Using Industry Standards to Exploit the Advantages and Resolve the Challenges of Multicore Technology

September 19, 2007

Markus Levy, EEMBC and Multicore Association

Enabling the Multicore Ecosystem

- Multicore Association™
  - Initial engagement began in May 2005
  - Industry-wide participation
  - Current efforts
    - Communications APIs
    - Debug API
- Embedded Microprocessor Benchmark Consortium® (EEMBC)
  - Industry benchmarks since 1997
  - Evaluating current and future MP platforms
    - Uncover MP bottlenecks
    - Compare multiprocessor vs. uni-processor

What Is Multicore’s Value?

- Is + < ?
- Is + = ?
- Is + > ?

Answer: It depends on the application

Multicore Issues to Solve

- Communications, synchronization, resource management between/among cores
- Debugging: connectivity, synchronization
- Distributed power management
- Concurrent programming
- OS virtualization
- Modeling and simulation
- Load balancing
- Algorithm partitioning
- Performance analysis

EMBC CERTIFIED

Perspective on COTS

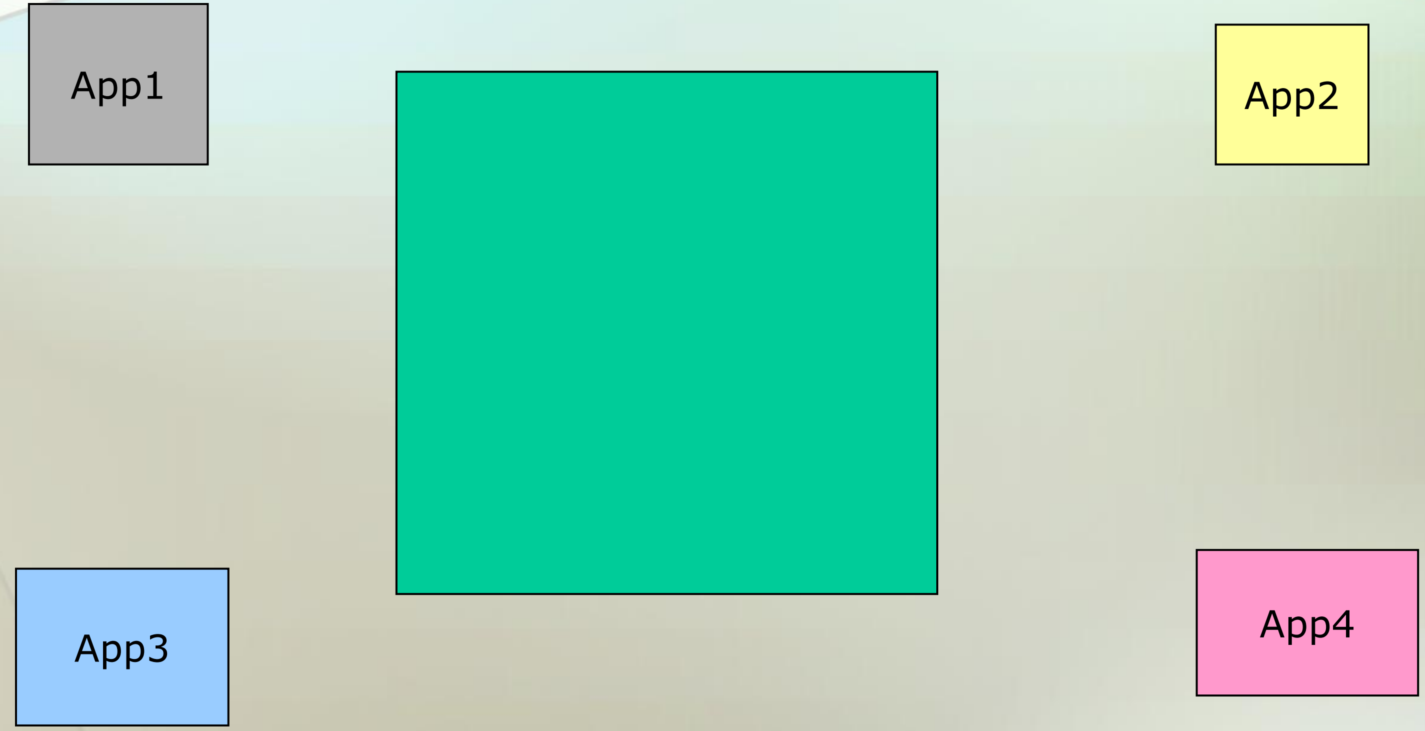
- ‘Ready-made’ solutions help overcome some of these challenges

- Extracting full benefits requires effort

- **COTS = Careful Optimizations To Succeed**

Multicore for Multiple Reasons

- Increase compute density
  - Centralization of distributed processing
  - All cores can run entirely different things; for example, four machines running in one package



App1

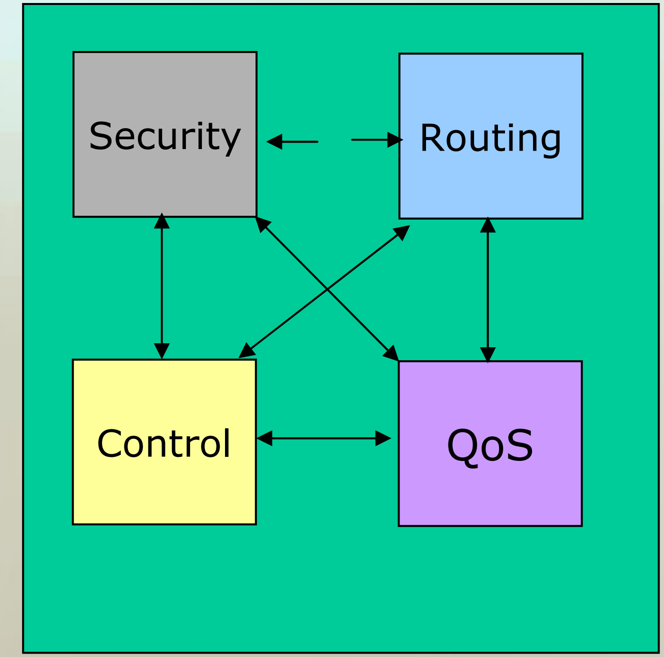
App2

App3

App4

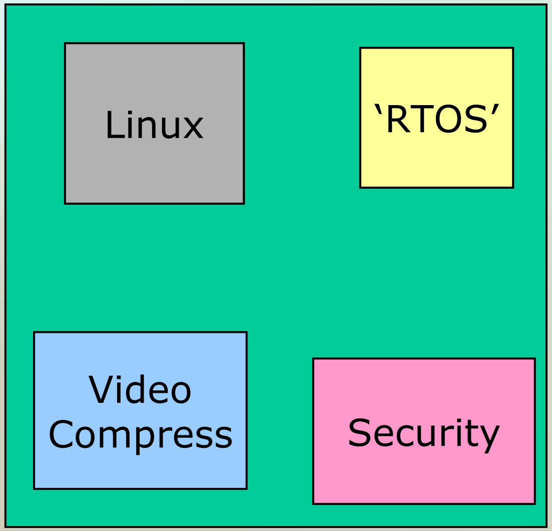
Multicore for Multiple Reasons

- Functional partitioning
  - While possible with multiprocessors, benefits from proximity and possible data sharing
  - Core 1 runs security, Core 2 runs routing algorithm, Core 3 enforces policy, etc.



