Scavenger:
Automating the Construction of Application-Optimized Memory Hierarchies

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

† Massachusetts Institute of Technology, ‡ Intel Corporation
§ European Space Agency, *NVIDIA Research

September 3rd, FPL 2015
Abstraction

- Abstraction hides implementation details and provides good programmability
Abstraction

- Abstraction hides implementation details and provides good programmability
Abstraction

- Abstraction hides implementation details and provides good programmability.

- Hardware is optimized for a set of applications and fixed at design time.
Abstraction

- Abstraction hides implementation details and provides good programmability

- Hardware is optimized for a set of applications and fixed at design time
Abstraction

- Abstraction hides implementation details and provides good programmability

- Hardware is optimized for a set of applications and fixed at design time

- Implementation details are handled by programmers
- Hardware can be optimized for the target application
Abstraction

- Abstraction hides implementation details and provides good programmability

- Hardware is optimized for a set of applications and fixed at design time
Abstraction

• Abstraction hides implementation details and provides good programmability

• Hardware is optimized for a set of applications and fixed at design time
Abstraction

- Abstraction hides implementation details and provides good programmability

- Hardware is optimized for a set of applications and fixed at design time
Abstraction

- Abstraction hides implementation details and provides good programmability

- Hardware is optimized for a set of applications and fixed at design time

- Platform hardware can be optimized for the target application
Application-Optimized Memory Subsystems

- Goal: build the “best” memory subsystem for a given application
  - What is the “best”?  
    - The memory subsystem which minimizes the execution time
  - How?  
    - A clean memory abstraction
    - A rich set of memory building blocks
    - Intelligent algorithms to analyze programs and automatically compose memory hierarchies
Observation

• Many FPGA programs do not consume all the available block RAMs (BRAMs)
  – Design difficulty
  – Same program ported from smaller FPGAs to larger ones
Observation

• Many FPGA programs do not consume all the available block RAMs (BRAMs)
  – Design difficulty
  – Same program ported from smaller FPGAs to larger ones

Goal: Utilizing spare BRAMs to improve program performance
LEAP Memory Abstraction

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

• Highly parametric
  – Cache capacity
  – Cache associativity
  – Cache word size
  – Number of cache ports

• Enable specific features/optimizations only when necessary
  – Private/coherent caches for private/shared memory
  – Prefetching
  – Cache hierarchy topology
Many FPGA programs do not consume all the BRAMs

**Goal:** utilize all spare BRAMs in LEAP memory hierarchy

**Problem:** need to build very large caches
Cache Scalability Issue

- Simply scaling up BRAM-based structures may have a negative impact on operating frequency
  - BRAMs are distributed across chip, increasing wire delay
Cache Scalability Issue

- **Solution**: trade latency for frequency
  - Multi-banked BRAM structure
  - Pipelining relieves timing pressure
Cache Scalability Issue

- **Solution**: trade latency for frequency
Banked Cache Overhead

- Simple kernel (hit rate=100%)
Banked Cache Overhead

- Simple kernel (hit rate=69%)
Results: Scaling Private Caches

- **Case study: Merger (an HLS kernel)**
  Merger has 4 partitions: each connects to a LEAP scratchpad and forms a sorted linked list from a stream of random values.
Private or Shared Cache?

• We can now build large caches
• Where should we allocate spare BRAMs?
  – Option1: Large private caches
  – Option2: A large shared cache at the next level
• Many applications have multiple memory clients
  – Different working set sizes and runtime memory footprints
Adding a Shared Cache

- Private Scratchpad
  - Scalable Cache

- Scratchpad Master Node

- Scratchpad Controller

- Central Cache (DRAM)

- Host Memory

- FPGA

- Host
Adding a Shared Cache

- **Private Scratchpad**
  - Scalable Cache
- **Scratchpad Master Node**
- **Scratchpad Controller**
- **Shared On-Chip Cache**
- **Central Cache (DRAM)**
- **Host Memory**

Connections:
- **Resp** from Scratchpad Master Node
- **Req** to Scratchpad Master Node

Roles:
- **FPGA**
- **Host**
Adding a Shared Cache

- Private Scratchpad
  - Scalable Cache
- Central Cache (DRAM)
- FPGA
- Host Memory
- Scratchpad Controller
- Shared On-Chip Cache
- Host

Consume all extra BRAMs
Automated Optimization

Pre-build database

User Kernel Generation
(Bluespec, Verilog, HLS kernel)

LEAP Platform Construction

BRAM Usage Estimation

Shared Cache Construction

FPGA Tool Chain

User frequency, memory demands
(ex: cache capacity)
Results: Shared Cache

• **Case study: Filter (an HLS kernel)**
  – Filtering algorithm for K-means clustering
  – 8 partitions: each uses 3 LEAP Scratchpads
Conclusion

• It is possible to exploit unused resources to construct memory systems that accelerate the user program.
• We propose microarchitecture changes for large on-chip caches to run at high frequency.
• We make some steps toward automating the construction of memory hierarchies based on program resource utilization and frequency requirements.
• Future work:
  – Program analysis
  – Energy study
Thank You