Rapid Design Space Exploration of Two-level Unified Caches

Jingyu Deng, Yun Liang, Guojie Luo, Guangyu Sun
Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
{dejiyu,ericlyun,gluo,gsun}@pku.edu.cn

Abstract—**Modern application specific system-on-chip platforms allow customization of caches. Such flexibility enables the designers to identify the suitable cache configurations through design space exploration of caches. Trace-driven simulation is widely used to obtain the cache hits and misses for design space exploration. However, simulation is normally slow. Meanwhile, as the embedded system moves toward cache hierarchies with multi-level caches, such expanded design space leads to extremely long simulation time. In this paper, we propose a rapid design space exploration technique for two-level unified caches. Given the application trace, our technique determines the cache hits and misses for multiple cache configurations in a single pass. Our exploration technique adopts a novel LRU linked list data structure, lookup tables, and search algorithms to effectively improve the exploration time. Experimental results indicate that our analysis is 7–239X times faster compared to the fastest known design space exploration technique, in estimating cache hits and misses for popular embedded benchmarks.**

I. Introduction

Caches have long been adopted to mitigate the speed disparity between fast processors and slow memories. In general, for a well-tuned cache hierarchy, most of the memory accesses can be fetched directly from caches instead of main memory by exploiting the spatial and temporal localities among the memory accesses in a program. The accesses to memory incur longer delay and much more power consumption compared to the accesses to caches. Hence, cache tuning and optimization can lead to significant performance gain and energy saving.

Modern application specific platforms allow cache customization, in particular cache design parameters. For example, soft-core processors in the FPGAs [1], [2] and application specific processors designs [3] allow cache customization at system design time. Meanwhile, this trend has also been reflected on the general purpose designs [4], [5], [6]. For example, NVIDIA's state-of-the-art Kepler architecture features with configurable L1 cache together with scratchpad memories.

The flexible cache architecture is a strength for cache optimizations, but is also a challenge for the system designers. The optimal cache configuration is application specific. Thus, system designers have to explore the entire cache design space to determine the cache hits/misses numbers for each cache configuration. The cache design parameters include the number of cache sets, the cache block/line size, and the degree of associativity. Hence, even for a single level cache, its design space consists of a large number of design points. More importantly, the design space is significantly expanded for multi-level unified caches due to the cache hierarchy and interlace of instruction and data caches. For multi-level unified caches, cache hierarchy and instruction and data caches can not be explored separately. For example, for a two-level unified cache, given N_1 L1 instruction cache configurations, N_2 L1 data cache configurations, and N_3 L2 unified cache configurations, the design space of two-level unified cache includes $N_1 \times N_2 \times N_3$ configurations in total.

The design space exploration of caches is a well studied problem. The most popular approach to obtain the cache hits/misses is trace-driven simulation [7], [8], [9], [10], [11]. The simulation based approaches take the memory access traces as inputs, and then mimic the cache behaviors for some hypothetical cache configurations, and outputs the cache hits and misses numbers. The cache hits/misses returned by the simulation are exact, but simulation could be slow especially when the design space is large. As an alternative to trace-driven simulation, analytical approaches have been proposed [12], [13], [14]. The analytical approaches use mathematical models to estimate the cache behavior. In theory, analytical approaches run fast and provide additional performance hints to the designers. However, they may fail to find the optimal cache configuration due to the inaccuracy of the models.

In this paper, we present a rapid design space exploration technique for two-level unified caches. Two-level caches can be designed to be either inclusive or exclusive. Compared to inclusive caches, exclusive caches have larger effective cache storage. Exclusive caches have been utilized in modern commercial processors including AMD Athlon [15] and ARM Cortex-A9 processor [16]. Thus, we focus on the exclusive caches in this work. Recently, Zang and Ross presented the first simulation framework (U-SpaCS) for two-level unified exclusive caches [11]. Their technique based on least recently used (LRU) stack achieves high speedup compared to the non-optimized trace-driven simulator Dinero [17]. However, there are two major drawbacks in their work. First, their technique is inefficient in terms of address lookup as it has to scan the entire LRU stack. Second, the L2 analysis is inefficient as it contains a unified L2 stack for all the cache configurations and compares the same copies of cache blocks multiple times.

Our technique adopts novel LRU linked lists, lookup tables, and fast search algorithms to improve the simulation time. More clearly, we use LRU linked list to store the cache contents for multiple cache configurations and use lookup tables to compute cache conflicts efficiently. Furthermore, for efficient L2 cache analysis, we use separate LRU linked lists for different L1 cache configurations and sort the L2 cache blocks based on their timestamp. This optimization ensures that one cache block is only compared once. We show that compared to the state-of-the-art two-level unified cache simulation U-SpaCS [11], our technique achieves 7–239X speedup in the design space exploration.

