Hardware Hash Functions

Harlan Kringen, Atefeh Mohseni, Bryan Tan

1 Introduction

In tiled many-core processor design, cores are fitted onto a single chip and interconnected via mesh-based networks. There are local and remote memory controllers for any tile (processing core) that are fitted onto the same chip [9]. When a core wants to load or store to a memory address, it maps cache line address to find the corresponding tile where its sharing status is stored. In this report we will explore different algorithms to efficiently map memory addresses to the caches given we have different tile sizes.

We explored two main approaches to solving this problem, program synthesis and parameterized hashing. The first approach is based on a technique used widely in the field of programming language theory. In program synthesis, we state a problem based on a specification of its constraints and allow a domain specific engine or logical solver to explore the problem space for satisfying solutions. In parametrized hashing, we investigate how to make existing hash functions sensitive to various parameters describing the bins or output of the function. We ultimately decide on a scheme based on voting theory. Finally, we canvas alternate places where we might find solutions to this problem, specifically in the field of machine learning.

2 Hash Functions and Associative Caches

As noted above, many-core architectures typically have several processors on chip, and these processors will have their own caches and routing logic to connect them to other processors. They might even have other peripherals associated such as floating point units. An example arrangement of processors in a many-core system, as well as a view of what a single tile looks like, are shown in Figures 1 and 2, both taken from the OpenPiton architecture [1].

In a many-core architecture with several identically sized caches, such as the OpenPiton system, the problem of finding the specific data associated with a memory address can be broken into two phases. In the first phase a memory address is homed to a cache on a specific tile. In the second phase the memory address is parsed to determine the exact line of the cache where the relevant data is stored. The first phase is the subject of our

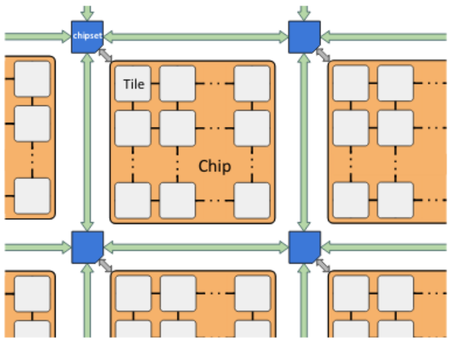

Figure 1: OpenPiton architecture

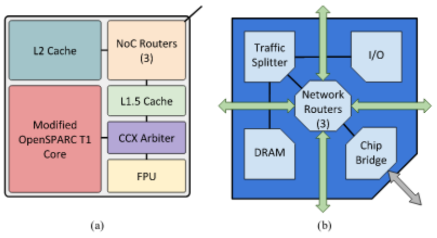

Figure 2: OpenPiton tile and chipset

investigations, and so we will canvas the second phase now, before diving into the details of the first phase.

The second phase of retrieving a cache line is accomplished by breaking down the address into subsets, with each subset of bits referring to a different level of the memory hierarchy. For instance, in Figure 3, we can see such a division. With tag and index, bits, we specify narrower regions of the cache, until the offset picks exactly which cache line holds that specific address.

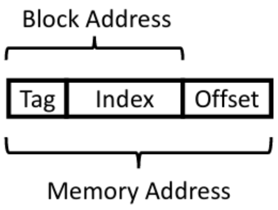

Figure 3: A memory address

In the first phase, because the data associated with the address can be stored in any cache, we are simply trying to distribute the addresses evenly across chips. This prevents traffic jams on the network interconnects between chips and gives all memory equal access times. To accomplish this, hardware architects employ hash functions, which are mappings from objects (the memory addresses) to identifiers, or hashes (the tiles). The mappings are deterministic, in that they compute the same hash for a key every time, are fast to compute (comprising of only bitvector operations), and evenly distribute the keys among the available hashes.

This scheme is inflexible however, if the sizes of the caches change. This is due to a fact about modern cache design, namely, associativity. Associativity describes how efficiently we can place data into the cache. In fully-associative caches, data may be placed anywhere. In a non-fully-associative, or “direct-mapped” cache, data is placed into the cache according to a simple scheme that possibly ejects memory already in the cache. Because of this, direct-mapped caches tend to show poor overall utilization, while fully-associative caches show increased utilization, and therefore suggest more efficient use of silicon.

