Era of Customization and Specialization

Jason Cong
Chancellor's Professor, UCLA Computer Science Department
cong@cs.ucla.edu
Director, Center for Domain-Specific Computing
www.cdsc.ucla.edu

Power Barrier and Current Solution

- 10’s to 100’s cores in a processor
- 1000’s to 10,000’s servers in a data center
Utilization Wall [G. Venkatesh et.al. ASPLOS’10]

- Assuming 80W power budget,
  - At 45 nm TSMC process, less than 7% of a 300mm² die can be switched.
- ITRS roadmap and CMOS scaling theory:
  - Less than 3.5% in 32 nm
  - Almost half with each process generation
  - Even further with 3-D integration.

Dark Silicon and the End of Multicore Scaling
[H. Esmaeilzadeh et. al., ISCA’11]

- Power wall:
  - At 22 nm, 31% of a fixed-size chip must be powered off
  - At 8 nm, more than 50%.
- A significant gap between what is achievable and what is expected by Moore’s Law
  - Due to power and parallelism limitations
  - Speedup gap of at least 22x at 8 nm technology
Next Big Opportunity – Customization and Specialization

Potential of Customization/Specialization

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>Power</th>
<th>Figure of Merit (Gb/s/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES 128bit key</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128bit data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.18mm CMOS</td>
<td>3.84 Gbit/s</td>
<td>350 mW</td>
<td>11 (1/1)</td>
</tr>
<tr>
<td>FPGA [1]</td>
<td>1.32 Gbit/s</td>
<td>490 mW</td>
<td>2.7 (1/4)</td>
</tr>
<tr>
<td>ASM StrongARM [2]</td>
<td>31 Mbit/s</td>
<td>240 mW</td>
<td>0.13 (1/85)</td>
</tr>
<tr>
<td>ASM Pentium III [3]</td>
<td>648 Mbit/s</td>
<td>41.4 W</td>
<td>0.015 (1000)</td>
</tr>
<tr>
<td>C Emb. Sparc [4]</td>
<td>133 Kbits/s</td>
<td>120 mW</td>
<td>0.0011 (1/100000)</td>
</tr>
<tr>
<td>Java [5] Emb. Sparc</td>
<td>450 bits/s</td>
<td>120 mW</td>
<td>0.0000037 (1/3,000,000)</td>
</tr>
</tbody>
</table>

[1] Amphion CS8320 on Virtex2 + Xilinx Virtex2 Power Estimator
[4] gcc, 1 mW/MHz @ 120 Mhz Sparc – assumes 0.25 u CMOS
[5] Java on JVM (Sun J2ME, non-JIT) on 1 mW/MHz @ 120 Mhz Sparc – assumes 0.25 u CMOS

Another Example of Specialization -- Advance of Civilization

- For human brain, Moore’s Law scaling has long stopped
  - The number neurons and their firing speed did not change significantly
- Remarkable advancement of civilization via specialization
- More advanced societies have higher degree of specialization
- Achieved on a common platform!

Example of Customizable Platforms: FPGAs

- Configurable logic blocks
- Island-style configurable mesh routing
- Dedicated components
  - Specialization allows optimization
  - Memory/Multiplier
  - I/O, Processor
  - Anything that the FPGA architect wants to put in!

More Opportunities for Customization to be Explored

Key questions:
Optimal trade-off between efficiency & customizability
Which options to fix at CHP creation? Which to be set by CHP mapper?

Research Scope in CDSC (Center for Domain-Specific Computing)
Examples of Energy-Efficient Customization

- Customization of processor cores
- Customization of on-chip memory
- Customization of on-chip interconnects

Extensive Use of Accelerators

- Accelerators provide high power-efficiency over general-purpose processors
  - IBM wire-speed processor
  - Intel Larrabee
- ITRS 2007 System drivers prediction: Accelerator number close to 1500 by 2022

- Two kinds of accelerators
  - Tightly coupled – part of datapath
  - Loosely coupled – shared via NoC
- Challenges
  - Accelerator extraction and synthesis
  - Efficient accelerator management
    - Scheduling
    - Sharing
    - Virtualization …
  - Friendly programming models
Managing accelerators through the OS is expensive

In an accelerator rich CMP, management should be cheaper both in terms of time and energy

- Invoke "Open"s the driver and returns the handler to driver. Called once.
- RD/WR is called multiple times.

