RC24100 (W0611-039) November 8, 2006
Computer Science

IBM Research Report

Efficient Hash Probes on Modern Processors

Kenneth A. Ross
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient Hash Probes on Modern Processors

Kenneth A. Ross
IBM T. J. Watson Research Center and Columbia University
kar@cs.columbia.edu

Abstract

Bucketized versions of Cuckoo hashing can achieve 95–99% occupancy, without any space overhead for pointers or other structures. However, such methods typically need to consult multiple hash buckets per probe, and have therefore been seen as having worse probe performance than conventional techniques for large tables. We consider workloads typical of database and stream processing, in which keys and payloads are small, and in which a large number of probes are processed in bulk. We show how to improve probe performance by (a) eliminating branch instructions from the probe code, enabling better scheduling and latency-hiding by modern processors, and (b) using SIMD instructions to process multiple keys/payloads in parallel. We show that on modern architectures, probes to a bucketized Cuckoo hash table can be processed much faster than conventional hash table probes, for both small and large memory-resident tables. On a Pentium 4, a probe is two to four times faster, while on the Cell SPE processor a probe is ten times faster.

1 Introduction

Hashing is a commonly used technique for providing access to data based on a key in constant expected time. It is used in database systems for joins, aggregation, duplicate-elimination, and indexing. In a database context, hash keys are typically single column values or a tuple of values for a small number of columns. Payloads may be pointers (or record identifiers) to records, or they may represent values of an aggregate computed using hash aggregation.

Consider Figure 1 that shows pseudo-code for a hash probe operation on a bucketized hash table with separate chaining. This code assumes that the hash table contains no duplicate keys and that zero is not a valid payload value.

How would this code perform on a modern processor? The bucket size `S` would be chosen to match the cache-line size of the processor, so that each bucket access generates just one cache miss. When accesses to the hash table (and overflow buckets) are accesses to RAM, the cost of those accesses is significant. On modern machines, an L2 (or L3) cache miss may have a latency of several hundred cycles, although some of that latency can be overlapped with other work. Branch mispredictions may also occur in the `for` loop, `if` test and `while` loop. Branch mispredictions can cost 20–30 cycles for machines with long pipelines.

```
h = hash(key) % TableSize;
m = &table[h];  // m points to a hash
                // bucket of size S
do {
  for(i=0 ; i < m->keys_in_bucket; i++) {
    if (m->key[i] == key)
      return m->payload[i]; }
  m = m->next_bucket;
} while (m != NULL);
return 0;
```

Figure 1. Pseudo-code for hash probe.

A more subtle problem caused by branches is the creation of control dependency chains that limit the processor's (and the compiler's) instruction scheduling ability. Modern out-of-order processors can choose the next instruction to execute from among a large number of pending instructions, as long as dependencies are not violated (e.g., a value is not read from a register before the preceding write has finished). Conditional branch instructions create control dependencies that prevent instructions occuring after the branch from being scheduled before the conditional test, and vice-versa.

We use an open addressing hash scheme based on a bucketized version of d-ary cuckoo hashing [6]. We will refer to this data structure as a "splash table".[1]

In this scheme, probes always take a fixed, constant time, and access a small fixed number of cache lines, typically 2, 3, or 4, depending on the parameters chosen. Hash buckets do not need any information beyond keys and payload data, and very high occupancy is achievable (see Section 2.1).

Our main contribution is to enhance the probe phase of

[1] As one entry is dropped into a hash bucket, it may cause another entry to "splash" into another bucket. (We propose a new name to be able to refer to it concisely, and to distinguish it from other Cuckoo hashing variants.)

this scheme in a way that significantly enhances its performance on modern architectures for probing both small (cache resident) and large (memory resident) data sets. Our improvements rely on two ideas: (a) The use of probe code that is free of conditional branches, and (b) The use of Single-Instruction Multiple-Datastream (SIMD) instructions to process multiple keys and payloads in parallel. By avoiding branches, we not only avoid branch mispredictions, but we also allow more flexibile instruction scheduling, leading to a higher degree of overlapping of latencies.