We can see that for a many-core architecture with caches that are at least partially associative, using a traditional hash function will result in overall worse utilization. Because memory addresses are mapped uniformly to all caches, larger caches will have proportionately fewer lines inhabited, and smaller caches will have all of their lines inhabited. To address this mismatch, we will investigate strategies for mapping memory addresses into caches proportionate to their sizes, thus maintaining high utilization across caches.

3 Solutions

3.1 Program Synthesis

Program synthesis [2] is a field that studies the problem of automatically *synthesizing* a program that can satisfy a given *specification*. Specifications may vary from informal specifications, such as test cases, examples, or natural language, to formal specifications, such as formal properties or logical formulas.

The characteristics of our problem make program synthesis a viable approach. Hash functions are typically simple with a few bitwise operations, so the search space is not too large relative to other types of programs. Through synthesis, we can formally guarantee properties about our hash function; for example, we may be interested in preventing strided addresses from mapping to the same block. We can also exploit patterns in program traces in order to improve performance for particular workloads.

3.1.1 Problem Formulation

To formulate the synthesis problem more precisely, suppose we have a list of caches $C_1, C_2, \ldots, C_N$ that together have a total of 2^k blocks. We then label all blocks by $B_1, \ldots, B_{2^k}$, so that the first $|C_1|$ blocks are in C_1, the next $|C_2|$ blocks are in C_2, and so on. Now, we want to synthesize a function $\mathcal{H} : \{0,1\}^n \to \{0,1\}^k$ that takes an n-bit address and computes a block B. Using B, we can use the above numbering scheme to determine which cache the address should be sent to.

This formulation is actually more general than is necessary, because we do not actually need to compute the block number. For example, we could take the output of $\mathcal{H}$ to directly be the cache number. However, the generality does not hurt; for example, a future work could use the actual block numbers themselves to account for differing associativity levels between the caches.

3.1.2 Sketching a Hash Function

To solve our problem, we use the method of sketching [5], where we define a *sketch* (i.e. a "template") of $\mathcal{H}$ containing *holes* (i.e. a "placeholder") that the synthesizer will be asked to *complete* with actual program fragments. We observe that $\mathcal{H}$ is usually of the following form:

- $\mathcal{H}$ will select some (or all) of the input bits $b_1, \ldots, b_n$ to use in some operations, and to which operations they should be provided.
- To compute the k-bit hash, the selected bits must be passed through a composition of operations.

Thus, to compute the hash, we use k holes for the output bits, where each hole is drawn from the following grammar:

$$H \quad \rightarrow \quad H_1 \,\mathsf{XOR}\, H_2 \mid H_1 \,\mathsf{AND}\, H_2 \mid H_1 \,\mathsf{OR}\, H_2 \mid \mathsf{NOT}\, H_1 \mid b_i$$

For example, one possible completion is $b_1 \ldots b_n \mapsto b_1 \ldots b_k$, e.g. take the first k bits.

3.1.3 Specifying Hash Behavior

Lastly, we must define what type of specification will be used for our synthesis problem. For simplicity, we decided on using a set of randomly generated addresses for test cases. The test cases are chosen so that there are $|C_1|$ addresses mapped to block C_1, $|C_2|$ addresses mapped to block C_2, etc. We then set up the synthesizer so that it attempts to find a function $\mathcal{H}$ so that the number of test cases with expected blocks exactly matching the output blocks of $\mathcal{H}$ is maximized. The results are shown in the evaluation section below. While this optimization objective is naive and simple, it leads to rather quick synthesis and serves as a good starting point for future study.

3.1.4 Search Strategy and Implementation

We adopted a purely logical approach for our search strategy. We encoded the synthesis problem as a constraint satisfaction problem that could be solved with off-the-shelf SMT solvers. We implemented our synthesizer in the Racket programming language using the Rosette EDSL [7].

3.2 Parameterized Hash Functions

In the previous section we looked at how to derive hash functions based on their desired characteristics. That is, we knew what the hash function should do, but not what it should look like. In this section we take the opposite tack, and directly construct a hash function

$\mathcal{H}$ that has the desired properties. In this vein, we also flip our take on $\mathcal{H}$ so that we look at it from the perspective of the caches as opposed to the memory addresses.

