Implementing Database Operations Using SIMD Instructions

Jingren Zhou
Columbia University
jrzhou@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

ABSTRACT

Modern CPUs have instructions that allow basic operations to be performed on several data elements in parallel. These instructions are called SIMD instructions, since they apply a single instruction to multiple data elements. SIMD technology was initially built into commodity processors in order to accelerate the performance of multimedia applications. SIMD instructions provide new opportunities for database engine design and implementation. We study various kinds of operations in a database context, and show how the inner loop of the operations can be accelerated using SIMD instructions. The use of SIMD instructions has two immediate performance benefits: It allows a degree of parallelism, so that many operands can be processed at once. It also often leads to the elimination of conditional branch instructions, reducing branch mispredictions.

We consider the most important database operations, including sequential scans, aggregation, index operations, and joins. We present techniques for implementing these using SIMD instructions. We show that there are significant benefits in redesigning traditional query processing algorithms so that they can make better use of SIMD technology. Our study shows that using a SIMD parallelism of four, the CPU time for the new algorithms is from 10% to more than four times less than for the traditional algorithms. Superlinear speedups are obtained as a result of the elimination of branch misprediction effects.

1. INTRODUCTION

Microprocessor performance has experienced tremendous improvements over the past decades. Multiple execution pipelines and speculative execution in modern CPUs provide an ever increasing number of inter-stage and intra-stage parallel execution opportunities. Another trend is the increasing availability of intra-instruction parallel execution provided by single-instruction-multiple-data (SIMD) tech-

nology. SIMD instructions reduce compute-intensive loops by consuming more data per instruction.

SIMD instructions were designed to accelerate the performance of applications such as motion video, real-time physics and graphics. Such applications perform repetitive operations on large arrays of numbers. Database engines also have this apparent characteristic of applying repetitive operations to a long sequence of records. Now that the CPU and memory performance of database systems has become the performance bottleneck for certain database applications [4, 11, 28], there is renewed interest in more fully utilizing available architectures. For example, there has been much recent work on CPU cache behavior in database systems [3, 6, 11, 18, 21, 22].

In this paper, we argue that SIMD technology provides new opportunities for database engine design and implementation. To the best of our knowledge, current database systems make little or no use of SIMD features. We study various kinds of operations in a database context, and show how the inner loop of the operations can be accelerated using SIMD instructions.

We have implemented various inner loop algorithms, and have evaluated their performance on a Pentium 4 machine. The Pentium 4 has a SIMD instruction set that supports SIMD operations using up to 128-bit registers. Other architectures have similar SIMD instruction sets that support processing of several data elements in parallel, and the techniques presented here are applicable to those architectures too. We focus on a Pentium because the SIMD instructions are among the most powerful of mainstream commodity processors, including 128-bit SIMD registers and floating point SIMD operations. Further, these SIMD instructions will be supported on Intel's 64-bit IA64 platforms, including the Itanium processor [16].

The obvious potential gain of SIMD instructions is parallelism. If we can process S elements at a time, we might expect to get close to a speedup of S. A less obvious benefit of using SIMD instructions is that in many cases we can avoid conditional branch instructions by doing arithmetic on the results of the SIMD operations. Conditional branch instructions are problematic in modern pipelined architectures, because if they are mispredicted, the instruction pipeline must be flushed and various other bookkeeping needs to be done to ensure consistent operation. The overhead of a branch misprediction can be significant. By avoiding such mispredictions, we sometimes observe a speedup exceeding S.

Our implementation assumes that the underlying data is stored columnwise as a contiguous array of fixed-length nu-

meric values. Some systems, such as Sybase IQ [2], Compaq Infocharger [13] and Monet [5] use this kind of columnwise approach. More recently, the PAX layout scheme proposes that each disk page be organized columnwise in a similar way [3]. As long as each page contains sufficiently many records, the CPU and memory performance of an operation on a collection of arrays (one per page) should be comparable to the performance on a single such array. Thus, our study is applicable to both in-memory databases as well as disk-based databases.

We consider the most important database operations, including sequential scans, aggregation, index operations, and joins. We present techniques for implementing these using SIMD instructions. In some cases, the SIMD implementation is straightforward. In other cases, the SIMD implementation is novel. We demonstrate that significant speedups in performance are achievable for each of these operations, with only moderate implementation effort to rewrite the inner loops for each operation. This rewriting can be done in `C` rather than in assembly language, using intrinsics provided in Intel's `icc` compiler.

The rest of this paper is organized as follows. We survey related work in Section 2. In Section 3, we demonstrate how SIMD techniques can be used in common data processing operations like scans and aggregation. In Section 4, we show how to use these SIMD techniques to speed up the performance of common index structures in a database system. In Section 5, we study joins. Additional SIMD techniques are briefly discussed in Section 6. We conclude in Section 7.

2. OVERVIEW OF SIMD TECHNOLOGY

SIMD technology lets one microinstruction operate at the same time on multiple data items. This is especially productive for applications that process large arrays of numeric values, a typical characteristic of multimedia applications.

SIMD technology comes in various flavors on a number of architectures, including "MMX," "SSE," and "SSE2" on Intel machines, "VIS" on SUN UltraSparc machines, and "3DNow!", "Enhanced 3DNow!" and "3DNow! Professional" on AMD machines. Additional vendors with SIMD architectures include Hewlett-Packard, MIPS, DEC (Compaq), Cyrix, and Motorola. For a detailed comparison of SIMD technology in different architectures, see [25].

Some of these SIMD technologies are limited; for example, Sun's VIS does not support floating point values in SIMD registers, and provides 64-bit rather than 128-bit registers [27]. We have performed experiments using Sun's VIS, but have found that if we require a minimum of 32-bits for data values, then the two-way SIMD improvement that is enabled by having 64-bit SIMD registers is not significant.

We illustrate the use of SIMD instructions using the packed single-precision floating-point instruction available on Intel SSE technology chips, and shown in Figure 1. Other architectures have similar instructions.

For this particular instruction, both operands are using 128-bit registers. Each source operand contains four 32-bit single-precision floating-point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, X1 and Y1, X2 and Y2, and X3 and Y3) in each operand. SIMD operations include various comparison, arithmetic, shuffle, conversion and logical operations [14].

Terminology: A basic numeric element (either integer or
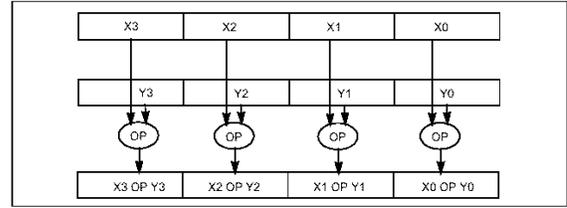
Figure 1: Packed Single-Precision Floating-Point Operation

floating point) is called a *word*. We let S denote the degree of parallelism available, i.e., the number of words that fit in a SIMD register. In Figure 1, $S = 4$. A memory-aligned group of S words is called a SIMD *unit*.

2.1 SIMD versus Vector Processors

Vector processors provide high-level operations that work on *vectors*, i.e., linear arrays of numbers [20]. A typical vector is able to contain between 64 and 512 64-bit elements. With a single instruction, operations can be performed in parallel on all elements. Specialized architectures with large numbers of simple processing elements are required.

Compared to vector registers, SIMD registers can hold a small number of elements. For example, at most four 32-bit values can be processed using SIMD instructions in a Pentium 4. Thus SIMD on commodity processors has a much lower degree of parallelism. On the other hand, the latency of load-store instructions (compared with vector processing) is much higher for vector processors than for SIMD on commodity processors. Further, the small size of SIMD instructions means they can also take advantage of pipelined execution in a superscalar processor, overlapping their work with other instructions.