II. Analysis Framework

Cache design involves with several parameters: cache line size (L), number of cache sets (N), associativity (A), and replacement policy. Then, the cache size is $L \times N \times A$. The

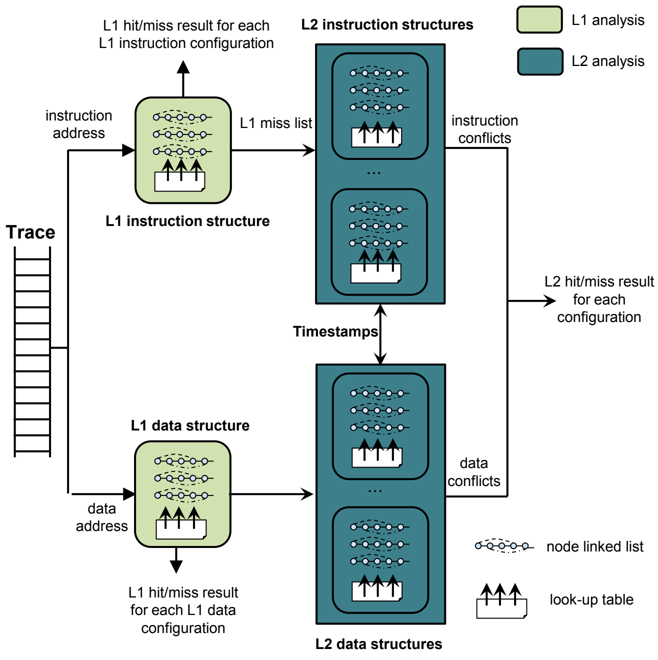

Fig. 1. Analysis Framework.

design space of a cache memory is $\{C(L,N,A)|L_{min} \leq L \leq L_{max}; N_{min} \leq N \leq N_{max}; A_{min} \leq A \leq A_{max};\}$, where L_{min} (L_{max}) is the minimal (maximal) line size, N_{min} (N_{max}) is the minimal (maximal) number of cache sets, and A_{min} (A_{max}) is the minimal (maximal) associativity, respectively. We define the design space of L1 instruction cache, L1 data cache and L2 unified cache as D^{L1}_{inst}, D^{L1}_{data}, and $D^{L2}_{unified}$, respectively. We consider LRU replacement policy. Under LRU replacement policy, we determine the cache hit or miss for an access by comparing its cache block conflicts with the cache associativity. The cache block conflicts of an access m is the number of cache blocks that are mapped to same cache set as m but more recently accessed than m.

In this work, we target a two-level unified cache hierarchy that consists of configurable L1 instruction and data cache and L2 unified cache. Each cache can vary its cache line size, number of cache sets, and associativity within its design space. L1 cache (instruction and data) and L2 cache have to be explored together due to the interlace of instruction and data caches through unified L2 cache. Thus, the total number of cache configurations of the two-level unified cache hierarchy is $|D^{L1}_{inst}| \times |D^{L1}_{data}| \times |D^{L2}_{unified}|$. For exclusive cache hierarchy, the L2 cache is filled with cache lines that are evicted from the L1 cache. More clearly, upon a L1 cache miss, the evicted cache line from L1 cache will be moved to L2 cache; if the requested cache line can be found in L2 cache, then it will be moved from L2 cache to L1 cache. Thus, cache lines are frequently moved between L1 and L2 caches. Using a common line size for L1 and L2 caches significantly simplifies the data movement and hardware design [18]. Thus, we consider L1 and L2 caches with the same cache line size. Finally, for a practical design, L2 cache should be larger than L1 cache.

Figure 1 presents the framework of our simulation framework. Our framework consists of two phases: L1 analysis and L2 analysis phase, respectively. The inputs to our framework is the memory address trace. Each address in the trace is associated with its access type (instruction or data). Depending on the access type, it either goes to L1 instruction analysis or L1 data analysis. The output of L1 analysis is the cache hits/misses for all L1 cache configurations. If the L1 access incurs a cache miss, then it proceeds to the L2 analysis. The output of the L2 analysis is the cache hits/misses for all L2 cache configurations.

