Structural Object Programming Model: Enabling Efficient Development on Massively Parallel Architectures

Mike Butts, Laurent Bonetto, Brad Budlong, Paul Wasson

HPEC 2008 – September 2008

- HPEC systems run compute-intensive real-time applications such as image processing, video compression, software radio and networking.

- Familiar CPU, DSP, ASIC and FPGA technologies have all reached fundamental scaling limits, failing to track Moore's Law.

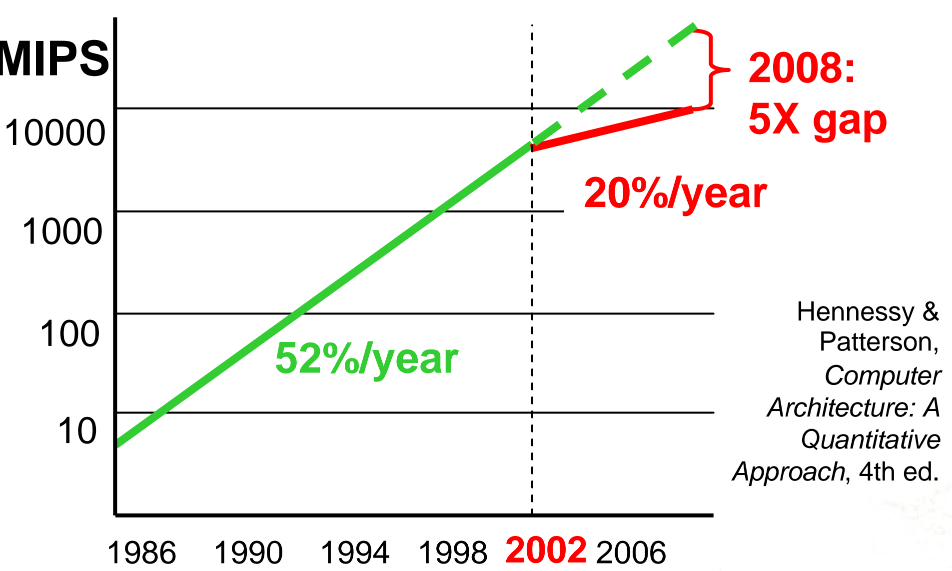
- A number of parallel embedded platforms have appeared to address this:
  - SMP (symmetric multiprocessing) multithreaded architectures, adapted from general-purpose desktop and server architectures.
  - SIMD (single-instruction, multiple data) architectures, adapted from supercomputing and graphics architectures.
  - MPPA (massively parallel processor array) architectures, specifically aimed at high-performance embedded computing.

- Ambric devised a practical, scalable MPPA programming model first, then developed an architecture, chip and tools to realize this model.

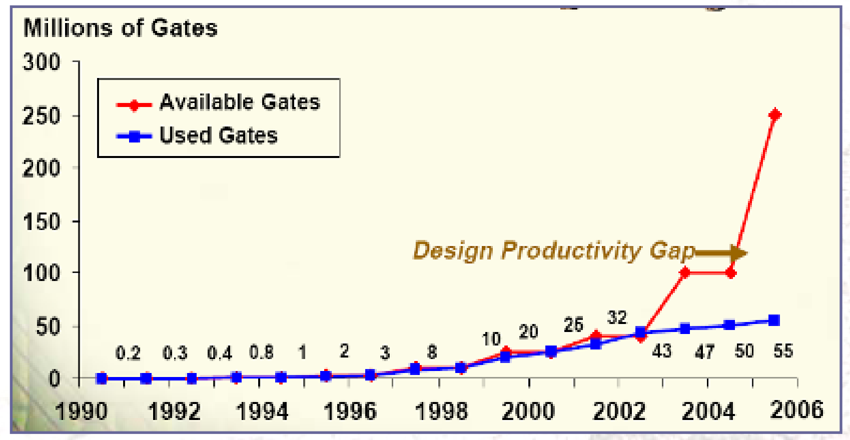
Scaling Limits: CPU/DSP, ASIC/FPGA

- Single CPU & DSP performance has fallen off Moore's Law
  - All the architectural features that turn Moore's Law area into speed have been used up.
  - Now it's just device speed.
- **CPU/DSP does not scale**

- ASIC project now up to $30M
  - NRE, Fab / Design, Validation
- HW Design Productivity Gap
  - Stuck at RTL
  - 21%/yr productivity vs 58%/yr Moore's Law
- ASICs limited now, FPGAs soon
- **ASIC/FPGA does not scale**



Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed.



Gary Smith, *The Crisis of Complexity*, DAC 2003

Parallel Processing is the Only Choice

Parallel Platforms for Embedded Computing

- Program processors in software, far more productive than hardware design
- Massive parallelism is available
  - A basic pipelined 32-bit integer CPU takes less than 50,000 transistors
  - Medium-sized chip has over 100 million transistors available.
- But many parallel chips are difficult to program.
- The trick is to

  1) Find the right programming model first,

  2) Arrange and interconnect the CPUs and memories to suit the model,

  3) To provide an efficient, scalable platform that’s reasonable to program.
- Embedded computing is free to adopt a new platform
  - General-purpose platforms are bound by huge compatibility constraints
  - Embedded systems are specialized and implementation-specific

Choosing a Parallel Platform That Lasts

- How to choose a durable parallel platform for embedded computing?
  – Don't want adopt a new platform only to have to change again soon.

- Effective parallel computing depends on common-sense qualities:
  – **Suitability:** How well-suited is its architecture for the full range of high-performance embedded computing applications?
  – **Efficiency:** How much of the processors' potential performance can be achieved? How energy efficient and cost efficient is the resulting solution?
  – **Development Effort:** How much work to achieve a reliable result?

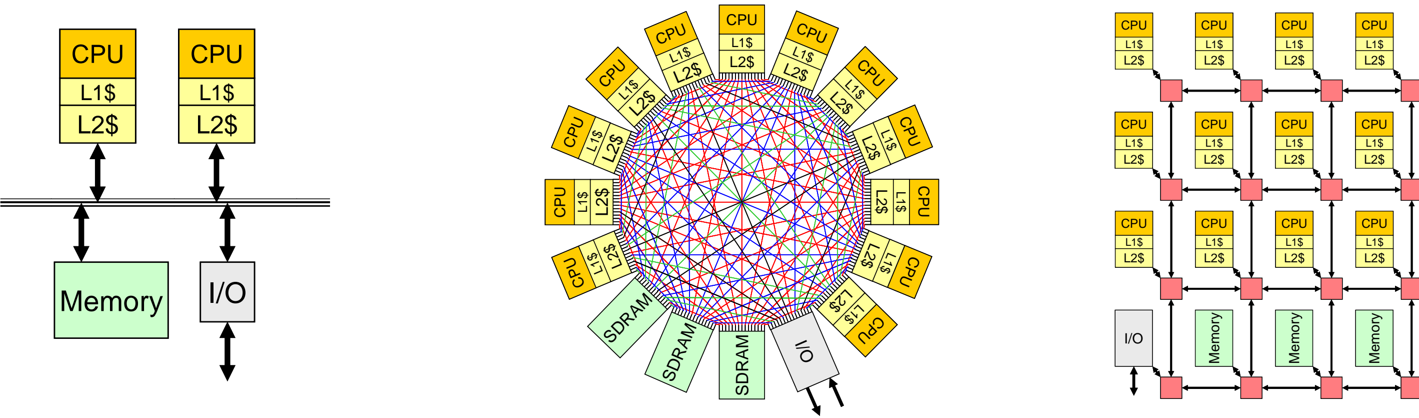
- Inter-processor communication and synchronization are key:
  – **Communication:** How easily can processors pass data and control from stage to stage, correctly and without interfering with each other?
  – **Synchronization:** How do processors coordinate with one another, to maintain the correct workflow?
  – **Scalability:** Will the hardware architecture and development effort scale up to a massively parallel system of hundreds or thousands of processors?

