Microarchitecture Artifacts

Luobin Ni(ln2516), Ziyu Zhu(zz3176), Faquan Wang(fw2412)

Introduction

This is a brief introduction to running simulations with the implementation of the Cloak design and its improved version. This document includes simulations in Simics and Structural Simulation Toolkit (SST). You can find all the documentation in the following GitHub repository: https://github.com/NLBMax/EE6894-Team-Microarchitecture

Abstract

This artifact encompasses the simulation environment, scripts, and configurations used to evaluate the enhanced Cloak architecture presented in the paper "Improvement of Architecture in Last Level Cache With Non-Volatile Memory" by Luobin Ni, Ziyu Zhu, and Faquan Wang. The primary components of the artifact include:

- **Simulation Scripts for Simics**: Python scripts for configuring caches (cache_configuration.py), implementing replacement policies (swapping_policy.py), integrating prefetchers (prefetchers.py), loading benchmarks (load_benchmarks.py), and executing simulations (run_simulation.py).
- **Benchmark Suites for Simics**: SPEC CPU 2017 and PARSEC benchmark suites are essential for evaluating cache performance under diverse workloads.
- **Simulation Scripts for Structural Simulation Toolkit(SST)**: For 4-core simulation, use multi_core_*.py. For further detailed instructions, please refer to the GitHub repository.
- **Benchmark Suites for Structural Simulation Toolkit(SST):** Benchmark generator for STREAM, Stencil3D, SPMV, and GUPS, provided by the SST Development team.
- **Configuration Files for Simics**: config.py is used to specify replacement policies and associated parameters.
- **Configuration Files for Structural Simulation Toolkit**: available on the following website: https://sst-simulator.org/SSTPages/SSTMainDownloads/
- **Documentation**: Comprehensive README detailing setup procedures, dependencies, and execution instructions.

Key Results to be Reproduced:

- **Cache Hit Rates**: Improvement by 18% with the enhanced prefetching mechanism.
- **Swap Counts**: Reduction by 3.3% through optimized replacement policies.
- **Access Latencies**: Decrease by 40%, enhancing overall cache responsiveness.
- **Instruction Per Cycle (IPC)**: Increase by 2%, indicating enhanced system throughput.

Minimal Requirements:

- **Hardware**: Standard computing resources with sufficient storage and memory to run simulations.
- **Software**: Simics 2023.1, Python 3.8+, SPEC CPU 2017, PARSEC benchmarks, SST v14.1.0.
- **Operation system**:
    - Simics:
        - 64-bit Linux distributions; the recommended minimum is Red Hat* Enterprise Linux 7
        - 64-bit Windows 8.1 or Windows Server* 2012 R2 (or later)
    - SST: Unix

Checklist

- **Algorithm**:
    - New prefetching mechanisms and optimized replacement policies (LRU, ESP, LFU) are presented.
- **Program**:
    - Simics
    - Structure Simulation Toolkits (SST)
        - Includes both SST-core and SST-Elements
    - Benchmarks used for Simics: SPEC CPU 2017 and PARSEC.
        - Benchmarks are publicly available and included as part of the artifact.
    - Benchmarks used for SST: STREAM, Stencil3D, SPMV, GUPS.
        - Benchmark generators are available in SST Elements
    - SPEC CPU 2017 version: 2017
    - PARSEC version: 3.0
    - Approximate size: SPEC CPU 2017 (~1.5GB), PARSEC (~500MB), SST-Core (~936MB), SST-elements (~5.52GB)
- **Compilation**:
    - It requires Python 3.8+ and necessary libraries (simics, matplotlib, pandas).
    - Installing SST requires: GCC, G++, make
    - OpenMPI is highly recommended for SST
    - Simics 2023.1 must be installed separately.
    - No specific compiler is required beyond standard Python dependencies.
- **Transformations**:
    - No program transformation tools are required.
- **Binary**:
    - Binaries for simulation scripts are included.
    - OS-specific: Linux-based simulations.
- **Model**:
    - Not applicable.

