ECE 576 – Engineering of Computer-Based Systems

• Profiling
  - Overview of Profiling Approaches
  - Static vs. Dynamic Profiling
  - ProMem – Dynamic Profiler
  - Non-intrusive dynamic profiler for frequent loop detection

Introduction

Hardware/Software Partitioning

- Profiling
  - Identify critical kernels within application
  - Gather accurate execution statistics
  - Application/critical kernels

Introduction

Our Solution: Add On-Chip Profiler Memory to Monitored Bus
- Accepts 1 pattern/cycle
- Keeps exact counts

Goal: Determine # of times each target pattern appears on the bus

Profiling Can Be Used to Solve Many Problems

- Optimization of frequently executed subroutines
- Mapping frequently executed code and data to non-interfering cache regions
- Synthesis of optimized hardware for common cases
- Identifying frequent loops to map to a small low-power loop cache
- Many Others!

Problem Definition

- Objective
  - Count number of times each target pattern appears on bus B
- Requirements
  - Accept input patterns on every clock cycle
  - Monitoring any bus, e.g., deeply embedded buses in SoCs
  - Non-intrusive
  - Exact target pattern count

Input Patterns

Bus B

Target Patterns

Target Pattern Counts

TP = (tp1, ..., tp m)

CTP = (ctp1, ..., ctp m)
Profiling Techniques - Software

- Instrumenting Software
  - Adding code to count frequencies of desired code regions
- Problems
  - Incurs runtime overhead
  - Possibly changes program behavior
  - Increase in code size

for ( ... )
{
  ... 
  ctpm += 1; 
}

Profiling Techniques - Software

- Periodic Sampling
  - Interrupt processor at periodic interval
  - Read program counter and other internal registers
- Problems
  - Disruption of runtime behavior during interrupt
  - Inaccurate

// ISR period = 10ms
ISR()
{
  //update profile info
}

Profiling Techniques - Software

- Simulation
  - Execute application on instruction set simulator
  - Simulator keeps track of profile information
- Problems
  - Difficult to model external environment which leads to inaccuracy
  - Extremely slow

Profiling Techniques - Hardware

- Logic Analyzer
  - Probes placed directly on bus to be monitored
- Problems
  - Cannot monitor embedded buses

Profiling Techniques - Hardware

- Processor Support
  - Mainly event counters
  - Monitored events include cache misses, pipeline stalls, etc.
- Problems
  - Few registers available
  - Reconfiguration needed to obtain a complete profile
  - Leads to inaccuracy

Profiling Techniques - Hardware

- Content-addressable memories (CAMs)
  - Fast search for a key in a large data set
  - Returns the address at which the key resides in a memory
- Types
  - Fully Associative
  - RAM coupled with a smart controller
Profiling Techniques - Hardware

- Fully Associative CAMs
  - Simultaneously compares every location with the key
- Problems
  - Does not scale well to larger memories
  - Increased access time as CAM size grows
  - Large Power Consumption

Profiling Techniques - Hardware

- RAM coupled with a smart controller
  - Efficient lookup data structure in memory such as a binary tree or Patricia Trie
- Problems
  - Multiple cycle lookup

Observations

- Not necessary to have 1 cycle look up
- Only need to accept one input pattern every cycle

Queueing

- Hold input patterns in queue until we are able to process them
- Problems
  - Does not work with patterns arriving every clock cycle

Pipelining

- Implemented in processors such that instructions can be executed every cycle
- Can we use pipelining to solve our problem?

Pipelined CAM

- Large CAMs required long access times
- Partition large CAM into several smaller CAMs
  - Requires pipelining to reduce access time
  - Provides solution to access time problem
  - Requires Large Area
  - Large Power Consumption
**Pipelined CAM**
- Entries can be stored in a CAM in any order
  - requires sequential lookup in pipelined CAM approach
- Is there a benefit to sorting the entries?
  - not necessary to search all entries
  - leads to faster lookup time
- Tree structure provides a inherently sorted structure
  - Search time remains a problem
  - Can we pipeline the structure?

**Pipelined Tree**
- Solves access time problem
  - One memory access per level
- Solves area problem
  - Single comparator per level
  - Each level grows by factor of two
  - For large memories, comparators are negligible

**Pipelined Binary Search Tree**
- Left child > Parent
- Right child < Parent

**Searching for Input Pattern: f**
- e < h, append 0 to address
- e > d, append 1 to address
- e < f, append 0 to address
- e = e, Found!