At first glance, it appears that Cuckoo-based hashing would be worse than a standard scheme for large hash tables because it requires at least two memory references. A standard scheme will typically need just one reference for most probes. However, modern CPUs can support multiple outstanding memory requests. On an architecture such as the Pentium 4 that has a non-blocking cache and a CPU that supports out-of-order execution, the two accesses can be overlapped, since they are independent [12]. We shall confirm this behavior experimentally. While the required memory bandwidth for splash tables is double that of a standard hash table, memory bandwidth is typically not a performance bottleneck. Overlapping of memory accesses is not possible for conventional hash methods because later memory accesses are dependent on the state (such as the address of the overflow bucket) of earlier memory accesses.

Zukowski et al. [18] have also used cuckoo hashing to improve probe performance. Their work is similar to ours in that it attempts to remove branch operations in order to improve the overall number of cycles per instruction. However, there are several significant differences: (a) Their work builds on [12] rather than [6], meaning that the space utilization is about half as good. (b) They do not demonstrate that probes scale beyond cache-resident tables. Instead, they propose a partitioning step that divides the data into cache-sized units. In contrast, our probe results scale to RAM-sized tables without partitioning. (c) Our evaluation includes a modern architecture (the Cell SPE) that does not provide out-of-order execution. (d) [18] does not use SIMD instructions. (e) [18] does not use a universal hash function.

We evaluate splash tables on several modern architectures. Our primary platforms are a Pentium 4 machine, and the Synergistic Processing Element (SPE) of the Cell Processor [9]. A Cell chip contains eight independent SPEs, each of which has a 256KB local store with a 6 cycle latency. The Cell chip also contains a conventional PowerPC core. The SPEs can be seen as specialized processors that can be used to accelerate tasks that map well to their SIMD design. For example, based on the present study, the SPEs could be used to offload (from the conventional PowerPC processor on the Cell chip) dimension table lookups for a foreign key join with a large fact table. We will also examine probe performance on the Cell PowerPC processor, and on an AMD Opteron processor.

Our probe results (Section 4) show improvements over conventional hash tables, for both small and large tables, on several modern architectures. Performance improves by a factor of 2 to 4 on a Pentium 4, and by a factor of 10 on a Cell SPE. These improvements in probe time come with some additional overhead at build time, as we quantify in Section 3. In applications where there are many more probes than insertions (such as database joins where one typically builds a hash table on the smaller relation) the net result is a significant performance improvement.

2 Outline of the Approach

Key/payload assumptions. We shall assume initially that both keys and payloads are 32 bit values. Longer keys and/or payloads will be discussed in Section 4.4. We assume that the hash table contains no duplicate keys[2] and that zero is not a valid payload value.

Hash functions. We use multiplicative hashing as our hash function. It has several important properties for our purposes: (a) It is universal [3]. (b) It can be computed very efficiently on most modern architectures [15]. (c) If our keys form an arithmetic progression, as might be expected for automatically-generated primary keys, then hash values are even more balanced than randomly generated values [10]. (d) It is amenable to vectorization, so that we can compute multiple hash functions at once.

We can interpret a 32-bit hash value v as a binary fraction, i.e., the representation of a number between zero and one whose digits after the (binary) decimal point are given by v. Given a hash table size s, we can obtain a slot number by multiplying s and v and shifting right 32-bits. This way, arbitrary table sizes are possible without using expensive division or modulus operations. Computing $s*v$ can be performed with one 32-bit multiplication (generating a 64-bit result). If s is a 16-bit number, as might be expected for a small hash table, then two 16-bit multiplications, a shift, and a 32-bit addition suffice.

2.1 Building the Table

The build process is essentially the same as that described by Erlingsson et al [6]. A key is hashed according to H hash functions, leading to H possible locations in the table for that key. Each location is a bucket capable of holding B entries. We consider two possible ways of selecting a single bucket to place the key: (a) Place the key in the first bucket with space; (b) Place the key in the least loaded bucket, with ties broken arbitrarily.

[2]Duplicate keys can be handled by making the payload be a pointer to a list of records with that key value.

	K1	K2
0	6	
1	1	17
2	12	24
3	73	
4	46	
5	75	54
6	68	
7	57	
8	84	
9	99	91

(a)

	K1	K2
0	6	1
1	17	91
2	12	24
3	73	
4	46	
5	75	54
6	68	
7	57	
8	84	
9	59	99

(b)

	K1	K2
0	6	1
1	17	91
2	12	24
3	73	
4	46	41
5	75	54
6	68	60
7	57	79
8	84	87
9	59	99

(c)

Figure 2. Splash Tables for Example 2.1.

