EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System

By Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang.

Chris Adamopoulos

Dept of Computer & Information Sciences

University of Delaware

CISC 879 : Software Support for Multicore Architectures

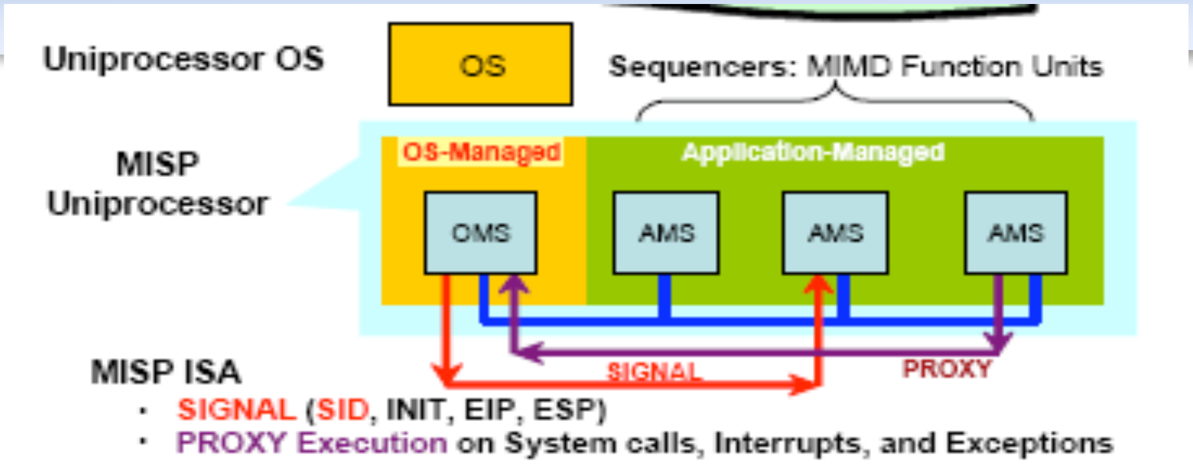
Outline

- MISP Background
- EXO Architecture
- CHI Runtime Environment
- Experiment Setup
- Performance Results
- Conclusion

MISP

- Multiple Instruction Stream Processing
  - An Multiple Instruction Multiple Data (MIMD) ISA
  - Developed by Intel to efficiently utilize “heterogeneous” cores
- Introduces two new types of resources:
  - Shred: User-level thread
  - Sequencers: abstract processing core capable in fetching and executing shreds
  - Two-types
    - OMS: OS-managed sequencer
    - AMS: Application-managed sequencer exposed by the programmer

MISP Processor


Uniprocessor OS
OS
Sequencers: MIMD Function Units
MISP Uniprocessor
OS-Managed
Application-Managed
OMS
AMS
AMS
AMS
SIGNAL
PROXY
MISP ISA
• SIGNAL (SID, INIT, EIP, ESP)
• PROXY Execution on System calls, Interrupts, and Exceptions

- MISP processor consists of two or more sequencers.
  - One managed by OS and one or more by applications
- AMS are directly managed by applications
  - Shreds are schedule by runtime environment, not by OS
  - Achieves parallelism for shreds are run concurrently and asymmetrically
- OMS acts as interface between OS and MISP processor
- To the OS, it see the MISP processor as ONE processor

EXOCHI

EXO Architecture:

- Extension of MISP Architecture
- MISP Exoskeleton
- Address Translation Remapping
- Collaborative Exception Handling

CHI Environment:

- C for Heterogeneous Integration
- Responsible for shred scheduling at runtime.
- Inline Assembly Support
- OpenMP Pragma Extension
- Work-Queuing
- CHI Runtime Environment

A Class of its Own

- Tightly-coupled: CPU manages threads for the co-processor and waits until execution is finished.
  - EXOCHI allows co-processors (AMS) to independently sequence and concurrently execute multiple streams at once.
- Loosely-coupled: CPU and co-processors separated and managed by OS and device drivers respectively
  - EXOCHI's sequencers are directly exposed to application programs and do not require OS management
  - Shred scheduling and communication supported by CHI runtime and shared virtual memory.