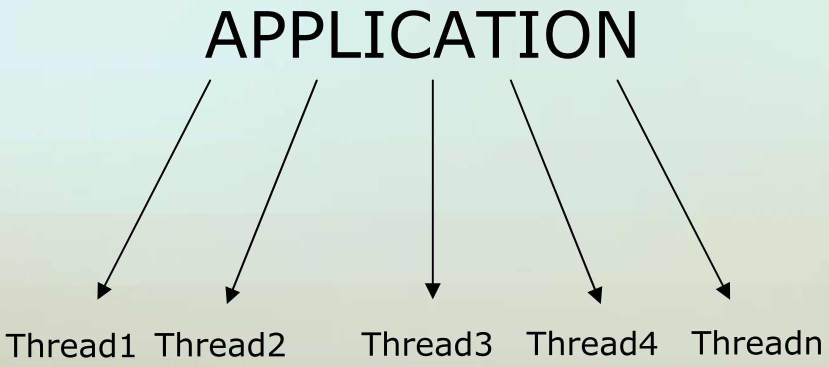
Multicore for Multiple Reasons

- Asynchronous multiprocessing (AMP)
  - Minimizes overhead and synchronization issues
  - Core 1 runs legacy OS, Core 2 runs RTOS, others do a variety of processing tasks (i.e. where applications can be optimized)



Multicore for Multiple Reasons

- Parallel pipelining
  - Taking advantage of proximity
  - The performance opportunity….



APPLICATION

Thread1 Thread2 Thread3 Thread4 Threadn

Benchmarking Multicore – What’s Important?

- Scalability where contexts exceed resources
- Single versus multiprocessor
- Memory and I/O bandwidth
- Inter-core communications
- OS scheduling support
- Efficiency of synchronization
- System-level functionality

Bird's Eye View to Multicore Performance Analysis

- Shared Memory
  - Semantics of a thread-based API
  - Supporting coherency
  - Applicable to MPC8641D, Intel x86
- Message Based
  - Heterogeneous MP solutions lack common programming paradigm
  - Distributed memory architectures
  - Applicable to i.MX devices, many others
- Scenario Driven
  - Black Box Approach
  - Performance based on real implementations
  - Define methodology, inputs, expected outputs

EEMBC CERTIFIED

Phase 1 Methodology for Benchmarking Multicore

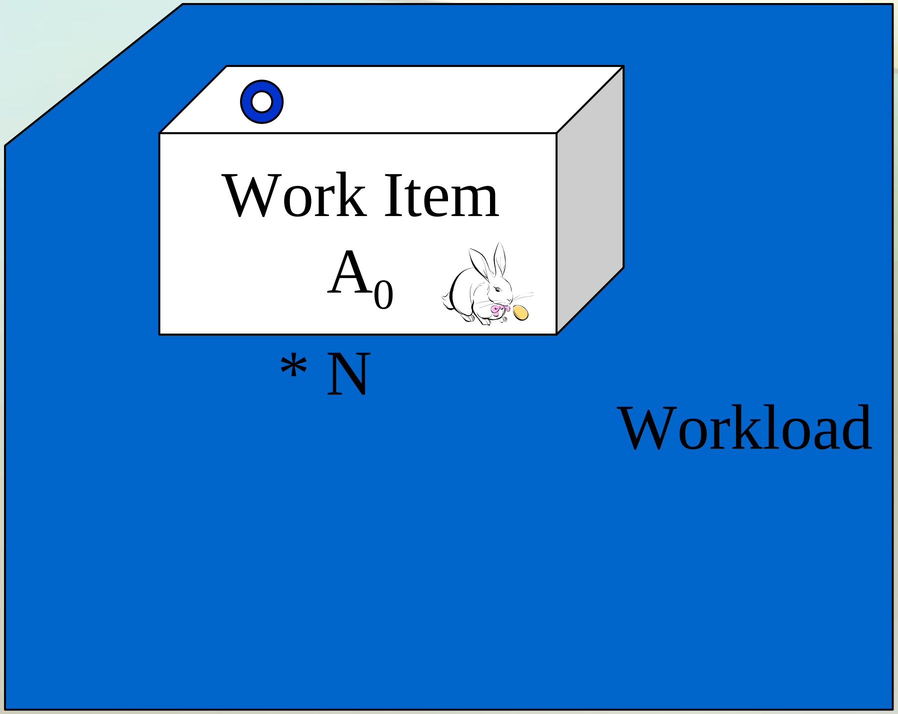
- Software assumes homogeneity across processing elements
- Scalability of general purpose processing, as opposed to accelerator offloading
- Contexts
  - Standard technique to express concurrency
  - Each context has symmetric memory visibility
- Abstraction
  - Test harness abstracts the hardware into basic software threading model
- EEMBC™ has patent-pending on methodology and implementation

Multicore Benchmarking Primer

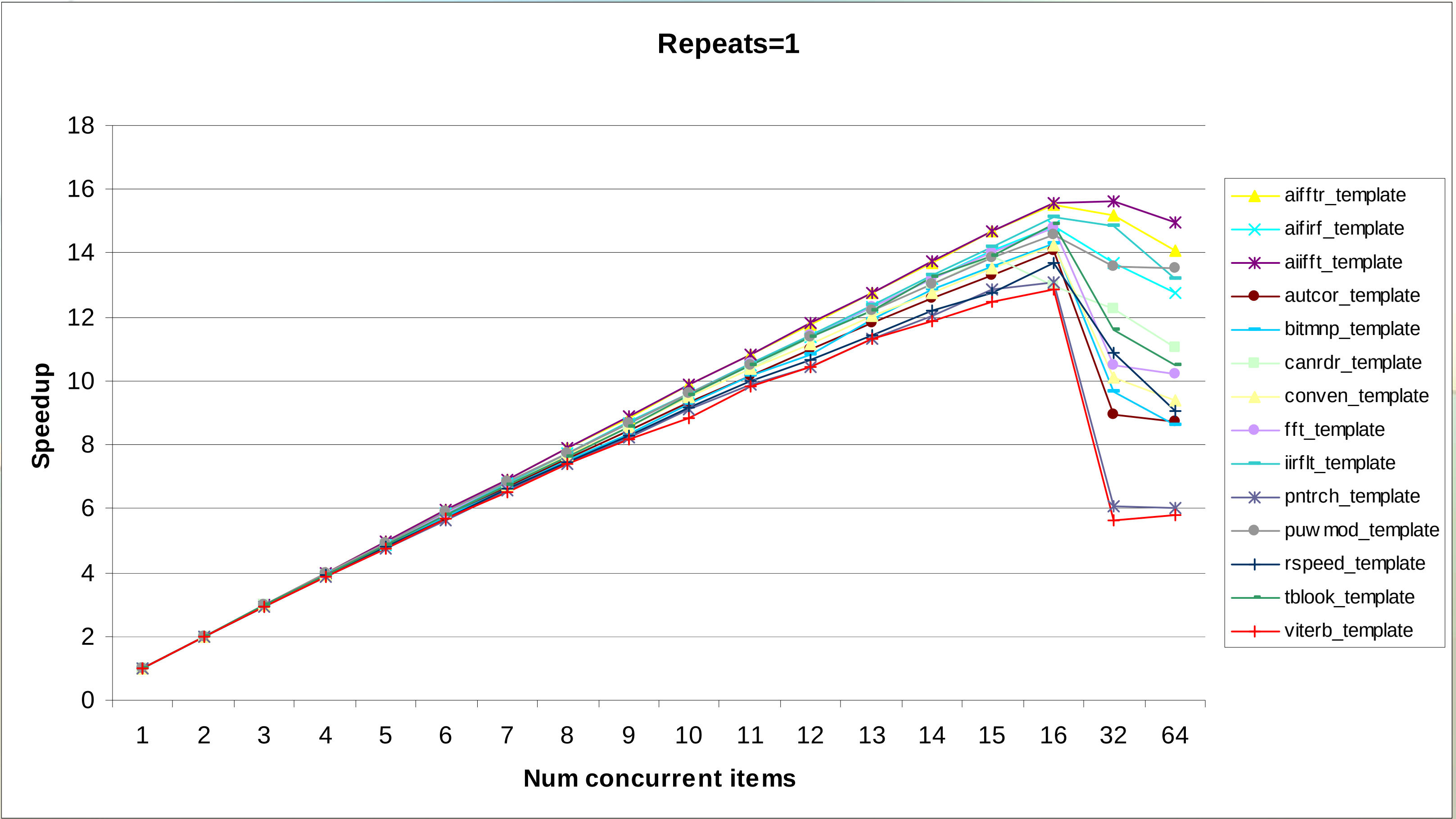
Multicore Performance Analysis is Multi-Faceted Problem
More Cores ≠ More Performance
A Major Hardware Architecture and Programming Issue