When the table is close to full, the system may encounter a key whose H candidate buckets are all full. In that case, we follow a reinsertion procedure [7]. Choose a bucket b at random from among the candidates, remove the key that had been inserted earliest from that bucket, and place the original key in b. The removed key is then inserted into one of its $H-1$ remaining candidate buckets. If all of these are full, the process is repeated recursively. If, after some large number[3] of recursive insertions, there is still no room for the key, then the build operation *fails*.

Hash buckets that are not full are padded with records having a zero payload.

Example 2.1 *Consider a splash table with $N = 10$ buckets and $B = 2$ entries per bucket. We will show just the keys; it is assumed that the payloads are inserted and moved along with the keys. For ease of exposition, assume that the two hash functions are $h_1(x) = x\%10$ and $h_2(x) = (x/10)\%10$, where $\%$ is integer modulus and $/$ is integer division. Thus, for a two-digit decimal number, the slots it maps to are precisely the digits of its representation. (In practice, we would use multiplicative hashing, as described in Section 2.) Suppose that we insert the keys $1, 12, 57, 73, 99, 91, 6, 46, 24, 17, 68, 84, 75, 54$ to the table in sequence. Key 1 can go in slots 1 or 0, both of which have count 0. Let us suppose that slot 1 is chosen. Key value 12 can then go in slots 2 or 1. Since slot 1 has a count of 1 and slot 2 has a count of 0, slot 2 is chosen. After the complete sequence of insertions we might reach the table shown in Figure 2(a), without any reinsertions. In this diagram, the earlier insertion is in the K1 column, and the later insertion is in the K2 column.*

Now suppose we wish to insert key 59. Slots 5 and 9 are both full. We choose one arbitrarily (say slot 9), remove the oldest key (99), and insert 59 into the table. Value 91 moves from the K2 position to the K1 position, and 59 is placed in the Key2 position. Value 99 is now reinserted into the table. Key 99 hashes to slot 9 according to both hash functions, and so we reinsert it into slot 9, and remove the key 91. Since key 91 came from slot 9, we attempt to insert it into its alternate slot, namely slot 1. Slot 1 is full, so we insert 91 and remove key 1 for reinsertion. Key 1 fits in slot 0, and we are done. The resulting table is shown in Figure 2(b).

Suppose that we continue by inserting the values $41, 60, 79, 87$ to get the table shown in Figure 2(c). The table is almost full: only one slot remains. (If there were to be no more insertions, we would be done.) If we were to insert a key that maps to slot 3, then we could achieve 100% occupancy in this example. On the other hand, inserting any key that does not map to slot 3 will cause a failure for this example, because there is only one key in the whole dataset that maps to slot 3. The reinsertion process would (if not terminated after a threshold) go on indefinitely.

[3] We use a value of 1000 as the limit for our experiments.

B	H	Load Factor
2	2	0.89
4	2	0.976
4	3	0.998
4	4	0.9997
8	2	0.997

Table 1. Failure thresholds for various B, H.

If an adversary has knowledge of the hash functions, he could devise a set of keys that never map to (say) the first half of the hash table according to any of the H hash functions. This would cause a hash table build to fail if the number of records to be inserted was more than half of the table's capacity.

There are two reasons why one should not be concerned about this possibility of failure. First, we use a class of hash functions that are *universal* [2]. If a hash function is chosen randomly from a universal class, then an adversary cannot create a degenerate dataset without knowledge of the particular function used. More practically, if for some reason a particular hash function behaves poorly for a given data set, a new hash function that will almost certainly behave well can be chosen fro the class.

Second, one can be careful to choose the size of the hash table so that there is enough space for recursive insertions to eventually find room in the hash table. There is a "thresholding" behavior in which builds almost always succeed when the load is below a threshold t, but almost always fail when the load is even slightly higher than t. Table 1 shows empirically derived threshold for various B and H; these are the load factors at which the build failure rate was one in a thousand, i.e., 0.1 percent. These values are similar to thresholds derived by others using different hash functions [6]. Even at very high occupancy levels, the build procedure succeeds with probability essentially 1.

Duplicate slots. It is possible that occasionally two or more of the H hash functions return the same slot of the

hash table, as we saw with key 99 in Example 2.1. This is undesirable because it gives less flexibility for the placement of some keys. One can avoid this behavior by adding code to the hash functions so that duplicates are not generated. However, this code would add to the cost of both probes and builds. For large enough hash tables, slot collisions of this kind are rare, and so we will not do anything special to avoid duplicate hash slots among the H hash values.

