XOR-Based Hash Functions

Hans Vandierendonck, *Member*, *IEEE*, and Koen De Bosschere, *Member*, *IEEE Computer Society*

Abstract—Bank conflicts can severely reduce the bandwidth of an interleaved multibank memory and conflict misses increase the miss rate of a cache or a predictor. Both occurrences are manifestations of the same problem: Objects which should be mapped to different indices are accidentally mapped to the same index. Suitable chosen hash functions can avoid conflicts in each of these situations by mapping the most frequently occurring patterns conflict-free. A particularly interesting class of hash functions are the XOR-based hash functions, which compute each set index bit as the exclusive-or of a subset of the address bits. When implementing an XOR-based hash function, it is extremely important to understand what patterns are mapped conflict-free and how a hash function can be constructed to map the most frequently occurring patterns without conflicts. Hereto, this paper presents two ways to reason about hash functions: by their null space and by their column space. The null space helps to quickly determine whether a pattern is mapped conflict-free. The column space is more useful for other purposes, e.g., to reduce the fan-in of the XOR-gates without introducing conflicts or to evaluate interbank dispersion in skewed-associative caches. Examples illustrate how these ideas can be applied to construct conflict-free hash functions.

Index Terms—XOR-based hash function, conflict-free mapping, null space, column space, skewed-associative cache, interbank dispersion.

1 Introduction

Hash functions are used in processors to increase the bandwidth of an interleaved multibank memory or to improve the utilization of a prediction table or of a cache. In each of these situations, conflicts can significantly hamper performance when they occur. However, they are easily avoided by means of a properly chosen hash function.

The easiest hash function to implement is the modulo function. In hardware, this hash function boils down to selecting the least significant bits of, e.g., an address. This hash function creates many conflicts for a number of frequently occurring access patterns, such as large power-of-2 strides. These conflicts are easily avoided by using a hash function that maps the elements of the frequent patterns without conflicts.

What constitutes a good hash function depends on the application, i.e., hash functions that perform well for interleaved multibank memories differ from hash functions that perform well for branch predictors. In any application, good hash functions have to perform well on two criteria. First, there is the added latency of computing the hash value. A latency of a few clock cycles is usually not acceptable for branch predictors and level-1 caches, but it may be acceptable for level-2 caches and interleaved memories. Second, the hash function must succeed in spreading the most frequently occurring patterns over all indices. This in turn depends on whether one is looking into branch prediction versus data memory accesses, but it may also depend on the executed programs and their input data.

This paper focuses on XOR-based hash functions, which compute each set index bit (or bank number bit) as the exclusive or (XOR) of a subset of the bits in the address. XOR-based hash functions have the benefit that the set index can be computed with a low latency (a multi-input XOR-gate) [1]. Furthermore, they perform well for all applications, ranging from interleaved memories to branch predictors. Understanding this type of functions is widely useful.

The goal of this paper is to provide information that is helpful in designing or selecting hash functions. We believe that the presented techniques will lower the barrier to reason about hash functions and that they will help designers to construct functions with desired properties. To achieve this goal, it is important to have a good mathematical representation of the hash function: Changes to the hash function should translate into changes in the number of conflicts. The commonly used matrix representation is not very useful in this respect. It represents a hash function as a binary matrix. The bit on row i and column j is 1 when the ith bit of the address is included in the jth bit of the set index. However, it is nearly incomprehensible how changing a bit in this matrix translates in a change in conflicts. We present two alternative representations: the null space and the column space of the matrix. The null space is the set of addresses that are mapped to the zero index. It is shown that making changes to this set does correlate with changes in conflict misses. A number of examples are given to show how the concept of null space simplifies the task of constructing a hash function that maps specific patterns without conflicts.

The second representation for hash functions discussed in this paper is the column space. The column space is the set of all linear combinations of the columns of the matrix. In some cases, it is easier to work with the column space than with the null space, e.g., when minimizing the fan-in of the XOR-gates without introducing conflicts.

• *The authors are with the Department of Electronics and Information Systems, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium. E-mail: {hans.vandierendonck, kdb}@elis.UGhent.be.*

Manuscript received 9 June 2004; revised 22 Dec. 2004; accepted 1 Mar. 2005; published online 16 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0195-0604.

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

The remainder of this paper is not included as this paper is copyrighted material. If you wish to obtain an electronic version of this paper, please send an email to bib@elis.UGent.be with a request for publication P105.075.pdf.