Project Report: Leveraging Near Memory Processing for Cuckoo Cycles

Jonathan Lister
Brown University
May 15th, 2019
Advisor: Maurice Herlihy

Abstract

Near-data processing is a hardware enhancement that uses lighter-weight processors and a "vault" of closely-coupled memory in order to efficiently run tasks with poor locality (that is, tasks in which most cache accesses are likely to be misses.) Prior work in my research group has been mostly focused on performance of data structures. This paper documents the results of delegating a major task in an application to near-data processing, as well as implementation challenges from a software development perspective.

1. Introduction

Cuckoo hashing [5] is a hashing technique in which each value in the table can be placed in one of two locations. If a value cannot be placed in one, it is placed in the other. If both spots are occupied, it is placed in one of the two locations, evicting the value that was there and forcing it to move to its other candidate position. This process occurs recursively until no more evictions occur–this means the hashing could fail if the recursive evictions attempt to evict the original value. The values and locations can be treated as a graph, where each value is an edge and each location a vertex. Cuckoo hashing thus fails if a cycle occurs.

Cuckoo cycles [6], developed by John Tromp, is a puzzle based on cuckoo hashing proposed as a proof-of-work for blockchain systems. The objective of the puzzle is to find a cycle of a specific length in a cuckoo graph generated by the hash function siphash, using a cuckoo hash as described above to find cycles.

2. Related Work

This project builds off Choe et al.'s work [2] on near-data processing. The simulator used is a modified version of the system described by Azarkhish et al. [1] My code uses the same high-fidelity simulator that was used to evaluate the performance of NDP-based data structures in Choe's paper. However, this project focuses on an application rather than a data structure.

Gao et. al [3] postulate that using multiple NDP cores can provide meaningful performance gains for map-reduce workloads. However, they assume that each core has an L1 cache and each NDP core has a method of communicating directly. In addition, they specify that map-reduce has high spatial locality. My work assumes no communication and no L1 cache, and chooses cuckoo cycles as an example because of its poor locality.

3. Architecture

The simulated system runs on a gem5-based simulation platform [4]. The simulation ran on a Lenovo Y510P laptop with 8 GB Ram, 2.4GHz processor running Ubuntu 14.04. The system uses 8 in-order host processors (ARMv7 Cortex-A15) [2] and one NPM core. I used the closed-page DRAM access policy described in Choe et al.'s paper, which contains more details of the system architecture. [2]. However, in order to make the cuckoo cycle code fit on the core, the text region in the memory map for NDP code had to be grown by 36 KB.

Table 1 shows some basic information about the host processor, and table 2 shows some basic information about the NDP core. This information was obtained by configuring the simulator and from Azarkhish's paper [1].

	Host System
CPU speed	2.0 GHz
L1 Data Cache Size	64 kB
L2 Cache Size	2 MB
L2 Cache Refill Latency (256 B)	102.3 ns

Table 1. Selected host system architecture details

	NDP details
CPU speed	2.0 GHz
Scratchpad size	8 kB
Scratchpad access time	0.3 ns
Vault Size	128 MB
Vault Access Time (4 B)	39.1 ns

Table 2. Selected NDP architecture details

4. Implementation

The mining program used is the lean miner from Tromp's repository on Github [7] with certain modifications in order to leverage near-data processing.

The lean miner's work is split into two phases: trimming and traversal.

- Trimming is an optimization which iteratively removes nodes with a degree of less than 2, as they cannot be included in cycles. A fixed number of rounds of trimming is chosen by the user. At each round, the degree of each node still in the graph is counted using a `twice_set`, which tracks whether each node has a degree of 0, 1, or greater than 1. After the degree of each node is counted, each node with degree less than 1 is deleted. Deleted nodes are tracked by a data region called a `shrinking_set`, which contains a bit for each node in the graph. The set starts out full, and a node i can be removed by setting the ith bit to 0.
- Traversal is when each path through the graph is taken to see if there are any cycles of the appropriate length. This makes use of the `cuckoo_hash` structure, which represents connections in the graph.

The baseline implementation is simply Tromp's lean miner. One enhanced implementation is mediating access to the large data regions by holding all three data structures in the NDP vaults and making a function call for each operation on those data structures (leaving all hashing to the host processor.) The final method is performing trimming in the NDP vault and then using the vector register to copy it back to main memory. This method involves computing hashes in the NDP core.