EXO Architecture Prototype

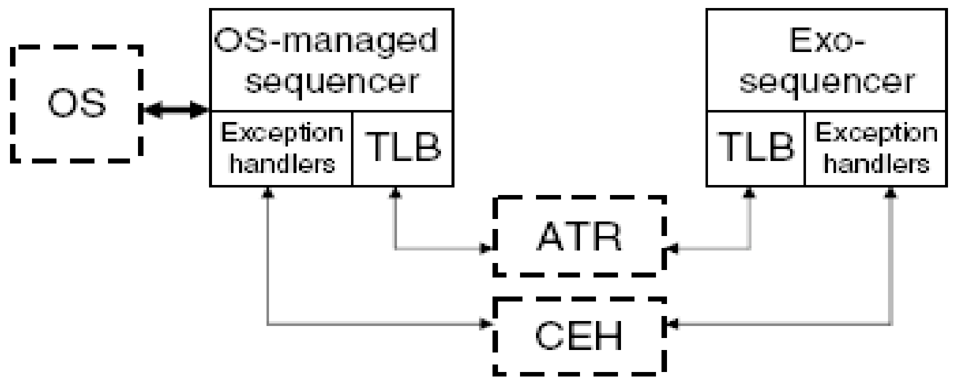


Figure 2. ATR and CEH between Heterogeneous Sequencers

- Provides support to the hardware component to EXOCHI
- Compose with Intel Core 2 Duo coupled with Intel Graphics Media Accelerator X3000 as heterogeneous co-processors.
  - Intel Core 2 Duo acts as the OS-managed sequencers (OMS)
  - X3000 are implemented as exosequencers supported by the MISP exoskeleton.
  - Exosequencers treated similar to application-managed sequencers.

MISP Exoskeleton

- Support interaction with OMS and exosequencers (AMS)
- Integrating AMS to MISP's signaling and communication mechanisms
  - Exposing AMS to applications and the programmer.
- OMS can create and dispatch shreds to be run on the AMS
  - No requirement from the OS

Address Translation Remapping

- Allows the OS to fix page faults made by the AMS in shared virtual memory
  - Necessary as EXOCHI's OMS and AMS have different ISA.
- When a TLB miss occurs:
  - Shred execution suspends and calls OMS for proxy execution
  - In MISP, OMS uses a proxy handler to contact and correct the page fault in place of the AMS
  - ATR recodes OMS page table entry to same format as the AMS.
  - Inserts OMS table entry into the AMS TLB and it will point to the identical physical page on the OMS to access the data.
  - AMS continues shred execution

Address Translation Remapping

- Benefits and support shared virtual memory space
    - Performs data communication and synchronization between OMS and AMS
    - Shared data structures can be transferred between different cores
    - Efficient as it does not heavily rely on data copying as with GPGPU
- ATR does not guarantee cache coherence
    - For a shared variable on OMS to be process on an AMS, OMS must flush its cache back to main memory
    - The reverse is true for AMS
    - Programmers utilize critical sections to prevent sequencers from reading incorrect data.

Collaborative Exception Handling

- Similar to ATR
- When an exception via instructions occurs on an AMS
  - In MISP, shred execution halted and instruction is replayed by the OMS
  - CEH allows the OS to directly handles the exception instruction by proxy
    - Via OS services, such as, Structural Exception Handling
  - When the exception is finished, AMS is updated with the results and resumed execution

C for Heterogeneous Integration

- Provides programming environment enabling AMS to be managed by user-level applications
- As opposed to other architectures relying on the CPU or OS to manage their threads
- The CHI runtime library is responsible for the scheduling of shreds amongst AMS
- Support for CHI's capabilities are a result for extending OpenMP pragmas for heterogeneous architectures.

Inline Assembly Support

- Programmers are able to utilize instructions and features for AMS in assembly
  - These instructions are not recognized by the compiler.
- Allows the performance for many sections to be custom optimized by the programmers.
- Compilers support can be extended to domain-specifics programming languages.
- For CHI, OpenMP “target” clauses specifies the target machine for which the assembly block should be assembled for.

```
pragma omp parallel target(...)
    __asm {
    .......
    }
```

OpenMP Pragma Extension

- “Parallel” pragma reconfigure to generate shreds for specified target machine.
- “Target” clause specifies target machine for which shreds will be spawned for.
- Programmers can exploit thread-level parallelism without worrying about how shreds are created, scheduled, and implemented.
- When “Parallel” pragma is encountered:
    - OMS shred, acting as the master thread, spawns shreds for target machine equal to num_threads
    - A call is put to the CHI runtime layer to dispatch and schedule shreds amongst the AMS.
    - Assembly block specified for AMS are executed concurrently and asymmetrically.

