LMC: Automatic Resource-Aware Program-Optimized Memory Partitioning

Hsin-Jung Yang†, Kermin E. Fleming‡, Michael Adler‡, Felix Winterstein§, and Joel Emer†

† Massachusetts Institute of Technology,
‡ Intel Corporation, § Imperial College London,

February 22nd, FPGA 2016
Motivation

Moore’s Law continues

- More transistors & memory controllers on modern FPGAs
  - Example: Xilinx VC709: two 4GB DDR3 memories
    Nallatech 510T: eight 4GB DDR4 memories + 2GB HMC
    Xeon + FPGA: three memory channels

It is difficult to fully utilize DRAM bandwidth

- Co-optimizing application cores and memory systems
- Porting an existing design to a new platform
  - Smaller FPGA -> Larger FPGA
  - Single FPGA -> Multiple FPGAs

Goal: Automatically optimizing the memory system to efficiently utilize the increased DRAM bandwidth
Utilizing Multiple DRAMs

• How to connect computational engines to DRAMs in order to maximize program performance?
  – Network topology: latency, bandwidth
  – On-chip caching
  – Area constraints

High design complexity!
Utilizing Multiple DRAMs

• How to connect computational engines to DRAMs in order to maximize program performance?
  – High design complexity: network, caching...

• Applications have different memory behavior

Need more bandwidth!
Utilizing Multiple DRAMs

• How to connect computational engines to DRAMs in order to maximize program performance?
  – High design complexity: network, caching...

• Applications have different memory behavior

Need more bandwidth!
Automatic Construction of Program-Optimized Memories

• A clearly-defined, generic memory abstraction
  – Separate the user program from the memory system implementation

• Program introspection
  – To understand programs’ memory behavior

• A resource-aware, feedback-driven memory compiler
  – Use introspection results as feedback to automatically construct the “best” memory system for the target program and platform
Abstraction

• Abstraction hides implementation details and provides good programmability

- Hardware can be optimized for the target application and platform
LEAP Memory Abstraction

LEAP Memory Block
- Simple memory interface
- Arbitrary data size
- Private address space
- “Unlimited” storage
- Automatic caching

interface MEM_IFC#(type t_ADDR, type t_DATA)
  method void readReq(t_ADDR addr);
  method void write(t_ADDR addr, t_DATA din);
  method t_DATA readResp();
endinterface
LEAP Private Memory

LEAP Memory with Multiple DRAMs

- **Naïve solution:** unified memory with multiple DRAM banks
LEAP Memory with Multiple DRAMs

- **Naïve solution**: unified memory with multiple DRAM banks

**Simplicity**
- More capacity
- Higher Bandwidth

**Difficulty**: Performance is limited
- Serialized requests
- Long latency for large rings

Can we do better?
LEAP Memory with Multiple DRAMs

- Distributed central caches and memory controllers
Private Cache Network Partitioning

- Program introspection
  - To understand programs’ memory behavior

Statistics file
Client A: 100
Client B: 10
Client C: 50
Client D: 20

Statistics Counter
Ex: # Cache Misses
# Outstanding Requests
Queueing Delays
Private Cache Network Partitioning

- Case 1: Memory clients with homogeneous behavior
Case 2: Memory clients with heterogeneous behavior

Private Cache Network Partitioning

Need more bandwidth!

Traffic:

100   10   50   20

Load-Balanced Partitioning

FPGA

Host

Host Memory
Private Cache Network Partitioning

- **Case 2: Memory clients with heterogeneous behavior**
  - Load-balanced partitioning
  - Classical minimum makespan scheduling problem

\[
x_{i,j} = \begin{cases} 
1 & \text{if client } j \text{ is mapped to controller } i \\
0 & \text{otherwise}
\end{cases}
\]

**ILP formulation:**

\[
\begin{align*}
& \text{minimize } t \\
& \text{s.t. } \sum_{j=1}^{n} x_{i,j} t_j \leq t, \quad i = 1, \ldots, m \\
& \sum_{i=1}^{m} x_{i,j} = 1, \quad j = 1, \ldots, n \\
& x_{i,j} \in \{0,1\}, \quad i = 1, \ldots, m, \quad j = 1, \ldots, n
\end{align*}
\]

**Approximation:**

Longest Processing Time (LPT) Algorithm
Private Cache Network Partitioning

- Case 3: Fractional load-balancing

\[
\begin{align*}
\text{minimize} & \quad t \\
\text{s.t.} & \quad \sum_{j=1}^{n} x_{i,j}t_j \leq t \\
& \quad \sum_{i=1}^{m} x_{i,j} = 1 \\
& \quad 0 \leq x_{i,j} \leq 1
\end{align*}
\]
LEAP Memory Compiler

- **Three-phase feedback-driven compilation**
  - **Instrumentation (optional):** to collect runtime information about the way the program uses memory
  - **Analysis:** to analyze the program properties and decide an optimized memory hierarchy
  - **Synthesis:** to implement the program-optimized memory
LEAP Memory Performance

• Baseline
LEAP Memory Performance

- Memory interleaving

![3D Graph showing memory performance](chart)
Case Study: Cryptosorter

- **Cryptosorter**: each sorter uses a LEAP private memory
Case Study: Filtering Algorithm

- Filtering algorithm for K-means clustering (HLS kernel)
  - 8 partitions: each uses 3 LEAP private memories
Coherent Cache Network Partitioning

- Baseline coherent memory

![Diagram of Coherent Cache Network Partitioning](image-url)
Coherent Cache Network Partitioning

• Coherent memory interleaving

Private cache network optimization can be combined.
Case Study: Heat Transfer

- **Heat transfer**: 16 engines, 1024x1024 frame

![Normalized Runtime Graph]

- **57% (96%) performance gain**

- **Private memory optimization only**
- **Private + coherent memory optimization**
Moving to Multi-FPGA Platforms

[Diagram showing a high-level overview of a multi-FPGA platform, with interconnected components like clients, private memory clients, central cache controllers, and local memory (DRAM banks).]
Performance on Dual FPGAs

Normalized Runtime

- Single VC707
- Dual VC707
- Single VC709
- Dual VC709

Merge Sort

4 Sorters

8 Sorters
Conclusion

• We introduce the LEAP memory compiler that can transparently optimize the memory system for a given application.

• The compiler automatically partitions both private and coherent memory networks to efficiently utilize the increased DRAM bandwidth on modern FPGAs.

• Future work:
  – More case studies on asymmetric memory clients
  – More complex memory network topologies
  – Dynamic cache partitioning
Thank You