Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores

Manuel Mohr, Carsten Tradowsky
Non-Cache-Coherent Architectures

Existing hardware coherence protocols hit scalability limit

Radical answer: abandon global cache coherence

Still provide shared memory

Most prominent example: Intel SCC
Non-Cache-Coherent Architectures

Existing hardware coherence protocols hit scalability limit
Radical answer: abandon global cache coherence
Still provide shared memory
Most prominent example: Intel SCC
Non-Cache-Coherent Architectures

Existing hardware coherence protocols hit scalability limit

Radical answer: abandon global cache coherence

Still provide shared memory

Most prominent example: Intel SCC
Non-Cache-Coherent Architectures

Existing hardware coherence protocols hit scalability limit

Radical answer: abandon global cache coherence

Still provide shared memory

Most prominent example: Intel SCC
Non-Cache-Coherent Architectures

- Existing hardware coherence protocols hit scalability limit

Coherence Wall

Radical answer: abandon global cache coherence
Still provide shared memory
Most prominent example: Intel SCC
Non-Cache-Coherent Architectures

- Existing hardware coherence protocols hit scalability limit
- Radical answer: abandon global cache coherence
  - Still provide shared memory
  - Most prominent example: Intel SCC

Image source: Wikipedia, used under CC BY-SA 3.0
Shared-Memory Programming Model
Shared-Memory Programming Model
Shared-Memory Programming Model

Feasible, but can be expensive
Shared-Memory Programming Model

\[ 0 \]

\[ \$1 \quad ? \quad \$0 \]
Shared-Memory Programming Model

Writeback

$1 \rightarrow 0$
Invalidate

Shared-Memory Programming Model
Shared-Memory Programming Model

Read

1

$\ 1$

$\ 1$

Feasible, but can be expensive
Feasible, but can be expensive
Partitioned Global Address Space (PGAS) Model

Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
Partitioned Global Address Space (PGAS) Model

Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.
Partitioned Global Address Space (PGAS) Model

Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.
Partitioned Global Address Space (PGAS) Model

Goal:
Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.
Goal:
Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.
Goal:
Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
Deep Copy

Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
Message Passing (MP)

1. Serialize $G$
2. Transfer $B$ to $B'$, e.g. using library
3. Deserialize $G'$ from $B'$

- Large memory overhead
- Serialization overhead
- $B$, $B'$ pollute caches
Message Passing (MP)

1. Serialize $G$ to $B$

- Large memory overhead
- Serialization overhead
- $B$, $B'$ pollute caches
Message Passing (MP)

1. Serialize $G$ to $B$
2. Transfer $B$ to $B'$, e.g. using library

Large memory overhead
- Serialization overhead
- $B$, $B'$ pollute caches
Message Passing (MP)

1. Serialize $G$ to $B$
2. Transfer $B$ to $B'$, e.g. using library
3. Deserialize $G'$ from $B'$
Message Passing (MP)

1. Serialize $G$ to $B$
2. Transfer $B$ to $B'$, e.g. using library
3. Deserialize $G'$ from $B'$

- Large memory overhead
- Serialization overhead
- $B$, $B'$ pollute caches
Message Passing via Shared Memory (MP-SHM)

1. Serialize $G$
2. Notify yellow core with $B$'s address & size
3. Invalidate $B +$ Just one buffer copy – Serialization overhead – $B$ pollutes cache
Message Passing via Shared Memory (MP-SHM)

1. Serialize $G$ to $B$

2. Notify yellow core with $B$'s address & size

3. Invalidate $B +$ Just one buffer copy – Serialization overhead – $B$ pollutes cache
1. Serialize $G$ to $B$
1. Serialize $G$ to $B$
1. Serialize $G$ to $B$ and write back $B$
1. Serialize $G$ to $B$ and write back $B$
Message Passing via Shared Memory (MP-SHM)

1. Serialize $G$ to $B$ and write back $B$
2. Notify yellow core with $B$’s address & size
Message Passing via Shared Memory (MP-SHM)

1. Serialize $G$ to $B$ and write back $B$
2. Notify yellow core with $B$’s address & size
3. Invalidate $B$
Message Passing via Shared Memory (MP-SHM)

1. Serialize $G$ to $B$ and write back $B$
2. Notify yellow core with $B$’s address & size
3. Invalidate $B$ and deserialize $G'$ from $B$
Message Passing via Shared Memory (MP-SHM)