Workloads, Work Items, Concurrency

- Workload can contain 1 or more work items, executed <N> times
- Can execute up to <N> work items concurrently
  - Performance limited by OS scheduling, # of cores, memory bandwidth
- Similar to spec-rate



Avoid Oversubscription

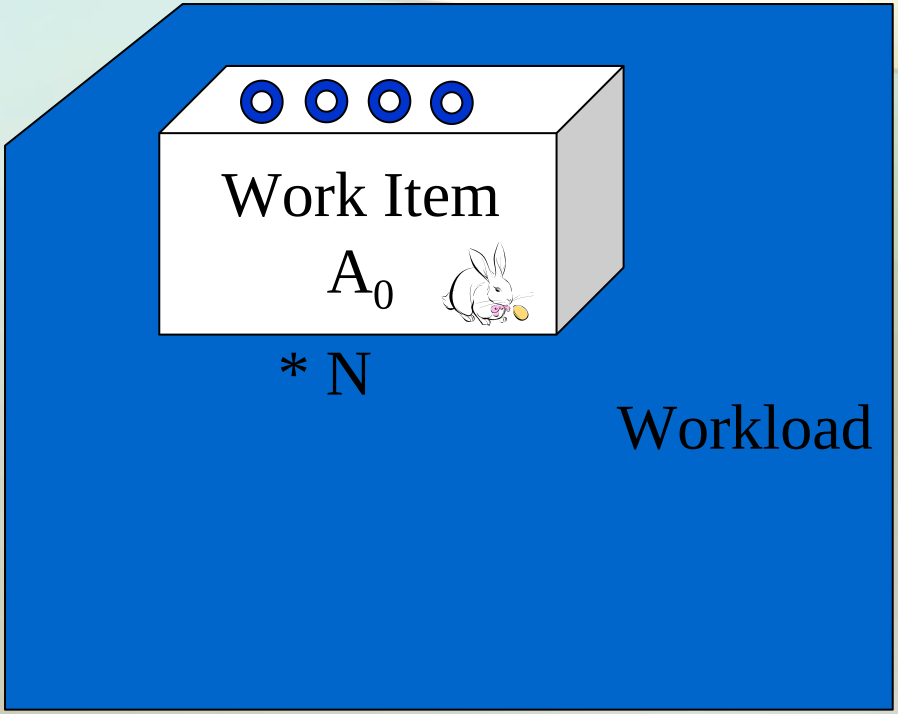


Linear speedup until reaching 16 concurrent items = number of cores
Effects vary based on algorithm and platform

Workloads Can Contain Multiple Contexts

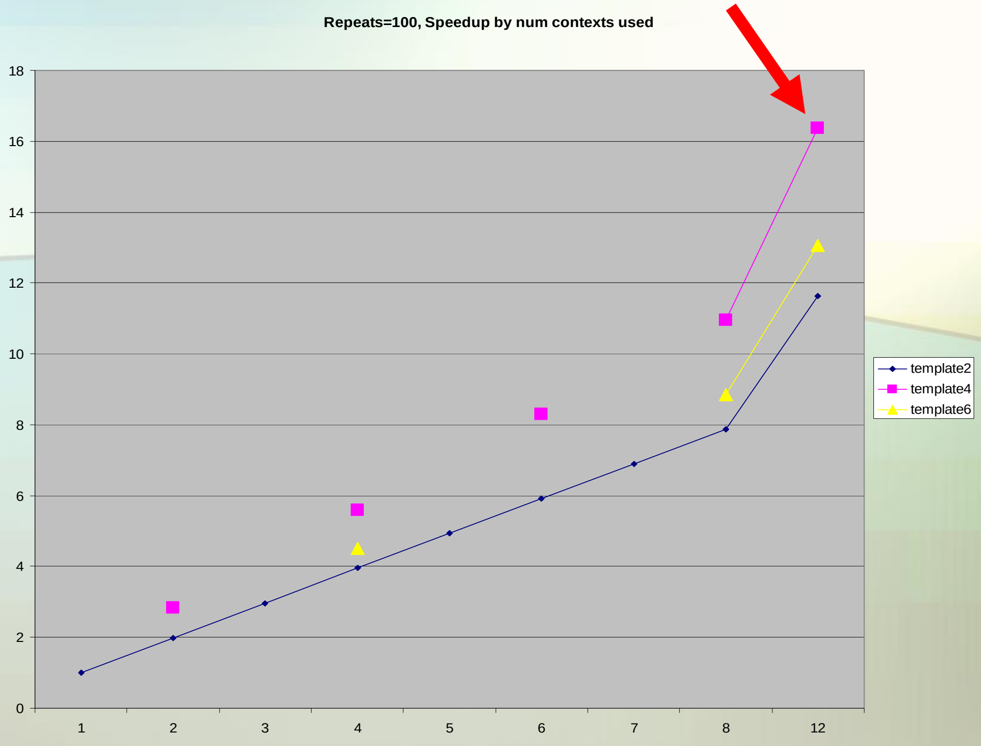
- Split Work Item into multiple contexts
- Increase parallelism within algorithms
- Always improves performance?