Counts. Insertions (but not probes) need to determine the number of available slots in a bucket. We could keep track of the number of valid keys in each bucket within the data structure itself. While this is the most straightforward alternative, it is not ideal because it imposes both a memory overhead and a potential additional source of cache misses. Instead, we simply count the number of zero payloads for a hash bucket. This counting can be performed without branches using a small number of SIMD instructions if the payloads are stored contiguously (see below).

2.2 Probing the Table

Given a hash table constructed as above, one needs to compute H hash functions and consult H slots of the hash table. For each slot, one compares each of the B keys against the search key for each slot, and return the payload of a match if one was found. In Example 2.1, a probe for key 27 would require looking in slots 2 and 7, and comparing all keys in these slots with 27.

Because we always do the same number of comparisons, loop unrolling allows us to avoid branches for the `for` and `while` loops of the probe implementation above.

We can also avoid branch mispredictions in the `if` test by converting the control dependency into a data dependency. A comparison operation generates a mask that is either all 0's (no match) or all 1's (a match). Such mask-generating comparisons are in the instruction sets of modern processors. The mask can be applied to each of the payloads, with the results ORed together. Since there are no duplicate keys, at most one of the disjuncts will be nonzero. An unsuccessful match returns zero.

A hash bucket is organized as follows. B keys are stored contiguously, followed by B contiguous payloads. This arrangement allows us to use SIMD operations to compare keys and process payloads. For example, if a SIMD register is 128 bits long (as in the Cell SPE and using SSE on the Pentium) then four keys or four payloads can be processed in parallel using a single machine instruction. As a result, a good choice for B would be a small multiple of 4. Further, when $B = 8$ the total size of a bucket is 64 bytes, which fits within the typical L2 cache line of modern processors. Thus there would be no more than H cache-line accesses.

A flowchart for the vectorized probe algorithm is given in Figure 3 for $H = 2$ and $B = 4$, using generic SIMD instructions that are close to the Pentium's SSE2 instructions. Solid lines represent the flow of data, usually in 128-bit vectors, while dashed lines represent a memory reference. If key $K7$ matches the probe K, then the output contains payload $P7$ in the leftmost SIMD slot. Note that even if an invalid key in a partially full bucket accidentally matches the hash probe, the corresponding payload is zero and the match will not affect the result.

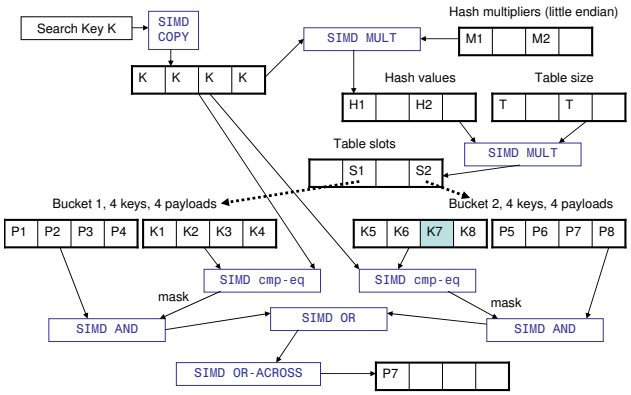

Figure 3. Probe flowchart (K7 matches K).

SSE2 only supports the direct extraction of 16-bit values from vectors. The `extract` operation (to get the slot number out of the SIMD vector to perform the memory reference) can thus be directly implemented in one instruction on the Pentium if the table size fits in 16 bits. Otherwise, two extractions, a shift and an OR (or a store to memory followed by a load) are required.

The Cell SPE does not directly support 32-bit multiplication, and so the hash function has to simulate 32-bit multiplication using 16-bit multiplication.[4] On the other hand, using 16-bit multiplication allows the simultaneous creation of four hash values in a single vector rather than two, which is useful for $H > 2$.

The `or_across` operation is directly supported by the Cell SPE instruction set; the SPE version takes one instruction where the Pentium version needs four.

3 Experimental Results: Build

In this section, we briefly study the build process. The time needed by an insertion operation at different load factors is given in Figure 4 for $B = 4$, $H = 2$ on a Pentium 4; the table capacity is 2^{25} elements, much larger than the

[4] Actually, since we're only using some of the bits of the 64-bit result, we can get away with simulating just part of 32-bit multiplication.