If we imagine each cache as a voter, with as many votes as cache lines, then a cache could cast votes on a memory address. We could then say whichever cache casts the most votes for a given address wins and is the hash of that address. This scheme is sensitive to the cache sizes from the outset. As it stands however, the cache with the most cache lines would always cast the most votes and would thus win every address. In some sense, this scheme is too sensitive to the cache sizes and we need to take a corrective action, *de-weighting* this sensitivity.

3.2.1 Caches as Voters

The maneuver we make is to arbitrarily select a distinguished cache and add a "modifier bonus" to it that increases the number of votes it can cast. For instance, we could add the number 4 to the distinguished cache, giving it a virtual 4 extra cache lines to vote with. This might make it the largest cache which would then make it the winner of the vote as well as the hash of the memory address. As long as we can remember how to pick the distinguished cache and how we derived the modifier bonus, and depending on how we choose the bonus, the larger caches should still overall win more memory addresses, and should fall into the distribution of base sizes.

Choosing the distinguished cache and modifier bonus can't be done *randomly* since that would make our hash function non-deterministic. But we can choose the distinguished cache by hashing. We do this by applying a standard, lightweight Pearson hash to the memory address, and then taking the modulus by the number of caches, to select one cache in particular. This strategy is deterministic and manages to always choose a pseudo-random cache, successfully de-focusing the largest one. A description of the algorithm is given below.

Algorithm 1 cache voting algorithm

```
Require: cache_array, a list of integers denoting corresponding cache size
Require: pearson, an impl of the 256-bit Pearson hash function
Require: maddr, a memory address
1: distinguished_tile ← pearson(maddr) % |cache_array|
2: modifier_bonus ← maddr % |cache_array|
3: cache-array[distinguished_tile] + = modifier_bonus
4: home_tile ← argmax(cache_array)
5: return home_tile
```

3.2.2 The Modifier Bonus and Implementation

For the modifier bonus, we can use some property of the memory address or the data itself. We tested 3 possibilities: memory address modulo the number of caches, sum of the positive bits in the address (popcount), and the median of the cache sizes. This weighting component is extremely important to obtaining good distributions of the data. We will see that the above examples are a good first pass, but definitely need more attention in future tests. The code was implemented in Python without any library dependencies.

4 Evaluation

4.1 Program Synthesis

To evaluate our synthesizer, we selected three different cache configurations and ran our synthesizer on a set of random test cases. For each configuration, the test cases consist of pairs of a 16-bit address and a block number appropriate for the configuration. We also experimented with hole depths of 1 and 2. The results are shown in Figure 4.

Configuration	Hole depth	Total test cases	Exact matches	Relative difference
8, 4, 4	1	64	10	25.0%
	2	64	25	21.9%
16, 8, 4, 4	1	128	15	7.81%
	2	128	17	10.9%
8, 8, 8, 8	1	128	15	50.0%
	2	128	23	25.0%

Figure 4: Cache configurations used with the synthesizer. The configuration column corresponds to the cache sizes used. Exact matches corresponds to the number of actual blocks matching expected blocks. Relative difference is the total deviation in the actual distribution of homed caches from the expected distribution of homed caches.

Note that increasing the depth tends to sometimes, but not always, decrease the relative difference rate. However, this comes at the cost of expanding the search space. There also need to be enough random test cases to cover all blocks and addresses in the configuration, or the synthesizer will risk overfitting like in machine learning approaches. For example, if all test cases end up having addresses less than some amount N, then the resulting hash function may not select the higher-valued bits of N.

This project made us realize that coming up with a good objective function and search strategy is very difficult. As can be observed in the table, the objective function of simply matching up blocks is not a good fit for the synthesis task. For example, the 8, 8, 8, 8 depth 1 configuration achieves a relative difference of 50% and would be completely outclassed by a simple hashing scheme. We also tried an objective function that minimizes the absolute difference, but we did not find a way to make it efficient enough to terminate within a reasonable amount of time. Future work could include optimizing the helper functions to be more amenable to logical encoding, devising a better objective function, and integrating machine learning techniques to improve the search strategy.

The sketch was comparably easier to construct than the objective function, but there is still room for improvement. We currently treat the sketch as a black box binary function, but we could incorporate aspects from more interesting hash functions like the parameterized hash function approach we have described in this report.