Symmetric Multiprocessing (SMP)

- Multiple processors share similar access to a common memory space
- Incremental path from the old serial programming model
  - Each processor sees the same memory space it saw before.
  - Existing applications run unmodified (unaccelerated as well of course)
  - Old applications with millions of lines of code can run without modification.
- SMP programming model has task-level and thread-level parallelism.
  - Task-level is like multi-tasking operating system behavior on serial platforms.
- To use more parallelism the tasks must become parallel: *Multithreading*
  - Programmer writes source code which forks off separate threads of execution
  - Programmer explicitly manages data sharing, synchronization
- Commercial SMP Platforms:
  - Multicore GP processors: Intel, AMD (not for embedded systems)
  - Multicore DSPs: TI, Freescale, ...
  - Multicore Systems-on-Chip: using cores from ARM, MIPS, ...

SMP Interconnects, Cache Coherency

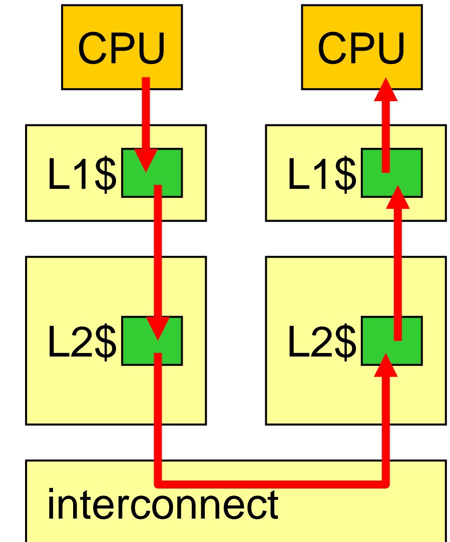
- Each SMP processor has its own single or multi-level cache.
- Needs a scalable interconnect to reach other caches, memory, I/O.



Bus, Ring: Saturates   Crossbar: N-squared   Network-on-chip: Complex

- SMP processors have separate caches which must be kept coherent
  - Bus snooping, network-wide directories
- As the number of processors goes up, total cache traffic goes up linearly, but the possible cache conflict combinations go up as the square.
  - Maintaining cache coherence becomes more expensive and more complex faster than the number of processors.

SMP Communication

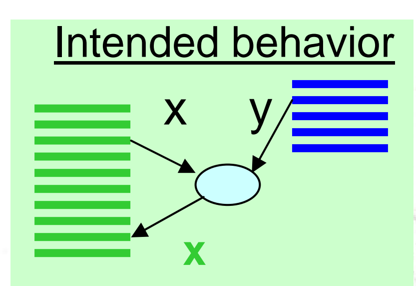


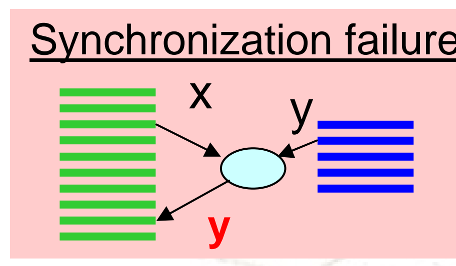
- In SMP *communication is a second-class function.*
  - Just a side-effect of shared memory.
- Data is copied five times through four memories and an interconnect.
  - The destination CPU must wait through a two-level cache miss to satisfy its read request.
- Poor cache reuse if the data only gets used once.
  - Pushes out other data, causing other cache misses.
- Communication thru shared memory is expensive in power compared with communicating directly.
- The way SMPs do inter-processor communication through shared memory is complex and expensive.

- SMP's multithreaded programming model is deeply flawed:
  Multithreaded programs behave unpredictably.

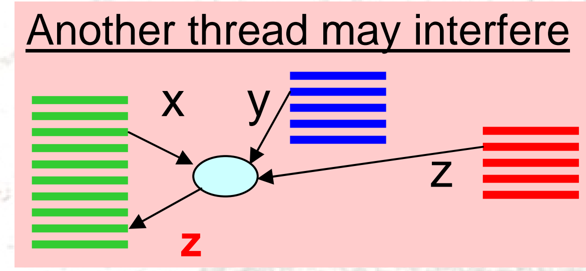
- Single-threaded (serial) program always goes through the same sequence of intermediate states, i.e. the values of its data structures, every time.
  - Testing a serial program for reliable behavior is reasonably practical.

- Multiple threads communicate with one another through shared variables:
  - Synchronization: partly one thread, partly the other
- Result depends on behavior of all threads.
  - Depends on dynamic behavior:
    indeterminate results.
    Untestable.





"If we expect concurrent programming to become mainstream, and if we demand reliability and predictability from programs, we must discard threads as a programming model." -- Prof. Edward Lee



SMP for HPECs Summary

- Easy use of general-purpose 2-to-4-way SMPs is misleading.
  - Big difference between small multicore SMP implementations, and massively parallel SMP's expensive interconnect, cache coherency
- SMPs are non-deterministic, and get worse as they get larger.
  - Debugging massively parallel multithreaded applications promises to be difficult.

Suitability: Limited. Intended for multicore general-purpose computing.

Efficiency: Fair, depending on caching, communication and synchronization.

Development effort: Poor: DIY synchronization, multithreaded debugging.

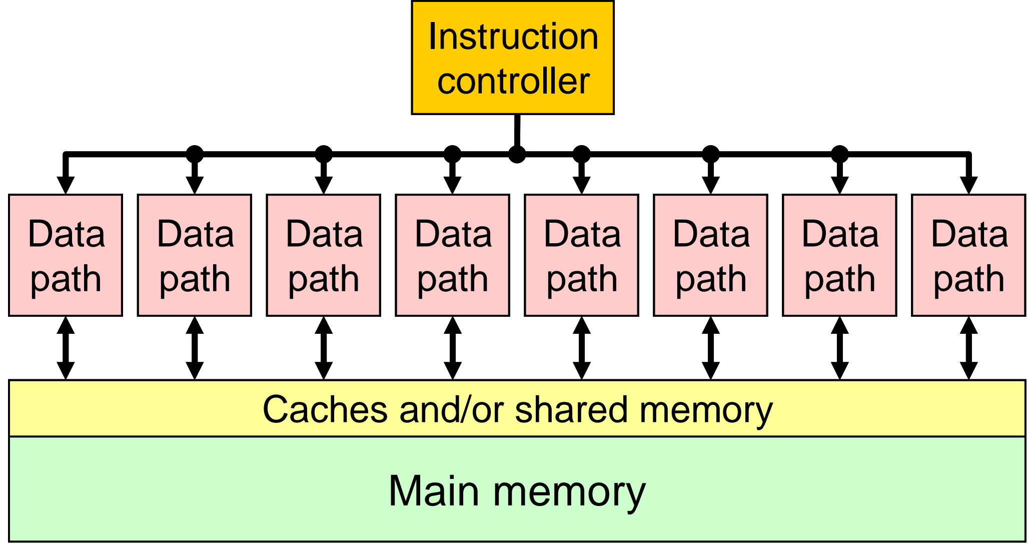
Communication: Poor: Complex, slow and wasteful.