For each level of cache, we use a group of LRU linked lists to store the cache contents for multiple cache configurations. We also use lookup tables to speedup the search of cache conflicts. For L1 cache analysis, the instruction and data can be separated. However, this is not the case for L2 cache analysis as instruction and data are stored together. For L2 analysis, we first compute the number of cache block conflicts from one type of access (either instruction or data), and then use the timestamp of current access (order in the trace) to determine the number of cache block conflicts from the other type of access. Finally, we sum the cache block conflicts together and determine the cache behavior.

III. Data Structures

To speedup the design space exploration, it is critical to design efficient data structures for data storage. LRU stack is used in [11]. However, for cache configurations with different cache sets, multiple LRU stacks are required [11]. This wastes storage and exploration time. Thus, we propose to use **LRU linked list**. The nodes in the linked list store not only the cache blocks but a few pointers that link them with the other nodes as shown in Figure 2. The cache blocks that appear in multiple cache configurations with different cache sets are only stored once. Through pointers, we can find the cache contents for multiple cache configurations with different cache sets. Thus, LRU linked list is a compact and efficient data structure compared to LRU stack. For L1 and L2 caches, we use a group of linked lists as shown in Figure 1. More concretely, for L1 cache, we use one group for instruction and data cache separately. L2 analysis depends on the results of L1 analysis. So, for L2 cache contents, we create one group of linked list for each L1 cache configuration. For each level of cache, the number of LRU linked list in the group is determined by the minimum number of cache set in the design space. For example, let us assume the minimum number of cache sets of L1 instruction cache is $N^{L1\_inst}_{min}$. Then, the group for L1 cache contains $N^{L1\_inst}_{min}$ LRU linked lists. Given a cache block m, it is mapped to the linked list based on its index ($m\%N^{L1\_inst}_{min}$).

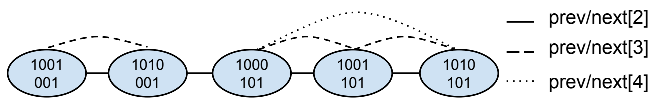

Fig. 2. LRU linked list.

Every node in the linked list is associated with a timestamp (order in the trace). We order the nodes in the linked list based on the LRU replacement policy. The leftmost (rightmost) node is the most (least) recently accessed node. More clearly, given a cache block access, if it can be found in the linked list (accessed before), then it will be moved to the head (leftmost) of the linked list; otherwise, a new node will be created and inserted to the head of the linked list.

A linked list stores the cache contents for multiple cache configurations with varying number of cache sets and associativity. Cache configurations with different block sizes are stored separately. For each linked list, we use an array of pointers to link the nodes together for different number of cache sets. For each node, pointers $prev[i]$ and $next[i]$ point to the previous and next node for 2^i number of cache sets, respectively. If the pointer $prev[]$ ($next[]$) is null, then it means the current node is the first (last) node. Figure 2 shows an example of $prev$ and $next$ pointers. To determine the cache behavior (hit or miss), we need to count the conflicting cache blocks that are more recently accessed than the current one. If the conflicts is less than the cache associativity, then it is a cache hit; otherwise, it is a cache miss. To speedup the search for conflicts, we define lookup tables $evict[][][]$. $evict[i][j][k]$ points to the 2^kth node in the cache set j for the cache with 2^i number of cache sets. Through $evict[][][]$ table, we can easily compute the cache block conflicts for different associativity, and then compare their timestamp with the current node to determine its cache behavior. Finally, we define insert, shift and remove operations to maintain the $prev[]$, $next[]$, and $evict[][][]$ data structures. Let us use insert operation as an example. Insert operation inserts a new node into the linked list. It first locates the head of the related linked list and then links it with the previous head of the linked list. All the nodes in the linked list are pushed backward by the insert operation. Thus, we have to update the $evict[][][]$ table. For example, we need to update the $evict[i][j][k]$ to its previous node. This is assisted by using the $prev[]$ pointer. Similarly, we maintain the data structures for shift and remove operations.

IV. Design Space Exploration

In this section, we will detail our L1 and L2 analysis algorithm. Given an address, it starts with L1 analysis and proceeds to L2 analysis if it incurs a L1 cache miss (Figure 1). In the following, we assume the access address is an instruction. The processes of instruction and data access are similar.

A. L1 analysis algorithm

Given an address, L1 analysis returns the cache hit or miss for all the cache configurations. If the current address is not accessed before, then it guarantees a cache miss for all the L1 and L2 cache configurations. For this case, we create a new node and insert it to the head of the linked list. If the current address is accessed before, then we determine its cache behavior using $evict[][][]$ table. We enumerate different cache configurations and obtain the timestamp using $evict[][][]$ table and then compare it with the timestamp of the current address. If the timestamp of current address is smaller, then it is a cache miss; otherwise, it is a cache hit.