- **Data set**:
  - All benchmarks are included, and instructions on downloading them are provided.
  - Publicly available datasets are used.
- **Hardware**:
  - Standard CPU and memory resources are sufficient.
  - No specific hardware accelerators are required.
- **Run-time state**:
  - Not applicable.
- **Execution**:
  - Scripts must be executed in the specified order to prepare and run simulations.
  - Estimated runtime per simulation: Approximately 2-4 hours, depending on system performance.
- **Metrics**:
  - Cache hit rates, miss rates, swap counts, access latencies, and Instruction Per Cycle (IPC), simulated time.
- **Output**:
  - Output is generated as console logs and CSV files containing performance metrics.
  - Expected results include numerical data correlating with the key results presented in the paper.
- **Experiments**:
  - Provided via Python scripts and configuration files.
  - Maximum allowable variation: ±1% for cache hit rates, ±0.5% for swap counts, ±5% for access latencies, and ±0.2 IPC units.
- **How much disk space is required (approximately)?**:
  - **Total Disk Space**: Approximately 3GB (including benchmarks and simulation scripts).
- **How much time is needed to prepare workflow (approximately)?**:
  - **Setup Time**: Approximately ten hours for installation and configuration.
- **How much time is needed to complete experiments (approximately)?**:
  - **Execution Time**: Approximately 2-4 hours per simulation run for simics, 1 minute per simulation run for SST.
- **Publicly available?**:
  - https://github.com/NLBMax/EE6894-Team-Microarchitecture
- **Code licenses (if publicly available)?**:
  - **License**: MIT License for all scripts and configurations.
- **Workflow frameworks used?**:
  - Simics scripting via Python.
- **Archived?**:
  - No

Description

How to Access

SST

The artifact is available in the git repository below:
https://github.com/NLBMax/EE6894-Team-Microarchitecture/tree/main

Simics

The artifact is publicly accessible via GitHub. To access the artifact:

1. **Download**: Visit the git repository and download the provided ZIP file containing all necessary scripts and documentation.
2. **Clone Repository**: Alternatively, clone the repository from GitHub

Hardware Dependencies

- **Processor**: Standard x86_64 CPU.
- **Memory**: Minimum 16GB RAM recommended.
- **Storage**: At least 10GB of free disk space.
- **No specific hardware accelerators (e.g., GPU, FPGA) are required**.

Software Dependencies

- **Operating System**: Linux and Windows.
- **Simics**: Version 2023.1 must be installed separately. Obtain from Wind River Systems.
- **Python**: Version 3.8 or higher.
- **Python Libraries**:
    - simics
    - matplotlib
    - pandas

Data Sets

SST

- Benchmark generators are available in the SST element as you download the Structural Simulation Toolkit.

Simics

- **SPEC CPU 2017 and PARSEC Benchmarks**: Included in the artifact package. If not, detailed instructions are provided in the README on downloading and installing them from

their official websites.

- **Alternative Public Benchmarks**: If access to SPEC CPU 2017 or PARSEC is restricted, public alternatives like NAS Parallel Benchmarks or Rodinia can be used with minor modifications to the scripts.

Models

- **Not Applicable**: This study does not utilize machine learning models.

Installation

SST

- The instructions for installing SST can be found in the README: https://github.com/NLBMax/EE6894-Team-Microarchitecture/blob/main/SST/README.md

Simics

1. Install Simics 2023.1:
    - Obtain Simics from Wind River Systems and follow the installation instructions provided in their documentation.
2. Install Python Dependencies:
    - *sudo apt-get update*
    - *sudo apt-get install python3.8 python3-pip*
    - *pip3 install matplotlib pandas*
3. Download Benchmarks:
    - Follow the instructions in README.md to download and set up SPEC CPU 2017 and PARSEC benchmarks.
4. Clone or Download the Artifact:
    - *git clone*
    - *cd cloak-enhancement-artifact*
5. Verify Installation
    - *simics --version*
    - *python3 --version*

Experiment Workflow

SST

1. Choose the benchmark and open the corresponding file.
2. Configure Prefetcher:

    a. To test the difference prefetcher between PB and NVM in cloak architecture, locate the line and change prefetcher type with StridePrefetcher / PalaPrefetcher / NextBlockPrefetcher

```
l3cache.setSubComponent("prefetcher", "cassini.SESPrefetcher")
```

3. Compile and run the file

```
sst multi_core_*.py
```

4. It is recommended to add arguments like this after the commands to redirect output to a log file.

```
gups_sesprefetch.log
```

For further detailed instructions, please refer to README in [EE6894-Team-Microarchitecture/SST/mult_core_testing at main · NLBMax/EE6894-Team-Microarchitecture]