OpenMP Work-Queuing

- CHI’s queuing model following producer-consumer method to support inter-shred dependencies.
- Relies on taskq and task constructs to ensure dependencies amongst shreds for the AMS.
- taskq pragma constructs an empty queue for each task construct of code to be executed serially.
- When taskq is encountered:
  - OMS call CHI runtime to pick one shred as a root shred.
  - Root shred execute a loop within taskq construct.
  - For each “task” encountered, CHI runtime created a child shred and places it in the queue only associated with that specific taskq construct and target machine.

CHI Runtime Support

- The key factor to creating, scheduling, and parallel execution of shreds.
- Responsible for handling exception instructions and managing shared virtual memory objects between OMS and AMS.
- Abstraction layer used to hide the detail in managing AMS from programmer.
- Purpose: allow applications to direct utilization of hardware features by calling to the source file instead to change the compiler.

CHI Runtime Support

- Descriptors are API's interpreting the attributes of shared variables by shreds.
- Efficient programming tool for AMS to successfully access shared data.
- Applications can harness AMS capabilities.
- #1 chi_alloc_desc(targetISA,ptr,mode,width,height)
  - Allocates and specify variable as input or output and its size
- #2 chi_free_desc(targetISA,desc)
  - Deallocates variable
- #3 chi_modify_desc(targetISA,desc,attrib id,value)
  - Modify variable attributes
- #4 chi_set_feature(targetISA,feature id,value)
  - Change global state for AMS for all shreds
- #5 chi_set_feature pershred(targetISA,shr id,feature id,value)
  - Change global state for AMS for one shred.

