Automatically Mapping Code on an Intelligent Memory Architecture

Jaejin Lee†, Yan Solihin‡, Josep Torrellas‡
†University of Illinois at Urbana-Champaign
‡Michigan State University
http://iacoma.cs.uiuc.edu/flexram
Overview

- Intelligent Memory Architecture
- Related Work
- Goals of This Study
- Mapping Algorithms
- Evaluation
- Conclusions
The Intelligent Memory Architecture

- Processors In Memory (PIM) Architecture Integrates processor logic and DRAM on a single chip.
- Two approaches:
  - IRAM, Shamrock, RAW, Smart Memories: the PIM architecture is main processing unit in the system
  - Active Pages, DIVA, FlexRAM: the PIM chips replace memory chips in the system (Intelligent Memory Architecture)
- The Intelligent Memory Architecture has a heterogeneous mix of processors.
The Intelligent Memory Architecture (cont’d)

- **P.host**: a wide-issue superscalar with a deep cache hierarchy.
- **P.mem**: a simple, narrow-issue superscalar with only a small cache.
- **Compiler controlled cache coherence**
  - P.host writeback dirty lines that might be read by P.mem.
  - P.host invalidates lines that might be written by P.mem.
Related Work

• Many different type of PIM architectures: IRAM, Shamrock, RAW, Smart Memories, Active Pages, DIVA, FlexRAM, etc.

• Previous work for Intelligent Memory architectures largely depend on programmers and focuses on running sections of code on a set of identical memory processors.

• Previous work in exploiting parallelism in a heterogeneous environment (Globus, Legion) has bigger granularity than ours and targets highly-distributed systems.
The goal of this study

• How to automatically program the Intelligent Memory Architecture?
  o A combination of static and run-time algorithms.
  o Partitioning code into smaller sections (*basic partitioning and advanced partitioning*).
  o Mapping the sections onto the best processor (*processor affinity estimation*).
  o Overlapping executions of the sections if possible.
• The algorithms are adaptive to the overheads.
Basic Partitioning

- Finds code sections (basic modules) that are easy to extract and have
  - homogeneous computing and memory behaviors
  - good locality.
- A basic module is a loop nest, where
  - each nesting level has only one loop
  - may span several subroutine levels.
Basic Partitioning (cont’d)

```fortran
N1 = N*2
DO I=1, N1
N2 = X * 4
DO J = 1, N2
X = ...
A(J,I) = ...
ENDDO
IF (X .LT. 1.0) THEN
X = ...
ENDIF
ENDDO
C(N) = ...
DO K = 1, N-1
B(K) = C(K+1)
ENDDO
```

```fortran
N1 = N*2
DO I=1, N1
N2 = X * 4
DO J = 1, N2
X = ...
A(J,I) = ...
ENDDO
IF (X .LT. 1.0) THEN
X = ...
ENDIF
ENDDO
C(N) = ...
DO K = 1, N-1
B(K) = C(K+1)
ENDDO
```
Advanced Partitioning

- Increases the grain size of the module, possibly reducing uniformity resulting in compound modules.
- Repeatedly applying *expansion* and *combining* steps.
- Expansion: similar to the basic partitioning

```plaintext
if P then
  ...
else
  M
endif
```

New module $M'$

```plaintext
if P then
  ...
else
  M
endif
```
Advanced Partitioning (cont’d)

- Combining: two adjacent modules with the same affinity are combined into a new module.

```
if P then
  M1
else
  M2
endif
```

```
if P then
  M1
else
  M2
endif
```

New module $M'$
Advanced Partitioning with Retraction

- Compound modules resulting from advanced partitioning may be very large and invoked very few times (harder run-time adaptation).
- The algorithm selects advanced modules that are expected to be invoked only 1-2 times.
- Peel-off statements until it reaches an all enclosing loop or a set of disjoint loops.
Mapping (Static)

- **Performance model (Delphi tool)**
  - For numerical applications.
  - Execution time $= T_{comp} + T_{memstall}$
    
    $T_{comp} = \max (T_{int}/N_{int}, T_{fp}/N_{fp}, T_{ldst}/N_{ldst}) + T_{other}$
    
    $T_{memstall} = \sum_{i=caches} (Miss\_penalty_i)$
  - Stack distance model for the number of misses.

- **Profiling**
  - For non-numerical applications
  - Gather execution time and the number of invocations for all modules and subroutines.
Mapping (Dynamic)

- Decision runs to determine affinity
- Coarse and CoarseR (adaptive to workload)

Invocation 1 2 3 4 5 •••

Coarse
- P.host
- P.mem

CoarseR
- P.host
- P.mem

- For both basic modules and compound modules
- Fine and FineF are similar to Coarse and CoarseR, but the decision runs are the first two iterations of each invocation (high overhead).
  - Only for basic modules
Overlapping Execution

- With basic partitioning.
- Module-wise parallel region and module-wise serial region.

Module-wise parallel region

- Module 1
- Module 2
- Module 3

P.host

- Module 1
- Module 2

P.mem

- Module 3
Module-wise Serial Region

- Static vs. Dynamic partitioning for overlapping execution to balance the load considering cache write-back and invalidation overheads.

```plaintext
do i=1,100
A(i) = A(i-1)+B(i)
C(i) = A(i)
enddo
```

```plaintext
P.host
```

```plaintext
P.mem
```

```plaintext
Dopipe
```

```plaintext
P.host
do i=1,100
A(i) = A(i-1)+B(i)
if mod(i,4)=0 then
  Wait
  Writeback
  Signal
endif
C(i) = A(i)
Enddo
```

```plaintext
P.mem
do i=1,100
if mod(i+3,4)=0 then
  Wait
endif
C(i) = A(i)
Enddo
```
**Evaluation Environment**

- Evaluated both numerical (by Polaris compiler) and non-numerical (by hand) applications.
- Mint based simulator.