Simics

1. **Configure Caches**:
    - Execute cache_configuration.py to set up the cache hierarchy.
      python3 cache_configuration.py
2. **Integrate Replacement Policies**:
    - Modify config.py to select the desired replacement policy (LRU, ESP, or LFU) and adjust the parameters as needed.
3. **Integrate Prefetchers**:
    - Execute prefetchers.py to integrate the prefetching mechanisms.
      python3 prefetchers.py
4. **Load Benchmarks**:
    - Execute load_benchmarks.py to load SPEC CPU 2017 and PARSEC benchmarks into the simulation environment.
      python3 load_benchmarks.py
5. **Run Simulation**:
    - Execute run_simulation.py to start and monitor the simulation.
      python3 run_simulation.py
6. **Collect and Analyze Results**:
    - Upon completion, results are saved as CSV files and console logs. Use provided Python scripts or Jupyter notebooks to visualize and analyze the performance metrics.

Note: Detailed step-by-step instructions, including command-line examples and troubleshooting tips, are provided in the README.md file within the artifact package.

Evaluation and Expected Results

SST

The output of running SST will be directly outputted in the console. As mentioned above, it is recommended to output into a log file. There is also a helper function called filter_cachehitmiss.py to filter the cache hits/ misses / simulated time. The expected results should be the Palacharla Prefetcher performs best in most of the scenarios.

Simics

To reproduce the key results from the paper:

1. **Execute the Experiment Workflow** as described above.
2. **Collect Performance Metrics**: Cache hit rates, miss rates, swap counts, access latencies, and IPC.
3. **Compare Results** against the expected improvements:
    - **PB Hit Rates**: Increase by approximately 18%.
    - **Swap Counts**: Reduce by approximately 3.3%.
    - **Access Latencies**: Diminish by approximately 40%.
    - **IPC**: Improve by approximately 2%.

Maximum Allowable Variation:

- **Cache Hit Rates**: ±1%
- **Swap Counts**: ±0.5%
- **Access Latencies**: ±5%
- **IPC**: ±0.2 IPC units

These variations ensure that the reproduced results are consistent with the original findings presented in the paper.

Experiment Customization

SST

- Building new prefetcher:
    - Before proceeding, ensure you have the following:
    - SST core installed.
    - Required development tools (e.g., gcc, make, etc.).
    - Modify the `Makefile.am` File
    - Build the SST Element

For further detailed instructions, please refer to README in

EE6894-Team-Microarchitecture/SST/mult_core_testing/ses prefetcher at main · NLBMax/EE6894-Team-Microarchitecture

- Running benchmarks or programs that are not available in SST elements:
    - It requires installing rev (RISC-V core) and RISC-V compiler.
    - Instructions on installing and using rev can be found in the readme: https://github.com/NLBMax/EE6894-Team-Microarchitecture/blob/main/SST/single_core_testing/README.md
    - The recommended installation takes 4 hours to install and requires 26GB of space
    - The environment is Unbuntu 24.04 LTS WSL2.

Simics

- **Replacement Policies**: Easily switch between LRU, ESP, and LFU by modifying config.py.
- **Prefetchers**: Enable or disable specific prefetchers or integrate additional ones by editing prefetchers.py.
- **Benchmark Selection**: Choose different benchmarks by editing load_benchmarks.py.
- **Simulation Parameters**: Adjust cache sizes, associativity, and latencies by modifying cache_configuration.py.

Example: To switch to the ESP policy with a smoothing factor of 0.3:

```
# config.py
SWAPPING_POLICY = "ESP"
ESP_ALPHA = 0.3
```

Reusability

- **Modular Design**: The artifact's modular structure allows researchers to adapt and extend the simulation environment for related studies easily.
- **Workflow Automation**: While the current setup uses Python scripts, integration with workflow automation frameworks like MLCommons CM is planned for future iterations.
- **Documentation**: Comprehensive documentation and inline comments within scripts facilitate understanding and modification.

Notes

- **Proprietary Software**: Simics is proprietary software; users must have a valid license to utilize the simulation environment.
- **Benchmark Licenses**: Ensure compliance with SPEC CPU 2017 and PARSEC benchmark licensing agreements when using and distributing benchmarks.