### Operation Latency (in Cycles)

<table>
<thead>
<tr>
<th>Operation</th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
<th>16 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke</td>
<td>214413</td>
<td>256401</td>
<td>266133</td>
<td>308434</td>
<td>316161</td>
</tr>
<tr>
<td>RD/WR</td>
<td>703</td>
<td>725</td>
<td>781</td>
<td>837</td>
<td>885</td>
</tr>
</tbody>
</table>

#### Overall Architecture of ARC

- **Architecture of ARC**
  - Multiple cores and accelerators
  - Global Accelerator Manager (GAM)
  - Shared L2 cache banks and NoC routers between multiple accelerators
Overall Communication Scheme in ARC

1. The core requests for a given type of accelerator (lcacc-req).

New ISA

- lcacc-req t
- lcacc-rsrv t, e
- lcacc-cmd id, f, addr
- lcacc-free id

Overall Communication Scheme in ARC

2. The GAM responds with a “list + waiting time” or NACK.
Overall Communication Scheme in ARC

3. The core reserves (lcacc-rsv) and waits.

New ISA
- lcacc-req t
- lcacc-rsrv t, e
- lcacc-cmd id, f, addr
- lcacc-free id

4. The GAM ACK the reservation and send the core ID to accelerator
5. The core shares a task description with the accelerator through memory and starts it (lcacc-cmd).
   - Task description consists of:
     - Function ID and input parameters
     - Input/output addresses and strides

6. The accelerator reads the task description, and begins working
   - Overlapped Read/Write from/to Memory and Compute
   - Interrupting core when TLB miss
7. When the accelerator finishes its current task it notifies the core.

8. The core then sends a message to the GAM freeing the accelerator (lcacc-free).
Accelerator Chaining and Composition

- **Chaining**
  - Efficient accelerator to accelerator communication

- **Composition**
  - Constructing virtual accelerators

Accelerator Virtualization

- Application programmer or compilation framework selects high-level functionality
- **Implementation via**
  - Monolithic accelerator
  - Distributed accelerators composed to a virtual accelerator
  - Software decomposition libraries
- Example: Implementing a 4x4 2-D FFT using 2 4-point 1-D FFT
Accelerator Virtualization

- Application programmer or compilation framework selects high-level functionality
- Implementation via
  - Monolithic accelerator
  - Distributed accelerators composed to a virtual accelerator
  - Software decomposition libraries
- Example: Implementing a 4x4 2-D FFT using 2 4-point 1-D FFT

Step 1: 1D FFT on Row 1 and Row 2

Step 2: 1D FFT on Row 3 and Row 4
Accelerator Virtualization

- Application programmer or compilation framework selects high-level functionality
- Implementation via
  - Monolithic accelerator
  - Distributed accelerators composed to a virtual accelerator
  - Software decomposition libraries
- Example: Implementing a 4x4 2-D FFT using 2 4-point 1-D FFT

Step 3: 1D FFT on Col 1 and Col 2

Accelerator Virtualization

- Application programmer or compilation framework selects high-level functionality
- Implementation via
  - Monolithic accelerator
  - Distributed accelerators composed to a virtual accelerator
  - Software decomposition libraries
- Example: Implementing a 4x4 2-D FFT using 2 4-point 1-D FFT

Step 4: 1D FFT on Col 3 and Col 4
Light-Weight Interrupt Support

CPU  GAM  LCA

Light-Weight Interrupt Support

CPU  GAM  LCA

Request/Reserve Confirmation and NACK Sent by GAM
Light-Weight Interrupt Support

CPU → GAM

LCA

TLB Miss
Task Done

Core Sends Logical Addresses to LCA
LCA keeps a small TLB for the addresses that it is working on
Light-Weight Interrupt Support

Core Sends Logical Addresses to LCA
LCA keeps a small TLB for the addresses that it is working on

Why Logical Address?
1- Accelerators can work on irregular addresses (e.g. indirect addressing)
2- Using large page size can be a solution but will effect other applications

Light-Weight Interrupt Support

It’s expensive to handle the interrupts via OS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency to switch to ISR and back (# Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core</td>
</tr>
<tr>
<td>Interrupt</td>
<td>16 K</td>
</tr>
</tbody>
</table>
Light-Weight Interrupt Support

Extending the core with a light-weight interrupt support

Two main components added:
- A table to store ISR info
- An interrupt controller to queue and prioritize incoming interrupt packets

Each thread registers:
- Address of the ISR and its arguments and lw-int source

Limitations:
- Only can be used when running the same thread which LW interrupt belongs to
- OS-handled interrupt otherwise
Programming interface to ARC

Evaluation methodology

- Benchmarks
  - Medical imaging
  - Vision & Navigation

<table>
<thead>
<tr>
<th>CPU</th>
<th>Ultra-SPARC III-i @ 2.0GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>1, 2, 4, 8, 16</td>
</tr>
<tr>
<td>Coherence protocol</td>
<td>MSI_MOSI_CMP_directory</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32 KB, 4 way set-associative</td>
</tr>
<tr>
<td>L2 cache</td>
<td>8 MB, 8-way set-associative</td>
</tr>
<tr>
<td>Memory latency</td>
<td>1000 cycles</td>
</tr>
<tr>
<td>Network topology</td>
<td>Mesh</td>
</tr>
<tr>
<td>Operating System</td>
<td>Solaris10</td>
</tr>
</tbody>
</table>
Application Domain: Medical Image Processing

