Continuous Runahead

Transparent Hardware Acceleration for Memory Intensive Workloads

Authors: M. Hashemi, O. Mutlu, Y. N. Patt
Presented at MICRO 2016
Problem Statement

- For various applications we would like to process large amounts of data.
- Frequent memory accesses lead to a lot of wait time.
- Runahead techniques want to reduce this wait time by prefetching and executing memory requests during wait time.
Quick Summary

Continuous Runahead explores a method to prefetch and execute instructions while a program is running to generate cache misses and subsequent memory loads. This leads to fewer cache misses while a program is executed and therefore to lower wait times on memory.
Overview

- Runahead Execution
- Continuous Runahead
  - Choosing and Storing Dependence Chains
  - CRE
- Performance evaluations
- Critic
- Discussion
RUNAHEAD EXECUTION
Runahead Execution

- What is Runahead Execution?
- Prefetching methods
  - Stream prefetcher
  - Global History buffer
- Current Limitations of Runahead Execution
Runahead Execution

- Memory accesses can cause full pipeline stalls
- Stalls use around 50% of execution time of a program
- Runahead uses instruction window to fetch and execute upcoming instructions

Rightarrow Fewer cache misses
Stream Prefetcher

- Defines stream of cache misses by looking at addresses close in memory
- Looks only in a defined direction
- Prefetches blocks of memory in said direction

More in “Memory Prefetching using Adaptive Stream Detection” by I. Hur and C. Lin
Global History Buffer

- Holds most recent miss addresses in FIFO order
- Ordered table allows to discard unused data
- Complete picture of cache miss history
- Small sized table


https://www.eecg.utoronto.ca/~steffan/carg/readings/ghb.pdf
Limitations of Prefetching

- Short duration of full-window stall
- Prioritisation of memory accesses
CONTINUOUS RUNAHEAD
Key Ideas

- Dynamically filter incoming dependence chains
  - Filter dependence chains generating memory accesses
- Execute dependence chains in a loop
- Loop executed on the Continuous Runahead Engine (CRE)
DEFINITIONS
Dependence Chain

- Set of dependent instructions leading up to a key instruction
- Generated by backtracking the data flow

Example of a dependence chain:
Computing the address for a memory access

```plaintext
ADD R5 + 1 -> R3

SHIFT R3 -> R4

ADD R4 + R3 -> R2

SHIFT R2 -> R1

MEM_LD [R1] -> R0
```
Full-Window Stall

- Instructions are retired in program order
- Long-latency instructions can block pipeline
- Instruction window is filled with incoming instructions
- Both instruction window is blocked and pipeline stalled is called full-window stall
IMPLEMENTATION
Dependence Chain Selection

- **Base Policy**
  - Select next memory access in buffer

- **PC based Policy**
  - Lists all PCs that caused LLC misses
  - Dependent on operation which is blocking retirement

- **Maximum-Misses Policy**
  - Finds and selects PC causing most cache misses

- **Stall Policy**
  - Tracks PCs causing full-window stalls
  - Selects chain causing most full-window stalls
Evaluation of the Policies

- Evaluation of the policies on a single core system using Runahead
- Using policies tracking most misses gives improved performance on most workloads

Comparisons of the policies
Selecting Instructions

- Small amount of instructions cause over 90% of full window stalls

Only a handful instructions need to be looped to be effective
Continuous Runahead Engine

- Strongly based on an enhanced memory controller
  See paper “Accelerating Dependent Cache Misses with an Enhanced Memory Controller” by M. Hashemi et al.

- Sits on the memory controller to reduce latency on memory loads
Architecture of the CRE

- 32-uop buffer to hold full dependence chains
- 32-entry physical register
- 4kB cache with 32-entry TLB
Handling Dependence Chains

- Upon generation TLB sends required load to the CRE
- TLB misses are sent to core of the CPU to resolve
- Dependence chains are continuously executed
- The running dependence chain is relaced every full-window stall
PERFORMANCE EVALUATION
Simulation Environment

- Execution-driven, cycle-level x86 simulator
- Single core system with
  - 256-entry reorder buffer
  - 32KB of instruction/data cache
  - 1MB LLC
- Combined with
  - GBH prefetcher
  - Stream prefetcher
RESULTS
CRE alone

- 34.4% performance gain over the no-prefetching baseline
- 11.9% performance gain over GHB prefetcher
CRE + GHB Prefetching

- 36.4% performance gain over the no-prefetching baseline
- 11.9% performance gain over GHB prefetcher
Memory Bandwidth Consumption

- Increased memory bandwidth consumption for stream prefetching and GHB on some applications
- Overhead drastically reduced with CRE
Points to take Home

- Solves limit on runahead distance by
  - Dynamically identifying critical dependence chains
  - Executing these in a loop
- Cheap and low-complexity hardware solution
- Significant performance gain on a variety of workloads
Formal Critique

- Positives
  - Written in an understandable way
  - Well structured

- Negatives
  - Relying heavily on the readers understanding of specific previous work
Positives regarding Content

- New idea on handling the specified problem
- Efficient solution using few additional resources
- Exploring variety of ways to combine previous solutions with described solution
Negatives regarding Content

- Potentially few workloads profiting from this
- Potential negative side effects caused by placing a CRE on the memory controller not explored
- Solution only for independent cache misses
QUESTIONS
DISCUSSION
Topics

- Alternatives for Implementation
- Could/Should we implement this in general purpose computers
- Performance on Multicore Systems
- Energy consumption
Alternatives for Implementation

- What do we need to be able to
- Is the CRE the only way to implement Continuous Runahead?
  - Simulations multi threading
  - Idle cores
Performance on Multicore Systems

Figure 17: Heterogeneous workload performance.

![Heterogeneous workload performance chart]

Figure 18: Homogeneous workload performance.

![Homogeneous workload performance chart]