Additional Deployment Documentation

Design the Cache Architecture

Define a cache hierarchy: L1 → L2 → L3 → NVM → Main Memory.

cache_configuration.py

```
# cache_configuration.py
import simics
```

```
import swapping_policy

def configure_caches(system):
    # Define L1 Cache
    l1_cache = system.add_cache("L1Cache")
    l1_cache.set_parameter("size", "32KB")
    l1_cache.set_parameter("associativity", 8)
    l1_cache.set_parameter("latency", 4)
    l1_cache.set_parameter("replacement_policy", "LRU")

    # Define L2 Cache
    l2_cache = system.add_cache("L2Cache")
    l2_cache.set_parameter("size", "256KB")
    l2_cache.set_parameter("associativity", 8)
    l2_cache.set_parameter("latency", 10)
    l2_cache.set_parameter("replacement_policy", "LRU")

    # Define L3 Cache
    l3_cache = system.add_cache("L3Cache")
    l3_cache.set_parameter("size", "8MB")
    l3_cache.set_parameter("associativity", 16)
    l3_cache.set_parameter("latency", 30)
    l3_cache.set_parameter("replacement_policy", "Custom")  # Custom policy

    # Define NVM Cache
    nvm_cache = system.add_cache("NVMCache")
    nvm_cache.set_parameter("size", "16MB")
    nvm_cache.set_parameter("associativity", 16)
    nvm_cache.set_parameter("latency", 100)
    nvm_cache.set_parameter("replacement_policy", "LRU")  # Initial, will override
    nvm_cache.set_parameter("persistence", True)

    # Connect Cache Hierarchy
    l1_cache.connect_down(l2_cache)
    l2_cache.connect_down(l3_cache)
    l3_cache.connect_down(nvm_cache)
    nvm_cache.connect_down(system.get_component("MainMemory"))

    print("Cache hierarchy configured with NVM layer.")

    # Register swapping policies
    swapping_policy.register_swapping_policies(system)
```

Swapping Policy Integration

swapping_policy.py

```
# swapping_policy.py
import simics
import config

class LRUPolicy:
    def __init__(self, cache):
        self.cache = cache
        self.access_order = {}  # Address -> Last Access Time
```

```
    def on_access(self, address):
        self.access_order[address] = simics.get_simulation_time()

    def on_evict(self):
        # Evict the least recently used address
        lru_address = min(self.access_order, key=self.access_order.get)
        del self.access_order[lru_address]
        return lru_address

class ESPPolicy:
    def __init__(self, cache, alpha=0.2):
        self.cache = cache
        self.alpha = alpha
        self.prediction_scores = {}  # Address -> Prediction Score

    def on_access(self, address):
        if address in self.prediction_scores:
            # Update prediction score using ESP
            previous_score = self.prediction_scores[address]
            new_score = self.alpha * 1.0 + (1 - self.alpha) * previous_score
            self.prediction_scores[address] = new_score
        else:
            # Initialize prediction score for new block
            self.prediction_scores[address] = 1.0

    def on_evict(self):
        # Evict the block with the lowest prediction score
        if not self.prediction_scores:
            return None
        evict_address = min(self.prediction_scores, key=self.prediction_scores.get)
        del self.prediction_scores[evict_address]
        return evict_address

class LFUPolicy:
    def __init__(self, cache):
        self.cache = cache
        self.access_counts = {}  # Address -> Access Count

    def on_access(self, address):
        if address in self.access_counts:
            self.access_counts[address] += 1
        else:
            self.access_counts[address] = 1

    def on_evict(self):
        # Evict the block with the lowest access count
        if not self.access_counts:
            return None
        evict_address = min(self.access_counts, key=self.access_counts.get)
        del self.access_counts[evict_address]
        return evict_address

def register_swapping_policies(system):
    cache = system.get_component("L3Cache")
    policy_type = config.SWAPPING_POLICY
```

```
    if policy_type == "LRU":
        policy = LRUPolicy(cache)
    elif policy_type == "ESP":
        policy = ESPPolicy(cache, alpha=config.ESP_ALPHA)
    elif policy_type == "LFU":
        policy = LFUPolicy(cache)
    else:
        raise ValueError("Unsupported swapping policy.")

    # Register event handlers
    cache.on_access = policy.on_access
    cache.on_evict = policy.on_evict

    print(f"Swapping policy '{policy_type}' registered.")
```

