CS 426

Current Directions
Memory Issues in 3D Architectures

- Benefits of a 3D chip over a 2D design
  - Reduction on global interconnect
    - Performance: reduced average interconnect length
    - Power: reduction in total wiring length
  - Higher packing density and smaller footprint
  - Support for realization of mixed-technology chips
Memory Issues in 3D Architectures

- Placement of processors and storage blocks
  - A challenging problem for 3D CMPs
  - Both power and performance depend on data communication distances
  - Frequently accessed data from certain storage blocks; these storage blocks should be placed close
  - Data sharing across processors should be considered
Heterogeneous Chip Multiprocessors

- A chip multiprocessor
  - High-complexity cores
  - Low-complexity cores

- Better resource-to-application mapping
  - Speed of a large core
  - Efficiency of a small core

- Alpha Cores
- EV8 is 80X bigger
- Only 2X – 3X performance improvement
Heterogeneous Chip Multiprocessors

- No single core is ideal for the universe of applications
  - Employ heterogeneity
  - Application - core mapping best suited for performance
- Higher computational capability per unit area
  - 63% performance improvement on equal-area (Kumar et al)
- Reduction in leakage power
  - Fewer transistors for a given computational capability
Heterogeneous Chip Multiprocessors

- **Best core-to-application mapping**
  - Execution characteristics of each application
  - Predict its future processing needs

- **Programming/compiling parallel applications**
  - Application developers typically assume that computational cores provide equal performance
  - Heterogeneity-aware compiling

- **Sensitivity of number of distinct core types**
  - Are two types enough?
CS 426

NoC Architectures
NoC Architecture

- $M \times N$ mesh architecture
- Node in the mesh
  - Processor
  - Memory module
  - Switch
Routers are roughly 1/5th the area of the processors.

Processors communicate using x-y routing.
- Used to estimate the cost of communication.
What are NoCs?

- Imagine if you can, a LAN on your chip!
- Communication over shorter distances (per hop)
- Predictable electrical parameters
- Reuse expert knowledge
  - Interconnection networking concepts
Evolution or Paradigm Shift?

- **Architectural paradigm shift**
  - Replace wire spaghetti by an intelligent network infrastructure

- **Design paradigm shift**
  - Busses and signals replaced by packets

- **Organizational paradigm shift**
  - Create a new discipline, a new infrastructure responsibility

![Diagram showing architectural and design paradigm shifts]
Future (current?) SoC Challenges

- Emergence of multi-core architectures
- Computation + Communication
  - 50% of SoC power consumption is wires
- Traditional bus-based communication cannot meet the challenges
  - Higher clock rates → longer distances → more power needed for communication
  - Bus availability challenges
  - Scalability challenges
Bus pros (😊) and cons ( Recruiting Assistant) No content (😊)

- Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.
- Bus timing is difficult in a deep submicron process.
- Bandwidth is limited and shared by all units attached.
- The silicon cost of a bus is near zero.
- Any bus is almost directly compatible with most available IPs, including software running on CPUs.
- The concepts are simple and well understood.
NoC: Good news

😊 Only point-to-point one-way wires are used, for all network sizes.
😊 Aggregated bandwidth scales with the network size.
😊 Routing decisions are distributed.
😊 NoCs increase the wires utilization (as opposed to ad-hoc p2p wires)
There’s no free lunch…

 опасно

 Internal network contention causes (often unpredictable) latency.
 опасно

 The network has a significant silicon area.
 опасно

 Bus-oriented techniques need smart wrappers.
 опасно

 Software needs clean synchronization in multiprocessor systems.
 опасно

 System designers need reeducation for new concepts.
Facts about NoC’s

- It is a way to decouple computation from communication
- The design is layered (physical, network, application…):
- Communication between processing elements in NoC takes place by encapsulating data in packets
- The elementary packet piece to which switch and routing operations apply is the flit
Topologies

- Heritage of networks with new constraints
  - Need to accommodate interconnects in a 2D layout
  - Cannot route long wires (clock frequency bound)

a) SPIN,
b) CLICHE’
c) Torus
d) Folded torus
e) Octagon
f) BFT.
Switching

- Again, techniques inherited from Computer and Communication Networks
- New constraints in silicon: area and power
  - Use as few buffers as possible
Current NoC Research focus

- **Design**
  - *Application mapping*: Map task graphs to a topology. Optimization techniques: branch and bound, GA, etc. – scalability of process?
  - *Data mapping*: After a preprocessing step of mapping tasks to processors, map communicating data to a topology. Optimization techniques, heuristics are used.
  - *Communication routing*: Usually a part of mapping, cost constrained
Background

- Intel Many Integrated Core (Intel MIC) architecture
  - Processing highly parallel workloads
  - Standard programming models (OpenMP and Cilk with a few extensions)

Knights Ferry
Packaged as a co-processor in a PCI-e card. With 32 cores running four threads apiece, this can process 128 threads at 1.2GHz.