1. Serialize $G$ to $B$ and write back $B$
2. Notify yellow core with $B$’s address & size
3. Invalidate $B$ and deserialize $G'$ from $B$

- Just one buffer copy
- Serialization overhead
- $B$ pollutes cache
Cloning (CLONE)

1. Traverse $G$ and write back
2. Notify yellow core with $G'$'s root
3. Traverse $G'$, invalidate & clone objects

+ No serialization
+ No temporary buffer
+ More cache-friendly
Cloning (CLONE)

1. Traverse $G$ and write back
2. Notify yellow core with $G$'s root
3. Traverse $G'$, invalidate & clone objects

- No serialization
- No temporary buffer
- More cache-friendly
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
1. Traverse $G$ and write back $o_i$
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
1. Traverse $G$ and write back $o_i$
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
3. Traverse $G$ & clone objects
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
3. Traverse $G$, invalidate & clone objects
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
3. Traverse $G$, invalidate & clone objects
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$'s root
3. Traverse $G$, invalidate & clone objects
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
3. Traverse $G$, invalidate & clone objects
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
3. Traverse $G$, invalidate & clone objects
Cloning (CLONE)

1. Traverse $G$ and write back $o_i$
2. Notify yellow core with $G$’s root
3. Traverse $G$, invalidate & clone objects

- No serialization
- No temporary buffer
- More cache-friendly
list = new LinkedList;
remote_op(list);
list = new LinkedList;
remote_op(list);
list = new LinkedList;
remote_op(list);

- Compiler has full view of types and controls data transfers
list = new LinkedList;
remote_op(list);

- Compiler has full view of types and controls data transfers
  → PGAS languages enable fully-automatic compiler-based implementation of cloning
    - Compiler generates type-specific writeback and invalidate functions
    - No need to modify existing programs
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, invalidate(addr)
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \texttt{invalidate(addr)}

⇒ Software iterates over all relevant addresses

\[
\textbf{for} \ x = S \ \textbf{to} \ E \ \textbf{step} \ \text{CACHE\_LINE\_SIZE} : \\
\text{invalidate}(x)
\]
Invalidation and write-back of **address ranges** \([S, E]\) 

Status quo: operate on individual cache lines, \(\text{invalidate}(\text{addr})\)

⇒ Software iterates over all relevant addresses

\[\text{for } x = S \text{ to } E \text{ step CACHE_LINE_SIZE:} \]
\[\text{invalidate}(x)\]

Why not support this in hardware?
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \texttt{invalidate(addr)}
  
  \[ \Rightarrow \]  
  Software iterates over all relevant addresses
  
  \[
  \text{for } x = S \text{ to } E \text{ step CACHE_LINE_SIZE:}
  \]
  
  \[
  \text{invalidate(x)}
  \]

- Why not support this in hardware?

---

\[
\begin{array}{c|c|c|c}
\text{Tag} & \text{Valid} & \text{Dirty} \\
\hline
0010 & 1 & 0 \\
1011 & 1 & 1 \\
0001 & 1 & 0 \\
1101 & 1 & 1 \\
\end{array}
\]
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \texttt{invalidate(addr)}

⇒ Software iterates over all relevant addresses

\[\text{for } x = S \text{ to } E \text{ step CACHE_LINE_SIZE: invalidate}(x)\]

- Why not support this in hardware?

![Diagram of cache controller and range controller](image)
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```plaintext
for x = S to E step CACHE_LINE_SIZE:
    invalidate(x)
```

- Why not support this in hardware?

![Diagram of cache and range controller](image)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, `invalidate(addr)`
- Software iterates over all relevant addresses
  ```
  for \(x = S\) to \(E\) step CACHE_LINE_SIZE:
  invalidate(x)
  ```
- Why not support this in hardware?

![Diagram of Cache Controller and Range Controller](image)

```plaintext
Cache Controller

CPU

invalidate \([S, E]\)

Range Controller

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, `invalidate(addr)`
  
  \[\Rightarrow\]  
  Software iterates over all relevant addresses
  
  ```
  \textbf{for } \textit{x = S to E step CACHE\_LINE\_SIZE:\ }
  \textit{invalidate(x)}
  ```

- Why not support this in hardware?

