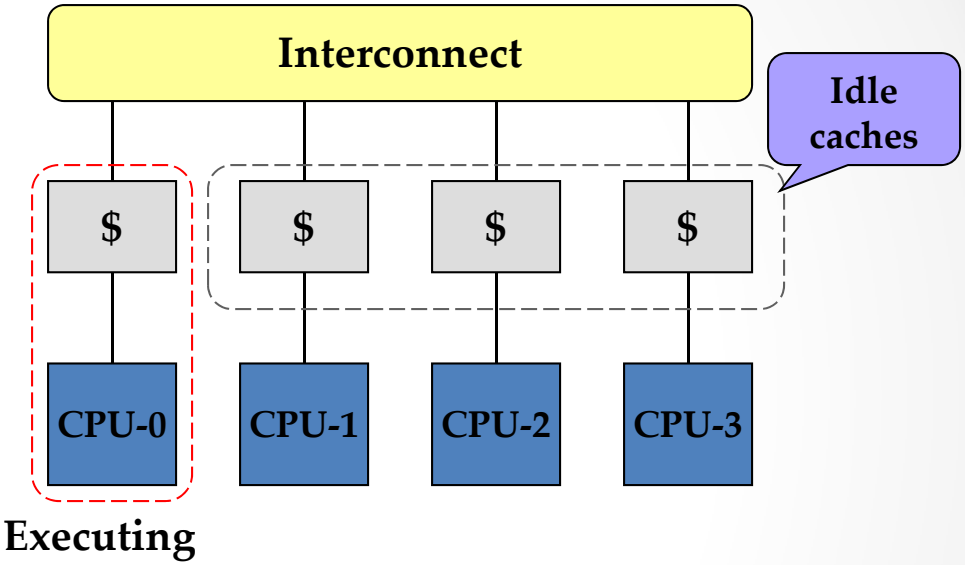
```
for i=1 to 100  {
      for j=1 to 1000      Loop1
            a[j] = a[j-1] + a[j+1]
       ……….
      for j=1 to 2000      Loop2
            b[j] = b[j-1] + b[j+1]
 }
```



- Parallel execution in hardware is not always possible
  - Lack of parallelism

Can We Exploit Idle Resources ?

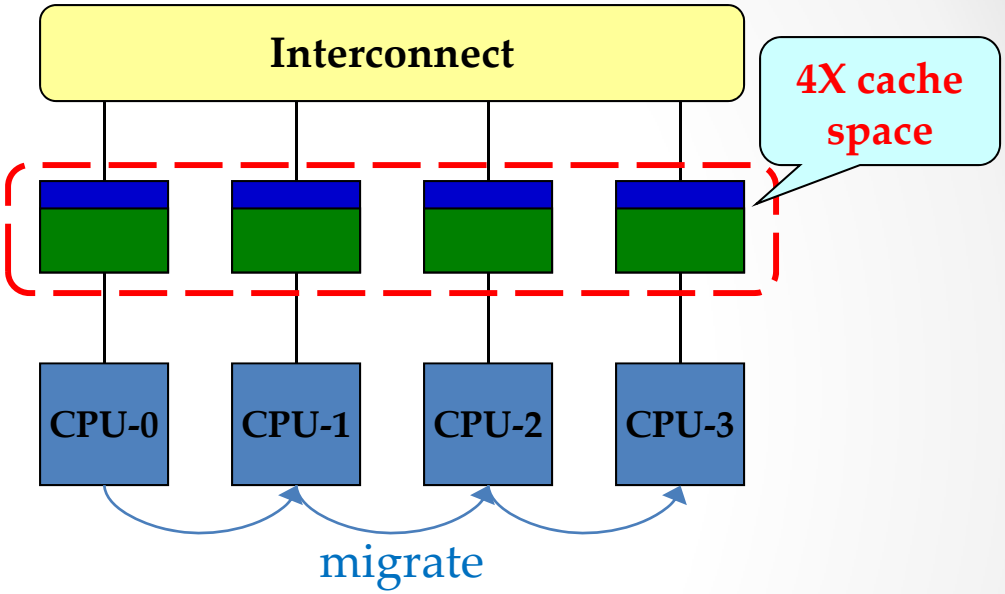
```
for i=1 to 100  {
      for j=1 to 1000      Loop1
            a[j] = a[j-1] + a[j+1]
       ……….
      for j=1 to 2000      Loop2
            b[j] = b[j-1] + b[j+1]
 }
```



- Can we exploit distributed caches even when the computation cannot be distributed?