2.2 The Limitations of Vectorizing Compilers

The goal of vectorizing compilers is to automatically detect opportunities where ordinary code can be transformed into code exploiting SIMD techniques. For example, `icc` does have vectorizing capabilities [15]. However, [15] describes various reasons why the compiler may fail to apply vectorization, including: stylistic issues (such as the use of global pointers or the use of moderately complex expressions), hardware issues (such as data alignment), and complexity issues (such as the use of function calls or nonassignment statements in a loop).

None of the code we wrote for the experiments could be vectorized by the `icc` compiler. The code contains several fundamental obstacles. For example, most of the code contains conditional branches in the inner loop. In other cases, like for aggregation, the hand-crafted SIMD code uses relatively subtle tricks in manipulating the conditional masks that would be very difficult for a compiler to produce. Other compilers for other SIMD architectures (such as Sun's compiler for their VIS instruction set) are similarly limited [27].

Thus, while it would be convenient to rely upon the compiler to do the SIMD transformations for us, it is clear that state-of-the-art compilers cannot do so. As a result, in order to take advantage of the benefits of SIMD, database programmers must be explicit in their use of SIMD, just as multimedia programmers are today.

2.3 Comparison Result Format

In Intel and AMD architectures, a SIMD comparison operation results in an *element mask* corresponding to the length of the packed operands. An element mask is a vector in which each packed element contains either all 1's or all 0's. For example, when OP is a comparison operation in Figure 1, the result is a 128-bit wide element mask containing four 32-bit sub-elements, each consisting either of all 1's (0xFFFFFFFF) where the comparison condition is true or all 0's (0x00000000) where it is false.

In other architectures, like SUN and MIPS, comparison instructions result in generation of a *bit vector*, in which a single bit represents the true/false result of the comparison between the corresponding elements in two operands. Intel and AMD architectures also support bit vectors through the application of one additional instruction that converts an element mask to a bit vector by selecting the most significant bits of each word. (We shall use this instruction several times in this paper; we call this operation SIMD_bit_vector.)

2.4 Data Organization

The data alignment for SIMD instructions is another issue that must be considered. For example, VIS instructions operate on 8-byte aligned data. For SSE and SSE2 instructions which operate on 128-bit registers, data must be on 16-byte boundaries to take maximal advantage of SIMD instructions. Although there are move instructions to allow unaligned data to be copied into and out of SIMD registers when using unaligned data, such operations are much slower than aligned accesses.

To better utilize SIMD instructions, it is necessary to make the source data contiguous, so that a whole SIMD unit can be loaded at once. One way to achieve this effect is to use columnwise storage, as discussed in Section 1.

2.5 Branch Misprediction

Conditional branch instructions present a significant problem for modern pipelined CPUs because the CPUs do not know in advance which of the two possible outcomes of the comparison will happen. CPUs try to *predict* the outcome of branches, and have special hardware for maintaining the branching history of many branch instructions. A mispredicted branch incurs a substantial delay; [4] reports that the branch misprediction penalty for a Pentium II processor is 17 cycles. For Pentium 4 processors, the *minimum* penalty is 17 cycles [1] with a slightly higher average penalty; the pipeline for a Pentium 4 is 20 stages deep. In our experimental results that measure the relative importance of branch misprediction effects, we assume that a branch misprediction incurs a 20 cycle delay. This corresponds to $20/c$ seconds in our graphs, where c is the clock rate of our processor. To make use of branches may minimize memory references or instruction counts, but it may impede progress in a pipelined architecture. Even a 5% misprediction rate cuts performance by as much as 20–30% in today's wide-issue processors [1].

2.6 SIMD Implementation

The direct way to use SIMD instructions is to inline assembly language into one's source code. However, this can be time-consuming, tedious and error-prone. Instead, we use SIMD intrinsics provided by Intel's C++ Compiler in this paper. Intrinsics are special coding extensions that allow one to use the syntax of C function calls and C variables instead of hardware registers. Generally speaking, direct assembly coding can outperform the use of intrinsics. Nevertheless, for fairness of comparison with algorithms coded in C, we use the provided intrinsics.

Our experimental results use single precision 32-bit floating point values as the element data type, unless otherwise mentioned. Since SSE and SSE2 registers are 128 bits, this choice means that $S = 4$. Our Pentium 4 machine runs at 1.8 GHz, has 1GB of Rambus RDRAM, and uses the RedHat Linux 7.1 operating system. We use Intel's C++ compiler with the highest optimization level. GNU's g++ compiler gives similar results for algorithms without SIMD instructions, but g++ does not have intrinsics for Pentium SIMD instructions. In this paper, we only report the results for Intel's C++ compiler.

On a Pentium 4, common access patterns such as sequential access are recognized by the hardware, and hardware prefetch instructions are automatically executed ahead of the current data references, without any explicit instructions required from the software. Since many of our proposed techniques do have a simple sequential access pattern, we benefit from this behavior by having reduced cache-miss latencies.

In some experiments we explicitly measure the number of branch mispredictions. Pentium machines have hardware performance counters that enable one to measure such statistics without any loss in performance. To measure these numbers, we used the Intel VTune Performance Tool.[1]

3. SCAN-LIKE OPERATIONS

In this section, we study operations that, in one way or another, process their data sequentially. Examples include searching on one or more unindexed attributes, creation of bitmaps based on selection conditions, and scalar aggregation. Grouped aggregation can be achieved by a combination of sorting followed by one scalar aggregation per group. Scan-like operations are also used as components of more complex operations that we will discuss in Section 4 and Section 5.

In practice, we need to consider inputs that may be unaligned, or where the number of elements may not be a multiple of S. Special pieces of code can be used for the first and last few elements, with the remaining "inner" part of the input array both aligned and of size being a multiple of S. Since the input is typically much larger than S, the overhead of these pieces of code to handle the ends is likely to be small. Thus, we focus below on inputs that are aligned and have size being a multiple of S.

All scan-like operations described in this section have the following high-level structure. In this pseudo-code fragment, N is the number of words, which is a multiple of S, and x and y are arrays corresponding to columns of our input records.

```
for i = 1 to N {
  if (condition(x[i])) then process1(y[i]);
  else process2(y[i]); }
```

[1]This tool runs under Microsoft Windows, so we had to compile the code (again using Intel's compiler) under Windows rather than Linux to perform this measurement on the same machine. We verified that the inner loop assembly code generated by the compilers was the same for both operating systems.

Different operations correspond to different implementations of the functions `condition`, `process1` and `process2`. Of course, a scan operation might need to test multiple columns, or to process multiple columns; our choice of single columns is for simplicity of presentation only.

The high-level structure of the corresponding SIMD code will look like the following.

```
for i = 1 to N step S {
   Mask[1..S] = SIMD_condition(x[i..i+S-1]);
   SIMD_Process(Mask[1..S], y[i..i+S-1]); }
```

SIMD instruction sets typically have rich comparison, logical and arithmetic instructions. For most `condition` functions containing comparison, logical or arithmetic operations, there is an equivalent SIMD version `SIMD_condition` formed by replacing each scalar operation `op` by its SIMD version `SIMD_op`. For example, if `condition(x)` were implemented as $(4 < x)$ && $(x <= 8)$ then `SIMD_condition(x[1..S])` would be `(FOUR SIMD_< x) SIMD_And (x SIMD_<= EIGHT)`, where `FOUR` and `EIGHT` are SIMD units containing S copies of 4 and 8 respectively.