Synchronization: Poor: DIY thread synchronization is difficult, dangerous.

Scalability: Poor: Interconnect architecture, communication through caches, and multithreaded synchronization problems indicate poor hardware and/or software scalability beyond the 2-to-8-way multicore level.

- **Massively parallel SMP platforms are unlikely to be well-suited to the development, reliability, cost, power needs of embedded systems.**

Single Instruction Multiple Data (SIMD)



- Tens to hundreds of datapaths, all run by one instruction stream
  - Often a general-purpose host CPU executes the main application, with data transfer and calls to the SIMD processor for the compute-intensive kernels.
- SIMD has dominated high-performance computing (HPC) since the Cray-1.
  - Massively data-parallel, feed-forward and floating-point-intensive
  - Fluid dynamics, molecular dynamics, structural analysis, medical image proc.
- Main commercial SIMD platforms are SMP / SIMD hybrids
  - NVIDIA CUDA (not for embedded), IBM/Sony Cell

SIMD can work well in supercomputing

- Massive feed-forward data parallelism is expected, so datapaths are deeply pipelined and run at a very high clock rate.
- Large register files are provided in the datapaths to hold large regular data structures such as vectors.
- SIMD performance is depends on hiding random-access memory latency, which may be hundreds of cycles, by accessing data in big chunks at very high memory bandwidth.

- Data-parallel feed-forward applications are common in HPC
  - Long regular loops
  - Little other branching
  - Predictable access to large, regular data structures

- In embedded systems, some signal and image processing applications may have enough data parallelism and regularity to be a good match to SIMD architectures.

SIMD can work poorly in HPECs

- SIMD's long pipelines can be very inefficient:
  - When there are feedback loops in the dataflow (x[i] depends on x[i-n])
  - When data items are only a few words, or irregularly structured
  - When testing and branching (other than loops)
- SIMD is not well suited to high-performance embedded applications
  - Often function-parallel, with feedback paths and data-dependent behavior
  - Increasingly found in video codecs, software radio, networking and elsewhere.
- Example: real-time H.264 broadcast-quality HD video encoding
  - Massive parallelism is required
  - Feedback loops in the core algorithms
  - Many different subsystem algorithms, parameters, coefficients, etc. are used dynamically, in parallel (*function-parallel*), according to the video being encoded.
- Commercial devices (CUDA, Cell) have high power consumption

SIMD for HPECs Summary

- SIMD architectures were developed for the massively data-parallel feed-forward applications found in scientific computing and graphics.

Suitability: Limited. Intended for scientific computing (HPC).

Efficiency: Good to Poor: Good for data-parallel feed-forward computation. Otherwise it gets Poor quickly.

Development effort: Good to Poor:
Good for suitable applications, since there is a single instruction stream.
Gets poor when forcing complexity, data-dependency into SIMD model.

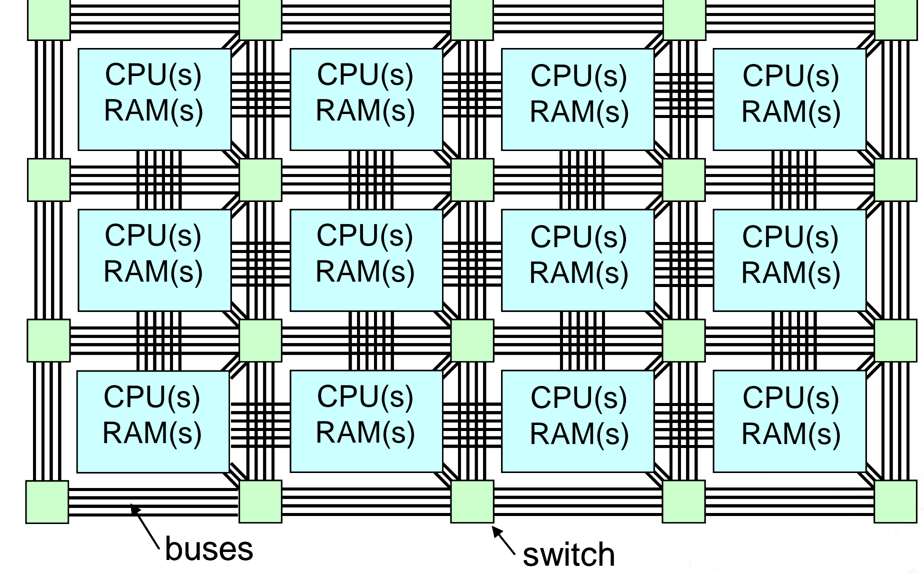
Communication and Synchronization: Good by definition, everything's always on the same step.

Scalability: Poor without lots of data parallelism available in the application. A few embedded applications have vector lengths in the hundreds to thousands, most don't.

- **Massively parallel SIMD platforms are unlikely to be well-suited to the most high-performance embedded system applications.**

Massively Parallel Processor Array (MPPA)

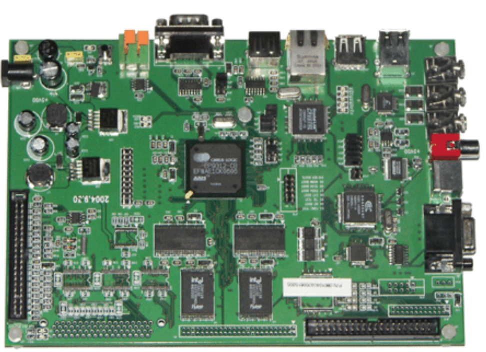
- Massively parallel array of CPUs and memories
- 2D-mesh configurable interconnect of word-wide buses
- MIMD architecture
  - Distributed memory
  - Strict encapsulation
  - Point-to-point communication

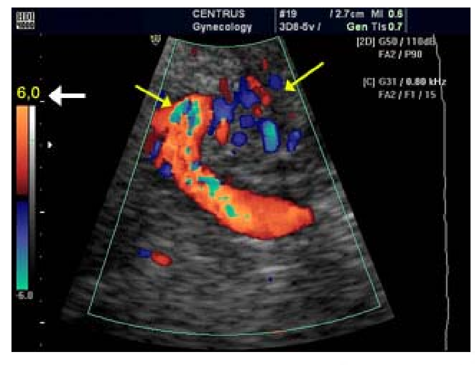


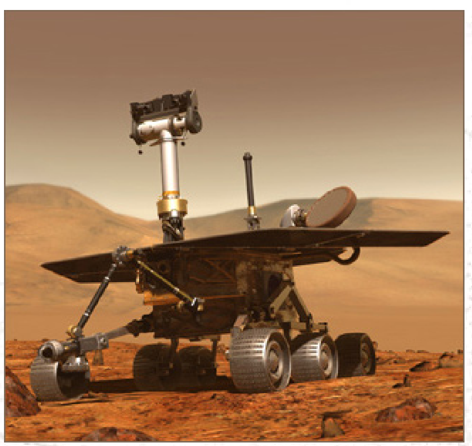
- Complex applications are decomposed into a hierarchy of subsystems and their component function objects, which run in parallel, each on their own processor.
- Likewise, large on-chip data objects are broken up and distributed into local memories with parallel access.
- Objects communicate over a parallel structure of dedicated channels.
- Programming model, communications and synchronization are all simple, which is good for development, debugging and reliability.