Can We Exploit Idle Resources ?

```
for i=1 to 100  {
      for j=1 to 1000      Loop1
            a[j] = a[j-1] + a[j+1]
       ……….
      for j=1 to 2000      Loop2
            b[j] = b[j-1] + b[j+1]
}
```



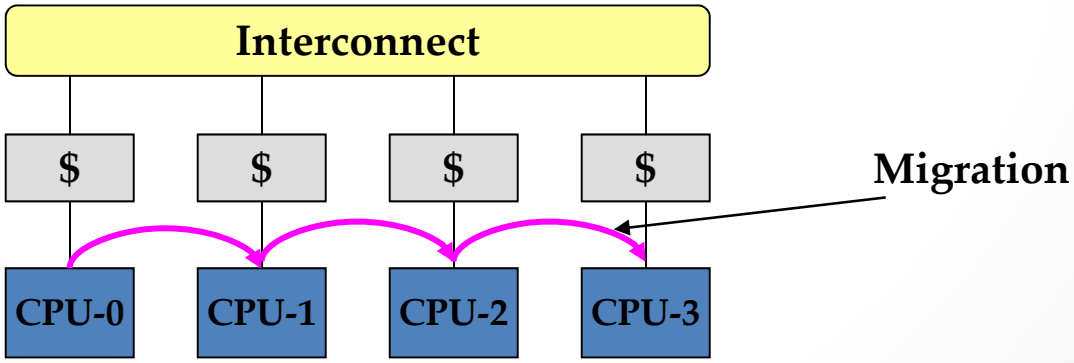
- *Data Spreading* uses core-to-core migration to distribute the data working set among multiple private caches.

By Relying on Entirely Software Techniques

- *Works on existing machines*
- Can extract application information easily to direct migration.
- Bridges the gap between complex hardware and higher program abstraction
  - Diverse memory hierarchy – shared/private, inclusive/exclusive

Our Software Techniques

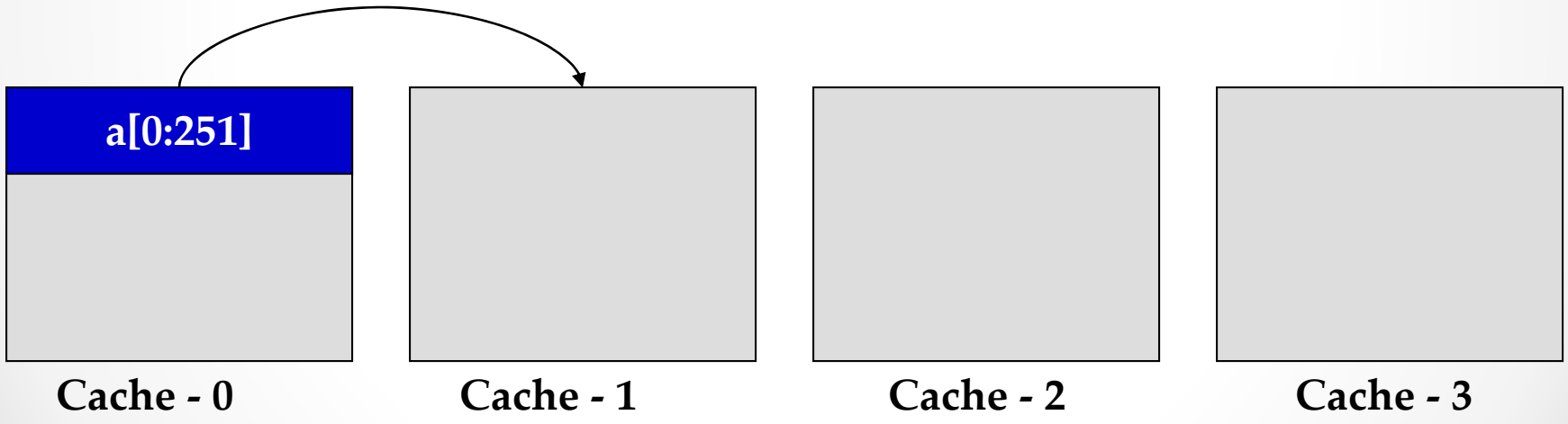
- *Retain serial execution*
- Target primarily memory intensive applications
- Use software controlled migration



Software Data Spreading

```
for i=1 to 100  {
    for j=1 to 1000      Loop1
        a[j] = a[j-1] + a[j+1]
    .....................
    for j=1 to 2000      Loop2
        b[j] = b[j-1] + b[j+1]
}
```