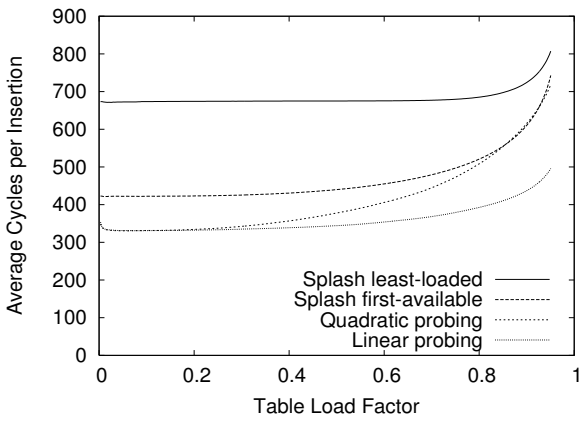

Figure 4. Amortized insertion time.

L2 cache. Using the least loaded bucket for every insertion performs worse than using the first bucket with space, since an additional cache miss is often incurred. Since the probe cost for cuckoo-based probes is independent of the load, the choice of ideal load is a trade-off between space usage and insertion cost. According to Figure 4, one could build a 50% full splash table for a cost of 450 cycles per insertion, while a 90% full table requires about 700 cycles per insertion. While the cost of insertions into a hash table with linear probing is less, the difference is only 20–40%.

4 Experimental Results: Probe

We have implemented the various algorithms in `C`. On the Pentium 4 we use Intel's `icc` compiler (version 9.0), which generated slightly more efficient code than `gcc`. On the Cell, we used IBM's `xlc` compiler (version 050418y) for the SPE. Maximum optimization was employed.

The first "generic" code version implements chained-bucket hashing as described in Figure 1, with bucket size `S` set to 64 bytes. This code contains no architecture-specific optimizations, and compiles on both the Pentium 4 and the Cell SPE. The second code version is similar to the first, except that it implements an open addressing hash table with quadratic probing rather than chained-bucket hashing. Both hash table variants are populated with a load factor of 0.75. We also consider quadratic probing with a load factor of 0.1, which represents trading off space for improved probe time. Table size is set to a power of two, so that a logical AND operation (rather than a remainder computation) can be used to calculate the hash slot.

The two other code versions are splash table probes implemented as described in Section 2.2. One of these versions uses SPE-specific SIMD instructions, while the other uses Pentium-4-specific SSE2 instructions. In both cases, these instructions were invoked using compiler intrinsics. When the table size fits in 16 bits, specialized versions of the hash and probe routines are used to save several instructions. Since the local store of the SPE is limited to 256KB, we limit the size of tables used on the Cell SPE. All code variants use multiplicative hashing.

The Cell SPE code is evaluated using IBM's `spusim`, which simulates the Cell SPE architecture, and is close to cycle-accurate. `spusim` allows one to determine how cycles were spent during program execution. For validation purposes, we also measure the performance of code running on a 2.4GHz IBM Cell blade. The Pentium 4 code is run on a 1.8 GHz Pentium 4 that is used solely for these experiments. The machine runs linux 2.6.10 and has a 256KB L2 cache, an 8KB L1 data cache, and 1GB of RAM. We measured the L2 latency, L1 latency, and TLB miss latency of the machine using the calibrator tool [11]; they were 273, 17, and 56 cycles respectively. During code execution we measured the values of hardware performance counters using the `perfctr` tool [14]. The branch misprediction penalty of the Pentium is assumed to be 20 cycles [17]. (It is 18 cycles on the Cell SPE [9].)

When presenting our results on the Pentium, we will be interested to know the effects of cache misses, branch mispredictions, and TLB misses on the final number of cycles needed. We multiply the counts for these events (obtained using the performance counters) by the latencies mentioned above. While this gives a reasonably accurate measure of the impact of branch mispredictions, it can overestimate the impact of cache misses and TLB misses because (a) they can be overlapped with other work, including other misses, and (b) multiple references to the same cache-line may be flagged as a miss multiple times even though a single miss penalty is paid. As a result of this overestimation, it may appear as though the aggregate L2 cache miss penalty exceeds the total execution time, an obviously inconsistent result. Nevertheless, it is very difficult to measure the overlapping and overcounting effects mentioned above to get a better estimate. We therefore include the results in this overestimated form, with the understanding that the true impact is some fraction of the plotted number of cycles.