◎ = context



Synchronization Overhead and Parallelization Side Effects

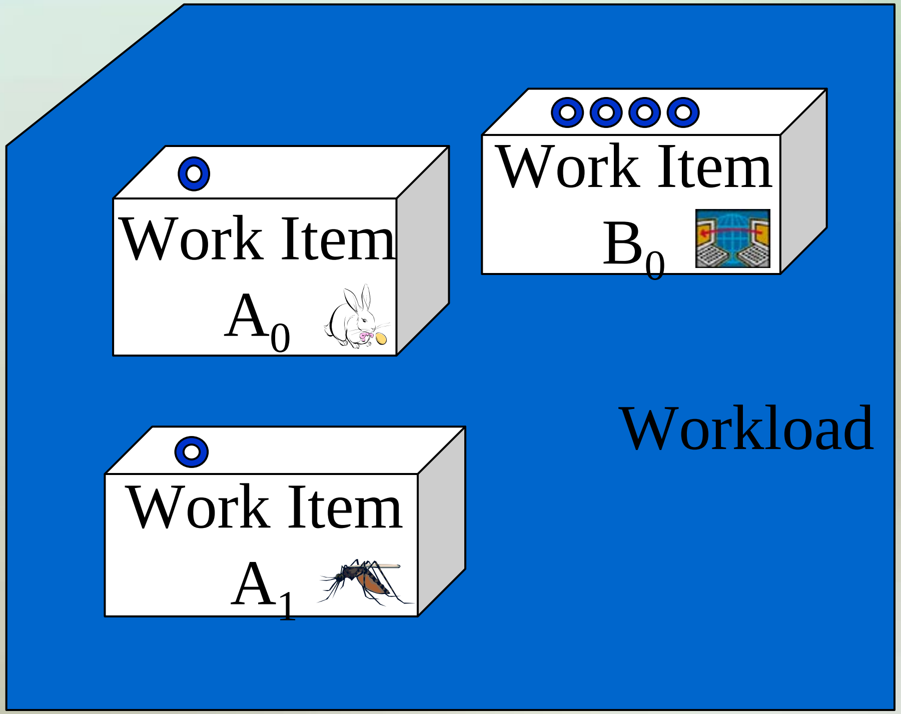
- template2: 1 instance of rgbhpg (rate)
- template4: 1 instance of rgbhpg dual context
- template6: 1 instance of rgbhpg quad context



Splitting job in 2 and using 6 concurrent items (total of 12 software contexts) is more efficient than using 12 concurrent items or splitting in 4 and using 3 concurrent items

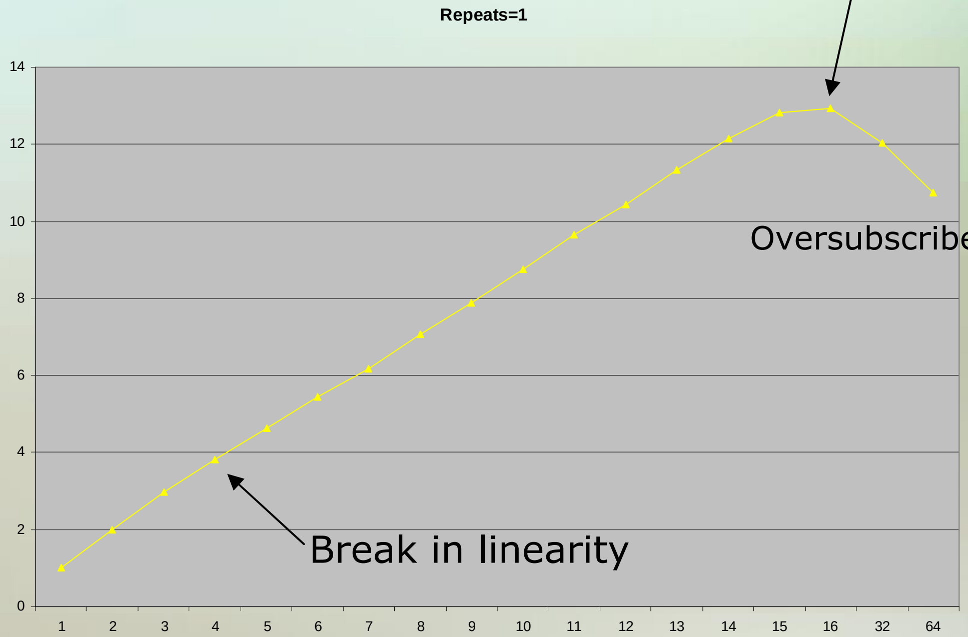
Increasing Workload Complexity and Realism

- Workload can contain a mixture of items
- Example
  - Item A applied to 2 different datasets
  - Item B explicitly using 4 execution contexts



Cache Effects Hit Early

- Combine filter items
  - 1 instance of rgbyiq using one context
  - 1 instance of rgbhpg using one context
  - 2 instances of rgbhpg using 2 contexts each
    - RGB to HPG conversion, splitting the image processing among 2 contexts



Max out at 13x

Oversubscribed

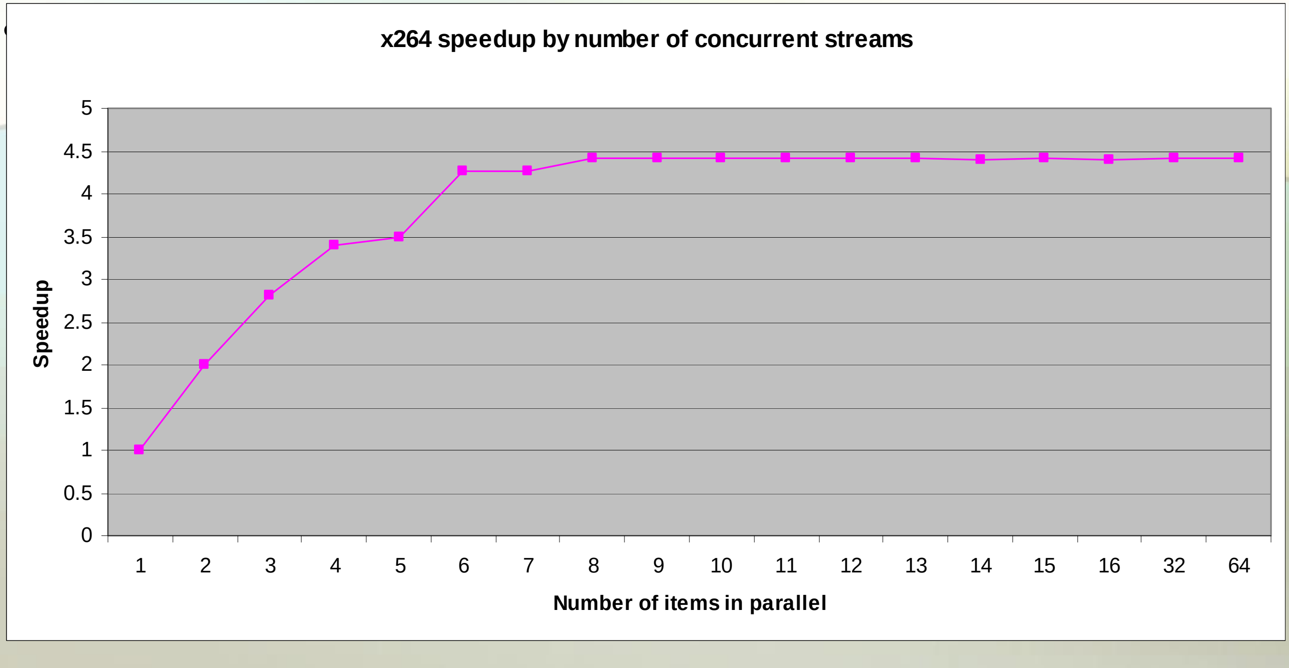
Break in linearity

Contexts

Memory Limitations

- Test workload for H.264 encoder
  - 2 contexts on each stream
  - 6 streams

Number of software contexts used is double the number of concurrent items (2 contexts per stream). Still, overall benefit over single core case is 9x rather then 16x on a 16 core system.



NOTE: Memory limited after 4 items in parallel

What Does This Picture Have in Common With Multicore?