A CHI Program Example

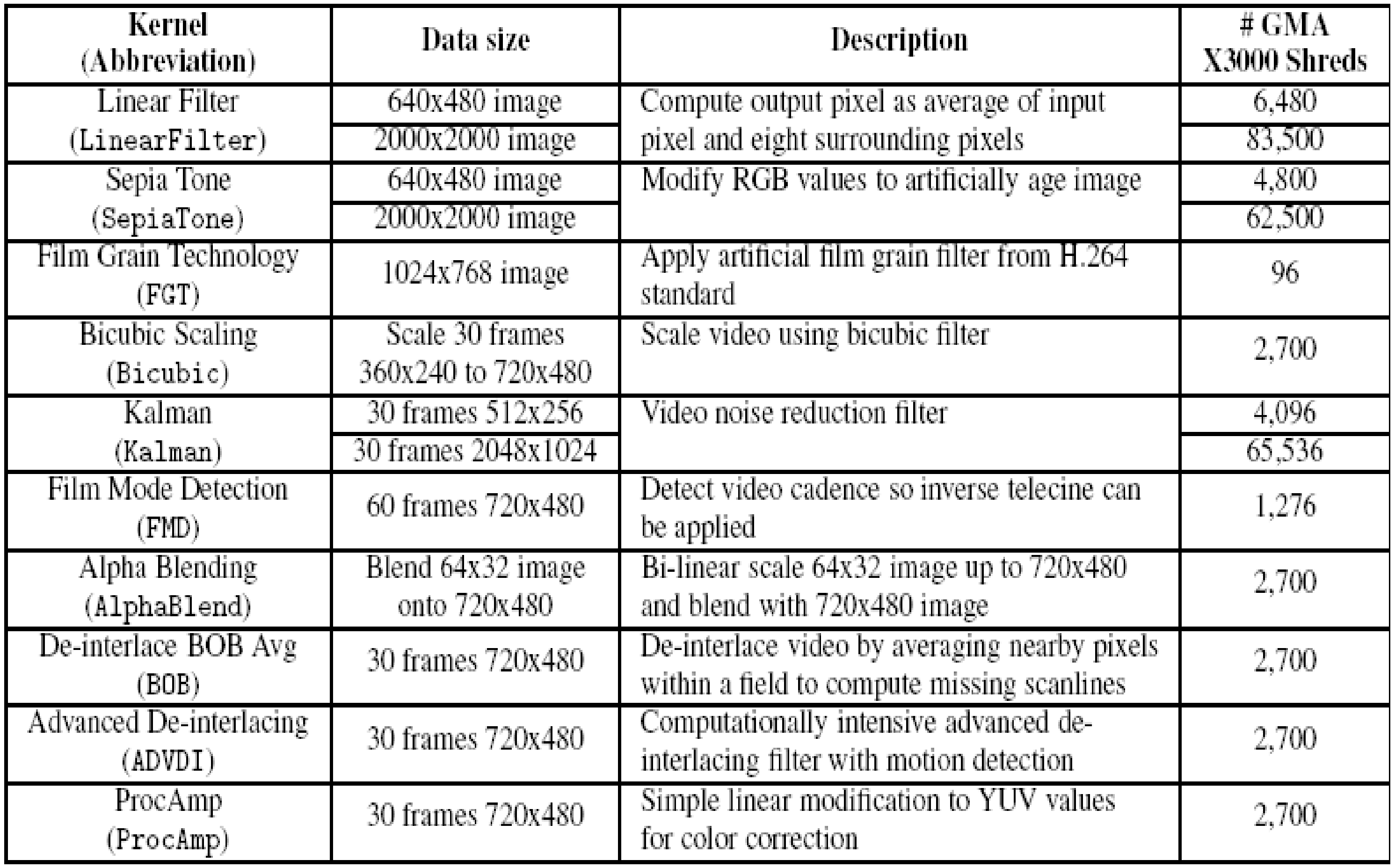
```
A_desc = chi_alloc_desc(X3000, A, CHI_INPUT, n, 1);
B_desc = chi_alloc_desc(X3000, B, CHI_INPUT, n, 1);
C_desc = chi_alloc_desc(X3000, C, CHI_OUTPUT, n, 1);
#pragma omp parallel target(X3000) shared(A, B, C)
  descriptor(A_desc,B_desc,C_desc) private(i) master_nowait
{
  for (i=0; i<n/8; i++)
    __asm
    {
      shl.1.w  vr1 = i, 3
      ld.8.dw  [vr2..vr9]   = (A, vr1, 0)
      ld.8.dw  [vr10..vr17] = (B, vr1, 0)
      add.8.dw [vr18..r25]  = [vr2..vr9], [vr10..vr17]
      st.8.dw  (C, vr1, 0) = [vr18..vr25]
    }
}
#pragma omp parallel for shared(D,E,F) private(i)
{
  for (i=0; i<n; i++)
    F[i] = D[i] + E[i];
}
```

Figure 6. CHI Code Example with GMA X3000 Pseudo-code

Experiment Setup

- EXOCHI prototype was tested on Intel Santa Rosa platform containing Intel Core 2 Duo as the OMS and 32 GMA X3000 as the AMS.
- A selection of benchmarks were configured due to their hold significant data and thread-level parallelism
  - Compiled with –fast and –Qprof_use options for aggressive optimization tuned to the Intel 2 Duo processor
    - Auto-vectorization and profile-guided optimization.
- Key factors for better performance include:
  - Wide SIMD instructions. (Vectors)
  - Predication Support
  - Large register file with 64 to 128 vector register on each AMS.
  - CHI inline assembly to configure code for better utilization of instructions and features for the X3000.