Despite the richness of the SIMD instruction set, there are conditions that are not easy to parallelize via SIMD. Examples include operations on data types that are variable length, such as strings. Thus we don't expect to be able to parallelize string operations such as the SQL `LIKE` operation.

Notice that there is no `if` test in the SIMD algorithm. If we can implement `SIMD_Process` in a way that avoids `if` tests, we avoid conditional branches and their potential for branch misprediction penalties.

In what follows, we describe various scan-like operations, and show how to derive implementations for `SIMD_Process` from those for `process1` and `process2`.

3.1   Return the First Match

The first scan-like operation is to return the first `y[i]` such that `x[i]` satisfies the condition function. That is, `process1(y)` is implemented as

```
process1(y) { result = y; return; }
```

with `process2` empty. The corresponding `SIMD_process` is

```
SIMD_Process(mask[1..S],y[1..S]) {
  V = SIMD_bit_vector(mask);
  /* V = number between 0 and 2^S-1 */
  if (V != 0) {
    for j = 1 to S
      if ( (V >> (S-j)) & 1 )   /* jth bit */
        { result = y[j]; return; } }
}
```

Note that if the selectivity of the condition function is low, so that `mask` is mostly zero, the `(V != 0)` test saves work.

A variation of the first match algorithm, that we call "unique match," applies when we know that the underlying data set contains at most one match for the search key. This happens in practice; for example, in a conventional B$^+$-tree, it is common to put just the unique keys in the leaves, with pointers to a list of records having the key. A unique-match algorithm would apply to the leaves of such a B$^+$-tree for exact-match searches of such a tree.

The variation saves work in the case where the `(V!=0)` test succeeds, i.e., there is a match among the S words in `y`.

We take advantage of the fact that we know there is exactly one match. The `SIMD_process` method is now

```
SIMD_Process(mask[1..S],y[1..S]) {
  V = SIMD_bit_vector(mask);
  /* V = number between 0 and 2^S-1 */
  if (V != 0) {
    result = SIMD_And(mask,y);
    for j = 0 to log_2(S)-1
      {result = SIMD_Or(result,
                  SIMD_Rotate(result,2^j));}
    return result[1]; }
}
```

The `for` loop rotates `result` by increasing powers of two. At the end of the for loop, all elements of the SIMD unit contain the required match result.

Figure 2(a) shows the performance of the original code and the SIMD-optimized code for a successful first-match query having a simple equality condition. The time reported is the average over 10^6 random search keys. The SIMD version is approximately 3 times faster than the unoptimized version. Figure 2(b) shows the same results for small node sizes, including the unique match variant. For smaller node sizes, where the relative impact of a successful match is higher, the unique match variant performs slightly faster.

We measured the number of branch mispredictions, and found low numbers for all three variants. The explanation is that the conditional test almost always fails, and is thus well predicted by the hardware. Thus the speedup is due primarily to the enhanced parallelism.

3.2   Return All Matches

When one returns all matches, we write the results to an array called `result`, indexed by a variable `pos` that initially points to the start of the `result` array. For this operation, `process1` is implemented as

```
process1(y) { result[pos++] = y; }
```

`SIMD_process` is similar to the SIMD first-match case; the only difference being that the very inner loop is given by

```
{ result[pos++] = y[j]; }
```

We call this SIMD alternative 1. A second implementation of `SIMD_Process` takes account of the fact that branch mispredictions within the inner loop might be common if the selectivity is intermediate, i.e., neither close to 0 nor 1. This alternative (similar to the "no-branch" algorithm of [24]) is given below.

```
SIMD_Process(mask[1..S], y[1..S]) {
  V = SIMD_bit_vector(mask);
  /* V = number between 0 and 2^S-1 */
  if (V != 0) {
    for j = 1 to S {
        tmp = (V >> (S-j)) & 1; /* jth bit */
        result[pos] = y[j];
        pos += tmp; } }
}
```

Although we do some extra copying, we eliminate many potential branch mispredictions. Figure 3 shows the performance of the three methods on a simple range query,
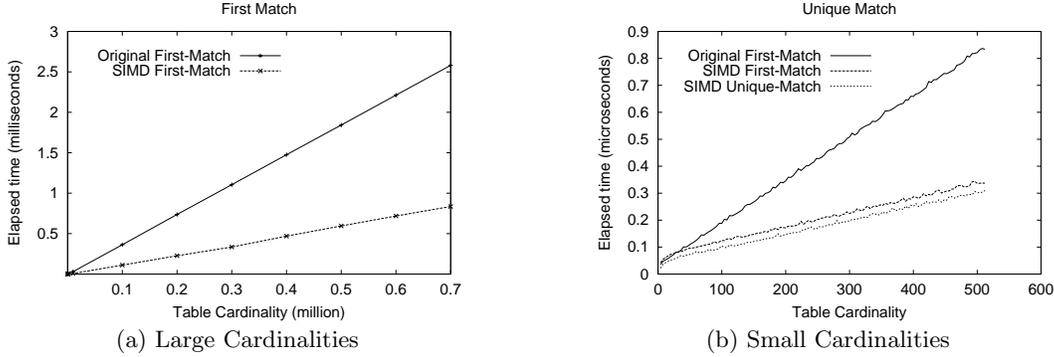
Figure 2: First Match and Unique Match Versus non-SIMD Code.

analogous to "SELECT y FROM R WHERE $x_{low} < x < x_{high}$". The time reported is the average over 5 runs. Figure 3(a) shows that at selectivity 0.2, the latter version of `SIMD_Process` is about 2.5 times faster than the original code. Figure 3(b) shows that the performance of the methods with branches is sensitive to the selectivity; intermediate selectivities yield branches that are intrinsically difficult for the hardware to predict.

Figure 3(c) breaks down the time for the three algorithms of Figure 3(b) into branch misprediction time (darker shade) and other time, measured at selectivity 0.2. The performance improvement is partially due to branch misprediction improvements, and partially due to increased parallelism. The relative benefits of parallelism are smaller here than for first-match because the cost of storing the matches is common to all algorithms, and is not done in parallel.

3.2.1 Building Bitmaps

The "find all matches" code could be slightly modified to generate a bitmap. This code could be used repeatedly to generate a bitmap index [19]. One would simply leave out the `V!=0` test and store the `V` values contiguously to form the bitmap. The performance issues for such an operation are thus similar to those for finding all matches.

3.3 Scalar Aggregation

Suppose we wish to aggregate column `y` for rows whose `x` values satisfy the given condition. In SQL, this corresponds to a scalar aggregate, i.e., an aggregation without a `GROUP BY` clause, but with a `WHERE` clause. Grouped aggregation can be achieved by first sorting the records by the grouping attributes, then performing several scalar aggregations. We consider incrementally computable aggregate functions, including sum, count, average, minimum and maximum. Average can be achieved by computing both sum and count.

For the non-SIMD code, the code for `process1` computes the aggregate function. For example, a sum aggregate might be coded as

```
process1(y) { result = result + y; }
```

The coding of other aggregates is analogous. We now consider SIMD versions of scalar aggregation.

Sum and Count

For these operations, we take advantage of the fact that a word consisting of all zeroes represents zero, both in standard integer and floating point representations. For example, Intel SIMD instructions are compatible with the IEEE

standard for binary floating-point arithmetic [14] that satisfies this property. (If this were not the case for some other architectures, our techniques for sum and count would resemble those for min and max given in the next section.)