Photo source: ZDNet
Background
Motivation: MIC Offloading

- Highly parallelizable code sections execute on MIC
Example: MIC Offloading

```c
#pragma omp parallel for
private(i, j, k) num_threads(16)
for (i=0; i<n; ++i) {
    for (j=0; j<n; ++j) {
        for (k=0; k<n; ++k) {
            arr3[i * n + j] +=
            arr1[i*n+k] * arr2[k*n+j];
        }
    }
}
```

```c
#pragma offload target(mic) \
in (arr1, arr2 : length(n*n)) \
out (arr3 : length(n*n))
```

```c
#pragma omp parallel for
private(i, j, k) num_threads(16)
for (i=0; i<n; ++i) {
    for (j=0; j<n; ++j) {
        arr3[i * n + j] = 0;
        for (k=0; k<n; ++k) {
            arr3[i*n+j] += …
            arr1[i*n+k] * arr2[k*n+j];
        }
    }
}
```
Motivation: MIC Offloading

- May incur overhead due to communication cost.
- Porting applications to MIC even with the straightforward programming model requires efforts.
## Project Goals

- **Programmer Productivity**
  - w/ offload pragma
    - Compiler selects the ones that will benefit from MIC.
  - w/ OpenMP pragma only
    - Compiler inserts pragma offload with correct in/out clauses.
  - w/o any pragmas
    - Compiler identifies the parallelizable sections and inserts pragma offload with in/out clauses.

- **Performance**
  - Various optimizations applied to maximize performance and to minimize communication overhead
CS 426

Improvements for Chip Multiprocessors
Agenda

● Performance Optimizations
  ■ Optimizing Parallelization
  ■ Dynamic Partitioning of Processing and Memory Resources
  ■ Locality-aware distributed loop scheduling

● Energy Optimizations
  ■ Selective Code/Data Migration
  ■ Workload Clustering

● Reliability Optimizations
  ■ Code Replication
CS 426

Optimizing Parallelization
Problems with current parallelization techniques

- Developed in context of high performance parallel machines
- Most of them parallelize one loop nest at a time
  - Cannot capture inter-nest relations well
- Their main goal is to minimize inter-processor communication
  - Not very suitable for chip multiprocessors
- What we need is data reuse oriented whole program parallelization
Example

for(i=1;i≤n; i++)
  for(j=1;j ≤n; j++)
    A[i][j] += B[j][i]+C[i][j]

for(i=1;i≤n; i++)
  for(j=1;j ≤n; j++)
    D[i][j] = D[i][j]+B[i][j]

for(i=1;i≤n; i++)
  for(j=1;j ≤n; j++)
    B[j][i] = B[j][i]+C[i][j]
Schemes

- **Original**
  - Unmodified/unparallelized code executed on a single processor

- **Nest-based**
  - Parallelizes each loop nest in isolation

- **Global [first nest]**
  - Parallelizes loop nests in their textual order

- **Global [ordered]**
  - Processes loop nests based on their importance

- **Constraint network (CN)**
  - Our approach
## Experimental setup and benchmarks

- 2050 lines of C++ code
- SimpleScalar
- 8 simple embedded cores
- L1 instruction and data caches of
  - 16KB
  - Two-way set-associative
  - 32 bytes of line size
- On-chip cache latency → 2 cycles
- Off-chip memory access latency → 90 cycles

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>Size</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Med-Im04</td>
<td>Medical image</td>
<td>311</td>
<td>825.55KB</td>
</tr>
<tr>
<td></td>
<td>reconstruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Laplace</td>
<td>Laplace Equation Solver</td>
<td>887</td>
<td>1,173.56KB</td>
</tr>
<tr>
<td>Radar</td>
<td>Radar imaging</td>
<td>497</td>
<td>905.28KB</td>
</tr>
<tr>
<td>Shape</td>
<td>Pattern recognition and shape analysis</td>
<td>761</td>
<td>1,284.06KB</td>
</tr>
<tr>
<td>Track</td>
<td>Visual tracking control</td>
<td>423</td>
<td>744.80KB</td>
</tr>
</tbody>
</table>
Comparison of schemes