Experiment Setup: Kernels



Kernel (Abbreviation)	Data size	Description	# GMA X3000 Shreds
Linear Filter (LinearFilter)	640x480 image	Compute output pixel as average of input pixel and eight surrounding pixels	6,480
	2000x2000 image		83,500
Sepia Tone (SepiaTone)	640x480 image	Modify RGB values to artificially age image	4,800
	2000x2000 image		62,500
Film Grain Technology (FGT)	1024x768 image	Apply artificial film grain filter from H.264 standard	96
Bicubic Scaling (Bicubic)	Scale 30 frames 360x240 to 720x480	Scale video using bicubic filter	2,700
Kalman (Kalman)	30 frames 512x256	Video noise reduction filter	4,096
	30 frames 2048x1024		65,536
Film Mode Detection (FMD)	60 frames 720x480	Detect video cadence so inverse telecine can be applied	1,276
Alpha Blending (AlphaBlend)	Blend 64x32 image onto 720x480	Bi-linear scale 64x32 image up to 720x480 and blend with 720x480 image	2,700
De-interlace BOB Avg (BOB)	30 frames 720x480	De-interlace video by averaging nearby pixels within a field to compute missing scanlines	2,700
Advanced De-interlacing (ADVDI)	30 frames 720x480	Computationally intensive advanced de-interlacing filter with motion detection	2,700
ProcAmp (ProcAmp)	30 frames 720x480	Simple linear modification to YUV values for color correction	2,700

Table 2. Media-Processing Kernels

Performance Speedup Over OMS

- Chart shows speedup factors for X3000 accelerators over Intel Core 2 Duo for all benchmarks.
- Two factors to the speedup performance:
  - Abundant shred-level parallelism
    - Stalls from context switch between shreds were covered up by numerous concurrent shred execution
  - Maximizing cache hit rate and bandwidth utilization with CHI runtime.
    - Programmers are able to order shreds in accessing adjacent macroblocks to take advantage of spatial and temporal localities.

Performance Speedup Over OMS

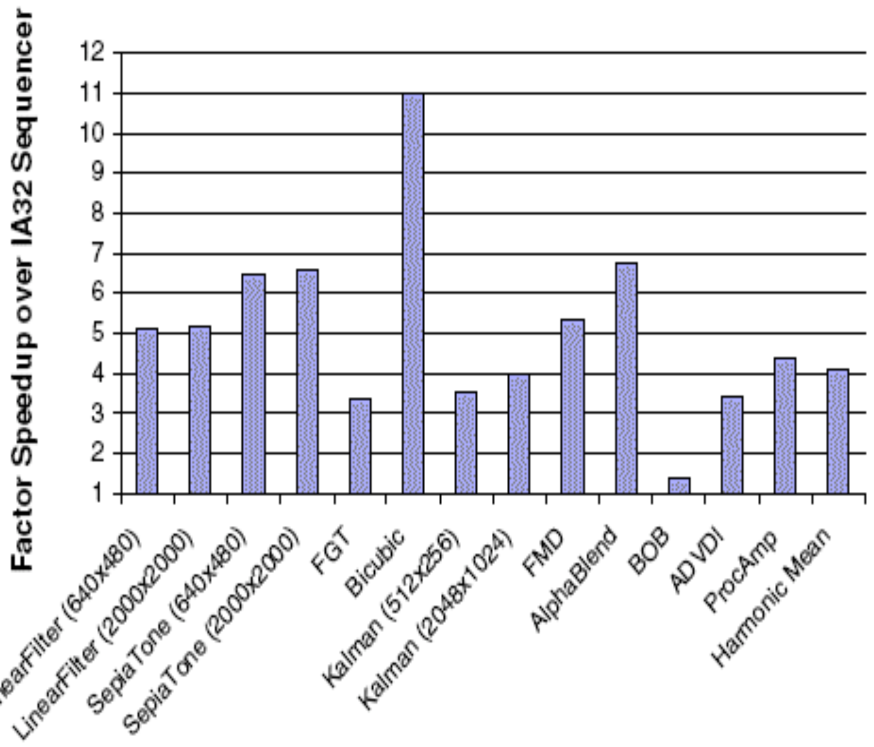
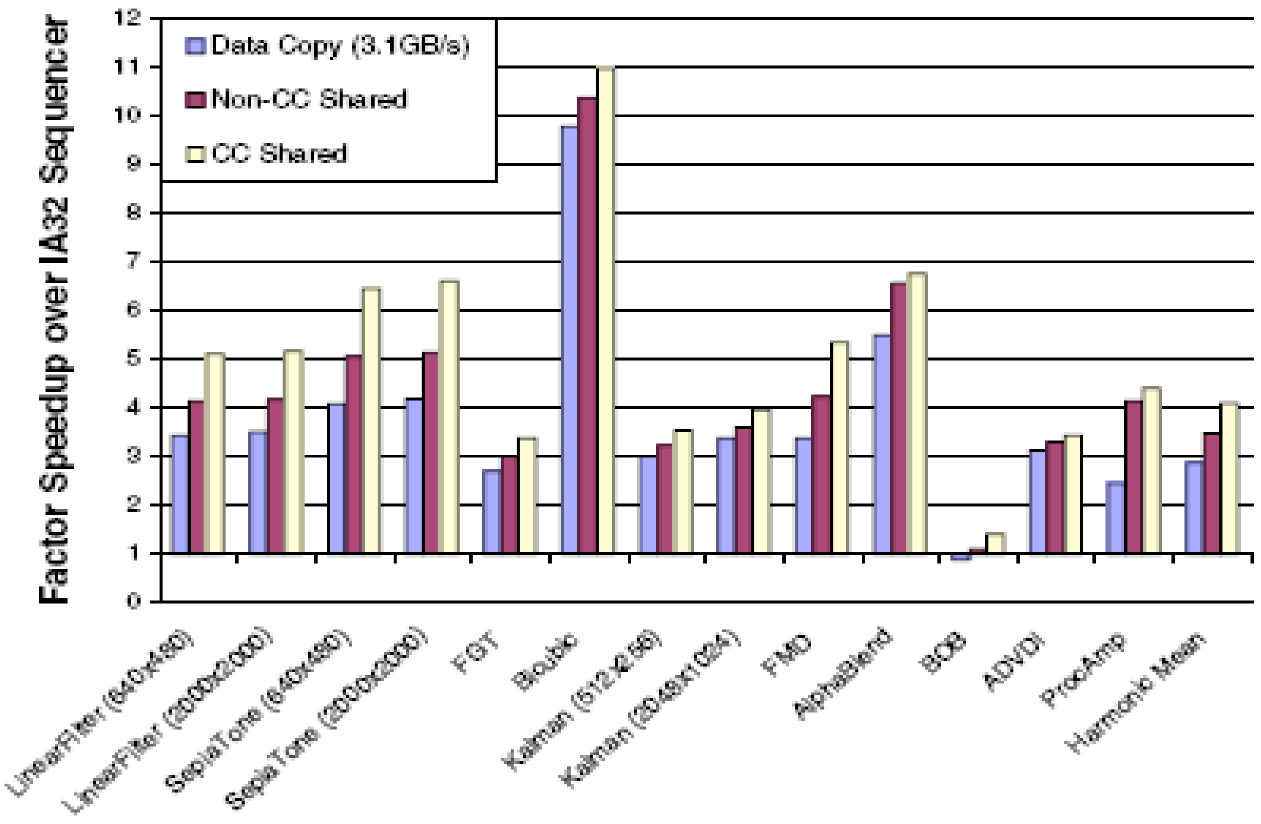