Suppose that we have a SIMD register called `sum` that is initialized to S zero words. `SIMD_Process` then looks like

```
SIMD_Process(Mask[1..S], y[1..S]) {
  temp[1..S] = SIMD_AND( Mask[1..S], y[1..S] );
  sum[1..S] = SIMD_+( sum[1..S], temp[1..S] );
}
```

The idea is to convert non-matched elements to zeroes. After finishing processing, we need to add the S elements of *sum* as the final result. This can be done with $\log_2(S)$ SIMD rotate and SIMD addition instructions.

The code for computing the count is identical to that for sum, except that the second argument to the `SIMD_AND` operation is a SIMD unit of words containing "1", rather than `y`. For count, one can optimize even further if one takes advantage of the fact that the word consisting of all 1's corresponds to the integer -1 in most signed integer representations. In that case, we can simply add the masks with a single SIMD instruction in the inner loop, and negate the result at the end. (Our performance experiments do not use this trick.)

Min and Max

Intel's SIMD instruction set provides a SIMD operation to compute the pairwise minimum of two SIMD units in parallel, and similarly for maximum. Assume that we initialize a SIMD unit `infinity` with the representation of the largest representable number. A SIMD unit called `min` is initialized to `infinity`. Then the min operation can be implemented using the following `SIMD_process` operation.

```
SIMD_process(mask[1..S], y[1..S]) {
 temp1[1..S] = SIMD_AND( mask[1..S], y[1..S] );
 temp2[1..S] = SIMD_ANDNOT( mask[1..S],
                            infinity[1..S] );
 temp1[1..S] = SIMD_OR( temp1[1..S], temp2[1..S] );
 min[1..S] = SIMD_MIN( min[1..S], temp1[1..S] );
}
```

The idea is to convert nonmatched elements to infinity so that they do not influence the minimum result. The final phase is to get the minimal element among the S elements in $min[1 \ldots S]$. This can also be done by $\log_2(S)$ shuffle and SIMD minimum instructions.

(a) Varying Table Size Selectivity = 0.2   (b) Table Size = 1,000,000 Varying Selectivity   (c) Branch Misprediction Impact
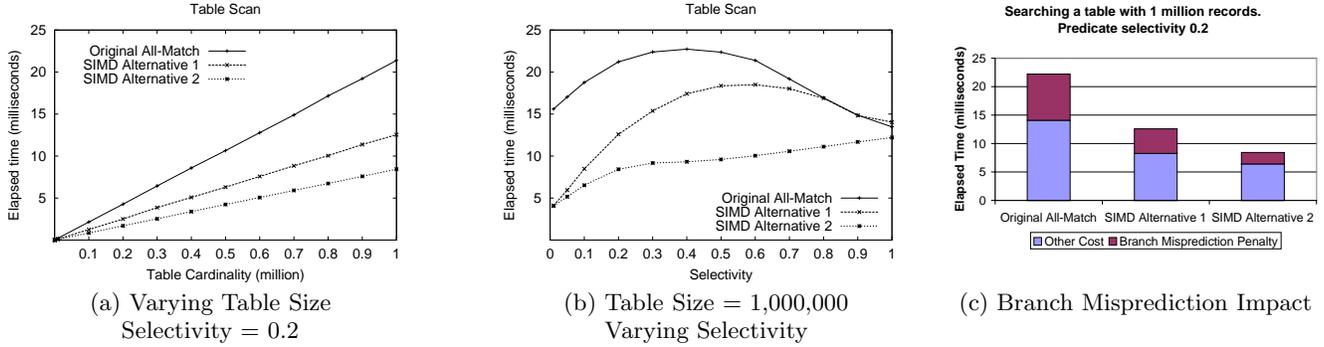
Figure 3: Table Scan (Return All the Matches)

The max operation is identical to the min operation, except that we use negative infinity rather than infinity, and apply a `SIMD_MAX` operation rather than a `SIMD_MIN` operation.

Figure 4 shows the performance and branch misprediction impact of a query corresponding to "SELECT AGG(y) FROM R WHERE $x_{low} < x < x_{high}$" for different types of aggregation functions. Table R has 1 million records and the predicate selectivity is 20%. There are no conditional branches in the SIMD-optimized algorithms. As a result, they are faster than the original algorithms by *more* than a factor of S ($S = 4$ in these experiments). 40% of the cost of the original algorithms is due to branch misprediction.
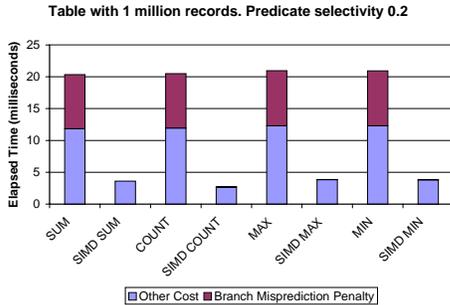

Figure 4: Aggregation and Branch Misprediction Impact

4. INDEX STRUCTURES

Index structures reduce computation time for answering queries, while consuming a modest amount of space. Common index structures include all kinds of tree-structured indexes, bitmap indexes and hash-based indexes. Various techniques for organizing index structures to be sensitive to modern architectures have been proposed, with particular emphasis on cache performance [21, 22, 11, 6, 17]. These techniques apply both to memory-resident indexes, as well as to the layout of a disk page within disk-based indexes. In this section, we study techniques for employing SIMD instructions to make index traversal more efficient. SIMD instructions can be combined with cache-conscious index designs to get the most out of modern CPU architectures.

We focus in particular on tree-based index structures. SIMD operations can be useful for other index designs too.

For example, they can speed up building hash indexes by allowing the computation of hash values in parallel.

Tree-based index structures proposed for use in database systems include ISAM, B$^+$ trees [7], and multi-dimensional indexes like Quad trees [10], K-D-B trees [23], and R-trees [12]. Searching over these indexes requires repetitively searching internal nodes from the root to the leaf nodes. Within each internal node, one must perform some computation to determine which child node to traverse. In the leaf nodes, similar searching is required.

We describe how SIMD techniques can be employed to efficiently search both internal nodes and leaf nodes. The first benefit we hope to achieve is that fewer instructions will be needed to locate the correct child, because SIMD instructions can compare multiple keys at the same time. The second benefit is that we will be able to employ simple arithmetic manipulations of the results of the SIMD instructions to determine the next step of the algorithm, as opposed to an explicit `if` test. As a result, we do not need any conditional branch instructions. Branch misprediction penalties are high, and index comparisons are likely to be a "worst-case" workload for the branch prediction hardware because each comparison is likely to be true roughly 50% of the time, in an unpredictable way. We may thus achieve significant time savings by replacing conditional branches by simple computation.

We first present techniques used in internal nodes of index structures in Section 4.1. In Section 4.2, we present techniques for leaf nodes. We use the B$^+$ tree as the basic structure in each section, and then discuss different processing techniques for different index structures.

4.1 Internal Nodes

A B$^+$-tree is a balanced index structure that is commonly used in commercial systems. A B$^+$-tree organizes its nodes (usually equal in size to disk pages) into a tree. Both internal and leaf nodes are at least half full.

An internal node stores an ordered sequence of keys. If n keys are used in a node, then there will be $n + 1$ child pointers. Leaf nodes consist of a sequence of entries for index key values. Each key value entry references the set of rows with that value. Traditionally, indexes have referenced each row via a *Row IDentifier*, or RID. A sequence of RIDs, known as an RID-list, is held in each distinct key value's entry in the B$^+$-tree leaf. We assume that keys are fixed length values; in our experiments we shall assume 32-bit keys. For variable length keys such as strings, one can obtain

many of the benefits of fixed length keys by storing a fixed length "poor man's normalized key" for each key [11].