Because we do not assume NDP cores have easy access to each other's memory, we simply use a single-threaded lean miner as the baseline implementation. We assume 19 edge bits (which means nodes are composed of 20 bits, which means the graph has a size of 2^{20}, and no partitioning.)

5. Challenges

The simulated NDP hardware is surprisingly fragile to how the code is written. Originally, the small code region of the executable was not large enough to fit the necessary parts of the lean miner application, so it had to be expanded by configurating simulation parameters. Additionally, attempting to run the whole puzzle solver on the NDP core necessitated further stack expansion, at which point the simulation slowed down dramatically and NDP core code **did not run**.

Additionally, the NDP code compiler would compile code that doesn't work correctly:

- The first issue is that by using a high-performance optimization level, the code wouldn't load. I noted that compilation appears to work but trying to run any NDP code results in a panic and error message.
- The second issue is that possibly due to the depth of stack frames, variables are sometimes placed within other variables. This made certain code run as an infinite loop:

```
int i;
for (i = 0; i < BLAKE2B_BLOCKBYTES; i++) {
    buffer[i] = 0;
}
```

If the buffer was allocated on the stack, `i` would be located somewhere inside the buffer, causing it to be reset to 0 in the middle of the loop. This problem was avoided by allocating these buffers in the vault.

6. Results

The runtimes, shown in table 3, are averaged over 5 different puzzle nonces: 97667, 236562, 127135, 385955, and 125247.

The puzzle was run in the simulator using three different techniques: running solely on the host with no NDP code and one worker thread (Base,) making an NDP call for every edit to one of the three major data structures (Delegated Data Structures,) and performing the expensive "trimming" process on the core and handing it back to the host (NDP Trim.)

The slowdown when using the NDP just to perform simple operations on the large data regions is expected, as there is an additional layer of communication added in order to accomplish this, but performing trimming in the NDP core is even slower. This is likely because the hash functions used (2-4 siphash and blake2b) involve a high amount of arithmetic computation on a relatively small memory region (i.e. small enough to be cached, if the NDP core had a cache.)

7. Conclusion

My work indicates naively implementing applications or routines using near-data processing may not provide a significant advantage, even if there is a significant portion of work that has no locality. This is because traditional CPU execution gives a much larger performance benefit for the work

	Nonce					
	97667	236562	127135	385955	125247	Average
Base Runtime (seconds)	1.503436	1.505234	1.511919	1.508958	1.505888	1.507087
Delegated Data Structures Runtime (seconds)	2.741559	2.74734	2.755014	2.750628	2.744232	2.747755
NDP Trim (Seconds)	4.104877	4.111344	4.123279	4.117365	4.115168	4.114407

Table 3. Runtimes in simulated environment using different techniques. Performance suffers when the NDP cores are used.

that can leverage the cache, such as computing the hashes for the cuckoo cycles puzzle. Any NDP-based application requires a compiler that can both leverage the properties of NDP and reliably produce bug-free, correct code.

8. Future Work

We plan to leverage other features of the NDP cores to get better performance for subroutines with good locality. Additionally, other use cases can be identified, such as hybrid data structures and dynamic reallocation of objects based on ”liveness” of specific members of a data structure.

References

[1] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*, pages 19–31, New York, NY, USA, 2016. Springer-Verlag New York, Inc. 1

[2] J. Choe, A. Huang, T. Moreshet, M. Herlihy, and R. I. Bahar. Concurrent data structures with near-data-processing: an architecture-aware implementation. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 19)*, SPAA ’19, New York, NY, USA, 2019. ACM. 1

[3] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT ’15, pages 113–124, Washington, DC, USA, 2015. IEEE Computer Society. 1

[4] E. A. Jiwon Choe. https://github.com/wldnjs1216/smc, 2018. Private repository, forked from https://iis-git.ee.ethz.ch/erfan.azarkhish/SMCSim. 1

[5] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004. 1

[6] J. Tromp. Cuckoo cycle: a memory bound graph-theoretic proof-of-work. Cryptology ePrint Archive, Report 2014/059, 2014. https://eprint.iacr.org/2014/059. 1

[7] J. Tromp. https://github.com/tromp/cuckoo, 2018. 2