We measure all Pentium 4 performance numbers in cycles. The Cell SPE is designed to operate at frequencies between 3 and 5 GHz [9]. Thus it is reasonable to assume that a cycle on an SPE is roughly the same amount of time as a cycle on the most recent Pentium 4 models.

Our performance results measure a large number of probes in a tight loop, simulating (part of) the probe phase of a hash join. The number of probes is large enough that probe costs dominate the initialization overheads. For the generic code, we interleave probes that are successful with probes that are unsuccessful. Splash table performance is not sensitive to whether or not the search is successful.

The x-axes of some figures show the total data structure size. This choice makes it easy to see transitions that hap-

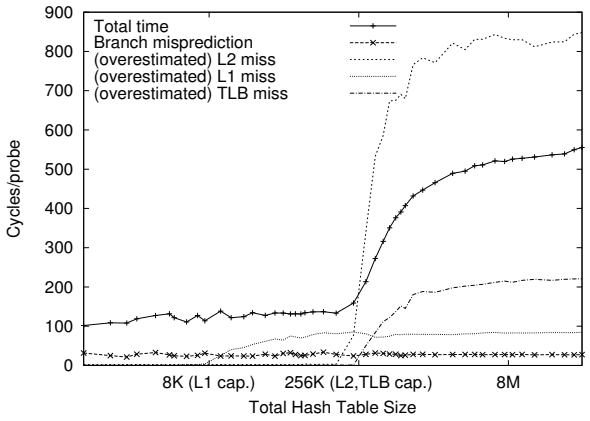

Figure 5. Chained-bucket hashing: Pentium.

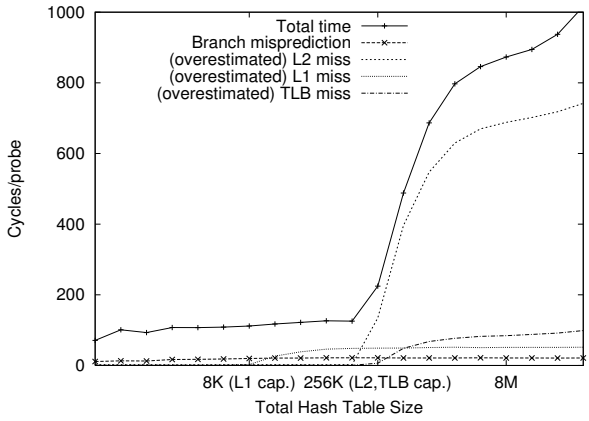

Figure 6. Quadratic Probing: Pentium.

pen when the table size goes beyond milestones such as the cache capacity. Splash tables can fit more entries into a fixed amount of memory than hash tables (Section 3). Thus, comparing the two methods at a given data structure size is somewhat biased in favor of hash tables.

4.1 A Fair Comparison?

To achieve good probe performance, we have optimized the probe phase of one hash algorithm so that (a) it does not use conditional branches, and (b) it uses SIMD instructions. To be fair, we should try to use the same optimizations to improve the performance of competing algorithms such as linear or quadratic probing, and chained-bucket hashing.

For chained-bucket hashing (Figure 1), it is possible to unroll the `for` loop if one is prepared to examine every slot (even empty slots) on each probe. If a bucket fits in a cache-line, the cost of checking all keys in the bucket may still be small relative to the cost of a cache miss. At the same time the inner `if` test can be converted to one using SIMD, changing the control dependency into a data dependency. We implemented this variant, and found that it was slightly slower than the original code.

To eliminate *all* branches, we must also unroll the `while` condition. If one kept track of the globally longest overflow chain, and required all probes to follow a chain of that length, then unrolling is possible. For lightly-loaded tables, it is likely that at most one overflow bucket is needed. Further, one can make the `next` pointer for a nonoverflowing bucket point to itself, so that processing the "second" bucket does not cause an extra cache miss. We also implemented this variant, but found that it was 50% worse than the original code, even though it had no branches. There are two reasons for this poor performance: (a) We are adding work to the common case (no overflow bucket) in order to make up for potential problems with rare cases (overflow buckets); and (b) chained-bucket hashing has a dependency in which the second cache miss for an overflow bucket cannot be overlapped with the cache miss for the original bucket; as a result, there are fewer opportunities for the processor to hide memory latency, compared with two independent memory accesses.