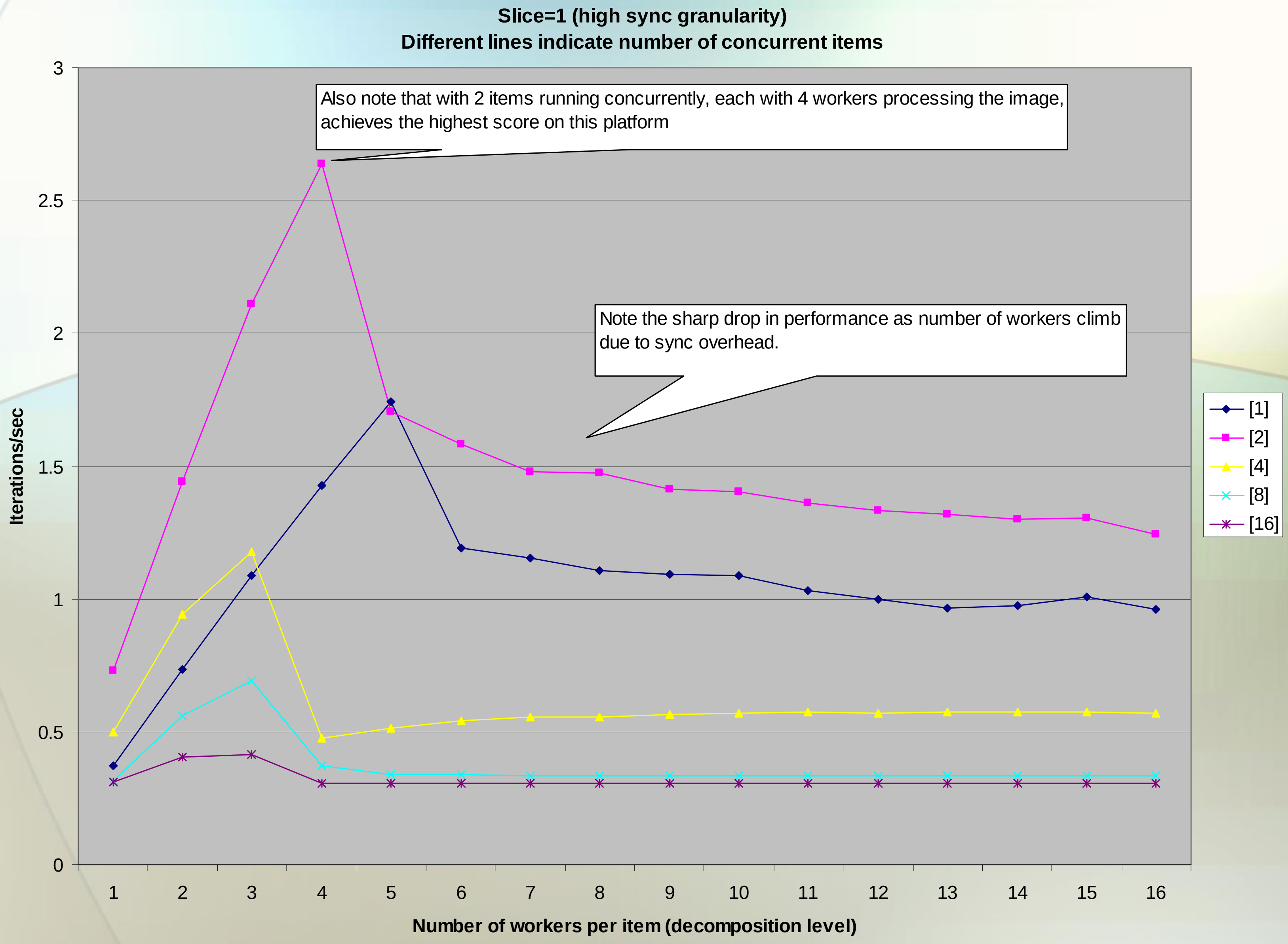
EMBC CERTIFIED

'Rotate' Benchmark Stresses Multicore in Many Ways

- A few points describing this benchmark.
  - Rotate greyscale picture 90 deg counter clock-wise.
  - Data shown is specifically for a 4 M-Pixel picture.
- Benchmark parameters allow different slice sizes, concurrent items, and number of workers
  - Workers = # of software contexts working on rotating a single picture concurrently
  - Items = # of pictures to process
  - Slice size = # of lines in a horizontal slice of the photo for each worker to process each time

Note: When total # SW Contexts <= # HW contexts, there is no difference.

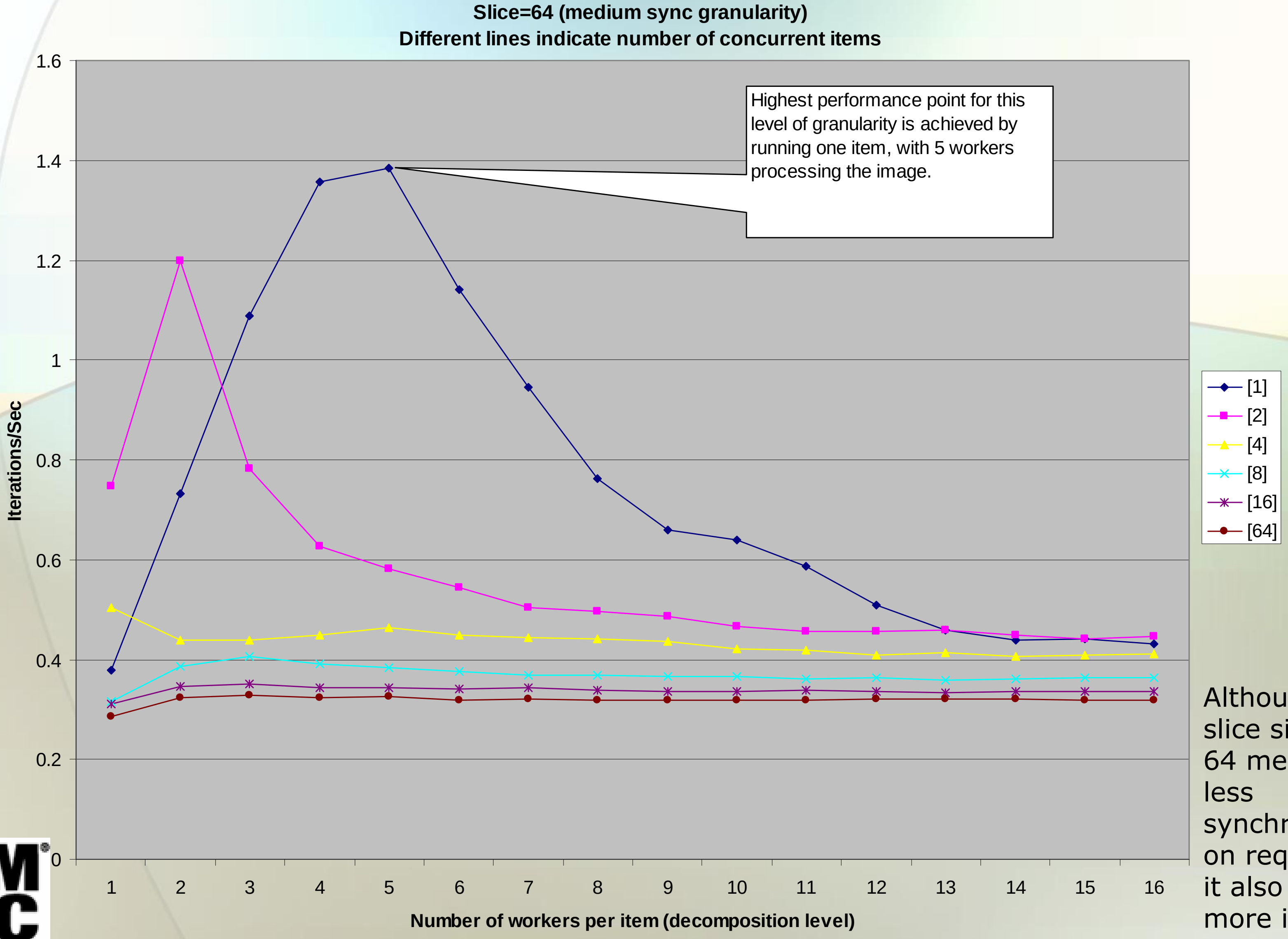
Balance Data Usage and Synchronization



Also note that with 2 items running concurrently, each with 4 workers processing the image, achieves the highest score on this platform

Note the sharp drop in performance as number of workers climb due to sync overhead.