4.2 Parameterized Hash Functions

For the voting algorithm, we approached simulation by treating all of the numbers in the ranges of $[0, 2^8)$ and $[0, 2^{16})$ as memory addresses and hashing them against a simple array of 4 caches of sizes 3, 2, 2, and 1. While we tested mainly with this setup, the number of caches and their sizes are easily configurable. The graphs below show the histograms of the homed data, namely, the number of addresses in the space that were mapped to the given cache. Ideally, we expect the number of addresses in each cache to follow the 3, 2, 2, 1 distribution. The metrics chosen were the modulo scheme, the sum of positive bits, and addition of the median number of caches.

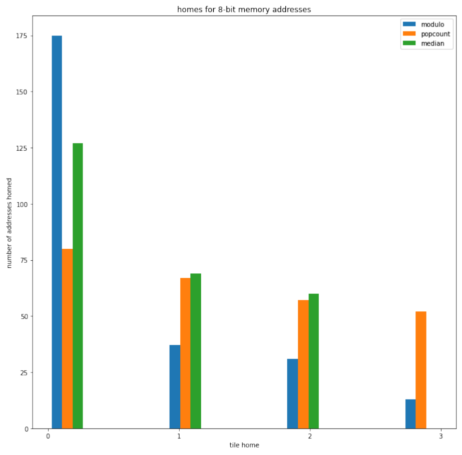

Figure 5: 256 bit memory space

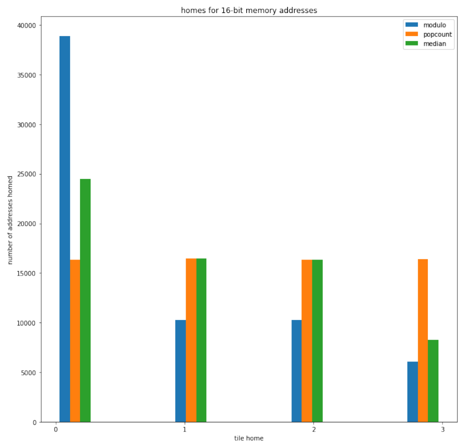

Figure 6: 65536 bit memory space

From the graphs we can see first that popcount is not a good metric for deciding homes as it collapses to the uniform distribution with enough data. Interestingly, the remaining two approaches, modulo and median, do roughly match the distribution of cache sizes, with most data going to the default largest cache, less going to the middle two largest caches, and the least going to the smallest cache. The median performs much closer to the ground truth, however it is not clear what this intuitively captures about the data. The modulo scheme seems to unduly weight the largest cache, but still roughly captures the relative caches sizes. While these weighting schemes don't quite capture the distribution exactly, it is clear that they can be modified to provide a more robust picture of the cache size distributions.

There are a number of improvements that we could make, including making sizes of the caches powers of 2, or simple arithmetic combinations of powers of 2, to mimic real world conditions. As noted previously, the weighting function is crucial to observing a smooth distribution matching the cache sizes. The options we tried should be expanded to find better matches.

Finally, the hash function we used was the Pearson function, which is traditionally based on a 256-bit table of randomized seed values. This table of seed values is implemented as a lookup table, which unfortunately, can be expensive in production many-core systems. The table can be replaced with alternative strategies, such as a subtraction of the memory address from 256. Ultimately, however, we can conclude that the above schemes, both synthesis and voting, are flexible and highly configurable approaches to parameterizing cache-based hash functions.

5 Related Work

Generally speaking, there are two categories of hashing methods: data-independent and data-dependent. Data-independent hashing methods usually generate a set of hash functions using randomization (without using any training data). In contrast, data-dependent hashing methods usually are using unsupervised, supervised or semi-supervised learning techniques to generate the results. Here we are going to explain some relevant hashing mechanisms to this report.

Operand locality: Authors in [4] proposed a mechanism for compute caches. In contrast to our problem, here the good results come out of locality of operands in the cache. Authors made some design choices in the cache organization such as defining banks that have multiple sets each and adding extra bits in address decoding for "bank bits" and "block partition" [4]. This approach can be used to remember adjacent strides and scatter them into different cache lines.

Using synthesis for a mapping algorithm: In [3] authors implemented a synthesizing methodology to map p-nested for loop algorithms into linear arrays. The method was based on a set of formal conditions on the sequential algorithm and the feasible "transformations" on them. The result was promising given the time that this paper has published.