For linear or quadratic probing, we could peform similar unrolling optimizations to use SIMD and to remove branches. The problems identified for chained bucket hashing re-appear. To remove all branches for linear probing, one would have to do work proportional to the *longest* sequence of contiguous keys for *every lookup*. Again, this is adding significant work to the common case to handle rare cases, and would not provide a performance improvement.

4.2 Pentium

Figure 5 shows the performance (measured in cycles per probe) of chained-bucket hashing on the Pentium 4. For tables that fit comfortably in the L2 cache, the performance is between 108 and 135 cycles/probe. However, once the hash table exceeds the L2 cache size of 256KB (which also coincides with the capacity of the TLB) the cost increases dramatically, exceeding 500 cycles/probe. The branch misprediction penalty does not seem to depend on the table size. For L2-cache resident tables, the branch misprediction penalty accounts for about 20% of the total cycles. The number of instructions retired per probe for these experiments was 55, independent of hash table size.

Figure 6 shows the performance of quadratic probing with a load factor of 0.75 on the Pentium 4. The performance is slightly better than chained-bucket hashing for small tables, but substantially worse for large tables, since probes may have to touch several cache lines when the initial hash slots are occupied by other keys.

Figure 7 shows the performance of a splash table with $B = 4$ and $H = 2$ on the Pentium 4. The L2 cache miss measurement is anomalous, and should be ignored.[5] For

[5]It appears that the Pentium 4 cache-miss performance counter has a

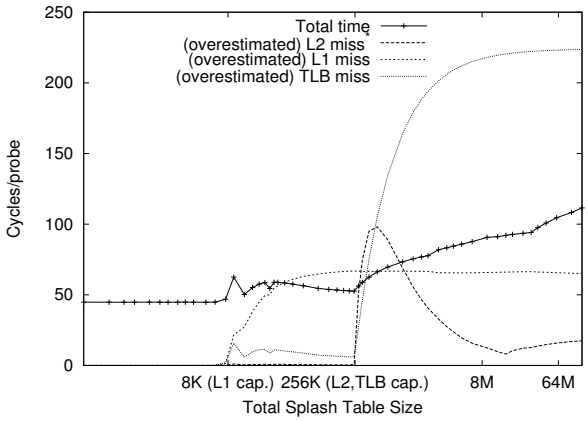

Figure 7. Splash tables: Pentium.

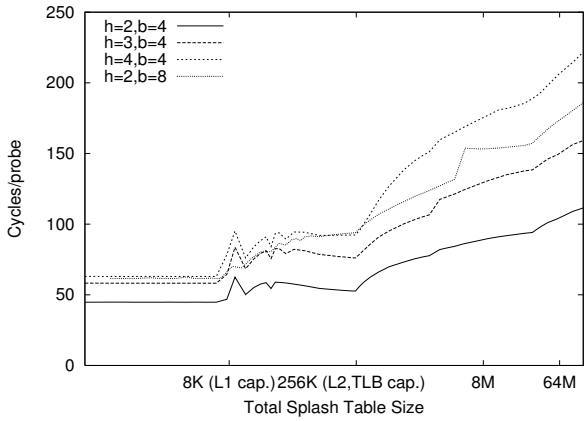

Figure 8. Varying B, H: Pentium.

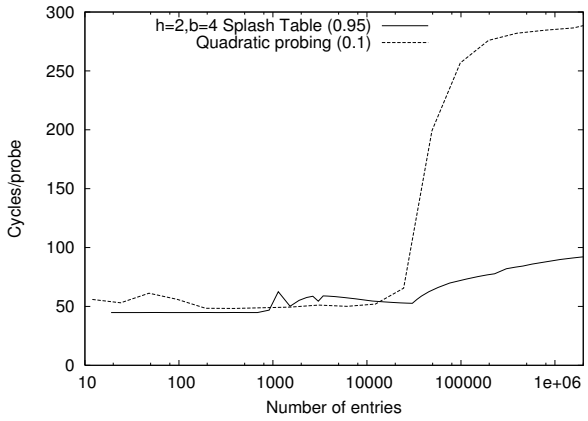

Figure 9. Splash tables versus lightly-loaded hash tables on a Pentium 4.

tables that fit comfortably in the L2 cache, the performance is between 45 and 63 cycles/probe. Once the splash table exceeds the L2/TLB capacity the cost increases modestly, to about 100 cycles/probe for a 64MB table. The branch misprediction penalty is essentially zero, and is not shown in the figure. The number of instructions retired per probe for these experiments was 27 when the table size fits in 16 bits, and 33 for larger table sizes.