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```plaintext
for x = S to E step CACHE_LINE_SIZE:
    invalidate(x)
```

- Why not support this in hardware?

![Diagram](image_url)

**Cache Controller**

**Range Controller**

**Range Buffers**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \(\text{invalidate}(\text{addr})\)

\[\Rightarrow\] Software iterates over all relevant addresses

\[\text{for } x = S \text{ to } E \text{ step CACHE_LINE_SIZE:} \]
\[\text{invalidate}(x)\]

- Why not support this in hardware?

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \(\text{invalidate}(\text{addr})\)
  - Software iterates over all relevant addresses
    \[\text{for } x = S \text{ to } E \text{ step } \text{CACHE\_LINE\_SIZE}: \text{invalidate}(x)\]
- Why not support this in hardware?

![Diagram of Cache and Range Controllers](image)

**Cache Controller**

**Range Controller**

**Range Buffers**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, invalidate(addr)

⇒ Software iterates over all relevant addresses

```plaintext
for x = S to E step CACHE_LINE_SIZE:
    invalidate(x)
```

- Why not support this in hardware?

![Diagram of hardware extension with CPU, Cache Controller, Range Controller, Range Buffers, Tag, Valid, Dirty]
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \(\text{invalidate}(\text{addr})\)

⇒ Software iterates over all relevant addresses

\[
\text{for } x = S \text{ to } E \text{ step CACHE_LINE_SIZE:} \quad \text{invalidate}(x)
\]

- Why not support this in hardware?

---

Diagram:

Image of a diagram showing the interaction between a CPU and cache controller with range buffers. The diagram illustrates the loading of an address range and the subsequent validation and invalidation process.
Evaluation Setup

Hardware
- FPGA prototype of non-cache-coherent many-core architecture
- 3 tiles, each 4 LEON3 cores
- 256 MiB shared DRAM
- Private L1$ per core, shared L2$ per tile
- No cache coherence between tiles
- No hardware-based range operations

Software
- Implemented cloning in compiler for PGAS language X10
- Input: X10 programs from IMSuite
  - 12 graph-based distributed algorithm kernels
## Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MP</th>
<th>MP-SHM</th>
<th>CLONE</th>
<th>CLONE MP</th>
<th>CLONE MP-SHM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>1.30</td>
<td>1.17</td>
<td>1.13</td>
<td>1.15×</td>
<td>1.03×</td>
</tr>
<tr>
<td>DST</td>
<td>9.35</td>
<td>7.94</td>
<td>7.35</td>
<td>1.27×</td>
<td>1.08×</td>
</tr>
<tr>
<td>BY</td>
<td>736.79</td>
<td>677.27</td>
<td>658.39</td>
<td>1.12×</td>
<td>1.03×</td>
</tr>
<tr>
<td>DR</td>
<td>83.22</td>
<td>82.13</td>
<td>80.42</td>
<td>1.03×</td>
<td>1.02×</td>
</tr>
<tr>
<td>DS</td>
<td>50.92</td>
<td>47.24</td>
<td>45.49</td>
<td>1.12×</td>
<td>1.04×</td>
</tr>
<tr>
<td>MIS</td>
<td>1.75</td>
<td>1.60</td>
<td>1.57</td>
<td>1.12×</td>
<td>1.02×</td>
</tr>
<tr>
<td>KC</td>
<td>27.10</td>
<td>25.86</td>
<td>25.84</td>
<td>1.05×</td>
<td>1.00×</td>
</tr>
<tr>
<td>DP</td>
<td>36.59</td>
<td>34.14</td>
<td>32.61</td>
<td>1.12×</td>
<td>1.05×</td>
</tr>
<tr>
<td>HS</td>
<td>43.86</td>
<td>34.81</td>
<td>34.00</td>
<td>1.29×</td>
<td>1.02×</td>
</tr>
<tr>
<td>LCR</td>
<td>14.24</td>
<td>11.92</td>
<td>11.88</td>
<td>1.20×</td>
<td>1.00×</td>
</tr>
<tr>
<td>MST</td>
<td>69.82</td>
<td>62.87</td>
<td>50.70</td>
<td>1.38×</td>
<td>1.24×</td>
</tr>
<tr>
<td>VC</td>
<td>1.60</td>
<td>1.30</td>
<td>1.26</td>
<td>1.27×</td>
<td>1.03×</td>
</tr>
</tbody>
</table>

Geomean: 1.17× 1.05×

- Running times and speedups over serialization-based approaches
## Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MP</th>
<th>MP-SHM</th>
<th>CLONE</th>
<th>CLONE MP</th>
<th>CLONE MP-SHM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>1.30</td>
<td>1.17</td>
<td>1.13</td>