**Searching for Input Pattern: e, f**
- e < h, append 0 to address
- e > d, append 1 to address
- e < f, append 0 to address
- e = e, Found!
Pipelined Binary Search Tree

Stage 0

Stage 1

Stage 2

Stage 3

Standard Memories

011 010 001 000 011 010 001 000

ProMem – Module Design

Input Pattern
Search Address
Enable

Search Address (Next Stage)
Enable (Next Stage)

Target Pattern Not Found – Enable Next Stage

Target Pattern Found

Compare =

Target Pattern Found - Update Count Value

When Target Pattern Found - Update Count Value
ProMem - Module Design

- **Pipeline Register**
- **Memories**
- **Module Controller**

ProMem - Interface

- **Simple Interface**
  - Internal interface
    - Enable signal
    - Connection to monitored bus
  - External interface
    - Read enable
    - Write enable
    - Connection to ProMem pattern input bus

ProMem - Layout

- **Efficient Layout**
  - Achieved by simply abutting each module with the next
  - Results in very short bus wires between each module

ProMem Results - Area*

- Module overhead only 1%
- Area obtained using UMC .18 technology library provided by Artisan Components

ProMem Results - vs. CAM

- CAM design is 46% larger than ProMem

ProMem Results - Timing vs. CAM

- CAM access time grows with CAM size
- ProMem access time remains constant (Due to Pipelining)
System Optimizations - Dynamic

- There are many dynamic optimization approaches
  - Dynamo performs dynamic software optimizations on the most frequently executed regions of code
  - Frequently executed regions of code can be remapped to non-interfering cache locations
  - Dynamic binary translation methods store translation results of frequently executed regions of code for quick look-up
  - Value profiling can determine runtime invariant variables for constant propagation and/or code specialization
  - Many others...

Dynamic Optimizations - Effectiveness

- For dynamic optimizations to be most effective, optimizations are typically applied to the most frequently executed regions of code
- For a large selection of the MediaBench benchmark suite, we observed that 90% of the execution time was spent in approximately 10% of the code
- Profiling is used to determine the critical regions of code

Previous Profiling Methods

- Desktop targeted profiling methods
  - Instrumentation and sampling
  - These methods are unsuitable for embedded systems
  - Causes disruption of run-time behavior

- Early methods used logic analyzers
- JTAG standard allows for internal registers to be read
- Typically used for testing and debugging
- Interrupts processor to write internal information to external pins

Profiling Methodology Goal

- The goal of our profiling approach is to design a profiling tool suitable for embedded systems to determine the most critical regions of code
Critical Region Detection - Operational Requirements

- Non-intrusion
  - Important for real-time systems
  - Minimizes the impact on current tool chains i.e. no special compilers or binary modification tools
- Low power
  - Battery operated systems
  - Systems with limited cooling
- Small area
  - Less significant due to the large transistor capacities of current and future chips
- Accuracy
  - Exact results are not required for the information to be useful -- instead, reasonable accuracy is acceptable

Frequent Loop Detection

- We analyzed the critical regions for various Powerstone and Mediabench benchmarks

All Critical Regions

- 15% - Subroutines with no inner loops
- 85% - Small inner loops

- We translate the problem of finding the critical regions to finding the frequently executed loops
- Short backwards branch (sbb) instruction is typically the last instruction of a loop

Percentage of Execution Time for Frequent Loops

- In addition to detection of frequent loops we also want to know the loops' percentage contribution to total execution time.

Application X

- Loop A - 10%
- Loop B - 10%
- Loop C - 80%

Application Y

- Loop A - 32%
- Loop B - 32%
- Loop C - 35%

Optimization of most frequent loop only = 80% for X and 35% for Y

Optimization of all critical loops = 80% for X and 100% for Y

Frequent Loop Detection - Cache Based Architecture

Cache Operation

Cache Operation - Conflict Resolution

- Resolve most conflicts using associativity and an LRU replacement policy for further conflicts
- Further conflicts may cause frequent loops to constantly be replaced in the cache - thrashing
- Our experiments did not suffer from this contention but a victim buffer may be added if necessary
Cache Operation - Frequency Width

- Our goal is to find the smallest possible cache needed to determine the frequent loops.
- We keep the cache small by allowing the frequency field width to be varied.
- If the frequency field is too small, saturations can occur and frequency information may be lost.

Cache Operation - Frequency Counter Saturation

- All frequencies are divided by 2 with a shift right (built as a special feature of the cache and activated by asserting the saturation signal to the cache).