Following [11], we physically organize an internal node as in Figure 5. The keys and child pointers are separately stored in ordered arrays. We require that the keys (and pointers) are properly aligned on a 16-byte address boundary to take maximal advantage of SIMD instructions. A page header inside each node stores the number of keys, key type, key size, empty space and other necessary information. The design of a leaf node is similar.
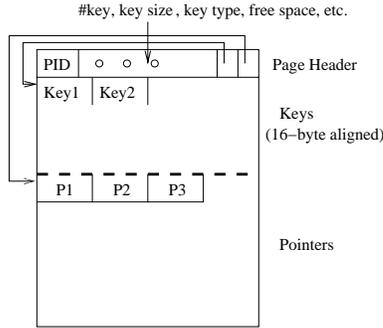

Figure 5: An example of B$^+$-tree internal node

For B$^+$ tree internal nodes, suppose our keys are in an array of the form $key_1, key_2, \ldots, key_n$ where key_i is an element of an ordered data type, n is a constant, and $key_i \leq key_{i+1}$. These n keys determine $n+1$ branches, and we assign each branch a *branch number* between 0 and n. Given a searching key K, the smallest i such that $key_i \leq K$ is the branch number; if no such i exists then the branch number is 0. B$^+$-Trees can use the branch number as an index into the array of child pointers.

Among different algorithms for searching within a page, binary search is preferable [26]. However, binary search involves a conditional test at every step. On some architectures such as the Pentium, the cost of branch mispredictions is high. Further, the kinds of comparisons needed for binary search represent a worst-case workload for branch prediction hardware because a branch will be taken roughly 50% of the time. As a result, we cannot expect the hardware to achieve less than a 50% misprediction rate.

4.1.1 Naive SIMD Binary Search

A first potential optimization is to use SIMD instructions in binary search. Instead of comparing one element at a time, we compare the key with S consecutive elements in one SIMD instruction. We start with the middle of the array, and proceed left or right as in binary search. If the SIMD comparison yields all zeroes, we move left; if all ones, we move right. If the comparison yields an intermediate value, we know the key is covered by the current SIMD unit of the array. The branch number can be computed directly by applying `SIMD_bit_vector` to the comparison result.

Naive SIMD binary search does not eliminate all branches, but it does shorten the comparison code by decreasing the number of comparisons required.

4.1.2 Sequential Comparison with SIMD

Some indexes, such as those designed for cache-locality, have a node size that is a small multiple of the cache line size. For these indexes, it may be more efficient to sim-
ply compare all keys in a node with the search key in a brute-force fashion. When implemented via SIMD instructions, such a method does not contain any complex branch instructions (thus avoiding branch mispredictions), and can take full advantage of the parallelism of SIMD. The basic idea is simple. We compare each SIMD unit with the searching key K. Then we count the number m of keys that are less than or equal to the searching key K. The implementation is identical to that for count aggregates in Section 3.3. Our experiments show that in some situations, sequential comparison works amazingly fast.

A variant of this method (we call it sequential search 2) checks each comparison result. If any comparison shows there is any key larger than the search key, we stop: There is no need to process the rest of the keys because they are all larger than the search key. The advantage of this variant is that it processes fewer keys (50% fewer on average) and performs fewer key comparison instructions. The disadvantage is that we need an extra conditional test in the inner loop to see if we have reached a key larger than the search key. We shall experimentally compare these two variants.

4.1.3 Hybrid Search

Consider an abstract search model, in which a logarithmic structure (such as a binary tree or binary search) is used to locate a particular sequence of elements, and then that sequence is sequentially scanned to find the particular search key. Suppose that all of these leaf-level sequences have length L. Then the overall complexity would be approximately $a \log_2(n/L) + bL + c$ for some constants a, b, and c. a would be the effective cost of processing a single comparison in the tree or binary search. b would be the effective cost of processing a single comparison in the sequential scan.

A straightforward analysis of this formula shows that the overall cost is minimized when $L = a/(b \ln 2)$. Interestingly, this minimum is independent of n. In an abstract model, one might imagine that $a \approx b$ since each is performing a single comparison. Such an analysis would suggest that one might as well have sequences of length $L = 1$, i.e., ordinary trees or arrays.

However, when one re-examines this trade-off in light of modern architectures, a different story emerges. In particular, suppose that the sequential scan is implemented using SIMD instructions as in Section 3. Then the relative sizes of a and b are impacted by two reinforcing effects. Firstly, since S comparisons can be done in parallel for the sequential scan, the effective value of b is reduced by close to a factor of S. Secondly, each binary search comparison has a 50% chance of incurring a branch misprediction penalty, while there are few (if any) branch mispredictions for the sequential search. This increases the relative cost of a by a factor equal to roughly half of the branch misprediction penalty. We shall see experimentally that the net result is an optimum L value of about 60 for our machine. It would be relatively easy to run a short configuration script to determine the appropriate L value for other machines at system initialization time.

Thus we propose a method called the "hybrid" method. We group the data into sequential segments of length L. Binary search is performed on the last element of all segments, until the correct segment is located. Finally, the correct segment is scanned sequentially.
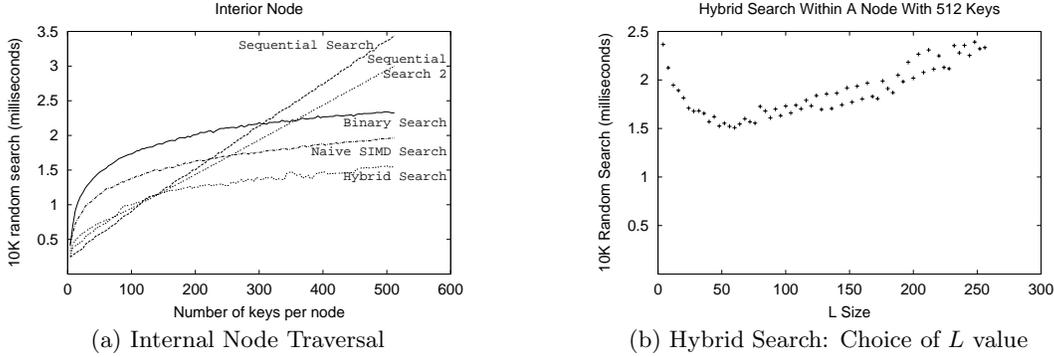
Figure 6: **Comparison of Search Methods**

(a) Internal Node Traversal   (b) Hybrid Search: Choice of L value

4.1.4 Experiments

We perform a simulation that corresponds to the traversal of a single internal node of a B$^+$-Tree. Thus a node contains just a key array and a pointer array, and those keys are ordered. By studying just the cost of traversing a single internal node we can isolate the improvement in the component of the cost for traversing an internal node.

We varied the number of keys per node from 4 to 512. Lookup keys are generated (randomly) in advance, so that we don't measure key generation time. Keys are 32-bit single-precision floating point numbers. Each traversal ends with the generation of the branch number for the node. We measure the searching time of 10,000 searches. We repeat each test five times and report the average time. Note that there are no cache misses after the first traversal, enabling us to concentrate on the computation cost that is the focus of this section.

After curve-fitting binary search and sequential search to compute a and b, we compute the optimum L for hybrid search as approximately 62.6. Figure 6(b) shows the performance of hybrid search within a node with 512 keys, varying the unit size. Performance is minimized when $L \approx 60$, matching our prediction.