<td>1.15×</td>
<td>1.03×</td>
</tr>
<tr>
<td>DST</td>
<td>9.35</td>
<td>7.94</td>
<td>7.35</td>
<td>1.27×</td>
<td>1.08×</td>
</tr>
<tr>
<td>BY</td>
<td>736.79</td>
<td>677.27</td>
<td>658.39</td>
<td>1.12×</td>
<td>1.03×</td>
</tr>
<tr>
<td>DR</td>
<td>83.22</td>
<td>82.13</td>
<td>80.42</td>
<td>1.03×</td>
<td>1.02×</td>
</tr>
<tr>
<td>DS</td>
<td>50.92</td>
<td>47.24</td>
<td>45.49</td>
<td>1.12×</td>
<td>1.04×</td>
</tr>
<tr>
<td>MIS</td>
<td>1.75</td>
<td>1.60</td>
<td>1.57</td>
<td>1.12×</td>
<td>1.02×</td>
</tr>
<tr>
<td>KC</td>
<td>27.10</td>
<td>25.86</td>
<td>25.84</td>
<td>1.05×</td>
<td>1.00×</td>
</tr>
<tr>
<td>DP</td>
<td>36.59</td>
<td>34.14</td>
<td>32.61</td>
<td>1.12×</td>
<td>1.05×</td>
</tr>
<tr>
<td>HS</td>
<td>43.86</td>
<td>34.81</td>
<td>34.00</td>
<td>1.29×</td>
<td>1.02×</td>
</tr>
<tr>
<td>LCR</td>
<td>14.24</td>
<td>11.92</td>
<td>11.88</td>
<td>1.20×</td>
<td>1.00×</td>
</tr>
<tr>
<td>MST</td>
<td>69.82</td>
<td>62.87</td>
<td>50.70</td>
<td>1.38×</td>
<td>1.24×</td>
</tr>
<tr>
<td>VC</td>
<td>1.60</td>
<td>1.30</td>
<td>1.26</td>
<td>1.27×</td>
<td>1.03×</td>
</tr>
</tbody>
</table>

**Geomean**

1.17×  1.05×

- Running times and speedups over serialization-based approaches
- Universal improvement by CLONE
- Speedups depend on structure of transferred data
Non-blocking range operations

- FPGA-based implementation based on LEON3 cache controller
- One range buffer
- Overhead compared to unmodified cache controller:

<table>
<thead>
<tr>
<th>Component</th>
<th>Absolute</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>1489</td>
<td>15.2%</td>
</tr>
<tr>
<td>Register</td>
<td>623</td>
<td>14.6%</td>
</tr>
<tr>
<td>LUT</td>
<td>1491</td>
<td>15.0%</td>
</tr>
<tr>
<td>BRAM</td>
<td>1</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

Data from benchmarks: 17 cache lines on average

Enough non-memory instructions to cover latency of range operations

⇒ Expected to take one cycle from view of CPU
Non-blocking range operations

- FPGA-based implementation based on LEON3 cache controller
- One range buffer
- Overhead compared to unmodified cache controller:

<table>
<thead>
<tr>
<th></th>
<th>Absolute</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>1489</td>
<td>15.2%</td>
</tr>
<tr>
<td>Register</td>
<td>623</td>
<td>14.6%</td>
</tr>
<tr>
<td>LUT</td>
<td>1491</td>
<td>15.0%</td>
</tr>
<tr>
<td>BRAM</td>
<td>1</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- Data from benchmarks: 17 cache lines on average
- Enough non-memory instructions to cover latency of range operations
  ⇒ Expected to take one cycle from view of CPU
Summary

Partitioned Global Address Space (PGAS) Model

Deep Copy

Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.
Summary

Partitioned Global Address Space (PGAS) Model

Goal: Efficiently deep copy pointed data structures between shared memory partitions on non-cache-coherent architectures

Cloning (CLONE)

1. Traverse G and write back $o_1$
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

Evaluation Setup

Hardware
- FPGA prototype of non-cache-coherent many-core architecture
- 3 tiles, each 4 LEON3 cores
- 256 MiB shared DRAM
- Private L1$ per core, shared L2$ per tile
- No cache coherence between tiles
- No hardware-based range operations

Software
- Implemented cloning in compiler for PGAS language X10
- Input: X10 programs from IMSuite
- 12 graph-based distributed algorithm kernels

Hardware Extension: Range Operations

Invalidation and write-back of address ranges $[S, E)$

Status quo: operate on individual cache lines, invalidate(addr)$

⇒ Software iterates over all relevant addresses for $x = S$ to $E$ step CACHE_LINE_SIZE: invalidate($x$)

Why not support this in hardware?

Cache Controller

Range Controller

I
S
E
Range Buffers
add r1, r2

Interesting point in the design space of non-cache-coherent systems