Average Values
Nest Based: 4.48
Global [First Nest]: 5.27
Global [Ordered]: 5.66
CN Based: 6.57

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Nest Based</th>
<th>Global [First Nest]</th>
<th>Global [Ordered]</th>
<th>CN Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Med-Im04</td>
<td>4.48</td>
<td>5.27</td>
<td>5.66</td>
<td>6.57</td>
</tr>
<tr>
<td>Track</td>
<td>5.66</td>
<td>6.57</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
## Impact of the number of processors

<table>
<thead>
<tr>
<th>Speedup</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nest Based</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>Global [First Nest]</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>Global [Ordered]</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>CN Based</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
</tr>
</tbody>
</table>

![Graph](image.png)

Scalability problem!
Dynamic Partitioning of Processing and Memory Resources
Resource Allocation

● Prior OS-based resource partition approaches
  ■ Advantage: transparent to applications and programmers
  ■ Drawback: application oblivious and reactive
● Our goal: *Proactive* resource partitioning scheme
Architecture and Resource Partition Schemes

(a) Abstract view of an MPSoC architecture with 8 processors

(b) Equal partitioning of resource across two applications

(c) Nonuniform partitioning of resources across two applications

(d) Nonuniform partitioning of resources across three applications
Components of Our Approach

● Offline Estimator
  ■ Generate an estimated profile
    – Behavior of each application under different numbers of processors and different on-chip memory space
    – Ideal number of processors
    – On-chip memory space requirement
  ■ Estimated profile information can be collected from static analysis or profiling

● Runtime Resource Partitioner
  ■ Divide the number of processors and the on-chip memory space among simultaneously-executing applications
  ■ Important factors:
    – Partition it fast
    – Reasonable partition
Components of Our Approach

Application → Profiler → Resource Partitioner

Application → Profiler → Resource Partitioner

Application → Profiler → Resource Partitioner

Processor & Memory Space Partitioning
Offline Estimator

- Divide each application into epochs of the same size
- Capture the ideal number of processors and on-chip memory spaces for each epoch
Runtime Resource Partitioner

- Resource partitioner is invoked when an application
  - Execution start
  - Execution cease
  - Move from one epoch to another
- Resource partition algorithm
  - For the current epoch of the $i^{th}$ application compute
    - $\alpha_i \rightarrow$ ideal processor count
    - $\beta_i \rightarrow$ ideal on-chip memory space respectively
  - Obtain
    - Number of processors $\rightarrow \theta_i = (\alpha_i / \sum_j \alpha_j) * 100$
    - Amount of on-chip memory $\rightarrow \gamma_i = (\beta_i / \sum_j \beta_j) * 100$
  - Allocate
    - $\theta_i\%$ of processors
    - $\gamma_i\%$ of on-chip memory space (LFU-based selection of victim regions)
Comparison of Different Schemes

- Compare our proposed scheme with four alternate schemes
  - **BASE**
    - Simple
    - OS-based management
  - **EQUAL**
    - Processors are partitioned equally
    - Memory space is partitioned equally
  - **MEM-OPT**
    - Optimize memory space partition
    - Processors are partitioned equally
  - **PROC-OPT**
    - Optimize processor partition
    - Memory space is partitioned equally
  - **OPT**:
    - Optimize the processors partition
    - Optimize memory space partition
Our proposed OPT scheme can reduce CEC by 26.6% over the BASE scheme on average for 8 different workloads.
Selective Code/Data Migration
Introduction and Motivation

- **Problem**: How to partition a given application into tasks?
  - Minimize inter-processor communication as much as possible
  - Inter-processor data communication requirements mostly known at runtime (hard to detect at compile-time)

- **Idea**: Migrate the task itself, instead of data
  - Performance & Power Benefits

- **Selective migration**: Task/data migration scheme that decides whether to migrate task or data
  - Runtime decision
  - Based on the statistics collected off-line through profiling
  - Depends on the dynamic execution path taken by the application
Architecture

- Multiple processors with local memories
- Communication between two processors occurs through explicit message passing: Owner
  - Reads data from local memory
  - Forwards it to the requester
- When an application code is parallelized for this architecture, each processor is assigned
  - *local code* → *task*
  - *local data* → *data*
  - stored in its *local memory*
- Our approach takes the *code parallelization* as an input
Selective Migration