Figure 6(a) shows the comparison of the different implementation techniques. Naive SIMD search is always better than binary search. SIMD-Sequential search is better than both algorithms when the number of keys is smaller than around 250. However, the cost of sequential search increases linearly with the number of keys and it becomes worse when dealing with a large number of keys. The second version of SIMD-Sequential search differs from the original sequential search in that it checks, after each unit comparison, whether it is necessary to compare more keys. The two variants are both linear, but with different slopes and intercepts. As a result, there is a transition point at which the preferred method of the two switches from the first to the second. However, both variants are asymptotically worse than binary search or naive SIMD search whose costs are logarithmic. Figure 6 also shows hybrid search with $L = 64$. Hybrid search combines the best of binary search and sequential search and performs well in all the cases. We advocate the use of sequential search when there is always a small number of keys (less than the optimal L value); otherwise use Hybrid search.

Figure 7 identifies the branch misprediction cost for the algorithms in Figure 6 at node size 128 and 512 keys. Not surprisingly, sequential search experiences the lowest branch misprediction penalty. On the other hand, on a 512-key node, it scans all 512 keys and the final cost is high. Sequential search 2 has nearly the same overall performance as sequential search; it searches fewer data elements but incurs more branch mispredictions. Hybrid search has fewer mispredictions than binary search and naive search; this difference explains most of the performance gap.

4.1.5 Quad Trees and R-Trees

An internal node of a quad tree in k dimensions has 2^k children. Searching an internal node involves determining which child "quadrant" to follow. A straightforward implementation of a node traversal would require k conditional tests.

Take a two-dimensional quad tree as an example. We number the four quadrants from 0 to 3 and store 4 pointers to different quadrants in an array. Instead of comparing with the x-axis and y-axis and combining the results, we can finish this operator with two SIMD instructions whenever $k \leq S$.

Figure 8 shows how to compare both the x-axis and y-axis at the same time, make a bit vector out of the result mask and easily get the quadrant number (and thus the offset to the quadrant pointer). There is no branch misprediction penalty at all.
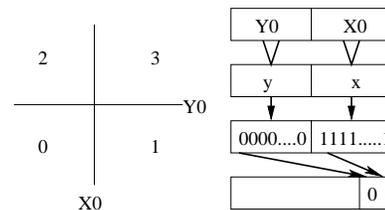

Figure 8: **SIMD Search Within A Quad Tree Node**

Internal nodes of an R-tree consist of many bounding boxes. Searching an R-Tree needs to check which bounding box contains the searching point. Normally, this requires up to two comparison and conditional tests per dimension. With SIMD operations, one can perform multiple tests in a single operation. In S dimensions, for example, we can compare the lower bounds in one test, and the upper bounds in another, resulting in two rather than $2S$ comparisons.

4.2 Leaf Nodes

Leaf nodes are similar to internal nodes except there are only N pointers corresponding to N keys. Also, for an exact-match search, we need to find an exact match in leaf nodes
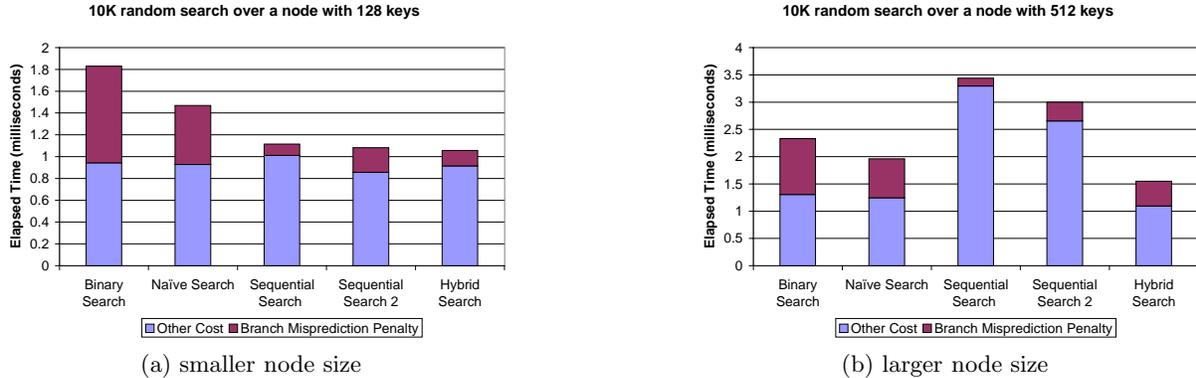
10K random search over a node with 128 keys

10K random search over a node with 512 keys

(a) smaller node size

(b) larger node size

Figure 7: Branch Misprediction Impact

rather than looking for the smallest key greater than or equal to the search key. The performance issues for leaf node design are very similar to those for internal node design, and the performance graphs are also similar. As a result, we expect that sequential search is the method of choice for leaf node sizes less than L, while hybrid search is preferred for larger leaves.

An interesting benefit of using sequential search in leaves is that leaves do not need to be stored in order. This makes the code for insertions, deletions and key modification simpler because much less copying needs to be done in the leaves.

4.2.1 Leaves in Multi-Dimensional Index Structures

Leaf nodes in multi-dimensional index structures are all similar. Instead of one key array in leaf nodes, we advocate storing each dimension in a separate array. The RID pointer array is the same. SIMD instructions enable us to compare S keys at a time in each dimension. After comparing S keys in every dimension and SIMD ANDing the results, we get a mask for S elements. The rest of the leaf node design can be done as for B$^+$ trees.

4.3 Overall Tree Performance

Using the techniques described above, we implemented a 3-level B$^+$-tree containing 10 million keys. Each node is a 4K page and is able to contain up to 500 pairs consisting of a 32-bit floating-point value and a 32-bit pointer. We use hybrid search in both internal nodes and leaf nodes. All the B$^+$ nodes have been preloaded in memory; we don't count the I/O cost.

For searching, there is around a 10% performance gain over the B$^+$-Tree with binary search. The speedup is not as much as for the CPU cost alone, because there are other costs incurred, such as cache miss penalties. Since insertion, deletion and update all perform an initial search, they also perform faster.

5. JOIN PROCESSING ALGORITHMS

The nested loop algorithm is the most general way of handling joins. It can handle both equi- and non-equi-joins. A nested loop join compares every record of one table against every record of the second table according to the join predicate(s). With SIMD techniques, there are three ways to implement the nested-loop join.

Duplicate-outer In the outer loop, fetch one join key from the outer relation and duplicate it S times to make a SIMD unit. In the inner loop, scan all the keys in the inner relation to find the matches. Using the techniques of Section 3.2, compare the unit with S join keys from the inner relation each time. If matches are found, produce join results accordingly.

Duplicate-inner In the outer loop, fetch S join keys from the outer relation to make a SIMD unit. In the inner loop, fetch one join key from the inner relation and duplicate it S times to make another SIMD unit. Compare both SIMD units, check the result, and produce join results accordingly.

Rotate-inner In the outer loop, fetch S join keys from the outer relation to make a SIMD unit. In the inner loop, fetch S join keys from the inner relation to make another SIMD unit. Compare the two units S times, rotating the inner unit by one word between comparisons. Check the comparison results and produce join results accordingly.

Rotate-inner can be evaluated in a slightly more efficient way if we have enough (more than S) SIMD registers: Precompute the S rotations of the *outer* records outside of the inner loop, and perform four separate comparisons (and no rotations) in the inner loop. As in Section 3, we assume that the join predicate is easily transformed into SIMD form using logical and arithmetic operations supported by the SIMD instruction set.