PGAS model exposes existence of multiple coherence domains

Compiler accelerates implicit data transfers via shared memory

Benefits from hardware support for range operations
**Summary**

**Partitioned Global Address Space (PGAS) Model**

**Goal:** Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures

**Cloning (CLONE)**

1. Traverse \( G \) and write back \( o_i \)
2. Notify yellow core with \( G \)'s root
3. Traverse \( G \), invalidate & clone objects

+ No serialization
+ No temporary buffer
+ More cache-friendly

**Evaluation Setup**

**Hardware**
- FPGA prototype of non-cache-coherent many-core architecture
- 3 tiles, each 4 LEON3 cores
- 256 MiB shared DRAM
- Private L1$ per core, shared L2$ per tile
- No cache coherence between tiles
- No hardware-based range operations

**Software**
- Implemented cloning in compiler for PGAS language X10
- Input: X10 programs from IMSuite
- 12 graph-based distributed algorithm kernels
Summary

**Partitioned Global Address Space (PGAS) Model**

- **Goal:** Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures.

**Cloning (CLONE)**

1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

**Evaluation Setup**

- **Hardware**
  - FPGA prototype of non-cache-coherent many-core architecture
  - 3 tiles, each 4 LEON3 cores
  - 256 MiB shared DRAM
  - Private L1$ per core, shared L2$ per tile
  - No cache coherence between tiles
  - No hardware-based range operations

- **Software**
  - Implemented cloning in compiler for PGAS language X10
  - Input: X10 programs from IMSuite
  - 12 graph-based distributed algorithm kernels

**Hardware Extension: Range Operations**

- Invalidation and write-back of address ranges \([S, E]\)
- Status quo: operate on individual cache lines, invalidate(addr)
- Software iterates over all relevant addresses
  
  ```
  for x = S to E step CACHE_LINE_SIZE:
      invalidate(x)
  ```

- Why not support this in hardware?

**Interesting point in the design space of non-cache-coherent systems**

- PGAS model exposes existence of multiple coherence domains
- Compiler accelerates implicit data transfers via shared memory
- Benefits from hardware support for range operations
Summary

**Partitioned Global Address Space (PGAS) Model**

- **Goal:** Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures

**Cloning (CLONE)**

1. Traverse G and write back o1
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

**Evaluation Setup**

**Hardware**
- FPGA prototype of non-cache-coherent many-core architecture
- 3 tiles, each 4 LEON3 cores
- 256 MiB shared DRAM
- Private L1$ per core, shared L2$ per tile
- No cache coherence between tiles
- No hardware-based range operations

**Software**
- Implemented cloning in compiler for PGAS language X10
- Input: X10 programs from IMSuite
- 12 graph-based distributed algorithm kernels

**Hardware Extension: Range Operations**

- Invalidation and write-back of **address ranges** \([S, E]\)
- Status quo: operate on individual cache lines, \(\text{invalidate}(\text{addr})\)
- Software iterates over all relevant addresses
  
  \[
  \text{for } x = S \text{ to } E \text{ step CACHE_LINE_SIZE; }
  \]
  
  \(\text{invalidate}(x)\)
- Why not support this in hardware?

Interesting point in the design space of non-cache-coherent systems

- PGAS model exposes existence of multiple coherence domains
- Compiler accelerates implicit data transfers via shared memory
- Benefits from hardware support for range operations
Backup
Why benchmarks without hardware extension?

- No conceptual obstacles
  - Implementation technique generally applicable
- Prototype platform has two-level cache hierarchy
- Range operations must work on both levels
- Caches very different
  ⇒ Redundant work to show feasibility of concept
- Definitely planned for future work