Randomizing Cache Index Generation for FPGA Implementation

Kentaro Hayashi Nagisa Ishiura

School of Science and Technology, Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, 669-1337, Japan

***Abstract*—This paper presents an FPGA (field programmable gate array) oriented cache index generation method to enhance cache hit ratio for embedded systems. Randomizing or hashed cache indexes are known to improve cache hit ratio, especially on direct mapping caches. In the context of embedded systems design, several researches have been conducted to find an appropriate hash function that reduces cache misses for a given program. Hash functions using 2-input exclusive-or gates are easily computed by hardware and yet achieve considerable cache miss reduction. However, they result in a waste of hardware resources if implemented by LUT-based FPGAs. Thus, in this paper, hash functions using LUTs (look-up tables) instead of XOR gates are proposed. Given a memory access trace, the best combination of truth tables and inputs for LUTs is searched by simulated annealing. Experiments on 12 SPEC CPU benchmarks shows that this method enhances the cache hit ratio over the XOR based method.**

I. Introduction

As the processor-memory gap is widening, cache memories are becoming more and more important, even in embedded systems. Since a subtle improvement on cache hit ratio leads to substantial speed-up of program execution, reduction of cache misses is one of the critical issues in system design.

In general purpose computing systems, hardware rather than software is altered to enhance cache hit ratio. For example, the starting addresses of the basic blocks in a given program are adjusted to reduce cache misses [1]. On the other hand, in embedded systems, where only a single program or a limited set of programs are executed, adjustment can be made on the hardware side.

In literature [2], a cache configuration, a combination of the total size (capacity) and the block size of the cache, to minimize cache misses are searched for a given program. Cache size can also be adjusted to improve WCET (worst case execution time) [3]. Another hardware oriented approach is to alter cache index functions. In the classical scheme, the lower c bits of the block numbers (where 2^c is the maximum number of the blocks accommodated in the cache) are used for the cache index. In general, however, any c bits computed from block numbers may serve as the cache index. By choosing a mapping that uniformly distributes resulting indexes, cache misses due to conflicts may be reduced [5], [6].

Vandierendonck [7] has proposed a randomizing index generation based on 2-input exclusive or (XOR) gates, which reduced cache conflict misses by 30% to 60%. Although it is easily implemented by hardware, it is not necessarily efficient if FPGA implementation is assumed, because a 2-input XOR function takes a single LUT (look-up table), though it can implement any logic function (typically of 4 or 5 inputs).

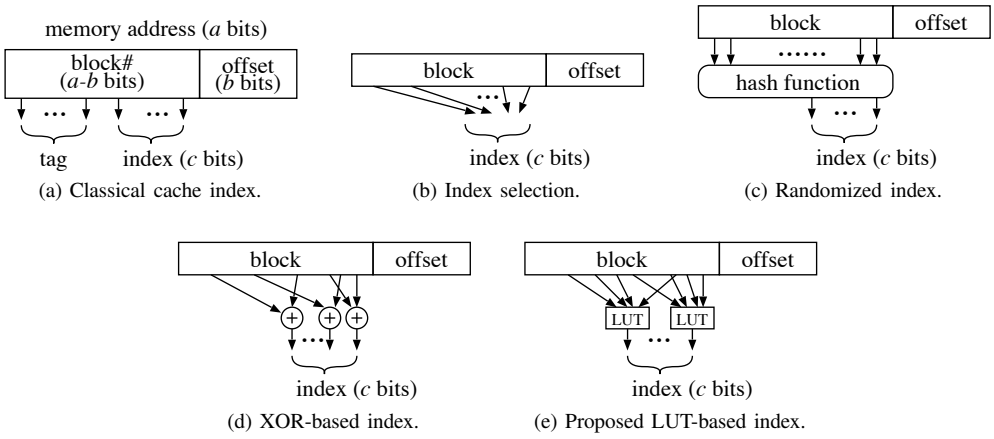
Thus, in this paper, randomizing cache index generation using LUTs is proposed. Each of the index bits is computed by an LUT, whose logic function and input bits are selected to enhance cache hit ratio. Simulated annealing based on cache simulation is employed to find optimal solution. Experiments on 12 SPEC CPU benchmarks shows that this method slightly enhances the cache hit ratio over the XOR based method.

II. Cache Index Generation

We deal with design of cache memories in embedded systems where hardware may be adjusted to a given program running on the system. We focus on generation of cache index bits in this paper.