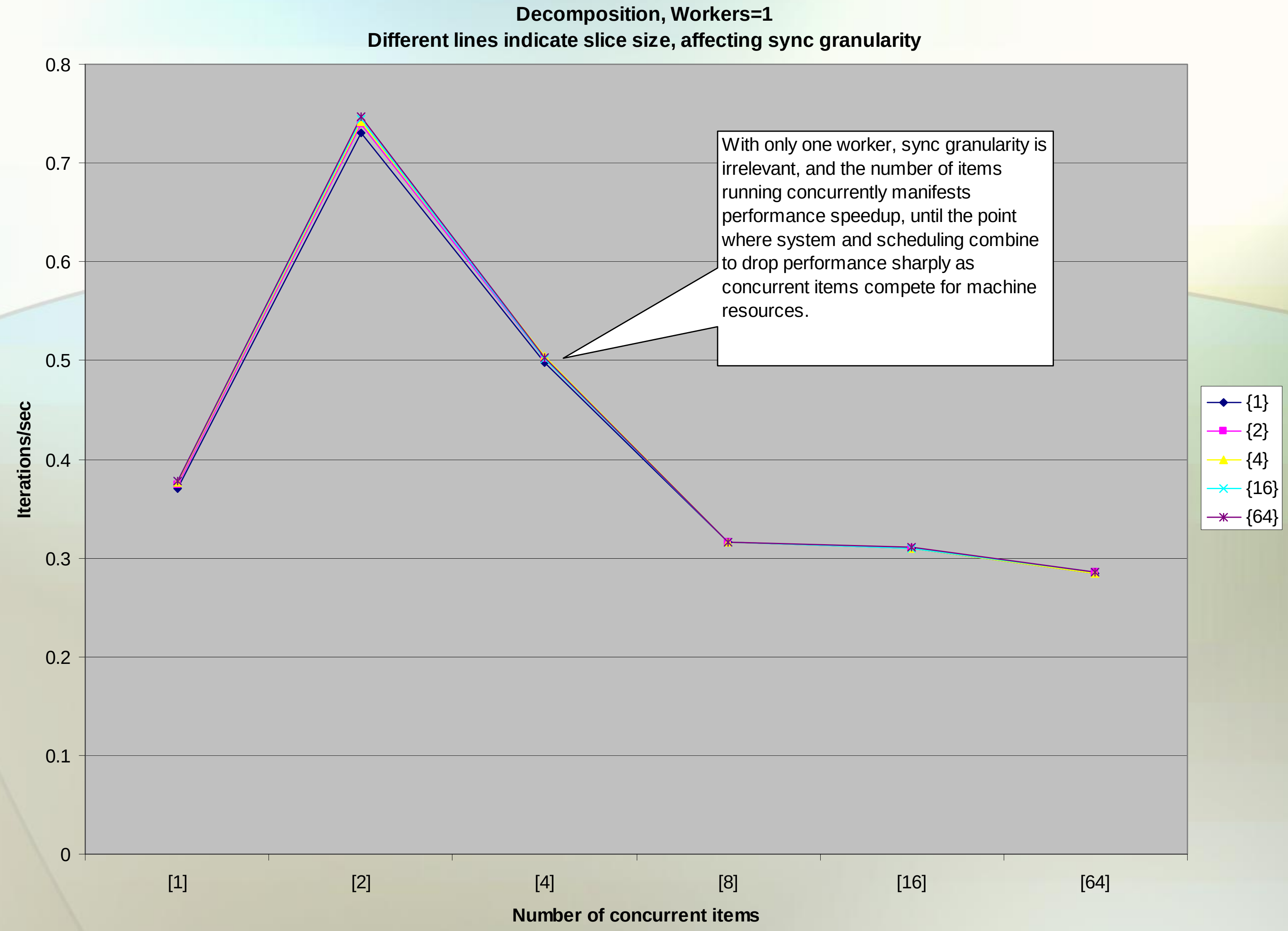
Lots of Data Slams Cache



Highest performance point for this level of granularity is achieved by running one item, with 5 workers processing the image.

Although slice size of 64 means less synchronization required, it also means more impact on cache.