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.host::P.mem</td>
<td>Frequency</td>
<td>800MHz :: 800MHz</td>
</tr>
<tr>
<td></td>
<td>Issue width</td>
<td>Out-of-order 6 :: In-order 2</td>
</tr>
<tr>
<td></td>
<td>Functional Units</td>
<td>4Int+4Fp+2Ld/St :: 2Int+2Fp+1Ld/St</td>
</tr>
<tr>
<td>P.host Caches</td>
<td>L1-Data</td>
<td>Write-through, 32KB, 2-cycle hit</td>
</tr>
<tr>
<td></td>
<td>L2-Data</td>
<td>Write-back, 1MB (512KB for non-numerical apps.), 10-cycle hit</td>
</tr>
<tr>
<td></td>
<td>Write-back overhead</td>
<td>5 + 1 □ num_cache_lines (background)</td>
</tr>
<tr>
<td></td>
<td>Invalidation overhead</td>
<td>5 + 1 □ num_cache_lines</td>
</tr>
<tr>
<td>P.mem Cache</td>
<td>L1-Data</td>
<td>Write-back, 16KB, 2-cycle hit</td>
</tr>
<tr>
<td>Memory and Bus</td>
<td>Memory Latency (cycles)</td>
<td>160 from P.host, 21 from P.mem</td>
</tr>
<tr>
<td></td>
<td>Bus Type</td>
<td>Split transaction, 16-B wide</td>
</tr>
</tbody>
</table>
Average Characteristics of Basic Modules

- Different applications have a different distribution of module affinity.

<table>
<thead>
<tr>
<th></th>
<th>Numerical Applications</th>
<th>Non-numerical Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Averages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Modules</td>
<td>13.2 (99.1%)</td>
<td>41.8 (63.0%)</td>
</tr>
<tr>
<td>Parallel Modules</td>
<td>11.4 (70.9%)</td>
<td>8.8 (1.3%)</td>
</tr>
<tr>
<td>Serial Modules</td>
<td>1.8 (28.2%)</td>
<td>33.0 (61.7%)</td>
</tr>
<tr>
<td>P.host Affinity</td>
<td>4.0 (37.0%)</td>
<td>31.0 (38.9%)</td>
</tr>
<tr>
<td>P.mem Affinity</td>
<td>9.2 (62.1%)</td>
<td>37.8 (38.9%)</td>
</tr>
<tr>
<td><strong>Average Number of</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invocations</td>
<td>442.9</td>
<td>182,177</td>
</tr>
<tr>
<td><strong>Average Module Size</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P.host cycles)</td>
<td>4,570 K</td>
<td>477 K</td>
</tr>
</tbody>
</table>
Overall Speedups

- Our algorithm delivers speedups that are comparable to the ideal speedup.

<table>
<thead>
<tr>
<th>Apps</th>
<th>P.host(alone)/AdvCoarseR</th>
<th>P.host(alone)/OverDyn</th>
<th>Amdahl’s 2 P.hosts</th>
<th>2-processor SGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swim</td>
<td>1.67</td>
<td>2.71</td>
<td>2.00</td>
<td>1.85</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>1.17</td>
<td>1.60</td>
<td>1.67</td>
<td>1.44</td>
</tr>
<tr>
<td>LU</td>
<td>1.26</td>
<td>1.22</td>
<td>1.04</td>
<td>0.99</td>
</tr>
<tr>
<td>TFFT2</td>
<td>1.42</td>
<td>1.22</td>
<td>1.91</td>
<td>0.80</td>
</tr>
<tr>
<td>Mgrid</td>
<td>1.05</td>
<td>1.55</td>
<td>1.94</td>
<td>1.47</td>
</tr>
<tr>
<td>Average</td>
<td>1.31</td>
<td>1.66</td>
<td>1.71</td>
<td>1.31</td>
</tr>
<tr>
<td>Bzip2</td>
<td>1.37</td>
<td>-</td>
<td>1.01</td>
<td>0.99</td>
</tr>
<tr>
<td>Mcf</td>
<td>1.37</td>
<td>-</td>
<td>1.01</td>
<td>1.00</td>
</tr>
<tr>
<td>Go</td>
<td>0.97</td>
<td>-</td>
<td>1.01</td>
<td>0.57</td>
</tr>
<tr>
<td>M88ksim</td>
<td>1.01</td>
<td>-</td>
<td>1.03</td>
<td>1.00</td>
</tr>
<tr>
<td>Average</td>
<td>1.18</td>
<td>-</td>
<td>1.02</td>
<td>0.89</td>
</tr>
</tbody>
</table>
Conclusions

- Different applications have different computing and memory behaviors.
- By using a combination of static and dynamic algorithms, we achieve comparable speedups to the ideal speedup on the 2-host multiprocessor systems.
- A heterogeneous mix of processors can be exploited cost-effectively.
Normalized Execution Time

Swim


Busy Memory Other Idle WB&INV Ideal
The diagram shows a comparison of normalized execution times for different scenarios in the Mgrid benchmark. The x-axis represents various strategies such as `P.host/alone`, `P.mem/alone`, `Ideal`, `Static`, `Coarse`, `CoarseR`, `Fine`, `FineF`, `AdvCoarse`, `AdvCoarseR`, `OverSta`, and `OverDyn`. The y-axis indicates normalized execution time, ranging from 0.0 to 1.1.

Each scenario is color-coded to represent different components: `Busy` (solid), `Memory` (crosshatched), `Other` (light solid), `Idle` (dark solid), `WB&INV` (diagonal dashed), and `Ideal` (white triangle). The visual representation allows for a clear comparison of performance across these strategies.