Figure 7. Speedup from Execution on GMA X3000 Exo-sequencers over IA32 Sequencer

Data Copying vs Shared Space

- Charts show speedup factors for X3000 in three configurations for data communication and synchronization.
- Testing how EXOCHI handles overhead.
- Three configurations:
  - Data copying: EXOCHI act similar to a message-passing multi-core machine
    - Susceptible to numerous memory transfer with high overhead.
  - Shared Virtual Address Space: All AMS have access to the same virtual memory space.
    - Must constantly flush dirty cache lines to memory.
  - Shared Space with Cache Coherence: Similar to previous configuration, but does not necessary rely on cache flushing or data copying.

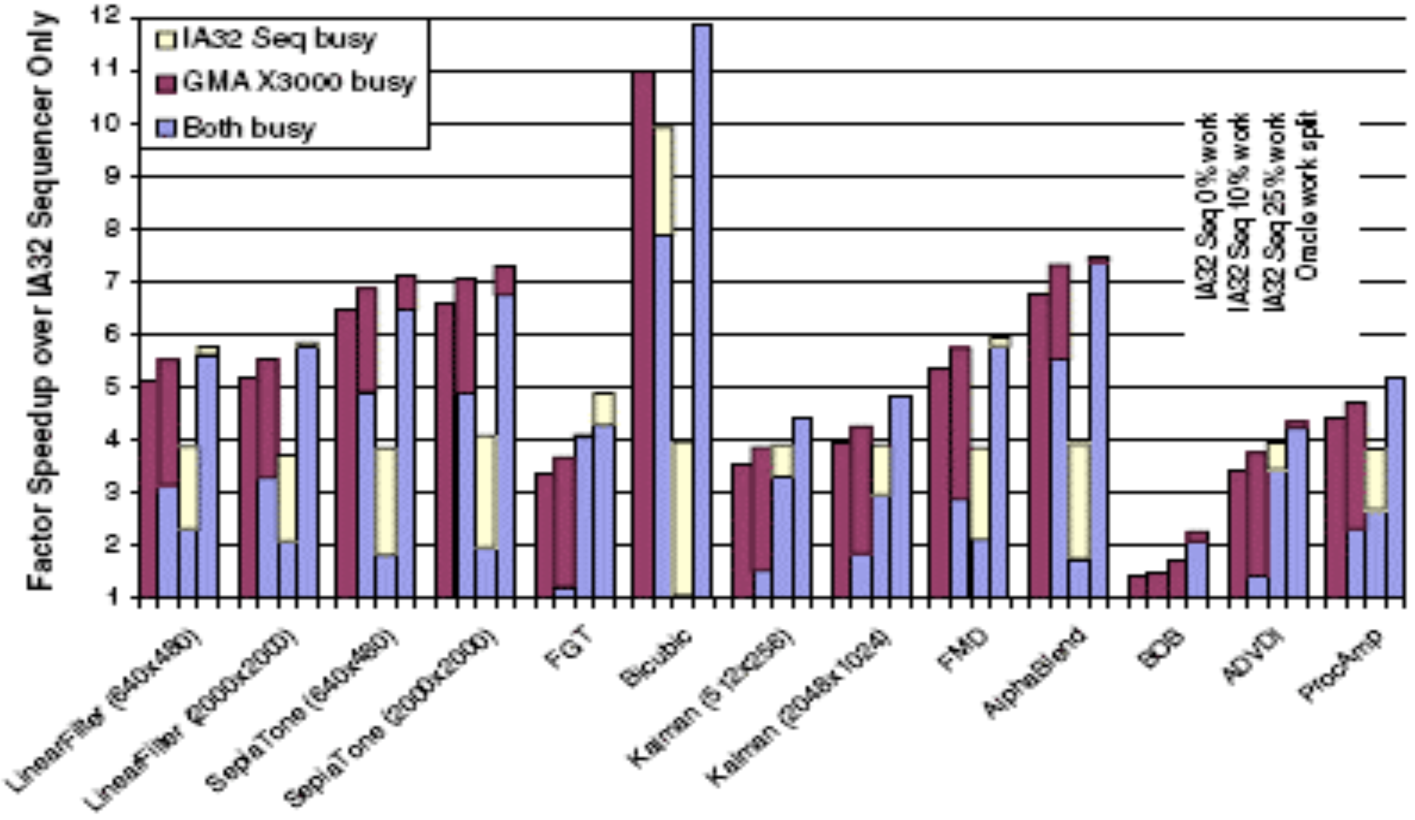
Data Copying vs Shared Space



Data Copying vs Shared Space

- Observations:
  - Significant degradation shown relying on data copying and using shared space with cache coherence.
  - Significant performance is preserved for most benchmark without cache coherency
  - Overhead and stall costs were covered up by the parellelization and interleaved execution between data copying/cache flushing and shred spawning.

Working Together



Working Together

- Chart indicating speedups when both OMS and AMS work together over OMS working by itself.
- Tested on work balances with OMS processing 0%, 10%, 25%, and oracle work split of shreds.
  - Oracle work split divides the shred work number in a way that both OMS and AMS finish at the same time.
- Performance speedup is severely lost mostly due to work imbalance.

Conclusion

- Changing the role of processor management resources to application and runtime
    - Yield increased performance over architectures with OS-based.
- Programs has direct access and can take full advantage for better optimization and performance.
- Most improvement was caused by the CHI runtime environment and OpenMP extension to support heterogeneous cores.
    - Shreds concurrently executed amongst a group of cores with little interference from OS or tightly-coupled CPU.