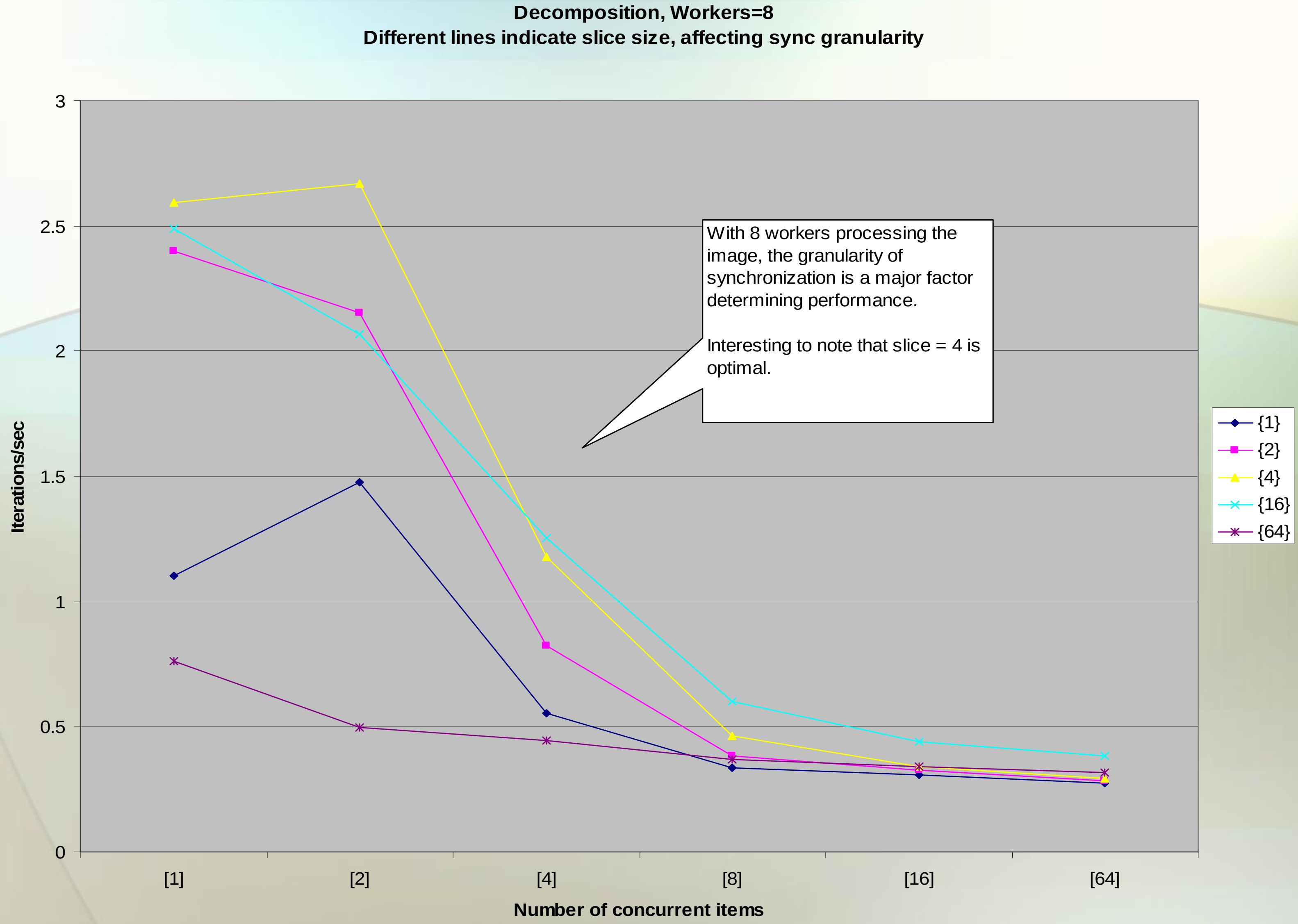
Critical to Optimize for Scheduling



With only one worker, sync granularity is irrelevant, and the number of items running concurrently manifests performance speedup, until the point where system and scheduling combine to drop performance sharply as concurrent items compete for machine resources.

Exposing Memory Bottlenecks

Slice of 2, 4,16 closest in performance.

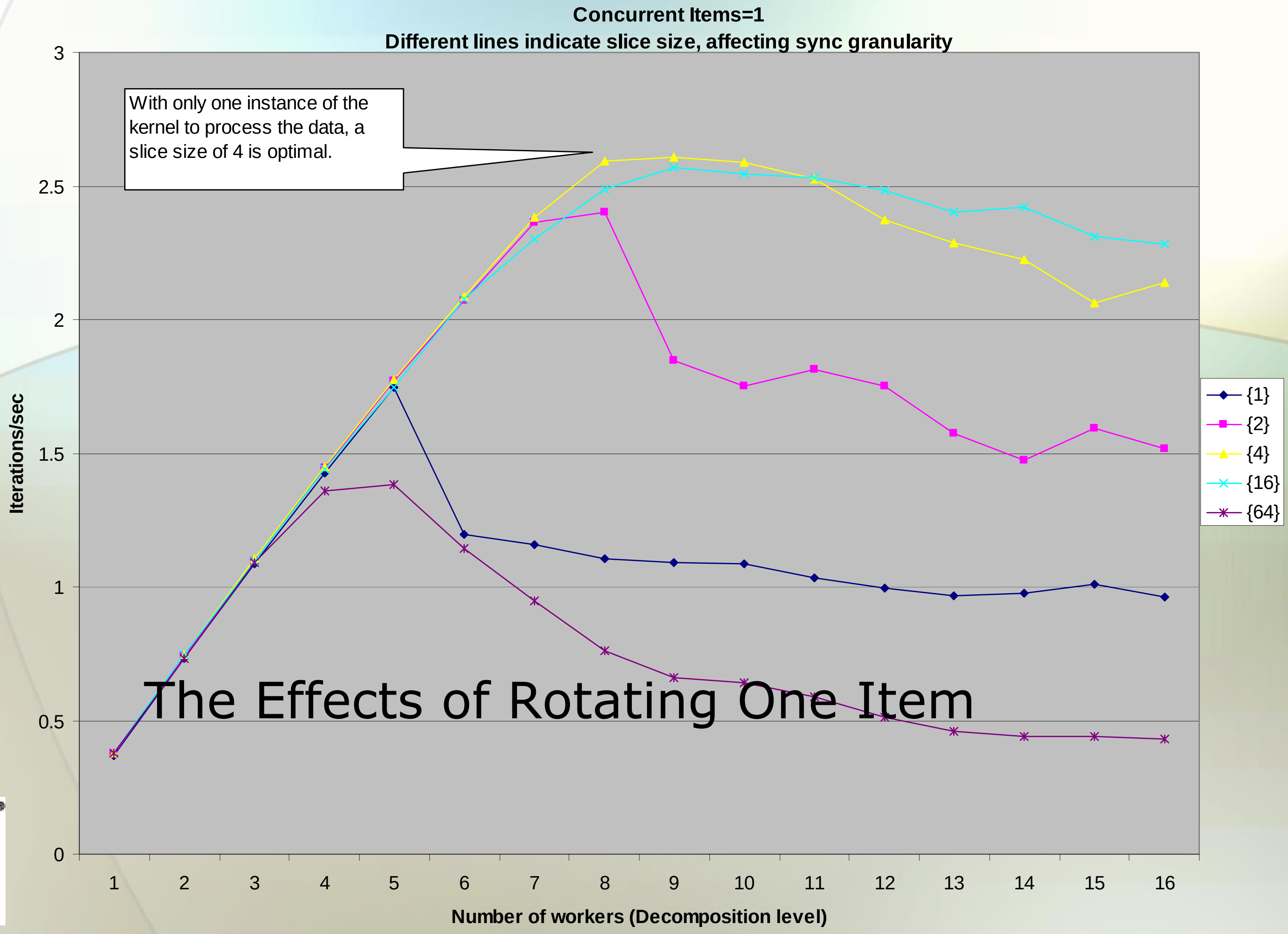


Decomposition, Workers=8
Different lines indicate slice size, affecting sync granularity

With 8 workers processing the image, the granularity of synchronization is a major factor determining performance.

Interesting to note that slice = 4 is optimal.