For our first study, we use the following four queries to show the performance of different algorithms (The key is integer type for the first query and floating-point type for the other three queries). The first two are equijoins over integer and floating point values, respectively. The fourth is a band join, in which records from one table must be within a constant-sized window of records from the other table [8]. The third is like a band join, except that the width of the band window varies from record to record.

```
Q1: SELECT ... FROM R, S WHERE R.Key = S.Key
Q2: SELECT ... FROM R, S WHERE R.Key = S.Key
Q3: SELECT ... FROM R, S
    WHERE R.Key < S.Key < 1.01 * R.Key
Q4: SELECT ... FROM R, S
    WHERE R.Key < S.Key < R.Key + 5
```

The outer relation R has 10^6 tuples and the inner relation S has 10^4 tuples. By making S the inner relation, we avoid cache thrashing since S fits entirely within the L1 cache of our processor. The join selectivity for the first two queries is around 0.0001. Queries 3 and 4 have join selectivity around 0.005. The result of the join is pairs of outer and inner RIDs of matching tuples. Subsequent tuple reconstruction is equal for all algorithms, and we do not include that in our comparison.
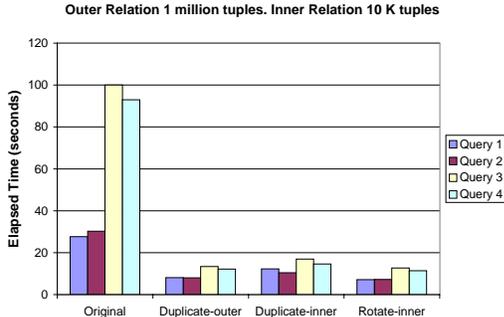

Figure 9: Nested-Loop Join

Figure 9 shows the elapsed time for the original join algorithm and three SIMD algorithms. All three methods take advantage of parallelism in SIMD instructions and are faster than the original nested-loop join. The `duplicate-inner` method is slower than the other two SIMD methods because the duplication of the inner keys happens in the inner loop, and must be performed $10^6 \cdot 10^4/4$ times. In contrast, the `duplicate-outer` method performs duplication just 10^6 times. The `rotate-inner` method is the fastest. It takes advantage of loading S SIMD variables outside the inner loop. None of the SIMD methods have a significant number of branch mispredictions.

For the simple integer and floating point join predicates, the SIMD algorithms are almost four times faster than the original algorithm. For queries 1 and 2, the number of branch mispredictions was small.

For queries 3 and 4, we see a factor of 9 improvement. Figure 10 shows the branch misprediction component of the cost for different algorithms for queries 3 and 4. There are two conditional tests in the predicates. As a result, roughly 40% of the time is consumed by branch misprediction effects in the original code. In contrast, none of the SIMD methods display significant numbers of branch mispredictions. The SIMD code directly processes the masks generated by SIMD compare instructions, without requiring an `if` test for each row. Further, when we directly implemented query 3 using just one slot of a SIMD register, we obtained an elapsed time of 44 seconds. Even if one discounts the branch misprediction overhead for the original code, 44 seconds is substantially smaller than the 60 seconds for the original code. The reason for this difference is that the Pentium 4's native floating point arithmetic instructions are significantly slower than its SIMD floating point arithmetic instructions. Thus, of the roughly 86 seconds saved, 30 were due to parallelism, 16 were due to faster processing of SIMD floating point arithmetic than native floating point, and 40 were due to avoiding branch mispredictions.

5.1 Optimizing for the Common Case

In practice, join selectivities are typically very small, meaning that a very small fraction of all record comparisons result in a match. Therefore, the (`V != 0`) tests on the bit-vector V representing the match results are worthwhile. Almost all of the time V will be zero, and we can avoid the work of checking the individual matches.

We can take this observation even further. Suppose that our SIMD architecture allows SIMD operations on smaller datatypes. On a Pentium 4, for example, there are SIMD operations for 8, 16, 32, and 64 bit datatypes, with corresponding S values of 16, 8, 4, and 2 respectively. Let f be a mapping from datatypes of size d bits to datatypes of size d' bits, where $d' < d$. Let $p(x, y)$ be a join predicate over size-d datatypes, and $p'(x, y)$ be another join predicate over size-d' datatypes. We say that the pair (f, p') is a *filter* for p if $p(x, y)$ implies $p'(f(x), f(y))$ for all x and y.

For example, if p and p' were both equality predicates, then any hash function f would enable (f, p') to be a filter for p. As another example, suppose that p and p' were bitwise comparison operations (possibly involving bitwise logical operations as subexpressions) that were identical except that p' operates on shorter datatypes. Then any function f that selected some d' bits out of a d bit word would enable (f, p') to be a filter for p.

Our basic idea is to perform an initial join comparison between S' smaller values, where $S' > S$. If some key-pair among the S' pairs matches, then we execute the S-way join as before for these records. However, if there is no match, then we skip to the next group of S' records. If it is common that none of the S' records matches, then we will have a net win because we can process S' records per comparison rather than S. Before performing the join, we materialize the results of the f function as a column of each input relation.

Example 1. Suppose that we were performing an equijoin on 32-bit keys. Let f_8 be a hash function that maps 32 bit keys to 8-bit values, and let f_{16} be a hash function mapping 32 bit keys to 16-bit values. If we use multiplicative hashing, we can assume that the probability of a collision in f_n is 2^{-n} since this scheme has been shown to be universal [9]. Let us assume that our join selectivity s was very small, say less than 0.0001, as is common. Then the probability that an m-way comparison of n-bit values is nonzero would be approximately $ms + 1 - (1 - 2^{-n})^m$. For $m = 16$ and $n = 8$, this probability is approximately 0.06. For $m = 8$ and $n = 16$, this probability is less than 0.001.

Thus, we could perform an initial test on 16-bit hash values, and less than one time in 1000 would we need to actually look at the original 32-bit values. The rest of the time we would be proceeding twice as fast through the data as we would with 32-bit comparisons. Similarly, we could proceed four times as fast through the data 94% of the time with 8-bit hash values, and need to check the original 32-bit data 6% of the time. The relative cost of handling nonmatches versus handling matches needs to be considered when determining the right trade-off. Figure 11(a) shows the measured performance for a simple join like query Q1 above. The join selectivity is varied by varying the domain cardinalities of the join attributes. Note that the smaller (inner) table fits within the cache, so that we do not expect the cache miss cost to be significant.
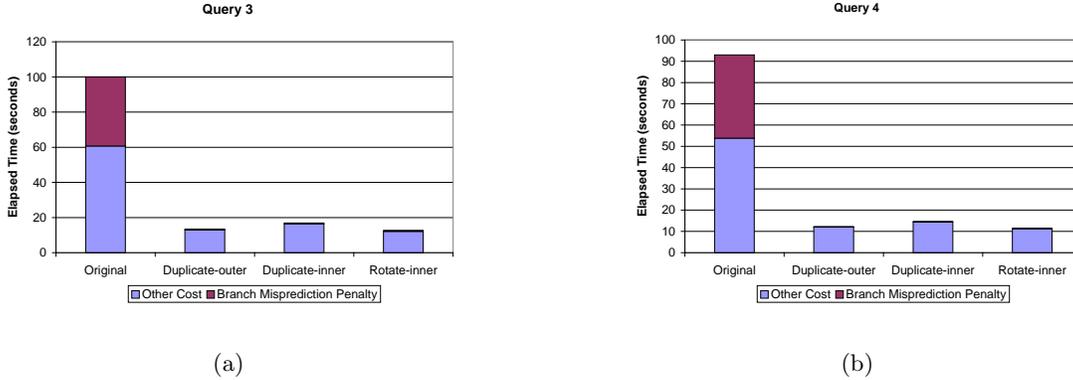
Figure 10: Branch Misprediction Impact

The results show that the 8-way SIMD code is the most efficient for small selectivities. That the 8-way code is better than the 4-way code follows from the doubling of the effective rate that data is processed, while incurring the expense of an explicit match of eight full keys just 0.1% of the time. More interestingly, the 8-way code beats the 16-way code even though the 16-way code is twice as fast as the 8-way code most of the time. However, 6% of the time the 16-way method has to perform an explicit match of 16 full keys. The cost of the explicit match is high enough that this difference is dominant.