CPU 0

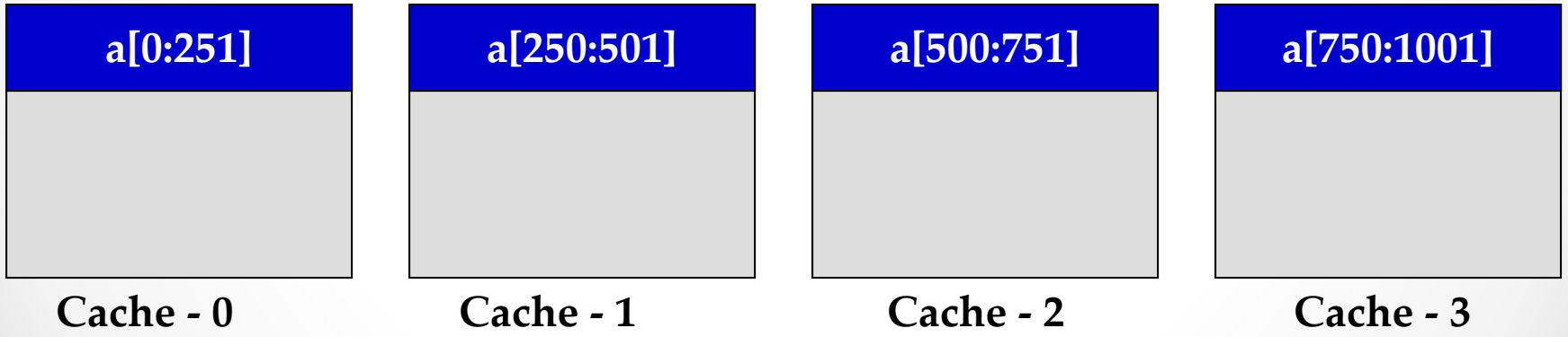


Iteration 1 – Spread **'a[0:1001]'**

Software Data Spreading

```
for i=1 to 100  {
    for j=1 to 1000      Loop1
        a[j] = a[j-1] + a[j+1]
    .....................
    for j=1 to 2000      Loop2
        b[j] = b[j-1] + b[j+1]
}
```

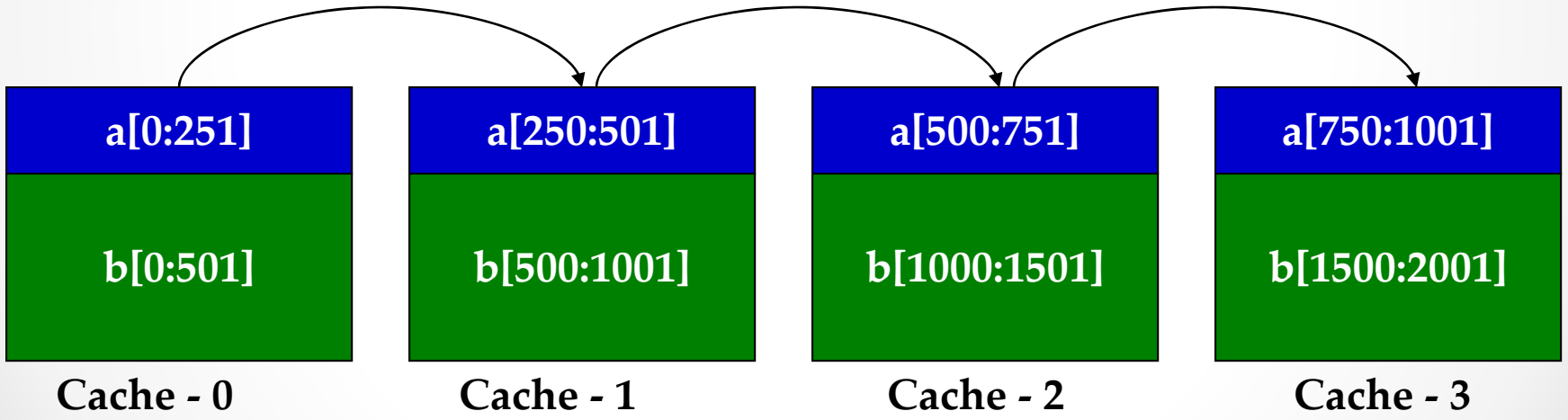
CPU 3



Iteration 1 – **'a[0:1001]'** spread across 4 caches

Software Data Spreading

```
for i=1 to 100  {
    for j=1 to 1000      Loop1
        a[j] = a[j-1] + a[j+1]
    ..........................
    for j=1 to 2000      Loop2
        b[j] = b[j-1] + b[j+1]
}
```