Massively Parallel Processor Array (MPPA)

- Developed specifically for high-performance embedded systems.
  - Video codecs, software-defined radio, radar, ultrasound, machine vision, image recognition, network processing...............
  - Continuous GB/s data in real time, often hard real-time.
  - Performance needed is growing exponentially.
- Function-parallel, data-parallel, feed-forward/back, data-dependent
- TeraOPS, low cost, power efficiency, and deterministic reliable behavior.

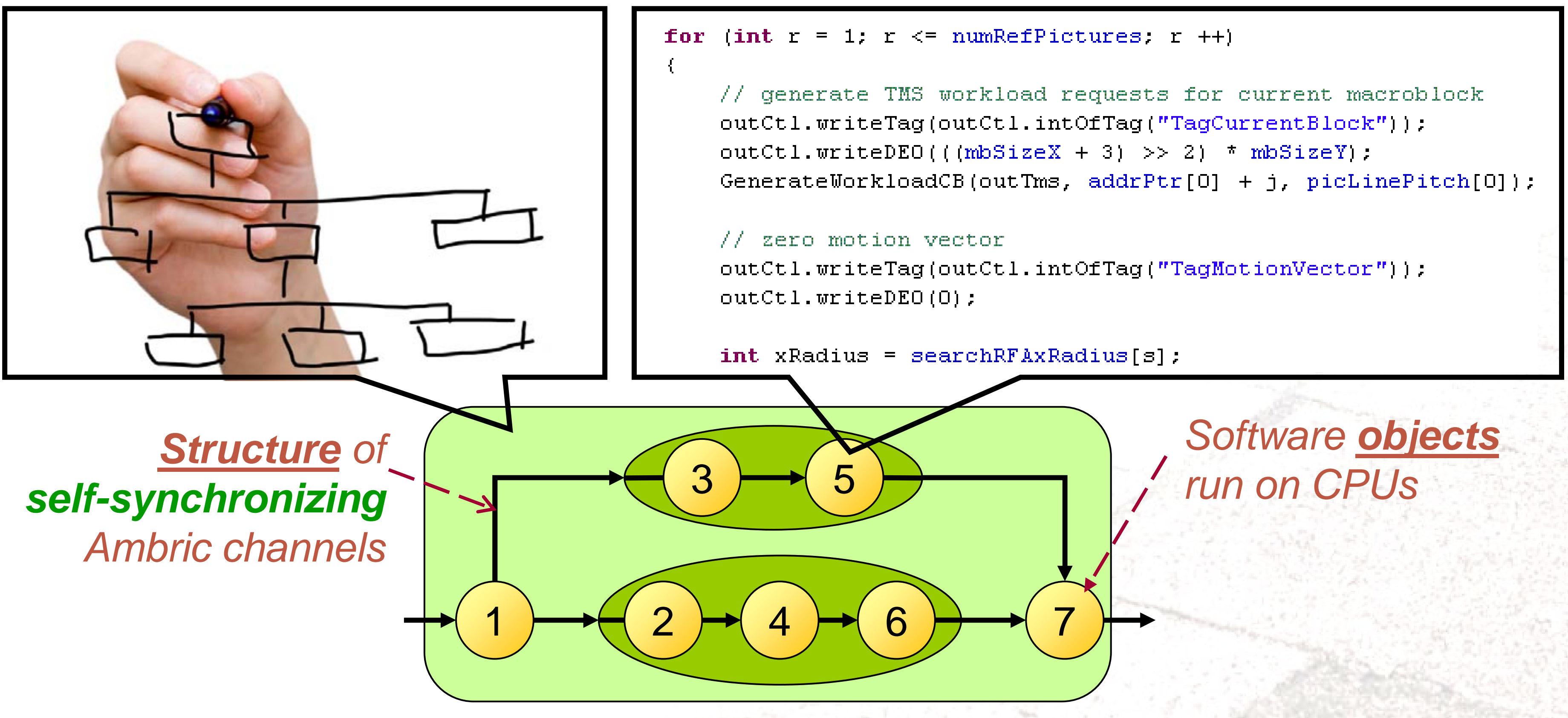






- Ambric MPPA platform objectives:
  1) Optimize performance, performance per watt
  2) Reasonable and reliable application development
  3) Moore's Law-scalable hardware architecture and development effort

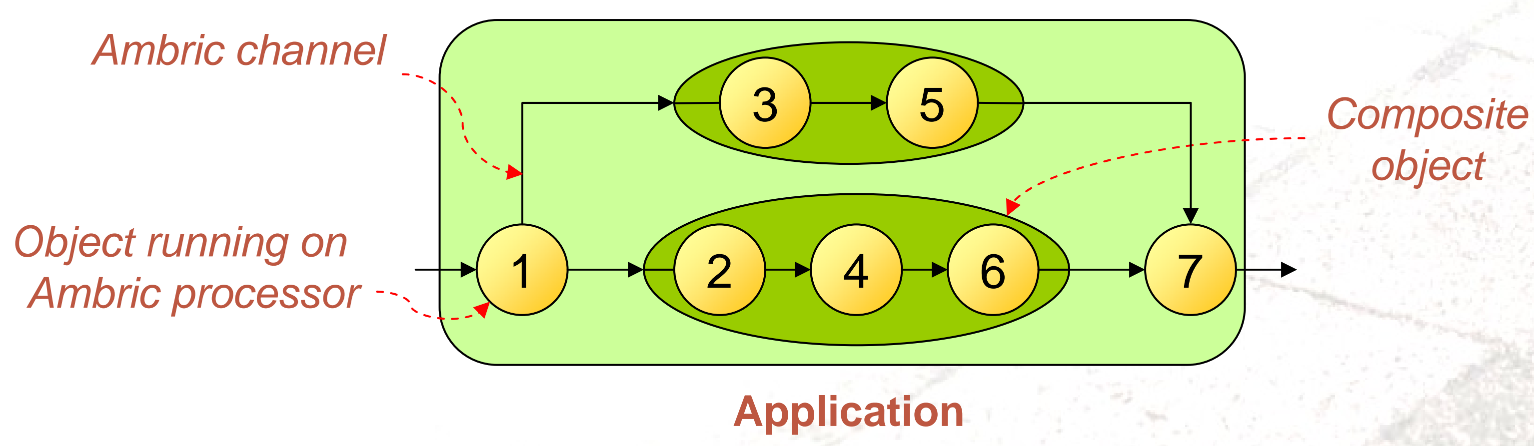
Ambric said: Choose the Right Programming Model *First*



