Power-Performance Models for Runtime Reconfiguration and Power Capping

Pietro Cicotti, **Ananta Tiwari (AT)**, Laura Carrington
EP Analytics, Inc.
PMaC/SDSC

MODSIM 2015

Presenter - Ananta Tiwari (ananta.tiwari@epanalytics.com)
Corresponding Author – Pietro Cicotti (pietro.cicotti@epanalytics.com)

Motivation

- Goals:
  - Support proactive run time decisions
  - Create integrated power/performance models
- Requirement:
  - Incur little runtime overhead
    - Use little information
    - Be queried quickly
- Approaches:
  - Instruction-Level Modeling
    - Constructed using single-instruction benchmarks
    - Correlates instructions in compute phase to performance/power
  - Statistical Modeling
    - Constructed on micro-benchmarks
    - Correlates performance hardware counters to performance/power

Use-case

- Run-time system activated or informed before a compute phase
  - Compute phases identified in the source code
    - Run-time system API calls added to the source code
  - Compute phases identified in the binary
    - Run-time system API calls added by binary instrumentation
  - Runtime queries models and selects optimal configuration
- Performance/power locally optimized
- Power cap globally imposed (out of scope)

Instruction-Level Models

- Instruction-level – measure cost in terms of performance and energy for all instructions
  - Benchmark the contribution of individual instructions
    - add r1_64b,r2_64b -> 1 cycle, 1.4nJ (2.6GHz), 1.2nJ (2.5GHz), …
  - Create a model that aggregates the contributions
    - loop@2.5GHz
      - Time=$2.5 \times 10^{-9} \times \alpha \times \sum cycles_i$
      - Energy=$\sum energy_i(2.5GHz)$
      - Power=Energy/Time
  - Offline
    - Measure the contribution of single instructions
    - Create the model
    - Analyze/instrument code
  - Online
    - Use information from static analysis before compute phase start and invoke run time system
    - Use information at run time dynamic execution
      - E.g. tune α for expected hit rate
    - Search performance/power space at different frequencies
    - Optimize: e.g. power limit and Energy Delay Product

Benchmarking Instructions

- Reduce benchmarking space
  - 300+ instructions in x86_64 ISA
  - Some instructions are *overloaded*
    - Different data types, number of operands, etc.
- Group instructions in *equivalence classes*
  - Members of an equivalence class have same latency and energy cost
  - E.g. [add]={add, sub, and, …}
- Approximate energy at different frequencies

Benchmarks

- Arbitrarily long sequence of embedded asm

```
for(i=0;i<n;++i)
  UNROLL(asm volatile ("subsd %%xmm1, %%xmm0\t\n"::));
```

```
0000000000400860 <main>:
  ...
  400900: f2 0f 5c c1        subsd  %xmm1,%xmm0
  400904: f2 0f 5c c1        subsd  %xmm1,%xmm0
  400908: f2 0f 5c c1        subsd  %xmm1,%xmm0
...
```

- Power/Energy measured for system (Watts), package (RAPL) and DRAM (RAPL)

Memory Operands

- Load/store instructions
- Instructions with memory operands
- Latency and energy depend on level servicing request
  - E.g. latency=4 cycles, 12 cycles, 54 cycles, 375 cycles
- ad-hoc benchmark to target a single level
- Need estimate at runtime of hit rates

Integration with Tools

- Compile time or static binary analysis
  - Determine instruction mix
- Tools (or programmer)
  - Identify compute phases and insert calls to run time system
  - setup ad-hoc model for a given compute phase
    - run time parameters: hit rates, optimization target, and power cap
  - if know or reasonably estimate possible, model is statically tuned
- Run time system
  - Receives parameters before computation phase
  - Runs model and selects optimal DVFS setting

Machine-Learning Approach

- Develop machine learning based model to inform power capping decisions
  - Models are trained using hardware counters
  - Explore the performance and power sensitivity of different computations when power-related hardware parameters change

Enabling Components

- Main enabling components
  - Modeling methodology that can encapsulate the relationship between hardware power states, application characteristics derived from hardware counters and power/performance responses
  - A set of computational kernels that are representative of most of the computations we see in HPC

Modeling Methodology

- Dimension reduction using correlation analysis and PCA
- Modeling technique: Cubist
  - Rule-based learning model built using a tree of linear regression models
  - Predictions are made using a linear model found at the leaf nodes of the tree
  - The choice of leaf is determined by the rules in non-terminal nodes that are also based on linear regression models
  - Has the capability of capture non-linear relationships between inputs

Modeling Methodology

- Prevent over-fitting:
  - Split the empirical dataset into training and validation sets
    - 60%-40% split: 60% used for training the model and 40% for validation
  - 10-fold cross validation to avoid over-fitting during model training
- Variable importance analysis to determine which predictors have the most impact on performance and power

Summary

- Two approaches:
  - Instruction-based
    - Estimates cost of instructions, directs models with instruction mix and optimization target
  - Status
    - Tuning and validating
    - Working on tools and run time system integration
- Future goals
  - Integration with run time system
  - Model ARM
  - Multi-resource management and power shifting
  - Local and global optimizations