Iteration 1 – **'b[0:2001]'** spread across 4 caches

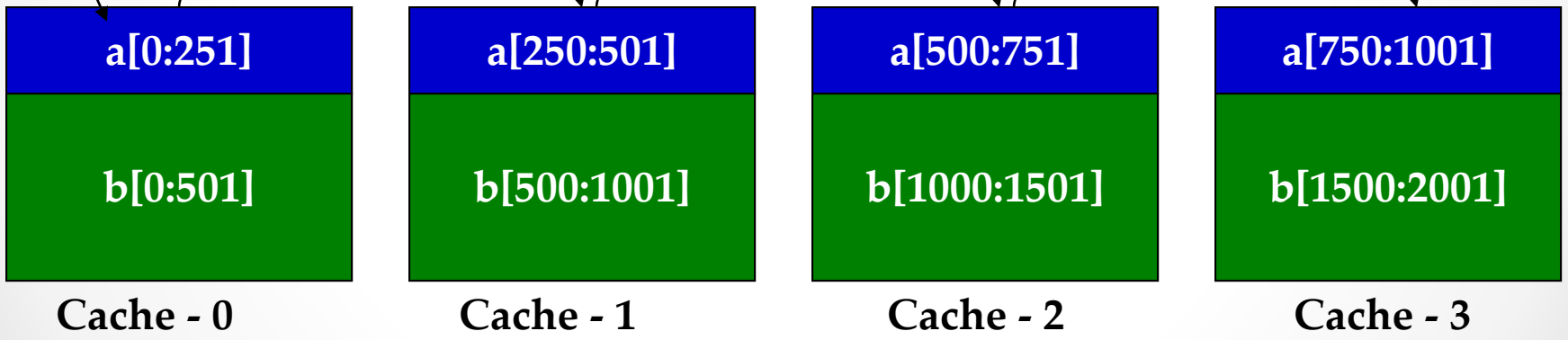
Software Data Spreading

```
for i=1 to 100  {
    for j=1 to 1000      Loop1
        a[j] = a[j-1] + a[j+1]
    .....................
    for j=1 to 2000      Loop2
        b[j] = b[j-1] + b[j+1]
}
```

CPU 0

'a[0:251]' in the cache

Computation Follows the Data

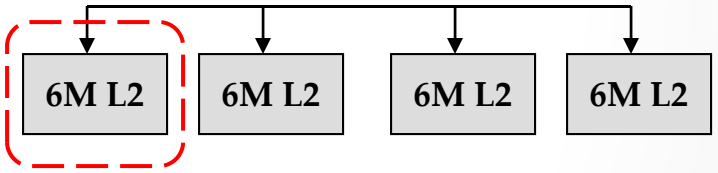


Iteration 2 – Hits in private cache

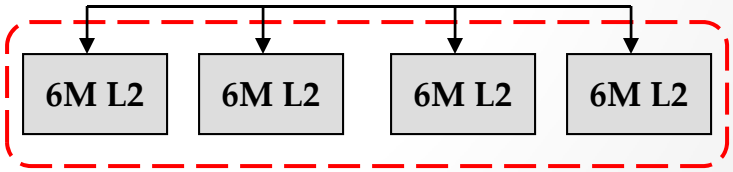
Software Data Spreading

```
for i=1 to 100  {
        for j=1 to 1000      Loop1
            a[j] = a[j-1] + a[j+1]
        .....................
        for j=1 to 2000      Loop2
            b[j] = b[j-1] + b[j+1]
}
```

- No Data Spreading
  - **98%** L2 Miss rate



- With Data Spreading
  - **2%** L2 Miss rate

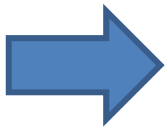


Data Spreading Issues

- What loops to spread?
  - Target memory-intensive
  - Needs the right type of reuse
  - Cannot migrate a loop and a containing loop
- When to migrate?
- How to migrate?