- Either
  - Use a single data migration (S)
  - Use a task migration (Q) followed by a data migration (Y)
- Decide based on the relative communication costs of S, Q, and Y

\[
P_i \quad S \quad P_j
\]

\[
Y = Q(S)
\]

\[
P_i \quad Q \quad P_j
\]

\[
Y = Q(S)
\]

\[
Y
\]
Selective Migration

- Need to distinguish the *global optimal* from *local optimal(s)*
  - Consider the subsequent communication requirements
  - Future use of task (Q)
  - Future use of data (S)
  - Consider multiple correlated communications together not just the current one
  - Make a decision that is globally optimal
- Target: communication energy reduction
- Compare with
  - Always migrate data
  - Always migrate task
Our Approach

- Major components

Profiling

Compile Time (Offline)
Profiling

- Unit of code migration
  - Function
  - Procedure
- Unit of data migration $\rightarrow K$
- For each application calculate
  - Communication energy cost of transferring each function/procedure over the network
  - Cost of transferring a data block of size $K$ over the network
- Perform this for each pair of processors
- This is a one time cost at compile time
  - An offline process
  - Will be compensated in the long run
Example

- $M_a : Y_1 = Q_1(S_1)$
- $M_b : Y_2 = Q_2(S_1)$
- $M_c : Y_3 = Q_3(S_1)$

$Q_1 + Y_1 + Q_2 + Y_2 < S_1 < Q_1 + Y_1 + Q_2 + Y_2 + Q_3 + Y_3$

Diagram:

- $Q_1, Q_2, Q_3$
- $S_1$
- $P_i$
- $P_j$

$Q_1 = Q_1(S_1)$
$Y_1 = Q_1(S_1)$
$Y_1 = Q_1(S_1)$
$Y_1 = Q_1(S_1)$
$S_1$

$M_a \Delta M_b \rightarrow Y_1 \rightarrow Y_1 \rightarrow S_1$
$Q_1 + Y_1 + Q_2 + Y_2 + Q_3 + Y_3 < S_1$
Experimental Results

- Applications are parallelized by exploiting task parallelism
- Schemes evaluated in this work use the same task parallelization
- Schemes
  - DMO: data migration only
  - TMO: task migration only
  - CDTM: selective task/data migration
Experimental Results

- Interprocessor communication energy
- 19% saving over DMO
- 39% saving over TMO
Workload Clustering
Scenario with no energy saving scheme

Processor Busy

Processor Idle
Energy Reduction Schemes

● There are two primary groups
  ■ Voltage scaling techniques
  ■ Processor shutdown schemes

● They can be applied using hardware or an optimizing compiler

● They are applied independently

● They are applied in disjoint manner
Processor Shutdown

● Saves leakage energy
● Not all processors are used.
● These processors spend energy.
● Shut off unused processors.
  ■ Low power mode
  ■ Another idea
    – Turn off processors by detecting that jobs have finished
    – Turn them on later if necessary
Scenario with processor shutdown

![Diagram showing processor states]

- **P₀**: Processor Busy
- **P₁**: Processor Busy
- **P₂**: Processor Busy
- **P₃**: Processor Busy and Idle
- **P₄**: Processor Busy and Idle
- **P₅**: Processor Idle and Processor Shutdown

Legend:
- Red: Processor Busy
- Green: Processor Idle
- Blue: Processor Shutdown
Voltage Scaling

- **Active Power** $\propto$ Voltage level$^2$
  - Need to reduce voltage level in order to increase energy savings.
  - But how? Scaling!
- **Frequency** $\propto$ Voltage levels
  - Time $\propto 1/$Frequency
  - So, Time $\propto$ Voltage levels

So, scale the voltage down to take advantage of extra available time and reduce dynamic energy consumption! But leakage increases!
Scenario with voltage scaling

- **P_0**: Processor Active
- **P_1**: Processor Active
- **P_2**: Processor Active
- **P_3**: Processor Voltage Scaled
- **P_4**: Processor Voltage Scaled
- **P_5**: Processor Idle
Intuition for a unified approach

- Processor shut down does not try to take advantage of voltage scaling.
- Pure Voltage scaling will not shut off idle processors.
- Job clustering is not being done.
- Hence a unified approach that optimally uses a combination of the two schemes on a per-job/processor basis is needed!
Unified approach

- Cluster jobs on as few processors as possible
  - Increases number of completely idle processors
  - They can be shut down
- Perform voltage scaling of those processors that have remaining slack.
- Question
  - How is it possible to select the optimal voltage level for a particular job?
  - How is it possible to determine the optimal clustering of jobs?
- Answer
  - Integer Linear Programming (ILP)
Scenario with voltage scaling and processor shutdown