Ambric's Structural Object Programming Model

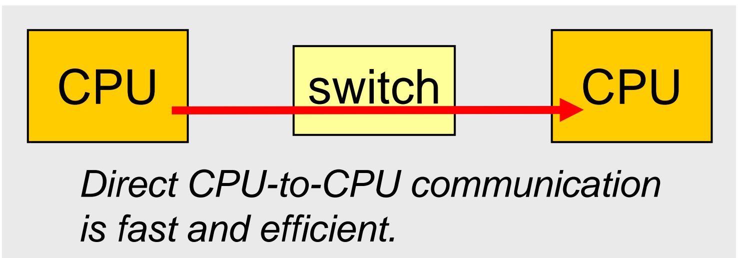
Structural Object Programming Model

- Objects are software programs running concurrently on an array of Ambric processors and memories
- Objects exchange data and control through a structure of self-synchronizing Ambric channels
- Mixed and match objects hierarchically to create new objects, snapped together through a simple common interface
- Easier development, high performance and scalability



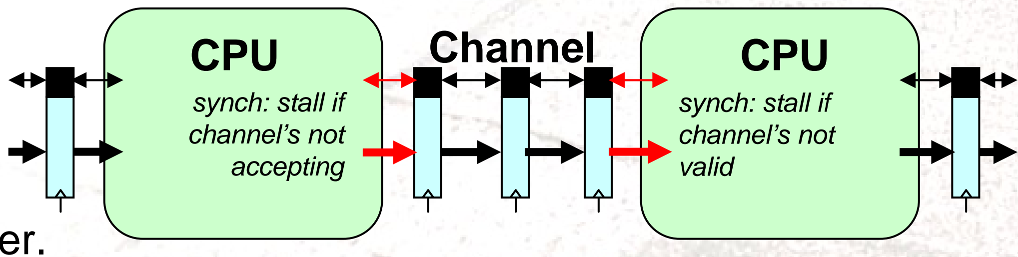
Communication and Synchronization

- Ambric configurable circuit-switched interconnect joins CPUs, memories
  - Dedicated hardware for each channel
  - Word-wide, not bit-wide
  - Registered at each stage
  - Scales well to very large sizes
  - Place & route take seconds, not hours (1000s of elements, not 100,000s)



Direct CPU-to-CPU communication is fast and efficient.

- Ambric channels provide explicit synchronization as well as communication
  - CPU only sends data when channel is ready, else it just stalls.
  - CPU only receives when channel has data, else it just stalls.
  - Sending a word from one CPU to another is also an event.
  - Keeps them directly in step with each other.
  - Built into the programming model, not an option, not a problem for the developer.

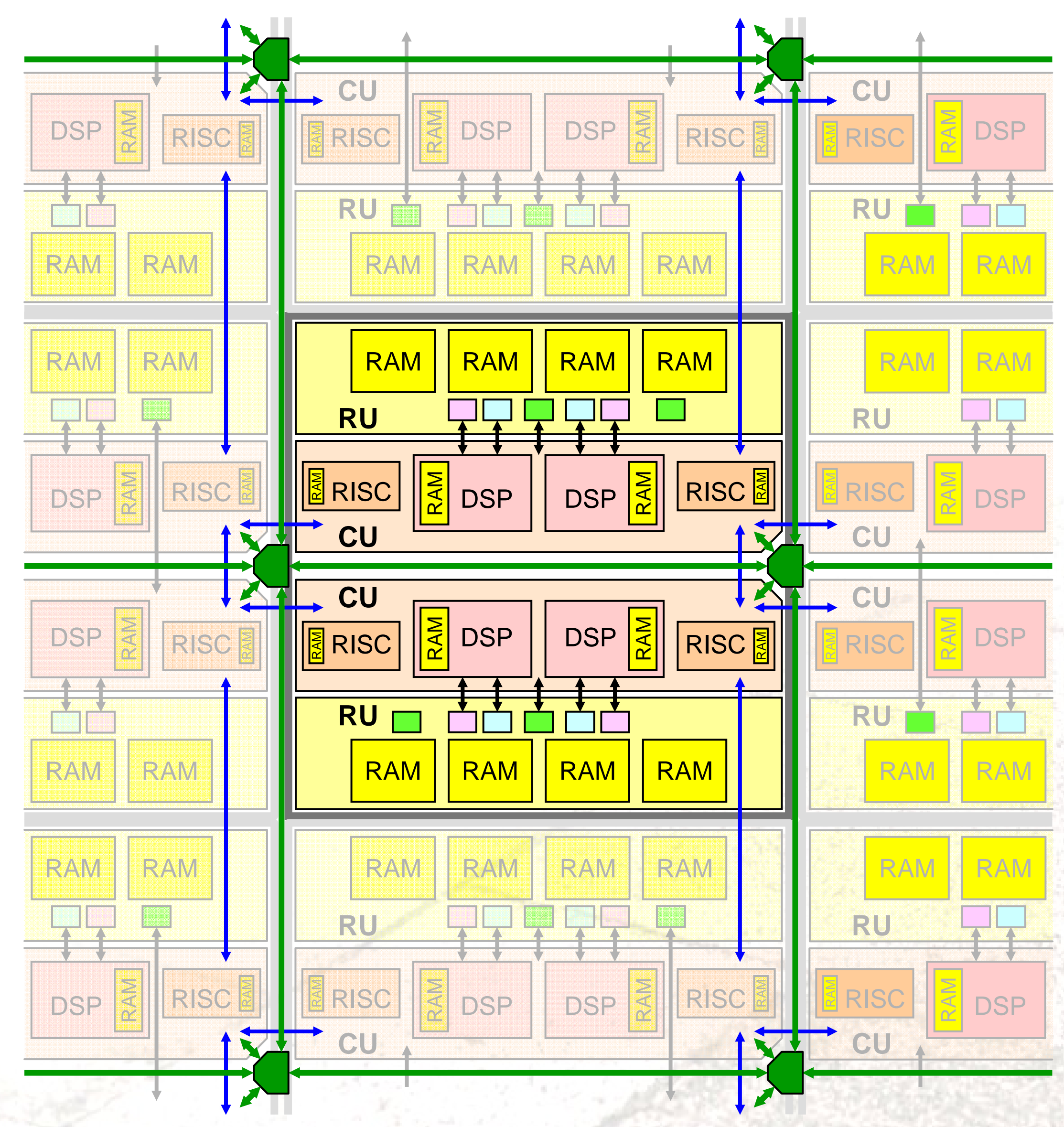


MPPA Determinism

- Recall the multithreaded SMP's difficulties and danger due to communicating and synchronizing through shared state.
- An MPPA has no explicitly shared state.
  - Every piece of data is encapsulated in one memory
  - It can only be changed by the processor it's connected to.
  - A wayward pointer in one part of the code cannot trash the state of another, since it has no physical access to any state but its own.
- MPPA applications are deterministic
  - Two processors communicate and synchronize only through a channel dedicated to them, physically inaccessible to anyone else.
  - No opportunity for outside interference
- MPPA applications have deterministic timing as well
  - No thread or task swapping, no caching or virtual memory, no packet switching over shared networks