Medical images exhibit sparsity, and can be sampled at a rate \( \ll \) classical Shannon-Nyquist theory:

\[
\min_{x} \sum_{\text{samples/voxel}} |\hat{x}| + \lambda \sum_{\text{voxels}} \|x\|_1
\]

compressive sensing

\[
\forall \text{coast: } u(x) = \sqrt{\sum_{\text{voxel}} w_j f_j(x)}^2 - 2x^2, \quad w_j = \frac{1}{2\|x\|^2}
\]

total variational algorithm

\[
\nu \frac{\partial u}{\partial x} + \nu \cdot v u
\]

fluid registration

\[
\frac{\partial u}{\partial t} + \nabla \cdot (F \partial u, F \partial v) = \frac{\partial u}{\partial x} + \lambda \delta (u - f(x, t))
\]

level set methods

\[
\frac{\partial u}{\partial t} + \nabla \cdot (\nabla u \cdot v) = 0
\]

Navier-Stokes equations

\[
\sum_{i} \frac{\partial U_i}{\partial x_j} - \sum_{j} \frac{\partial U_j}{\partial x_i} = \sum_{i} \frac{\partial ^2 U_i}{\partial x_i x_j} + f_i(x, t)
\]

Area Overhead

<table>
<thead>
<tr>
<th>Core</th>
<th>NoC</th>
<th>L2</th>
<th>Deblur</th>
<th>Denoise</th>
<th>Segmentation</th>
<th>Registration</th>
<th>SPM Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of instance/Size</td>
<td>1</td>
<td>1</td>
<td>8MB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Area(mm^2)</td>
<td>10.8</td>
<td>0.3</td>
<td>39.8</td>
<td>0.03</td>
<td>0.01</td>
<td>0.06</td>
<td>0.01</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>18.0</td>
<td>0.5</td>
<td>66.2</td>
<td>3.4</td>
<td>0.8</td>
<td>6.5</td>
<td>1.2</td>
</tr>
<tr>
<td>Total ARC: 14.3 %</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- AutoESL (from Xilinx) for C to RTL synthesis
- Synopsys for ASIC synthesis
  - 32 nm Synopsys Educational library
- CACTI for L2
- Orion for NoC
- One UltraSparc IIIi core (area scaled to 32 nm)
  - 178.5 mm^2 in 0.13 um (http://en.wikipedia.org/wiki/UltraSPARC_IIIi)
Experimental Results – Performance
(N cores, N threads, N accelerators)

Performance improvement over SW only approaches:
on average 168x, up to 380x

Performance improvement over OS based approaches:
on average 51x, up to 292x

Experimental Results – Energy
(N cores, N threads, N accelerators)

Energy improvement over SW-only approaches:
on average 241x, up to 641x

Energy improvement over OS-based approaches:
on average 17x, up to 63x
What are the Problems with ARC?

♦ Dedicated accelerators are inflexible
  - An LCA may be useless for new algorithms or new domains
  - Often under-utilized
  - LCAs contain many replicated structures
    - Things like fp-ALUs, DMA engines, SPM
    - Unused when the accelerator is unused

♦ We want flexibility and better resource utilization
  - Solution: CHARM

♦ Private SPM is wasteful
  - Solution: BiN

A Composable Heterogeneous Accelerator-Rich Microprocessor (CHARM) [ISLPED’12]

♦ Motivation
  - Great deal of data parallelism
    - Tasks performed by accelerators tend to have a great deal of data parallelism
  - Variety of LCAs with possible overlap
    - Utilization of any particular LCA being somewhat sporadic
  - It is expensive to have both:
    - Sufficient diversity of LCAs to handle the various applications
    - Sufficient quantity of a particular LCA to handle the parallelism
  - Overlap in functionality
    - LCAs can be built using a limited number of smaller, more general LCAs: Accelerator building blocks (ABBs)

<table>
<thead>
<tr>
<th>ABBs</th>
<th>Denoise</th>
<th>Deblur</th>
<th>Registration</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float Reciprocal (FInv)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Float Square-Root (Fsqrt)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Float Polynomial-16 (Poly16)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Float Divide (FDiv)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

♦ Idea
  - Flexible accelerator building blocks (ABB) that can be composed into accelerators
  - Leverage economy of scale
Micro Architecture of CHARM

- **ABB**
  - Accelerator building blocks (ABB)
  - Primitive components that can be composed into accelerators
  - **ABB islands**
    - Multiple ABBs
    - Shared DMA controller, SPM and NoC interface

- **ABC**
  - Accelerator Block Composer (ABC)
    - To orchestrate the data flow between ABBs to create a virtual accelerator
    - Arbitrate requests from cores

- **Other components**
  - Cores
  - L2 Banks
  - Memory controllers

An Example of ABB Library (for Medical Imaging)

<table>
<thead>
<tr>
<th>ABBs</th>
<th>Denoise</th>
<th>Deblur</th>
<th>Registration</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float Reciprocal (FInv)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Float Square-Root (FSqrt)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Float Polynomial-16 (Poly16)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Float Divide (FDiv)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Example of ABB Flow-Graph (Denoise)

\[ \frac{1}{\sqrt{\sum_{i=0}^{6} (X_i - Y)^2}} \]
Example of ABB Flow-Graph (Denoise)

\[ \frac{1}{\sqrt{\sum_{i=0}^{6} (X_i - Y)^2}} \]

ABB1: Poly
ABB2: Poly
ABB3: Sqrt
ABB4: Inv
**LCA Composition Process**

1. **Core initiation**
   - Core sends the task description: task flow-graph of the desired LCA to ABC together with polyhedral space for input and output

Task description

```
10x10 input and output
```

```
Core -> ABC
```

```
ABB ISLAND1: x, y
ABB ISLAND2: x, w
ABB ISLAND3: z, w
ABB ISLAND4: y, z
```
### LCA Composition Process

#### 2. Task-flow parsing and task-list creation
- ABC parses the task-flow graph and breaks the request into a set of tasks with smaller data size and fills the task list.