We assume a direct mapping cache memory in this paper. Let the logical memory address space be 2^a bytes and hence the memory address be specified with a bits. Let the cache block size be 2^b bytes and the cache can accommodate 2^c blocks. Then, as shown in Fig. 1 (a), the lower b bits of the a address bits represents offset. The upper $a-b$ bits represents the block number and c bits from the block number serve as a cache index.

In a classical scheme, the lower c bits of the block number are used for the cache index (Fig. 1 (a)). However, any c bits from the block number may serve as the cache index (Fig. 1 (b)). By choosing favorable c bits for a given program, cache conflict can be reduced [4]. This idea is further generalized



(a) Classical cache index. (b) Index selection. (c) Randomized index.

(d) XOR-based index. (e) Proposed LUT-based index.

Fig. 1. Cache index generation.

so that c bits are computed by a hash function from block numbers (Fig. 1 (c)). This randomizes cache indexes which is expected to reduce cache conflicts.

The hash function must be computed as fast as possible. Vandierendonck [7] has proposed a randomizing index generation based on 2-input exclusive or (XOR) gates. The 2 inputs for each of c XOR gates are selected to minimize the cache misses for a given program.

III. LUT-Based Cache Index Generation

A. Overview

Although the 2-input XOR method needs only c XOR gates, each XOR gate consumes an LUT if implemented by an FPGA. Since, typical LUTs implement logic functions of 4 or 5 inputs, we may use this freedom to further enhance cache hit ratio.

Fig. 1 (e) is the hardware configuration to generate cache index proposed in this paper. Each of the c bits is computed by an LUT whose truth table and input bits are chosen so that cache misses are minimized. The optimal combination of the truth tables and input bits are searched by simulated annealing.

B. Logic Functions for LUTs

We assume in this paper a memory-based LUT with m inputs, which can compute any m-input logic function by configuring its truth table consisting of 2^m bits. Basically, we choose one of the 2^{2^m} logic functions for each LUT. However, this might lead to huge search space, so we also consider two limited classes of logic functions as well.

One is the class of logic functions whose onset size is 2^{m-1}, where the onset of m-input logic function f is defined as $\{x \in \{0,1\}^m \mid f(x) = 1\}$. We hereafter refer to a function of this class as a "1/2-function." Since the output value of a 1/2-function becomes 1 with a probability of 1/2, each of the c index bits would equally partition the block number space. Moreover, since the m-input XOR function is a 1/2-function, this is a natural extension of the 2-XOR method.

The other class of logic functions we consider in this paper is the m-input symmetric functions, which are defined as logic functions whose output values for any input vector are the same as their values for any permutation of that input vector. This is also an extension of the 2-XOR method, for the m-input XOR function is a symmetric function.

C. Inputs to Each LUT

The m inputs to each LUT are chosen from bits representing block numbers. Let the j-th input of LUT ($j = 0, 1, \cdots, m-1$) is taken from the i_j-th bit of the block number ($i_j = 0, 1, \cdots, a-b-1$). Then we impose $i_0 < i_1 < \cdots < i_{m-1}$. For example, in the case of $m = 4$, we allow $(2, 5, 9, 13)$ but not $(3, 3, 5, 7)$ nor $(4, 8, 5, 10)$. This is because different input permutations are already incorporated in the logic functions.

IV. Searching for Optimal Configuration by Simulated Annealing

Given a trace, a sequence of the memory addresses accessed by a program, we try to find a combination of logic functions and inputs for the c LUTs that minimizes cache misses. In this paper, we employ simulated annealing for the search.

The truth table of the function of an m-input LUT is represented by a bit vector of size 2^m. The m inputs to the each LUT is also represented by a bit vector of length $a-b$ whose m bits are ones and the other bits are zeros. Namely, a state in our simulated annealing is represented by a bit vector of size $m \cdot (2^m + (a-b))$.