B. L2 analysis algorithm

For each possible cache set number in L2 cache, L2 analysis involves four steps: (1) computing the cache block conflicts from instruction accesses (Algorithm 1) (2) sorting (3) computing the cache block conflicts from data accesses (Algorithm 2) (4) computing the total cache block conflicts and determine cache hit or miss. For L2 unified cache, it stores both instruction and data accesses. Thus, we have to consider the cache block conflicts from both of them. In the first step, we compute the conflicts from instruction accesses and record the timestamp of current access for every L1 instruction cache miss configuration. Then, we sort the L1 instruction cache miss configurations in descending order based on their timestamp. In the third step, we count the cache block conflicts from the data accesses. Finally, we can determine the cache behavior by comparing the cache associativity with the sum of cache block conflicts from both instruction and data accesses. The LRU Linked list are maintained in LRU order. Thus, after sorting, we can walk through the linked list for the data accesses and collect the cache block conflicts in one scan. The sorting step reduces the complexity from $O(n^2)$ to $O(n)$.

Algorithm 1 Compute cache conflicts from instruction accesses

```
$N\ is\ the\ number\ of\ sets\ in\ L2$;
$i = log_2 N$;
for each L1 inst cache miss config $Ci$ do
    $Let\ x\ be\ the\ current\ node\ in\ the\ linked\ list$;
    $timestamp[Ci] = x.insert\_time$;
    $conf\_i[Ci] = 0$;
    $tmp = x.prev[i]$;
    while $tmp \neq Null$ do
        $conf\_i[Ci] = conf\_i[Ci] + 1$;
        $tmp = tmp.prev[i]$;
        if $conf\_i[Ci] >= Max\_Associativity$ then
            $break$;
        end if
    end while
    $remove\ x$;
end for
```

Algorithm 1 describes the details of the computation of cache block conflicts from instruction accesses. In this step, we output two arrays for all L1 instruction cache miss configurations. $conf\_i$ array records the number of instruction cache block conflicts for different L1 instruction cache configurations. $timestamp$ array records L2 timestamp for different L1 instruction cache configurations. For each L1 instruction cache miss configuration Ci, we find the current access x in the corresponding L2 linked list group. Then, we record the timestamp of this configuration and compute the cache conflicts by traversing from x to the head.

Algorithm 2 Compute cache conflicts from data accesses

```
$i = log_2 N$;
$Let\ idx\ be\ the\ index\ of\ cache\ set$;
for each L1 data cache config $Cd$ do
    $conf\_d = 0$;
    $tmp = evict[i][idx][0]; /*TheHeadOfList*/$
    for each L1 inst cache miss config $Ci$ do
        while $tmp.insert\_time > timestamp[Ci]$ do
            $conf\_d = conf\_d + 1$;
            $tmp = tmp.next[i]$;
            if $conf\_d >= Max\_Associativity$ then
                $break$;
            end if
        end while
    end for
end for
```

Algorithm 2 describes the details of the computation of cache block conflicts from data accesses. Let $conf\_d$ be the number of data accesses conflicts. We first enumerate L1 data cache configurations. For each L1 instruction cache miss configuration, we determine $conf\_d$ by comparing the timestamp of the nodes in the current linked list with the timestamp of the instruction cache configuration. Since all timestamps of the nodes in the list are in descending order, we can compute $conf\_d$ for each timestamp by scanning the list only once. Starting from the head node of the list, we increase $conf\_d$ by one after every hop. We continue this until we reach a node which has smaller timestamp and compute the total conflicts.

V. Experiments

We evaluate our framework by comparing with the state-of-the-art two-level unified cache simulation technique U-SpaCS [11]. Both our framework and U-SpaCS target two-level unified exclusive caches and both can simulate multiple cache configurations in a single pass. All the experiments are performed on a Intel Xeon 2.40GHz CPU with 16GB memory.

We try a set of embedded applications from MiBench benchmark suite [19] and larger general-purpose applications from SPEC2000. The address traces of these benchmarks are generated using SimpleScalar [20]. For some SPEC2000 benchmarks, the address trace could be extremely large. For those cases, we use a fraction of the trace (the first 50M references) for evaluation. For our design space, we vary the block size of the cache hierarchy from 8 to 64 bytes. For L1 instruction and data cache, we vary the number of cache sets from 4 to 64 and associativity from 1 to 8; for L2 unified cache, we vary the number of cache sets from 4 to 256 and associativity from 1 to 16. So, the L1 cache size is up to 64K and the L2 cache size is up to 256K. Finally, for a realistic cache design, the L2 cache should be larger than the L1 cache. Given this constraint, there are totally 32,200 two-level unified cache configurations in our design space.