- **Needed ABBs:** “x”, “y”, “z”
- **With task size of 5x5 block, ABC generates 4 tasks**

![Task-flow parsing diagram]

#### 3. Dynamic ABB mapping
- ABC uses a pattern matching algorithm to assign ABBs to islands.
- Fills the composed LCA table and resource allocation table.

<table>
<thead>
<tr>
<th>Island ID</th>
<th>ABB Type</th>
<th>ABB ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>1</td>
<td>Free</td>
</tr>
</tbody>
</table>
LCA Composition Process

3. Dynamic ABB mapping
   - ABC uses a pattern matching algorithm to assign ABBs to islands
   - Fills the composed LCA table and resource allocation table

<table>
<thead>
<tr>
<th>Island ID</th>
<th>ABB Type</th>
<th>ABB ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>1</td>
<td>Free</td>
</tr>
</tbody>
</table>

LCA Composition Process

4. LCA cloning
   - Repeat to generate more LCAs if ABBs are available

<table>
<thead>
<tr>
<th>Core ID</th>
<th>ABB Type</th>
<th>ABB ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>1</td>
<td>Busy</td>
</tr>
</tbody>
</table>
LCA Composition Process

5. ABBs finishing task
   - When ABBs finish, they signal the ABC. If ABC has another task it sends otherwise it frees the ABBs

<table>
<thead>
<tr>
<th>Island ID</th>
<th>ABB Type</th>
<th>ABB ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>1</td>
<td>Busy</td>
</tr>
</tbody>
</table>

DONE

LCA Composition Process

5. ABBs being freed
   - When an ABB finishes, it signals the ABC. If ABC has another task it sends otherwise it frees the ABBs

<table>
<thead>
<tr>
<th>Island ID</th>
<th>ABB Type</th>
<th>ABB ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>1</td>
<td>Busy</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>1</td>
<td>Free</td>
</tr>
</tbody>
</table>
LCA Composition Process

6. Core notified of end of task
   - When the LCA finishes ABC signals the core

<table>
<thead>
<tr>
<th>Island ID</th>
<th>ABB Type</th>
<th>ABB ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>1</td>
<td>Free</td>
</tr>
</tbody>
</table>

ABC Internal Design

- **ABC sub-components**
  - Resource Table (RT): To keep track of available/used ABBs
  - Composed LCA Table (CLT): Eliminates the need to re-compose LCAs
  - Task List (TL): To queue the broken LCA requests (to smaller data size)
  - TLB: To service and share the translation requests by ABBs
  - Task Flow-Graph Interpreter (TFGI): Breaks the LCA DFG into ABBs
  - LCA Composer (LC): Compose the LCA using available ABBs

- **Implementation**
  - RT, CLT, TL and TLB are implemented using RAM
  - TFGI has a table to keep ABB types and an FSM to read task-flow-graph and compares
  - LC has an FSM to go over CLT and RT and check mark the available ABBs
CHARM Software Infrastructure

- ABB type extraction
  - Input: compute-intensive kernels from different application
  - Output: ABB Super-patterns
  - Currently semi-automatic
- ABB template mapping
  - Input: Kernels + ABB types
  - Output: Covered kernels as an ABB flow-graph
- CHARM uProgram generation
  - Input: ABB flow-graph
  - Output:

Evaluation Methodology

- Simics+GEMS based simulation
- AutoPilot/Xilinx+ Synopsys for ABB/ABC/DMA-C synthesis
- Cacti for memory synthesis (SPM)
- Automatic flow to generate the CHARM software and simulation modules
- Case studies
  - Physical LCA sharing with Global Accelerator Manager (LCA+GAM)
  - Physical LCA sharing with ABC (LCA+ABC)
  - ABB composition and sharing with ABC (ABB+ABC)
- Medical imaging benchmarks
  - Denoise, Deblur, Segmentation and Registration
Area Overhead Analysis

- **Area-equivalent**
  - The total area consumed by the ABBs equals the total area of all LCAs required to run a single instance of each benchmark.