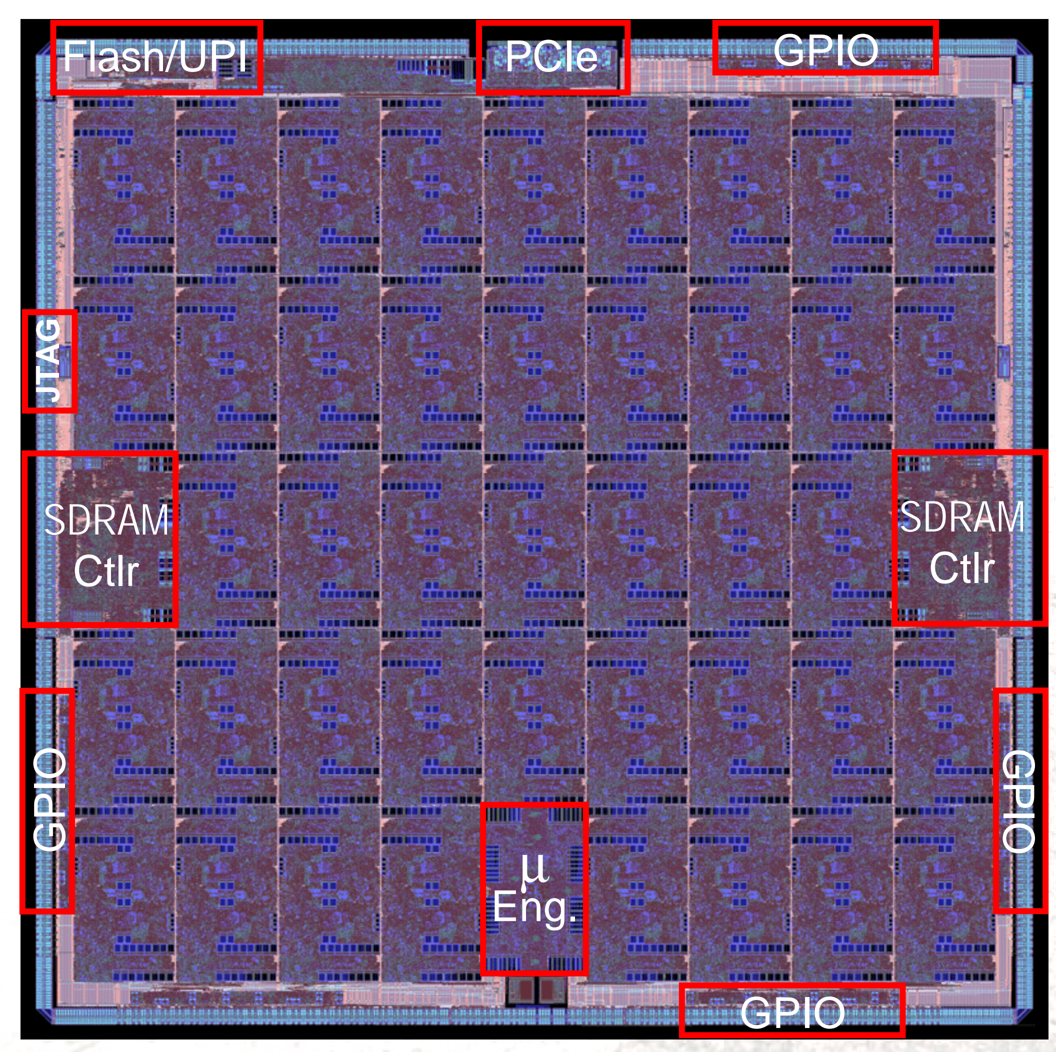
Ambric MPPA Hardware Architecture

- Brics connect by abutment to form a core array
- Each bric has
  - 4 streaming 32-bit DSPs
    - 64-bit accumulator
    - dual 16-bit ops
  - 4 streaming 32-bit RISCs
  - 22KB RAM
    - 8 banks with engines
    - 8 local memories
- Configurable interconnect of Ambric channels
  - Local channels
  - Bric-hopping channels



Ambric Am2045 Device

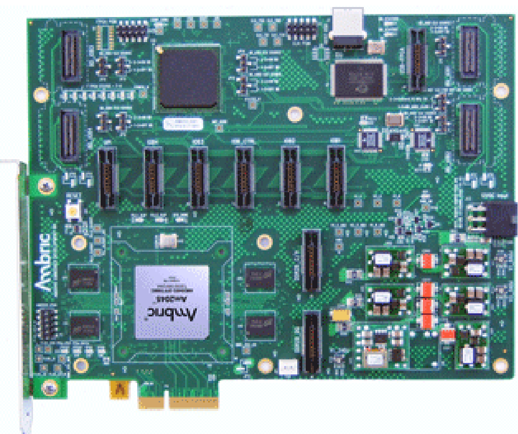
- 130nm standard-cell ASIC
  - 180 million transistors
- 45 brics
  - 336 32-bit processors
  - 7.1 Mbits dist. SRAM
  - 8 μ–engine VLIW accelerators
- High-bandwidth I/O
  - PCI Express
  - DDR2-400 x 2
  - 128 bits GPIO
  - Serial flash
- In production since 1Q08



Ambric Performance, Applications

- Am2045: array of 336 32-bit CPUs, 336 memories, 300 MHz
  - 1.03 teraOPS (video SADs), 126,000 32-bit MIPS, 50.4 GMACS
  - Total power dissipation is 6-12 watts, depending on usage.

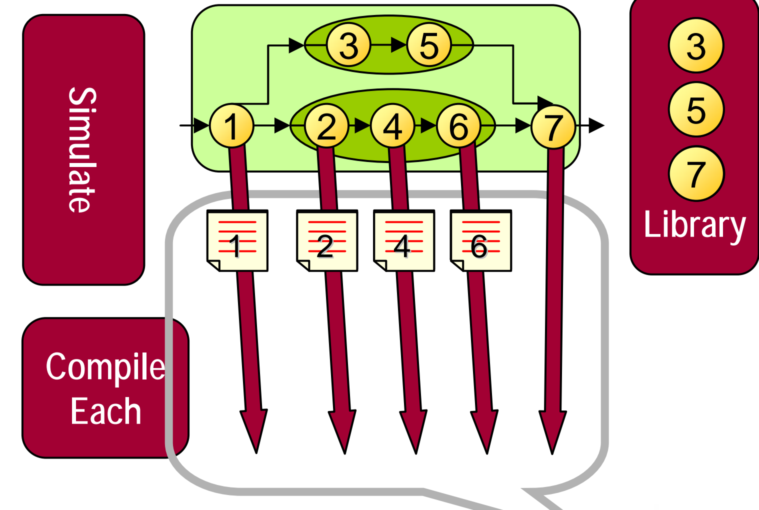
- Video Processing Reference Platform
  - PCIe plug-in card
  - Integrated with desktop video tools
  - Accelerates MPEG-2 and H.264/AVC broadcast-quality HD video encoding by up to 8X over multicore SMP PC.