When join selectivities increase, the probability that an explicit match of the full keys is needed increases, thus affecting the trade-off among the methods. Note that the query optimizer has enough information to choose the preferred method as long as it has a join selectivity estimate; given the use of multiplicative hashing, the only other statistics that need to be gathered for cost estimation are not data-dependent.

The following example demonstrates that there are realistic scenarios where this approach helps for nonequijoins. For nonequijoins such as this, the only practical method for performing the join in modern database systems is the nested loop join.

Example 2. Suppose have two tables `wants(Client, Facility)` and `supplies(Location, Facility)`. For each client, we want to find every location that has all the facilities that client wants. An SQL formulation of the query is

```
SELECT Client, Location
FROM wants C, supplies S
WHERE (SELECT Facility FROM supplies G
        WHERE G.Location = S.Location)
      Contains
      (SELECT Facility FROM wants H
       WHERE H.Client = C.Client)
```

Now suppose that there are many clients and suppliers, but the number of facilities is limited. To optimize the performance of this query, we encode the facilities via bitmaps. If the number of distinct facilities is smaller than d, we may use d-bit integers to represent either facilities required or facilities available. Then we have two new tables `wants(Client,`

`bitmap)` and `supplies(Location,bitmap)` with one row per client and location respectively. The query can then be written as

```
SELECT Client, Location
FROM wants C, supplies S
WHERE (C.bitmap & S.bitmap = C.bitmap)
```

(Different vendors express bit operations in different ways, but bit operations are supported in all major commercial relational database systems.) A filter for this join consists of taking the low order bits of the facilities bitmap, and using those for the comparison in the `WHERE` clause. Figure 11(b) shows the performance of this bitwise join. The keys for the inner table are bitmaps corresponding to the binary representations of the numbers 1 to 10,000. The range of keys for the outer table is varied to achieve different join selectivities. The 8-way SIMD version is the best.

The join selectivity for both of these examples is sufficiently small that branch misprediction does not play a role in the performance. The performance gain is primarily due to the enhanced parallelism.

6. ADDITIONAL SIMD TECHNIQUES

There are several specialized SIMD instructions designed for multimedia instructions. For example, a Pentium 4 has an operation to compute the sum of the absolute differences of two vectors of numbers in a single instruction. Such operations might be useful for nearest-neighbor query processing. Other operations, such as the Euclidean distance between two points, can be computed more efficiently in SIMD, even though there is no explicit instruction to do so.

There are SIMD instructions that we might wish for in future SIMD designs. For example, the first-match and all-matches code could be improved if we had a shuffle instruction that performed its shuffle in a data-dependent way. A shuffle instruction moves the words in the source SIMD unit into another order in the destination unit, with duplication or omission allowed. (The Pentium 4 supports only fixed shuffles, so that data-dependent shuffles have to be programmed.)

A SIMD indirect pointer lookup would also be potentially useful, allowing one to retrieve data from multiple locations at the same time. This would enable SIMD processing of indirectly referenced data.
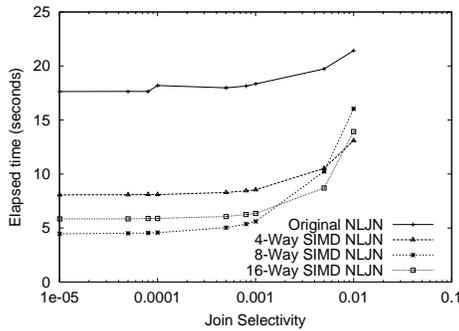
(a) Example 1
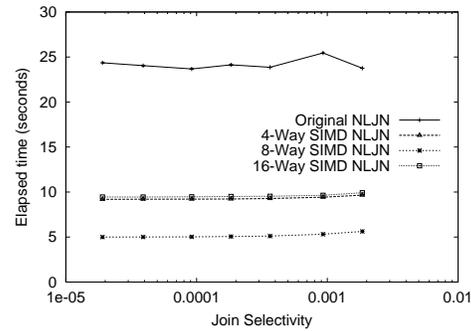
(b) Example 2

Figure 11: Nested-Loop Join with Filters.

7. CONCLUSION

Many database applications have become CPU bound, as I/O bandwidth and RAM capacities have increased. We have shown that designing database algorithms to utilize SIMD technology significantly improves their CPU performance. The two main reasons for the performance improvement are the inherent parallelism of SIMD, and the avoidance of branch misprediction effects. Programming SIMD inner loop code can be done in a high-level language, with relatively small code fragments required. Thus, we argue that it is relatively easy for implementors of high-performance database systems to take advantage of such techniques.

8. REFERENCES

[1] http://lava.cs.virginia.edu/bpred.html.

[2] Sybase IQ. http://www.sybase.com/products/archivedproducts/sybaseiq.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of VLDB Conference*, 2001.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of VLDB conference*, 1999.

[5] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of VLDB Conference*, 1999.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of ACM SIGMOD Conference*, 2001.

[7] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[8] D. J. DeWitt, J. Naughton, and D. Schneider. A comparison of non-equijoin algorithms. In *Proceedings of VLDB Conference*, 1991.

[9] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25:19–51, 1997.

[10] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[11] G. Graefe and P. Larson. B-tree indexes and CPU caches. In *Proceedings of ICDE Conference*, 2001.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD on Management of Data*, pages 47–54, 1984.

[13] InfoCharger Engine. Optimization for decision support solutions. 1998. (available from http://www.tandem.com/prod_des/ifchegpd/ifchegpd.htm).

[14] Intel Inc. Intel architecture software developer's manual. 2001.

[15] Intel Inc. Intel C++ compiler user's manual. 2001.

[16] Intel Inc. Intel IA64 architecture software developer's manual. 2001.

[17] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of ACM SIGMOD Conference*, 2001.

[18] S. Manegold, P. Boncz, and M. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *Proceedings of VLDB Conference*, 2000.

[19] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of ACM SIGMOD Conference*, 1997.

[20] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaugmann, 1996.

[21] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of VLDB Conference*, 1999.

[22] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. In *Proceedings of ACM SIGMOD Conference*, 2000.

[23] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. *ACM SIGMOD on Management of Data*, pages 10–18, 1981.

[24] K. A. Ross. Conjunctive selection conditions in main memory. In *ACM Symposium on Principles of Database Systems*, 2002.

[25] N. Slingerland and A. J. Smith. Multimedia extensions for general purpose microprocessors: A survey. Technical report CSD-00-1124, University of California at Berkeley, 2000.

[26] H. R. Strong, G. Markovsky, and A. K. Chandra. Search within a page. *JACM*, 26(3), 1979.

[27] SUN Microsystem Inc. VIS instruction set user's manual. 2000.

[28] Times Ten Performance Software Inc. TimesTen 4.3 and Front-Tier 2.3 product descriptions, 2002. Available at http://www.timesten.com.