- **Total CHARM area is 14% of the 1cm x 1cm chip**
  - A bit less than LCA-based design

<table>
<thead>
<tr>
<th>Name</th>
<th>(A(\mu m^2))</th>
<th>(P(mW))</th>
<th>Total#</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDIV</td>
<td>4940</td>
<td>0.264</td>
<td>12</td>
</tr>
<tr>
<td>Poly16</td>
<td>38276</td>
<td>1.608</td>
<td>96</td>
</tr>
<tr>
<td>Freq</td>
<td>3503</td>
<td>0.141</td>
<td>12</td>
</tr>
<tr>
<td>FSqrt</td>
<td>58683</td>
<td>1.83</td>
<td>8</td>
</tr>
<tr>
<td>SPI-4KB 1R/W</td>
<td>13591</td>
<td>17.6</td>
<td>288</td>
</tr>
<tr>
<td>SPI-56KB 1R/W</td>
<td>2545</td>
<td>7</td>
<td>72</td>
</tr>
<tr>
<td>ABC</td>
<td>8383</td>
<td>0.066</td>
<td>1</td>
</tr>
</tbody>
</table>

Results: Improvement Over LCA-based Design

- **N’x’ has N times area - equivalent accelerators**

- **Performance**
  - 2.5X vs. LCA+GAM (max 5X)
  - 1.4X vs. LCA+ABC (max 2.6X)

- **Energy**
  - 1.9X vs. LCA+GAM (max 3.4X)
  - 1.3X vs. LCA+ABC (max 2.2X)

- **ABB+ABC has better energy and performance**
  - ABC starts composing ABBs to create new LCAs
  - Creates more parallelism
**Results: Platform Flexibility**

- Two applications from two unrelated domains to MI
  - Computer vision
    - Log-Polar Coordinate Image Patches (LPCIP)
  - Navigation
    - Extended Kalman Filter-based Simultaneous Localization and Mapping (EKF-SLAM)
- Only one ABB is added
  - Indexed Vector Load

![Normalized Performance Chart](image)

<table>
<thead>
<tr>
<th></th>
<th>MAX Benefit over LCA+GAM</th>
<th>AVG Benefit over LCA+GAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPCP</td>
<td>3.64X</td>
<td>2.46X</td>
</tr>
<tr>
<td>EKF-SLAM</td>
<td>3.04X</td>
<td>2.05X</td>
</tr>
</tbody>
</table>

**Examples of Energy-Efficient Customization**

- Customization of processor cores
- Customization of on-chip memory
- Customization of on-chip interconnects
Memory Management for Accelerator-Rich Architectures [ISLPED’2012]

- Providing a private buffer for each accelerator is very inefficient.
  - Large private buffers: occupy a considerable amount of chip area
  - Small private buffers: less effective for reducing off-chip bandwidth
- Not all accelerators are powered-on at the same time
  - Shared buffer [Lyonsy et al. TACO’12]
  - Allocate the buffers in the cache on-demand [Fajardo et al. DAC’11][Cong et al. ISLPED’11]
- Our solution
  - BiN: A Buffer-in-NUCA Scheme for Accelerator-Rich CMPs

Buffer Size vs. Off-chip Memory Access Bandwidth

- Buffer size ↑ - off-chip memory bandwidth ↓: covering longer reuse distance [Cong et al. ICCAD’11]
- Buffer size vs. bandwidth curve: BB-Curve
- Buffer utilization efficiency
  - Different for various accelerators
  - Different for various inputs for one accelerator
- Prior work: no consideration of global allocation at runtime
  - Accept fixed-size buffer allocation requests
  - Rely on the compiler to select a single, ‘best’ point in the BB-Curve

![Buffer Size vs. Off-chip Memory Access Bandwidth](chart)

- High buffer utilization efficiency
- Low buffer utilization efficiency

Input image: cube(28)  cube(52)  cube(76)

Denoise

Buffer size (KB)
**Resource Fragmentation**
- Prior work allocates a contiguous space to each buffer to simplify buffer access
- Requested buffers have unpredictable space demand and come in dynamically: resource fragmentation
- NUCA complicates buffer allocations in cache
  - The distance of the cache bank to the accelerator also matters
- To support fragmented resources: paged allocation
  - Analogous to a typical OS-managed virtual memory
- Challenges:
  - Large private page tables have high energy and area overhead
  - Indirect access to a shared page table has high latency overhead

```
Shared buffer space: 15KB
Buffer 1: 5KB, duration: 1K cycles
Buffer 2: 5KB, duration: 2K cycles
Buffer 3: 10KB, duration: 2K cycles
```

**BiN: Buffer-in-NUCA**
- Goals of Buffer-in-NUCA (BiN)
  - Towards optimal on-chip storage utilization
  - Dynamically allocate buffer space in the NUCA among a large number of competing accelerators
- Contributions of BiN:
  - Dynamic interval-based global (DIG) buffer allocation: address the buffer resource contention
  - Flexible paged buffer allocation: address the buffer resource fragmentation
**Dynamic Interval-based Global (DIG) Allocation**

- Perform global allocation for buffer allocation requests in an interval
  - Keep the interval short (10K cycles): Minimize waiting-in-interval
  - If 8 or more buffer requests, the DIG allocation will start immediately
- An example: 2 buffer allocation requests
  - Each point \((b, s)\)
    - \(s\): buffer size
    - \(b\): corresponding bandwidth requirement at \(s\)
    - Buffer utilization efficiency at each point: \(\frac{(b_{j} - b_{j-1})}{(s_{j} - s_{j-1})}\)
  - The points are in non-decreasing order of buffer size

![Buffer allocation diagram](image)