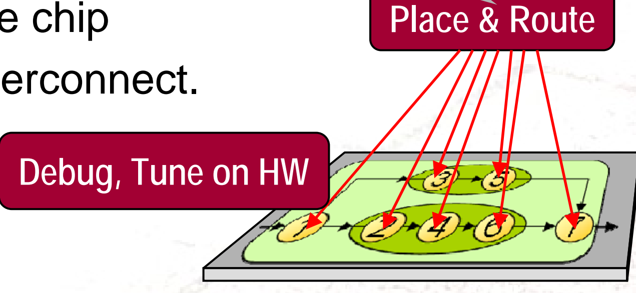


- Embedded Development Platform
  - Four 32-bit GPIO ports
  - USB or PCIe host I/F
  - End user applications in video processing, medical imaging, network processing.

Ambric Tools

- aDesigner Eclipse-based IDE
  - Processor objects are written in standard Java subset or assembly
  - Objects are taken from libraries
  - Structure is defined in aStruct, a coordination language*
  - Simulate in aDesigner for testing, debugging and analysis
  - Objects are compiled and auto-placed onto the chip
  - Structure is routed onto chip's configurable interconnect.





- On-chip parallel source-level in-circuit debugging and analysis
  - Any object can be stopped, debugged and resumed in a running system without disrupting its correct operation. *Self-synchronizing channels!*

**Coordination languages allow components to communicate to accomplish a shared goal, a deterministic alternative to multithreaded SMP.*

MPPA for HPECs Summary

- MPPA hardware is dense, fast, efficient and scalable.
- Efficient, reliable applications, reasonable development effort, scalability.

Suitability: Good. Developed specifically for embedded systems.

Efficiency: High. Data or functional parallel, feed-forward or feedback, regular or irregular control.

Development effort: Good. Modularity and encapsulation help development and code reuse. Whole classes of bugs, such as bad pointers, synchronization failures are impossible. Testing is practical and reliable.

Communication: Good. Direct comm'n is fast, reliable, deterministic, efficient.

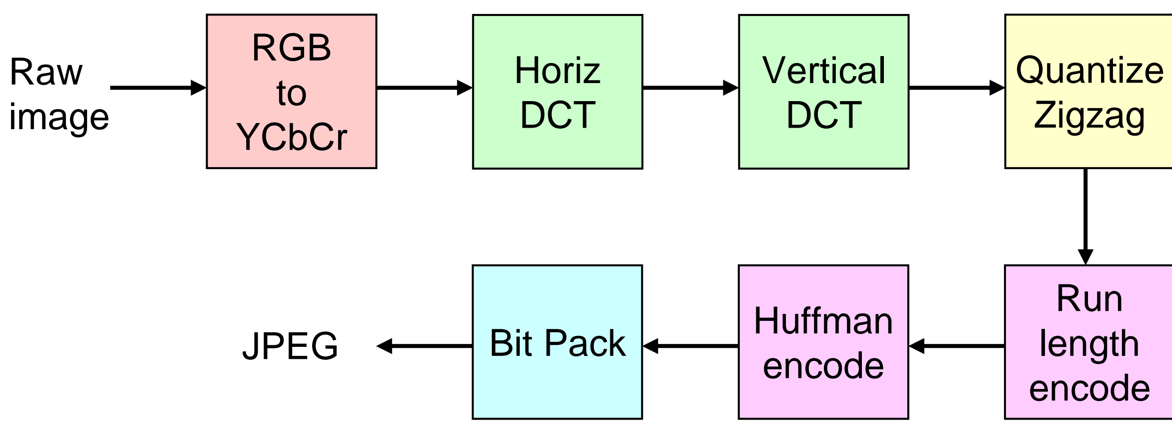
Synchronization: Good. Built in. Fully deterministic.

Scalability: Good. Hardware is free from many-to-many interconnect and caching, and can easily use GALS clocking for very large arrays. Software development is free from SMP's multithreading and SIMD's vector-length scaling limits, and leverage from code reuse is effective.

MPPA Development Example

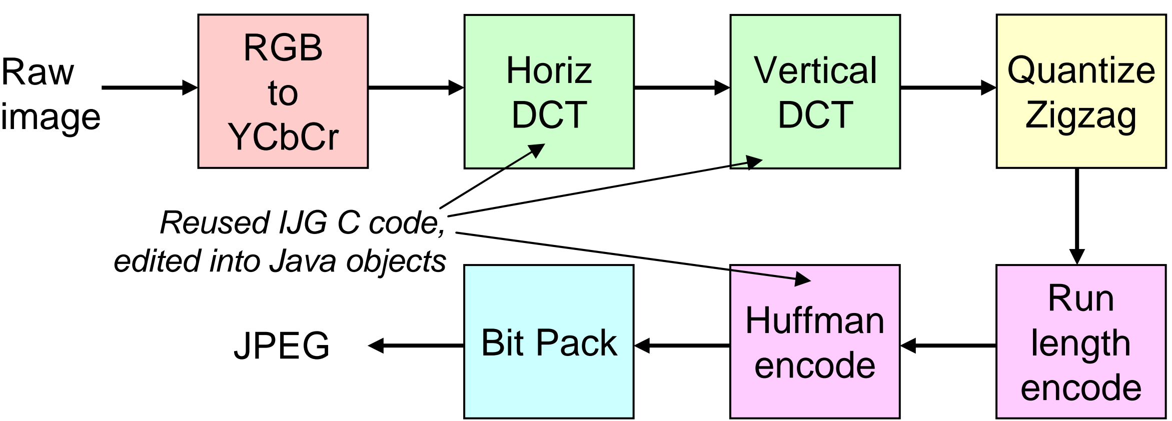
- JPEG is at the root of nearly all image and video compression algorithms.
  - A JPEG encoder is a realistic example of a complete HPEC application,
  - While remaining simple enough to be a good example.
- A video-rate JPEG encoder was implemented on the Ambric Am2045.
- A three phase methodology was used:
  1. Functional implementation
     - Start with a simple HLL implementation, debug it in simulation
  2. Optimization
     - Refine the objects into final implementation, using cycle budgets
  3. Validation and tuning
     - Target the real chip, check with real data, observe and tune performance
- This is the same implementation process used for more complex video codecs (H.264, MPEG2, etc.), and other applications in general.

Example: Video-rate JPEG



- All video compression standards share many functional blocks with JPEG:
- Most video codecs operate on small blocks of pixels (8x8 or similar sizes)
- On which they perform similar operations such as:
  - color space mapping,
  - transformation into the frequency domain (DCT or similar algorithms),
  - quantization,
  - run-length and Huffman encoding, etc.