TABLE I. Runtime comparison of U-SPaCS and our analysis.

Benchmark	Our (sec)	U-SPaCS (sec)	Speedup
basicmath	139,912	1,582,995	11.31
crc32	10,523	73,938	7.03
dijkstra	25,517	699,355	27.41
fft	80,582	2,857,595	35.46
ispell	18,268	222,168	12.16
jpeg	24,081	889,832	36.95
patricia	151,631	5,830,628	38.45
qsort	146,151	3,941,893	26.97
rijndael	64,320	629,678	9.79
sha	7,110	166,105	23.36
stringsearch	3,617	61,439	16.99
susan	55,300	13,224,244	239.14
bzip2	32,333	6,844,510	211.69
gzip	42,985	5,847,761	136.04
swim	33,641	1,270,129	37.76
average	55,731.4	2,942,818	52.8

We first compare the cache hits/misses of our technique and U-SpaCS for both L1 and L2 caches. The cache statistics are exactly the same. We also verify the numbers with the functional cache simulator sim-cache in Simplescalar.

The simulation time are shown in Table I. Our framework is significantly faster (7 - 239X speedup) compared to U-SpaCS. On average, our technique is about 53X faster than U-SpaCS. We also notice that the speedup varies across different applications. In fact, the achieved speedup depends on the number of unique cache blocks. In U-SpaCS simulation, its efficiency depends on the LRU stack size (the number of unique cache blocks). Given an application with large working set, U-SpaCS implementation wastes significant time in LRU stack scan and redundant cache block comparison. In contrast, our technique leverages the linked list with pointers and avoids redundant comparison. For example, for the larger benchmarks such as *bzip2* and *susan*, which contain a large number of cache blocks, the speedup of our technique is above 200X.

VI. Conclusion

In this paper, we develop a rapid design space exploration technique for two-level unified caches. Our technique adopts a novel LRU linked list data structure, lookup tables, and search algorithms to effectively improve the exploration time. Experimental results indicate that our analysis is up to 239X times faster (average 53X) compared to the fastest known cache design space exploration technique.

VII. Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (No. 61300005).

References

[1] Microblaze processor. Xilinx, Microblze processor.http://www.xilinx.com/tools/microblaze.htm.

[2] Nios processor. Altera, Nois Embedded Processor System.http://www.altera.com/devices/processor/nios2/ni2-index.html.

[3] Tensilica, Xtensa processor. http://www.tensilica.com.

[4] NVIDIA. Kepler GPUs www.nvidia.com/object/nvidia-kepler.html.

[5] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO*, pages 248–259, 1999.

[6] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. *SIGARCH Comput. Archit. News*, 31(2):136–146, 2003.

[7] M. S. Haque, A. Janapsatya, and S. Parameswaran. Susesim: a fast simulation strategy to find optimal l1 cache configuration for embedded systems. In *CODES+ISSS*, pages 295–304, 2009.

[8] R. L. Mattson et al. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[9] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 13(1), 1995.

[10] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170, 1997.

[11] W. Zang and A. Gordon-Ross. A Single-Pass Cache Simulation Methodology for Two-level Unified Caches. In *ISPASS*, 2012.

[12] A. Ghosh and T. Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):419–440, 2004.

[13] Y. Liang and T. Mitra. An analytical approach for fast and accurate design space exploration of instruction caches. *ACM Trans. Embed. Comput. Syst.*, 13(3):43:1–43:29, December 2013.

[14] Y. Liang and T. Mitra. Static analysis for fast and accurate design space exploration of caches. CODES+ISSS, pages 103–108, 2008.

[15] AMD Athlon Processor. http://www.amd.com/us/products/desktop/processors/athlon/Pages/AMD-athlon-processor-for-desktop.aspx.

[16] ARM Cortex-A9 Processor. http://www.amd.com/us/products/desktop/processors/athlon/Pages/AMD-athlon-processor-for-desktop.aspx.

[17] J. Edler and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/~markhill/DineroIV/.

[18] Y. Zheng, B. T. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *ISPASS*, pages 89–96, 2004.

[19] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization*, pages 3–14, 2001.

[20] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59 – 67, 2002.