- The accelerator and BiN manager (ABM)
- Arbitration over accelerator resources
- Allocates buffers in the shared cache (BiN management)

**Accelerator-Rich CMP with BiN**

- Overall architecture of ARC [Cong et al. DAC 2011] with BiN
  - Cores (with private L1 caches)
  - Accelerators
    - Accelerator logic
    - DMA-controller
    - A small storage for the control structure
  - The accelerator and BiN manager (ABM)
    - Arbitration over accelerator resources
    - Allocates buffers in the shared cache (BiN management)
  - NUCA (shared L2 cache) banks

![Architecture diagram](image)

- Core sends the accelerator and buffer allocation request with the BB-Curve to ABM.
- ABM performs accelerator allocation, buffer allocation in NUCA, and acknowledges the core.
- The core sends the control structure to the accelerator.
- The accelerator starts working with its allocated buffer.
- The accelerator signals to the core when it finishes.
- The core sends the free-resource message to ABM.
- ABM frees the accelerator and buffer in NUCA.

![Process flow diagram](image)
**Flexible Paged Allocation**

- Set the page size according to buffer size: Fixed total number of pages for each buffer
- BIN manager locally keep the information of the current contiguous buffer space in each L2 bank
  - Since all of the buffer allocation and free operations are performed by BIN manager
- Allocation: starting from the nearest L2 bank to this accelerator, to the farthest
- We allow the last page (source of page fragments) of a buffer to be smaller than the other pages of this buffer
  - No impact on the page table lookup
  - The max page fragment will be smaller than the min-page
  - The page fragments do not waste capacity since they can be used by cache

![Diagram of BIN manager allocating pages among cache banks]

**Buffer Allocation in NUCA**

- Total buffer size
  - Buffers are allocated on-demand
  - Set an upper-bound of the total buffer size: reduce the impact on cache
  - State-of-the-art cache partitioning can be used to dynamically tune the upper bound
    - E.g., [Qureshi & Patt, MICRO’06]
- Buffer allocations among cache banks
  - Distribute the imposed upper bound onto cache banks
    - Avoid creating high contention in a particular cache bank
  - State-of-the-art NUCA management schemes can be used to further mitigate contention introduced by buffer allocation
    - E.g., page re-coloring scheme [Cho & Jin, MICRO’06]
**Hardware Overhead of BiN Management**

- Storage:
  - 32 SRAMs: contiguous spaces info in cache banks
    - 7-entry: at most 7 contiguous spaces in a 64KB cache bank with a min-page of 4KB
      
      ![Contiguous Spaces](image)

  - 14 bits wide (10 bits: the starting block ID, 4 bits: the space length in terms of min-page)
  - 8 SRAMs: the BB-curves of the buffer requests
    - 8-entry: at most 8 BB-Curve points
    - 58 wide: 2B for the buffer size and 3B for the buffer usage efficiency
  - Total storage overhead: 768B, area: 3,282um

- Logic:
  - 9,725um² @ 2GHz (Synopsys DC, SAED library @ 32nm)
  - An average latency of 0.8us (1.2K cycles @ 2GHz) to perform the buffer allocations

- The total area of the buffer allocation module is less than 0.01% for a medium size 1cm² chip

---

**Simulation Infrastructure & Benchmarks**

- Extend the full-system cycle-accurate Simics+GEMS simulation platform to support ARC+BiN

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4 Ultra-SPARC III+ cores @ 2GHz</td>
</tr>
<tr>
<td>L1 data/instruction cache</td>
<td>32KB for each core, 4-way set-associative, 64B cache block, 3-cycle access latency, pseudo-LRU, MESI directory coherence by L2 cache</td>
</tr>
<tr>
<td>L2 cache (NUCA)</td>
<td>2MB, 32 banks, each bank is 64KB, 8-way set-associative, 64B cache block, 6-cycle access latency, pseudo-LRU</td>
</tr>
<tr>
<td>Network on chip</td>
<td>4X8 mesh, XY routing, wormhole switching, 3-cycle router latency, 1-cycle link latency</td>
</tr>
<tr>
<td>Main memory</td>
<td>4GB, 1000-cycle access latency</td>
</tr>
</tbody>
</table>

- Benchmark: 4 medical imaging applications in a medical imaging pipeline
  - Use the accelerator extraction method of [Cong et al., DAC'12]
  - Accelerator is synthesized by AutoESL, from Xilinx

- Experimental benchmark naming convention
  - mP-n: m copies of pipelines, the input to each is a unique n³ pixels image
    - No Fragmentation: Used to show the gain of DIG allocation only
  - mP-mic: m copies of pipelines, the inputs are randomly selected
    - Fragmentation occurs: Used to show the gain of both DIG and paged allocation
Reference Design Schemes

- **Accelerator Store (AS) [Lyonsy, et al. TACO’12]**
  - Separate cache and shared buffer module
  - Set the buffer size 32% larger than maximum buffer size in BiN: overhead of buffer-in-cache
  - Partition the shared buffer into 32 banks distributed them to the 32 NoC nodes