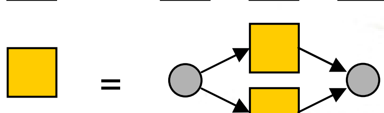
Phase I: Functional implementation



- Goal: Create a functionally correct design as quickly as possible, as a starting point for the fully optimized implementation.
- Do a natural decomposition of the functions into a small number of objects
  - Naturally parallel, intuitive to the developer
- Write objects in high-level language (Java) for simulation
  - No physical constraints apply
  - Based on the IJG JPEG library source code (Java here is very similar to C)
- Simulate with test vectors (aSim)
  - Developed both JPEG encoder and decoder, so each could test the other

Phase II: Optimization Methodology

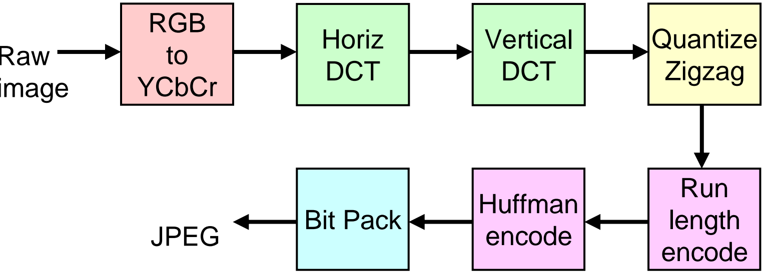
- Goal: Improve speed to meet application requirements.
- Design a cycle budget for each part of the workflow.
  - Like DSP designs, the developer writes software, which can be optimized.
  - Like FPGA or ASIC designs, the developer can trade area for speed.
- Many ways to speed up an object:
  - Functional parallelism: Split the algorithm across a pipeline:
  - Data parallelism: Multiple copies on separate data:
  - Optimize code: use assembly code, dual 16-bit ops



- Optimizing with assembly code is simpler than DSP, not needed as often.
  - Simpler code, simpler processors (no VLIW)
  - With many processors available, only optimize the bottlenecks
- This phase may be done in simulation with a testbench and/or on real hardware with live data.

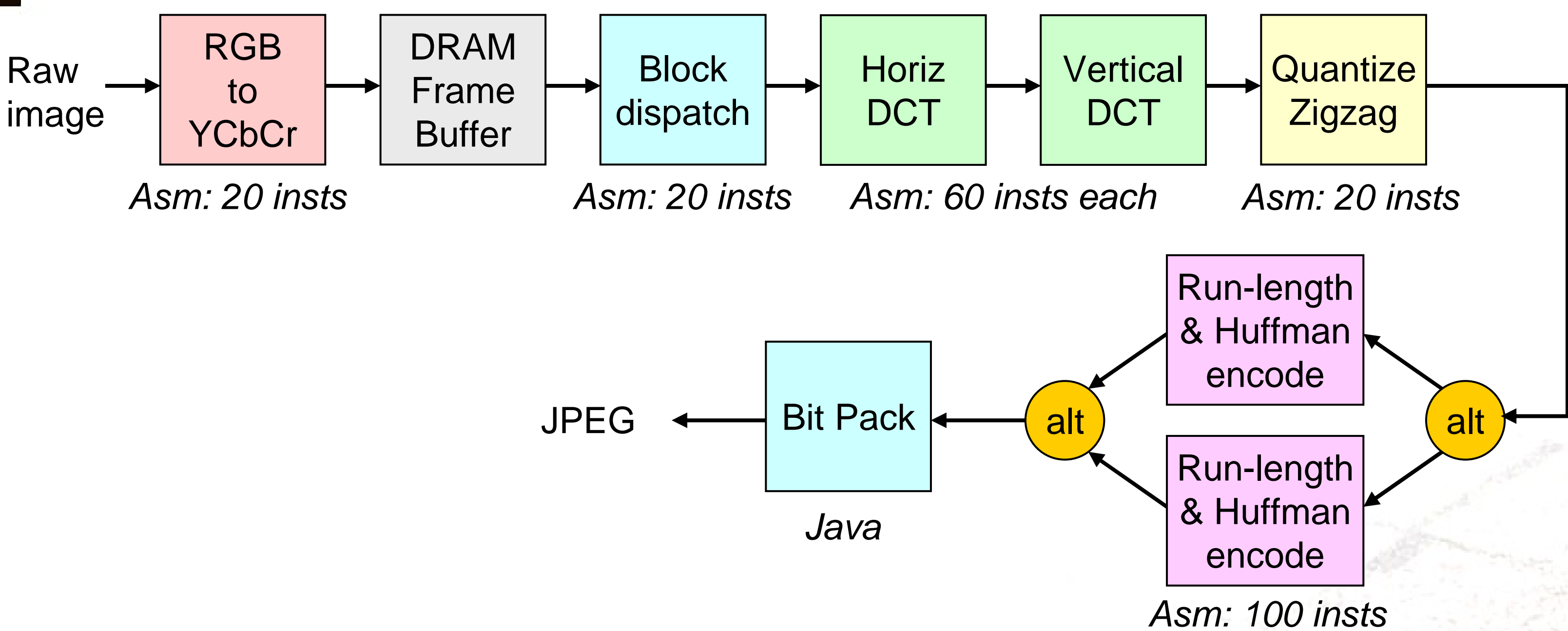
Phase II: Optimization Process

- Goal: 640x480 at 60 fps,
  running on Am2045 at 300 MHz



- Cycle budgets:
  - 5.4 cycles per input byte (pixel color)
  - 9 cycles per Huffman code
  - 90 cycles per 32-bit output word
- Optimize code in aDesigner simulation, one object at a time
  - Use its ISS and profiling tools to see how many cycles each object takes
  - Color conversion, DCT, quantize, zigzag: use assembly, with dual 16-bit ops
  - Run-length and Huffman encoding: use assembly, keep one sample at a time
  - Packing: Java is fast enough
- Parallelize any objects that still miss the cycle budget
  - Run-length and Huffman encoding: 2-way data parallel objects

Phase II: Optimization Result



- 300 lines of assembly code was written
  - Normal programming, nothing too 'creative'
- Fits in 2 brics out of 45 brics in Am2045
  - Completely encapsulated: Other applications on the same chip have no effect

Phase III: On-chip Validation & Tuning

- Thoroughly validate the application on hardware in real time:
  - aDesigner automatically compiles, assembles, places and routes the application onto the processors, memories and interconnect channels.
  - Am2045's dedicated debug and visibility facilities are used through aDesigner's runtime debugging and performance analysis tools.
- Resulting JPEG encoder
  - Uses < 5% of the Am2045 device capacity.
  - Runs at 72 frames per second throughput (vs. 60 fps target).
- The JPEG implementation on the Ambric Am2045 MPPA architecture illustrates an HPEC platform and development methodology that can easily be scaled to achieve levels of performance higher than any high-end DSP and comparable to those achieved by many FPGAs and even ASICs.

What Ambric Developers are Saying…

"Solving **real time high definition video processing** and digital cinema coding functions poses some unique programming challenges. **Having an integrated tool suite** that can simulate and execute the design in hardware **eases development of new products and features for high resolution and high frame-rate imaging** …"

Ari Presler, CEO of Silicon Imagin

"…designers are getting our implementation done in **half the time**. Our engineers are even **having fun** using the tool!…"

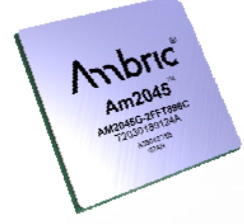
Shawn Carnahan, CTO, Telestream

*…s of numerous software development … market, the **quality of** … **designer tool suite is** … rous tests on … **fied development** … **rocessor***

…erest Consultants Inc.

"Most applications are compiled … are development, the design, debug, edit, and re-run cycle is nearly … essor communication and synchronization is simple. Sending and receiving a word … el is so simple, just like reading or writing a processor register. This kind of development is much easier and cheaper and achieves long-term scalability, performance, and power advantages of massive parallelism."

Chaudhry Majid Ali and Muhammad Qasim, Halmstad University, Sweden



www.Ambric.com

References: See workshop paper

Thank you!