The objective function, to be minimized, for a state is the cache miss counts for a given trace. This is computed by simulating the behavior of the cache

TABLE I
EXPERIMENTAL RESULT.

benchmark name	trace size	cache hit rate				
		classical	XOR [7]	LUT-G	LUT-H	LUT-S
astar	6,763	26.90%	72.29%	72.91%	72.56%	72.36%
bwaves	10,005	40.15%	75.99%	75.88%	75.92%	75.83%
gamess	9,715	40.48%	76.09%	76.44%	76.14%	76.28%
hmmer	5,278	28.63%	69.06%	69.19%	69.46%	69.34%
lbm	7,865	22.11%	75.49%	76.55%	76.62%	76.64%
libquantum	3,607	24.20%	70.58%	71.06%	71.58%	71.19%
namd	8,377	30.15%	73.56%	73.61%	73.62%	74.05%
povray	5,695	57.54%	81.93%	81.90%	81.84%	82.00%
specrand	3,086	22.62%	72.36%	72.65%	72.91%	72.55%
wrf	12,540	49.08%	79.15%	79.23%	79.29%	79.34%
zeusmp	10,431	39.66%	75.46%	75.82%	75.63%	75.67%
GemsFDTD	9,701	40.53%	76.06%	76.07%	76.13%	76.27%
average	6,015	33.60%	74.75%	75.03%	75.07%	75.05%
average CPU [s]		—	23.51	46.42	46.00	46.31

Intel Core i5-4310U 2.00GHz with 3.8GB RAM

for the given trace. It takes time proportional to the size of the trace.

The initial state is determined randomly. A neighbour state of a state is obtained by changing either the truth table or the inputs of one of the LUTs. A truth table may be replaced by any other truth table. If 1/2-functions or symmetric functions are assumed, the replacement is done within the same class of functions. On the other hand, as for the inputs, a change is allowed only on one of the m inputs. In other words, the hamming distance of the two vectors to represent the inputs differ by 2.

V. EXPERIMENTAL RESULT

A search program was implemented in C language which ran on Linux, Mac OSX, etc. Experiments were conducted on 12 benchmarks from SPEC CPU. The programs were compiled into x86 binaries and the traces of the instruction addresses were captured. Assumed cache block size was 2^4 bytes and cache size was 2^3 blocks. The initial temperature, the cooling rate, the final temperature were 1.0, 0.9, and 0.01, respectively, and the iteration at each temperature was set to 1,000.

TABLE I shows the result of an experiment, which lists the cache hit ratio for 12 benchmarks from SPEC CPU, average hit ratio and average CPU time spent for the search. Row "trace size" shows the number of memory accesses. Rows "classical" and "XOR" are of the conventional methods, while rows "LUT-G", "LUT-H", and "LUT-S" are of the proposed methods with general logic functions, 1/2 functions, and symmetric functions. Compared with the classical method, randomized index drastically enhances the cache hit ratio. Proposed methods further improves the hit ratio over the XOR-based method [7], if not significantly. The CPU time necessary for the search was as much as twice of the XOR, which we consider acceptable.

VI. CONCLUSION

We have presented an FPGA oriented randomizing cache index generation method to enhance hit ratio. It was confirmed that there was some room to improve hit ratio by replacing 2-input XOR gates by LUTs. It is a future work to revise details of the search procedure and to conduct experiments on a lot more instances and different settings.

ACKNOWLEDGMENT

Authors would like to express our thanks to all the members of Ishiura Laboratories of Kwansei Gakuin University for their advice and discussion for this work.

REFERENCES

[1] H. Tomiyama and H. Yasuura: "Code Placement Techniques for Cache Miss Rate Reduction," *ACM TODAES*, vol. 2, no. 4, pp. 410–429 (Oct. 1997).

[2] Y. Afek, D. Dice, A. Morrison: "Cache Index-Aware Memory Allocation," in *Proc. International Symposium on Memory Management (ISMM 2011)*, pp. 55–64 (June 2011).

[3] H. Falk and H. Kotthaus: "WCET-Driven Cache-Aware Code Positioning," in *Proc. CASES 2011*, pp. 145–154 (Oct. 2011).

[4] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, J. M. Garca: "ASCIB: Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses, " in *Proc. ISLPED 2012* pp. 51–56 (July 2012).

[5] A. J. Smith: "Cache Memories," *ACM Computing Surveys*, vol. 14, issue 3, pp. 473-530 (Sept. 1982).

[6] N. Topham and A. González: "Randomized Cache Placement for Eliminating Conflicts," *IEEE Trans. Computers*, vol. 48, no. 2, pp. 185–192 (Feb. 1999).

[7] H. Vandierendonck: "Application-Specific Reconfigurable XOR-Indexing to Eliminate Cache Conflict Misses," in *Proc. DATE 2006*, pp. 357–362 (Mar. 2006).