- **BiC [Fajardo, et al. DAC’11]**
  - BiC dynamically allocates contiguous cache space to a buffer
  - Upper bound: limiting buffer allocation to at most half of each cache bank
  - Buffers can span multiple cache banks

- **BiN-Paged**
  - Only has the proposed paged allocation scheme

- **BiN-Dyn**
  - Based on BiN-Paged, it also performs dynamic allocation without consideration of near future buffer requests
  - It responds to a request immediately by greedily satisfying the request with the current available resources

- **BiN-Full**
  - This is the entire proposed BiN scheme

Impact of Dynamic Interval-based Global Allocation

- BiN-Full consistently outperforms the other schemes
  - The only exception: 4P-mix3
    - 1.32X larger capacity of the AS can accommodate all buffer requests

- Overall, compared to the accelerator store and BiC, BiN-Full reduces the runtime reduction by 32% and 35%, respectively
Impact on Energy

♦ AS consumes the least per-cache/buffer access energy and the least unit leakage
  ▪ Because in the accelerator store the buffer and cache are two separate units

♦ BiN-Dyn
  ▪ Saves energy in cases where it can reduce the off-chip memory accesses and runtime
  ▪ Results in a large energy overhead in cases where it significantly increases the runtime

♦ Compared with the AS, BiN-Full reduces the energy by 12% on average
  ▪ Exception: 4P-mix-{2,3}
    • The 1.32X capacity of AS can better satisfy buffer requests

♦ Compared with BiC, BiN-Full reduces the energy by 29% on average

Examples of Energy-Efficient Customization

♦ Customization of processor cores
♦ Customization of on-chip memory
♦ Customization of on-chip interconnects
**Terahertz VCO in 65nm CMOS**

- Demonstrated an ultra high frequency and low power oscillator structure in CMOS by adding a negative resistance parallel tank, with the fundamental frequency at 217GHz and 16.8 mW DC power consumption.
- The measured 4th and 6th harmonics are about 870GHz and 1.3THz, respectively.

*higher harmonics (4th and 6th harmonics) may be substantially underestimated due to excessive water and oxygen absorption and setup losses at these frequencies.*

**Generating Terahertz Signals in 65nm CMOS with Negative-Resistance Resonator Boosting and Selective Harmonic Suppression**
- Symposium on VLSI Technology and Circuits, June 2010

---

**Use of Multiband RF-Interconnect for Customization**

- In TX, each mixer up-converts individual baseband streams into specific frequency band (or channel)
- N different data streams (N=6 in exemplary figure above) may transmit simultaneously on the shared transmission medium to achieve higher aggregate data rates
- In RX, individual signals are down-converted by mixer, and recovered after low-pass filter
Mesh Overlaid with RF-I [HPCA ’08]

- 10x10 mesh of pipelined routers
  - NoC runs at 2GHz
  - XY routing
- 64 4GHz 3-wide processor cores
  - Labeled aqua
  - 8KB L1 Data Cache
  - 8KB L1 Instruction Cache
- 32 L2 Cache Banks
  - Labeled pink
  - 256KB each
  - Organized as shared NUCA cache
- 4 Main Memory Interfaces
  - Labeled green
- RF-I transmission line bundle
  - Black thick line spanning mesh

RF-I Logical Organization

- **Logically:**
  - RF-I behaves as set of N express channels
  - Each channel assigned to src, dest router pair (s,d)
- **Reconfigured by:**
  - remapping shortcuts to match needs of different applications
Latest Progress: Die Photo of STL-DBI Transceiver

Controller Side: Memory Side

- Active Area: 0.12mm² (15% smaller than Ref. [4])

Comparison with State-of-the-art

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>65nm</td>
<td>180nm</td>
<td>130nm</td>
<td>40nm</td>
<td>65nm</td>
</tr>
<tr>
<td>Link Type</td>
<td>SBD</td>
<td>Bidirectional</td>
<td>Bidirectional</td>
<td>Bidirectional</td>
<td>SBD</td>
</tr>
<tr>
<td>Total Power</td>
<td>14.4mW(BB)</td>
<td>17.6mW(RF)</td>
<td>87mW</td>
<td>95mW</td>
<td>14.4mW</td>
</tr>
<tr>
<td></td>
<td>12mW (BB)</td>
<td>11mW (RF)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chip Area</td>
<td>0.12mm²</td>
<td>0.52mm²</td>
<td>0.30mm²</td>
<td>0.9mm²</td>
<td>0.14mm²</td>
</tr>
<tr>
<td>FoM</td>
<td>2.08</td>
<td>0.11</td>
<td>0.21</td>
<td>0.17</td>
<td>1.30</td>
</tr>
</tbody>
</table>

\[
\text{FoM} = \frac{\text{DR} / \text{pin}}{\text{Area} \cdot \text{Power}} = \frac{\text{Gb} / \text{pin}}{\text{mm}^2 \cdot \text{mJ}}
\]

* [1] K.-I. Oh, et al., JSSC2009 (Samsung)
* [2] K.-S. Ha, et al., ISSCC2009 (Samsung)
Research Scope in CDSC (Center for Domain-Specific Computing)