- $P_0$: Processor Active
- $P_1$: Processor Voltage Scaled
- $P_2$: Processor Idle
- $P_3$: Processor Idle
- $P_4$: Processor Idle
- $P_5$: Processor Shutdown
Scenario with unified approach including workload clustering

- Processor Active
- Processor Voltage Scaled
- Processor Shutdown
A Linear Program (LP) is a problem that can be expressed as follows:

\[
\text{minimize } cx \\
\text{subject to } Ax = b \\
x \geq 0
\]

- \(x\) is a vector of vector of variables to be solved for.
- \(A\) is a matrix of known coefficients.
- \(c\) and \(b\) are vectors of known coefficients.

*http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html#Q1*
Mathematical programming model

- **Dynamic Energy computation**
  - If a processor $p$ executes a job $j$ at voltage level $v$
    - Add the energy spent in doing so to the overall sum

  \[
  D\_\text{Energy} = \sum_{p} \sum_{j} \sum_{v} X(p, j, v) \times \text{Job\_Energy}(j, v)
  \]

- **Leakage Energy computation**
  - If a processor $p$ is not shutdown, i.e. it is busy
    - It spends leakage energy, add this to the overall sum

  \[
  L\_\text{Energy} = \text{Leakage\_Value} \times \sum_{p} \text{Busy}(p)
  \]
Results - Energy Savings

![Normalized Energy Consumptions](image)

*Figure 1: Normalized energy consumptions.*
Code Replication
Introduction

- Reliability issues in the existence of transient errors are becoming an increasingly critical challenge for embedded designs
- Recent research has underlined the importance of protection mechanisms against soft errors
- Hardware approaches are not suitable for embedded systems
  - Costly mechanisms
  - Multiple factors should be balanced out carefully
  - For optimizing reliability, one needs to consider issues such as performance, power and memory overheads
ECM Architecture

- A shared memory based ECM architecture
  - Each processor has its L\textsubscript{1} instruction and data caches
  - A shared on-chip unified L\textsubscript{2} cache
  - A large off-chip main memory
- Each processor can be operated under different voltage/frequency levels
  - Dynamically switching from one voltage/frequency level to another during execution takes time and energy
  - But these transitions do not occur very frequently
Our Approach

- Strike a balance between the two extreme schemes
- Reduce energy consumption in replicas as much as possible in the case where no errors occur
- At the same time, reduce the time required to recover from an error as much as possible when it occurs
- Execute replicas with scaled down voltage/frequency
- Replicas and the primary copies start executing at the same time
Our Approach

- **When no errors**
  - Determined by the successful termination of the primary copy
  - Terminate the replica
  - Since the replica has operated with lower voltage/frequency so far, we save energy, compared to the case where the replica is executing with the highest voltage/frequency available

- **When an error occurs in the primary copy**
  - Primary copy is aborted
  - Replica is switched to the highest voltage/frequency level to minimize the time to complete the task
Our Approach

Error Free

Transient Error
Our Approach

● X-axis: supply voltage/clock frequency pair under which a replica is run
● Energy-spent-in-replicas (ESR): The amount of energy spent in a replica up until the point the associated primary copy has finished its execution successfully or it has signaled an error and aborted
● Time-to-recovery (TTR): The time it takes to recover
● Region of exploration: Bounds for the acceptable ESR and TTR values
● Our goal is to study the impact of a selected voltage/frequency value on the values of ESR and TTR for a set of embedded applications
Our Approach

![Graph showing energy spent in replicas (ESR) and time-to-recovery (TTR) across different voltage/frequency levels.

Key:
- **ESR**: Energy Spent in Replicas
- **TTR**: Time-To-Recovery

Legend:
- Red circles: ESR
- Blue diamonds: TTR

Voltage/Frequency Levels:
- f1/V1 (highest)
- f2/v2
- f3/v3
- f4/v4
- f5/v5
- f6/v6
- f7/v7
- f8/v8 (lowest)

Region of Exploration:
- The region between f3/v3 and f6/v6, where both ESR and TTR are at their lowest and highest values, respectively.
Compiler’s Role

- Compiler is used to parallelize the given sequential application to be executed on an ECM
  - We parallelize outermost loops of the loop nests in the application
- Compiler creates the replica computations and inserts coordinating code in both replica and primary copy to enable switching between them at runtime
- Interaction between the primary copy and the replica
  - Primary copy should be able to detect whether it fails or succeeds
  - Primary copy should interact with the corresponding replica
Compiler’s Role