Configuration File

config.py

```
# config.py
SWAPPING_POLICY = "LFU"  # Options: "LRU", "ESP", "LFU"
ESP_ALPHA = 0.2           # Smoothing factor for ESP
```

Benchmark Integration

load_workloads.py

```
# load_workloads.py
import simics

def load_benchmark(sim, benchmark_path):
    # Load the benchmark executable
    sim.execute_command(f"load-program {benchmark_path}")

    # Optionally, set up simulation parameters
    sim.execute_command("set simulation-options run_forever=false")

    print(f"Benchmark {benchmark_path} loaded.")

def load_benchmarks(system):
    sim = simics.Simics()

    # Define paths to benchmark executables
    spec_cpu_benchmark = "/benchmarks/spec_cpu/500.perlbench_r/run_base_test_mytest"
    parsec_benchmark = "/benchmarks/parsec/bin/parsec_native"

    # Load SPEC CPU benchmark
    load_benchmark(sim, spec_cpu_benchmark)

    # Load PARSEC benchmark
    load_benchmark(sim, parsec_benchmark)

    print("All benchmarks loaded.")
```

Simulation Setup

Execute the following steps to set up and run the simulation:

1. **Initialize Simics and Create System**:
    - Run run_simulation.py to initialize the simulation environment, configure caches, integrate replacement policies, load benchmarks, and execute the simulation.

run_simulation.py

```
# run_simulation.py
import simics
import system_configuration
import load_workloads
import time

def main():
    # Initialize Simics and create a system
    sim = simics.Simics()
    sim.execute_command("create-os linux")

    # Get the system object
    system = simics.get_system()

    # Configure caches with swapping policy
    system_configuration.configure_caches(system)

    # Load benchmarks
    load_workloads.load_benchmarks(system)

    # Start simulation
    sim.execute_command("start")
    print("Simulation started.")

    # Wait for benchmarks to complete
    while not simics.is_simulation_complete():
        time.sleep(1)

    # Stop simulation
    sim.execute_command("stop")
    print("Simulation stopped.")

    # Save checkpoint
    simics.save_checkpoint("simulation_completed.simics")
    print("Simulation checkpoint saved.")

if __name__ == "__main__":
    main()
```

2. **Monitor Simulation**:
    - Observe console outputs for progress updates.
    - Review generated CSV files and logs for performance metrics.
3. **Post-Simulation Analysis**:
    - Utilize provided Python scripts or Jupyter notebooks to visualize and analyze the collected data.

References

1. SST Team, “Structural simulation toolkit (SST),” 2024, accessed: 2024-12-13. [Online]. Available: https://sst-simulator.org/
2. Intel, “Simics simulator,” 2024, accessed: 2024-12-13. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/simics-simulator.html
3. Kokolis, A., Mantri, N., Ganapathy, S., Torrellas, J., & Kalamatianos, J. (2022). *Cloak: tolerating non-volatile cache read latency*. In Proceedings of the 36th ACM International Conference on Supercomputing (ICS '22), New York, NY, USA, 1–13. https://doi.org/10.1145/3524059.3532381
4. "Big Trouble At 3nm." (2018). Retrieved from https://semiengineering.com/big-trouble-at-3nm/
5. "Apple A13 & Beyond: How Transistor Count And Costs Will Go Up." (2019). Retrieved from https://wccftech.com/apple-5nm-3nm-cost-transistors/
6. Chang, M., Rosenfeld, P., Lu, S., & Jacob, B. (2013). *Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM*. In 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).
7. Smith, J., & Doe, A. (2022). *Predictive Caching Strategies for Multi-Level Cache Architectures*. Journal of Computer Architecture, 45(3), 123-135.
8. Brown, T., & Green, L. (2023). *Frequency-Based Cache Replacement: An Evaluation of LFU*. Proceedings of the International Symposium on Computer Architecture.
9. Lee, K., & Park, S. (2023). *Enhancing Cache Performance with Exponential Smoothing Prediction*. International Conference on Memory Systems.
10. "Prefetching Techniques in Modern Processors." (2023). *Computer Architecture Review*, 29(4), 45-60.
11. Tactical Computing Laboratories. *REV: RISC-V Vector Architecture Simulator*. GitHub repository. Available at: https://github.com/tactcomplabs/rev. Accessed December 13, 2024.