Customizable Heterogeneous Platform

- Domain-specific modeling (healthcare applications)
- Application modeling
- Architecture modeling
- CHP creation
- Customizable computing engines
- Customizable interconnects

Design once
Invoke many times

CHP Mapping Overview

Goal: Efficient mapping of domain-specific application to customizable hardware

Adapt the CHP to a given application so as to optimize performance/power efficiency

Domain-specific applications

Programmer

Abstract execution

Application characteristics

CHP architecture models

Source-to-source CHP Mapper (Rose)

Domain-specific programming model
(Domain-specific coordination graph and domain-specific language extensions)

C/C++ code

ROSE SAGE IR

Reconfiguring and optimizing back-end (LLVM)

Binary code for fixed & customized cores

Unified Adaptive Runtime system
(maps tasks across CPUs, GPUs, Accelerators, FPGA processors)

CHP architectural prototypes
(CHP hardware testbeds, CHP simulation toolbed, full CHP)
**xPilot: Behavioral-to-RTL Synthesis Flow [SOCC’2006]**

- **Behavioral spec. in C/C++/SystemC**
- **Frontend compiler**
- **SSDM**
  - RTL + constraints
- **FPGAs/ASICs**

- **Advanced transformation/optimizations**
  - Loop unrolling/shifting/pipelining
  - Strength reduction / Tree height reduction
  - Bitwidth analysis
  - Memory analysis

- **Core behavior synthesis optimizations**
  - Scheduling
  - Resource binding, e.g., functional unit binding
  - Register/port binding

- **μArch-generation & RTL/constraints generation**
  - Verilog/VHDL/SystemC
  - FPGAs: Altera, Xilinx
  - ASICs: Magma, Synopsys

- **Advanced transformtion/optimizations**
  - Loop unrolling/shifting/pipelining
  - Strength reduction / Tree height reduction
  - Bitwidth analysis
  - Memory analysis

**AutoPilot Compilation Tool (based UCLA xPilot system)**

- Platform-based C to FPGA synthesis
- Synthesize pure ANSI-C and C++
  +, GCC-compatible compilation flow
- Full support of IEEE-754 floating point data types & operations
- Efficiently handle bit-accurate fixed-point arithmetic
- More than 10X design productivity gain
- High quality-of-results

Developed by AutoESL, acquired by Xilinx in Jan. 2011
**Programming Model and Runtime Support [LCTES12]**

- Concurrent Collection (CnC) programming model
  - Clear separation between application description and implementation
  - Fits domain expert needs
- CnC-HC: Software flow CnC => Habanero-C(HC)
- Cross-device work-stealing in Habanero-C
  - Task affinity with heterogeneous components
- Data driven runtime in CnC-HC

---

**CnC Building Blocks**

- Steps
  - Computational units
  - Functional with respects to their inputs
- Data Items
  - Means of communication between steps
  - Dynamic single assignment
- Control Items
  - Used to create (prescribe) instances of a computation step
**HC-1ex architecture**

- "Commodity" Intel Server
  - Intel® Xeon® Processor
  - Intel® Memory Controller Hub (MCH)
- Convey FPGA-based coprocessor
  - Application Engine Hub (AEH)
  - Application Engines (AEs)
  - XC6vlx760 FPGAs
  - 80GB/s off-chip bandwidth
  - 54W Design Power
- Direct Data Port

**Runtime Support Experimental results**

**Performance for medical imaging kernels**

<table>
<thead>
<tr>
<th></th>
<th>Denoise</th>
<th>Registration</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num iterations</td>
<td>3</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>CPU (1 core)</td>
<td>3.3s</td>
<td>457.8s</td>
<td>36.76s</td>
</tr>
<tr>
<td>GPU</td>
<td>0.085s (38.3 ×)</td>
<td>20.26s (22.6 ×)</td>
<td>1.263s (29.1 ×)</td>
</tr>
<tr>
<td>FPGA</td>
<td>0.190s (17.2 ×)</td>
<td>17.52s (26.1 ×)</td>
<td>4.173s (8.8 ×)</td>
</tr>
</tbody>
</table>
Experimental Results (Cont'd)

- Execution times and active energy with dynamic work stealing

Static vs Dynamic binding

- Static binding
- Dynamic Binding
Concluding Remarks

♦ Despite of end of scaling, there is plenty of opportunity with customization and specialization for energy efficient computing
♦ Many opportunities and challenges for architecture support
  - Cores
  - Accelerators
  - Memory
  - Network-on-chips
♦ Software support is also critical

Acknowledgements: CDSC Faculty

Aberle (UCLA)  Baraniuk (Rice)  Bui (UCLA)  Chang (UCLA)  Cheng (UCSB)  Cong (Director) (UCLA)
Palsberg (UCLA)  Potkonjak (UCLA)  Reinman (UCLA)  Sadayappan (Ohio-State)  Sarkar (Associate Dir) (Rice)  Vese (UCLA)
More Acknowledgements

Mohammad Ali Ghodrat

Michael Gill  Hui Huang  Chunyue Liu  Yi Zou  Beayna Grigorian

This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127, GSRC under contract 2009-TJ-1984.