- Primary copy detects whether it fails or succeeds by implementing loop invariants
  - We use Daikon, a dynamic invariant detector

Algorithm 1 Matrix Multiplication

1: Input: Matrices A and B
2: Output: Matrix C
3: for $i \leftarrow 1 \ldots N$ do
4:     for $j \leftarrow 1 \ldots N$ do
5:         for $k \leftarrow 1 \ldots N$ do
6:             $C[i][j] = A[i][k] * B[k][j]$
7:         end for
8:     end for
9: end for

Invariants

size(A) = size(B)
size(A) = size(C)
Elements of A, B, and C != null
A, B, and C do not contain any duplicates
address(B[i1][i2])=address(A[i1][i2])+D1
address(C[i1][i2])=address(A[i1][i2])+D2
Loop indices $i, j, k \geq 0$
Experimental Results
CS 426

Memory Hierarchy Design
Memory Hierarchy Design (2/5)

- Memory hierarchy management has been well studied
  - Caches
  - From the performance perspective
- Relatively less attention
  - Software managed memories
  - Optimizing energy behavior
- Software-managed hierarchies can be preferable against hardware counterparts
  - Able to design a customized memory hierarchy that suits the needs of the application
  - Data flow is managed by software
    - Energy-efficient → Dynamic lookup in hardware
A strategy for designing application-specific memory hierarchy for CMPs

- An optimizing compiler
  - Analyzes the application code
  - Detects the data sharing pattern across processors

- An ILP (integer linear programming) solver
  - Determines the sizes of the memory components
  - How these memory components are shared
  - What data each memory component is to hold
Memory Hierarchy Design (4/5)
Example

- An example on-chip memory hierarchy design and data allocation
  (a) After phase 1
  (b) After phase 2

<table>
<thead>
<tr>
<th>Processor</th>
<th>D₁</th>
<th>D₂</th>
<th>D₃</th>
<th>D₄</th>
<th>D₅</th>
<th>D₆</th>
<th>D₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Phase 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Energy Results

- On average: ILP-ML $\rightarrow 21.3\%$, ILP-SL $\rightarrow 12.1\%$, ILP-ML$^*$ $\rightarrow 9.9\%$
Our Approach

- Code compression strategy based on the CFG.
- Initially all basic blocks (BB) are compressed.
- Decompress only the blocks that are predicted to be needed in the near future.
- Decide when to compress a BB.
- Save as much memory space as possible.
- Frequent compressions and decompressions.
  - Performance degradation
  - May be in the critical path.
Our approach works on a control flow graph (CFG).
- Nodes: Basic blocks
- Edges: Control flow (conservative)
Basic Block Compression

- **K-edge algorithm**: To compress an executed basic block when the $k^{th}$ edge following it is traversed.

- How to select a suitable value for the $k$ parameter?
  - Very small $k$
  - Very large $k$

- **Multi-threaded approach**:
  - execution thread and compression thread

Compress $B_1$
Basic Block Decompression

Decompression Strategies

On-Demand Decompression

Pre-decompression

K-edge, Pre-decompress-all

K-edge, Pre-decompress-single
An example CFG fragment that can be optimized using pre-decompression.

Execution thread has just left $B_0$ ($k=2$).
The cooperation between the three threads during execution.
Experimental Evaluation

Memory Space Usage

- Adi
- Cordic
- Mxm
- Rawcaudio
- Tomcatv

Legend:
- On-Demand
- Pre-Single
- Pre-All
CS 426

Optimizations for Memory Banks
Memory Banking

- **Main memory energy consumption**
  - Constitutes a large percentage of overall energy.
  - Especially in data-intensive embedded applications.

- **Banking is a potential solution.**
  - Uniform banks: partition the memory space into equal-sized banks
  - Each bank can be power-managed independently
  - Low-power operating mode support

- **The effectiveness of low-power management increases as the bank idleness increases**
Banked Memory

- Banks are of the same size.
- Banks can be controlled independently.
- Low-power operating modes can be used.
Power Modes

- Each low-power mode has
  - Different energy consumption (per cycle).
  - Resynchronization cost (wake up penalty).
Our Approach

- **Non-uniform banking**
  - Banks can be of different sizes.
  - In embedded systems that execute a single application.
Experimental Evaluation

- Energy improvement brought by the Non-uniform over the Uniform
- The overall average reduction in energy consumption is 10.4%.

![Percentage Reduction in Energy w.r.t different bank sizes](image)