Supervised hash algorithms:

The authors in [8] proposed a hashing method that is implemented using column generation based convex optimization. Considering a set of constraints, their proposed hashing method is capable of learning compact hash codes. Their hash functions are learned iteratively using column generation.

The authors of [6] showed that learned indexes outperforms traditional hash algorithms because utilizing the distribution of data being indexed can provide significant benefits. Figure 7 shows how resistant their approach is to hash collisions in comparison to traditional hashing.

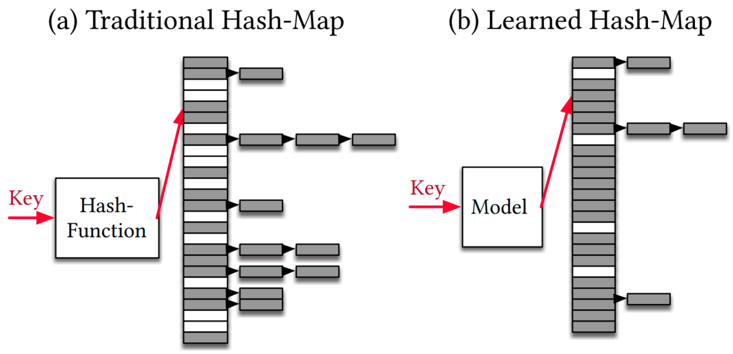

Figure 7: Traditional Hash-map vs Learned Hash-map [6]

6 Reflection

This project began as an exploration of applying tools from programming language theory to hardware architecture. The notion of a hash function for homing memory to tiles is a well-defined component of a processor and has the advantage of being relatively abstract, without requiring large amounts of hardware-specific knowledge. While we ultimately derived two interesting strategies for solving this problem, one of which was indeed based on a tool from programming language theory, the progress we made was built on a couple of weeks of background reading as well as a fair amount of trial and error.

We began by reading about caches and homing strategies in manycore designs until we felt we had a pretty good idea of what the bounds of the problem were. We also spent some time reading about hash functions, learning about their typology, including the universal, cryptographic, and hardware varieties. On the programming language front, we decided to explore a synthesis methodology, and worked to build a few examples of simple synthesis tasks over the theory of bitvectors. Throughout our investigations, we also learned about how problems similar to this were addressed in the machine learning space.

Our first few attempts at approaching the problem stalled for a few reasons. While the requirements of the problem are easy to state in prose, it was not clear how to state this in our synthesis tool. Namely, we want a hash function that distributes memory addresses to tiles according to a specific distribution in aggregate and is based on simple bitvector operations. As we went back-and-forth trying to express these requirements, we ended up tackling the problem from the analytic approach as well as the synthesis. It turned out that thinking about the problem from both ends is how we eventually were able to provide *two* distinct solutions to the problem. The code, sweat, and tears are all included in the following repo, including programs in both Racket and Python that may be explored by the interested reader: `https://github.com/atefehmohseni/CS_254_final_project`

7 Conclusion

In this study we worked on hardware hash functions to efficiently map memory addressees to caches of different sizes. We tried two approaches: program synthesis and parameterized hashing. In the first approach, we tried to sketch a hash function that applies different operators to different parts of a memory address. In the second approach we simulated a hash function as a voting scheme that maps addresses to cache lines based on bin sizes. The experimental results of both approaches were successful but there is still room for improvement. In future work it would be interesting to combine both approaches or even explore machine learning based hash functions.

References

[1] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff. Openpiton: An open source manycore research framework. ASPLOS, 2016.

[2] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[3] P. Lee and Z. M. Kedem. Synthesizing linear array algorithms from nested for loop algorithms. In *Transactions on Computers*. IEEE, 1988.

[4] A. Shaizeen, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[5] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News*, 34(5):404–415, Oct. 2006.

[6] K. Tim, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *International Conference on Management of Data*, SIGMOD'18, 2018.

[7] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[8] L. Xi, G. Lin, C. Shen, A. Hengel, and A. Dick. Learning hash functions using column generation. In *International Conference on Machine Learning*. PMLR, 2013.

[9] L. Ye, S. Kato, and M. Edahiro. Analysis of memory system of tiled many-core processors. In *IEEE Access*, 2019.