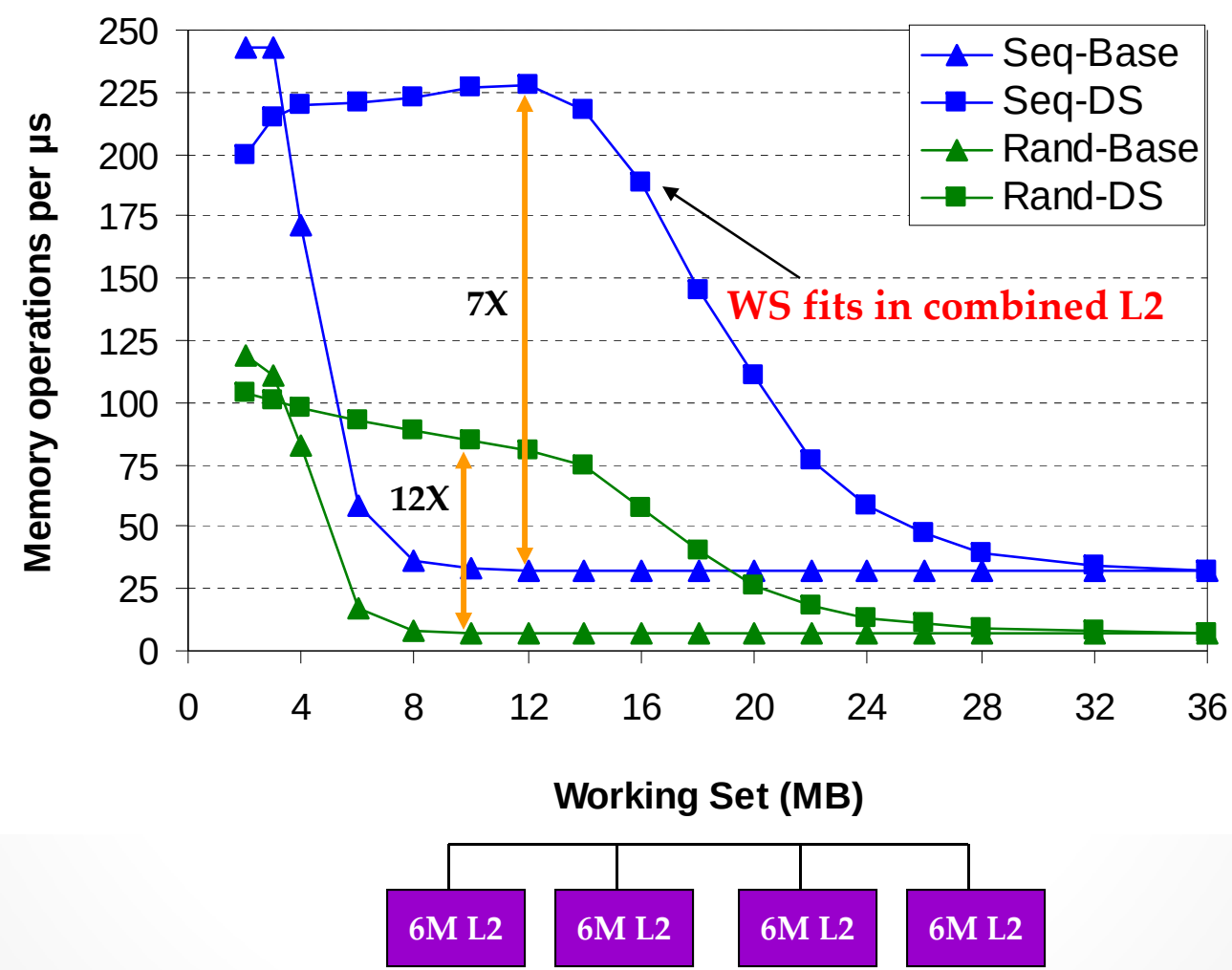
How to Spread Candidate Loops

```
for i=1 to 100 {          Loop0
    for j=1 to 1000       Loop1
        a[j] = a[j-1] + a[j+1]
    ....................
    for j=1 to 2000       Loop2
        b[j] = b[j-1] + b[j+1]
}
```



```
for i=1 to 100 {
    for k=0 to 3 {
        Migrate_to_CPU (k)
        for j=k*250+1 to 250*(k+1)
            a[j] = a[j-1] + a[j+1]
    }
    for k=0 to 3 {
        Migrate_to_CPU (k)
        for j=k*500+1 to 500*(k+1)
            b[j] = b[j-1] + b[j+1]
    }
}
```

Results – Dual Socket Core2Quad



Migration as a First-Class Compiler Primitive

- New architectures require new tools for effective compiler optimization
- Migration enables us to get the best of both worlds – distributed caches when we have distributed work, aggregated caches when we don't
- (removed)

Non-traditional Parallelism – the Big Points

- We need to exploit every opportunity to bridge the gap between available hardware parallelism and software parallelism
- The more parallel the hardware, the more performance is dominated by serial execution.
- Non-traditional parallelism enables parallel speedup of serial code.
- We exploit available threads/cores to
  - Precompute memory addresses
  - Generate new, improved code
  - Aggregate private cache space
- What other opportunities are there?

Questions?

- Collaborators on various projects I talked about today.
  - Jamison Collins
  - Hong Wang
  - John Shen
  - Christopher Hughes
  - Yong-Fong Lee
  - Dan Lavery
  - John P. Shen
  - Weifeng Zhang
  - Brad Calder
  - Md Kamruzzaman
  - Steven Swanson