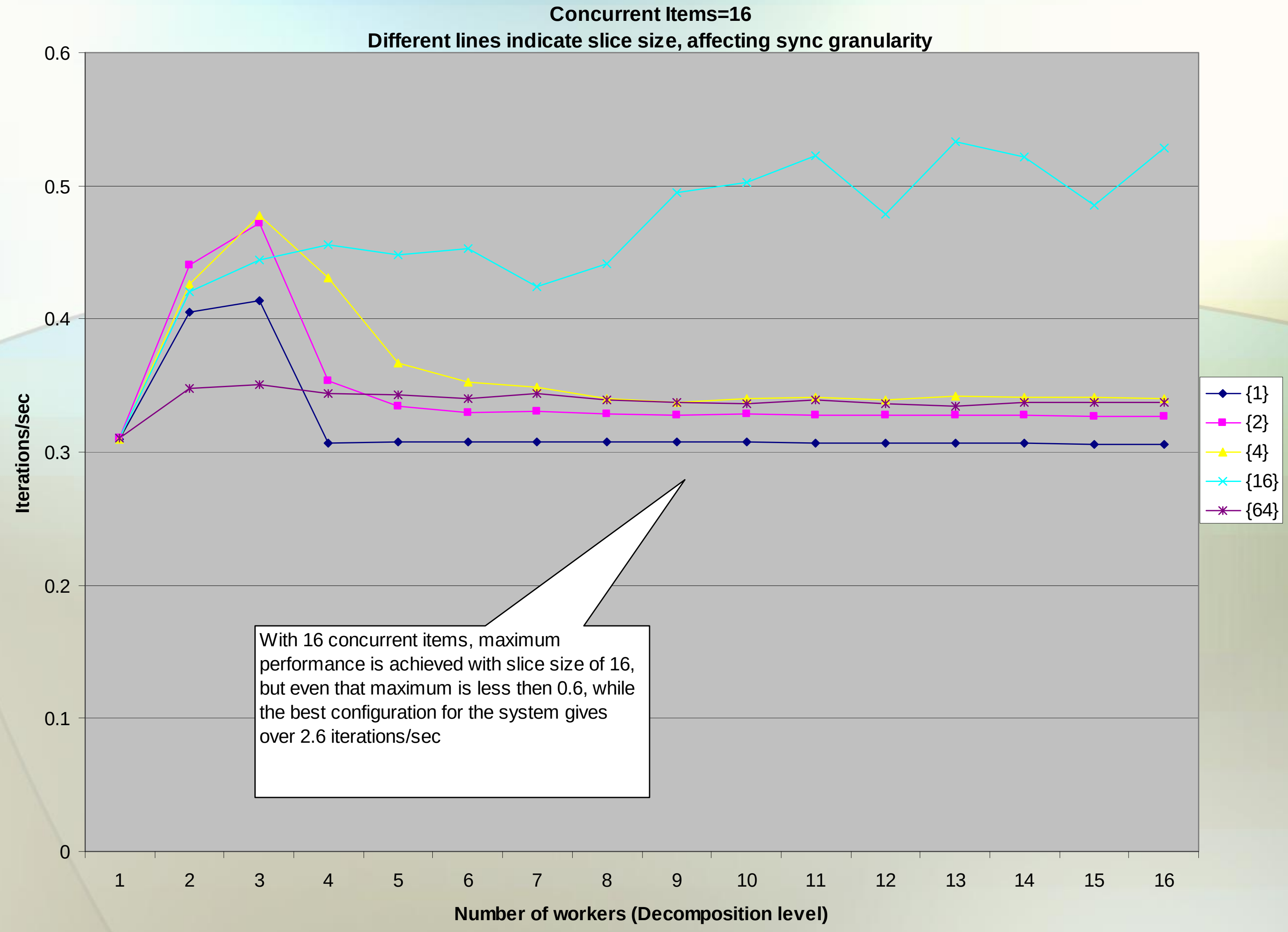
Less is More



With only one instance of the kernel to process the data, a slice size of 4 is optimal.

The Effects of Rotating One Item

More is Less



With 16 concurrent items, maximum performance is achieved with slice size of 16, but even that maximum is less then 0.6, while the best configuration for the system gives over 2.6 iterations/sec

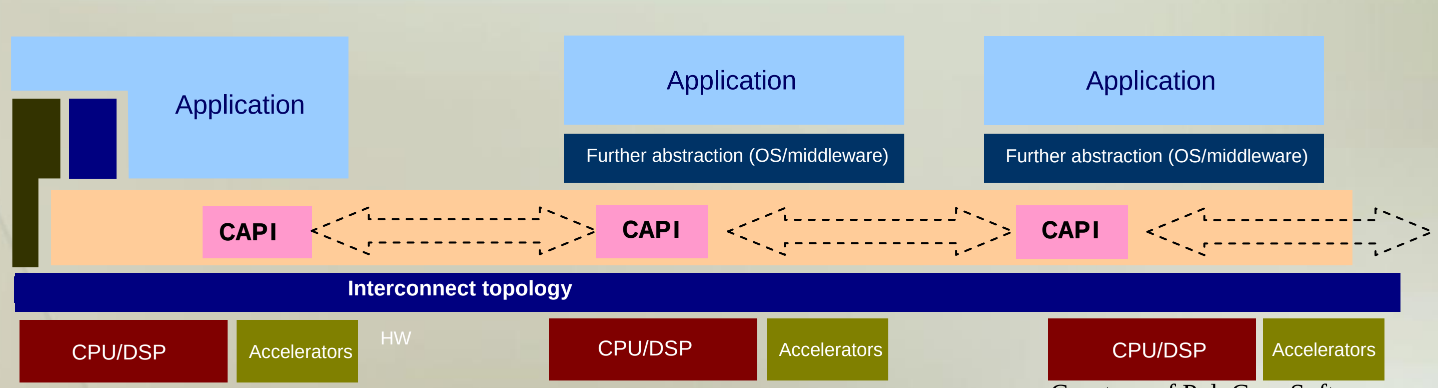
Communication API (MCAPI) Target Domain

- Multiple cores on a chip and multiple chips on a board
  - Closely distributed and/or tightly-coupled systems
- Heterogeneous and homogeneous systems
  - Cores, interconnects, memory architectures, OSes
- Scalable: 2 – 1000’s of cores
  - Assumes the ‘cost’ of messaging/routing < computation of a node

CAPI Features and Performance Considerations

- Very small runtime footprint
- Low-level messaging API
- Low-level streaming API
- High efficiency

- Seeking reviewers for pre-release version

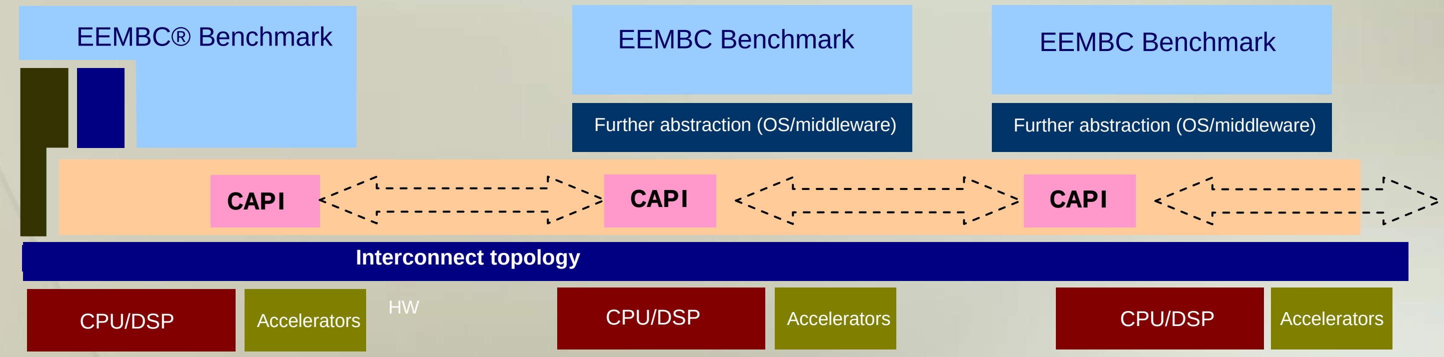


Courtesy of PolyCore Software

Benchmark Implementation on a Message-Based Platform

- Uses CAPI as underlying infrastructure and test harness
  - Portable implementation, even with a heterogeneous platform

- Divide applications and/or algorithms into smallest possible functional modules
  - Each module is compiled separately



Multicore Opportunities and Challenges

- Multicore technology is inevitable
- It’s time to begin implementing
- The Multicore Association™ will help ease the transition
- EEMBC® will help analyze the performance benefits
  - Patent pending on new multicore benchmark technique