The difference between Figures 5 and 7 is dramatic: a factor of two for small tables, and a factor of four for large tables. The improvement is attributable to several factors: (a) Eliminating the branch misprediction penalty; (b) Reducing the number of instructions needed per probe through the use of loop unrolling and SIMD operations; (c) Overlapping multiple cache misses, because the elimination of branching allows the CPU to better schedule multiple dependency chains through the instruction pipeline.

Figure 8 compares the total number of cycles taken by various splash table configurations on the Pentium 4. The $H = 2, B = 4$ configuration is about 30% better than the $H = 3, B = 4$ configuration. All configurations beat the chained-bucket hashing method. The small increment in the performance of the $B = 4$ methods at a table size of 2M corresponds to the transition from a 16-bit table size to a 32-bit table size.

We conclude this section by comparing a 95% full splash table with a 10% full hash table employing quadratic probing. This kind of hash table represents a choice to trade off space for improved probe time. Figure 9 shows the results on the Pentium 4. For tables smaller than the L2 cache, the time performance of the two methods is comparable, while the hash table uses 9.5 times as much space. However, for tables larger than the L2 cache, splash tables perform about three times better.

The measured number of cycles per probe in [18] appears smaller than what is presented here for L1-cache resident tables. However, [18] is measuring a probe method that does less work. In particular, the probe returns when it has the index of the matching record.

4.3 Cell

Figure 10 compares the total number of cycles taken by a splash table with $H = 2, B = 4$, and the two hashing methods on the Cell SPE. Since the SPE has no caching mechanism, the performance is not sensitive to the hash table size. The configuration shown corresponds to a splash table of size 128KB, and a hash table of size 155KB containing the same number of entries. (Recall that the SPE has only 256KB of local memory.) The total number of instructions per probe for the splash table is 29, comparable to that for the Pentium. These 29 instructions are executed in 20.5 cycles according to the simulator. When run on an actual Cell SPE processor, the time taken was 21.1 cycles per probe. The SPE has two execution pipelines that can

design flaw that causes it to ignore certain kinds of L2 misses [1], including the kind encountered in this code.

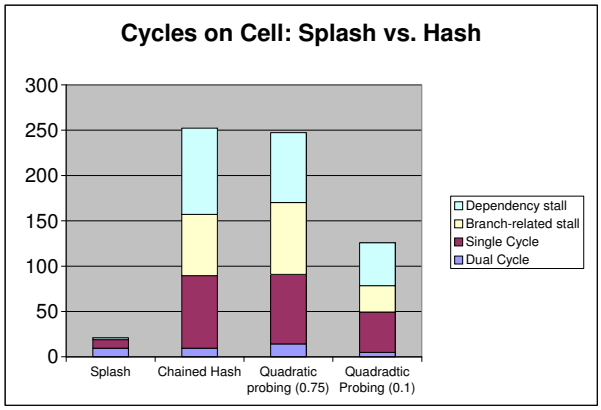

Figure 10. Simulated Cell SPE Performance.

execute memory operations in parallel with computation. About half of the useful cycles were spent executing two instructions ("dual cycle" in the figure). For a deeper comparison of the Cell SPE and the Pentium, see Section 4.5. The 20.5 cycles for the splash table is an order of magnitude better than the 250 cycles needed for the hash table.

For the chained-bucket hash table, 125 cycles per probe were needed, double that of the Pentium. The Cell SPE has very simple branch-prediction logic that (in the absence of compiler-generated hints) predicts that a conditional branch will not succeed. The SPE therefore suffers a higher misprediction penalty than the Pentium, as can be seen in Figure 10. The final component of the time taken for the hash table is the dependency-related stalls. Because the code branches often, it effectively becomes a single dependency chain. If an instruction takes 7 cycles to complete, and the next instruction needs to know the result of the first instruction, then no work can be done until the first instruction completes. (The SPE does not execute code out-of-order; it is up to the compiler to schedule instructions efficiently.) In contrast, the splash table implementation allows the compiler to interleave instructions from several consecutive probes. Because each probe is independent, there are far fewer dependency-induced stalls. See Section 4.5 for more discussion of this point. The results for quadratic probing are similar to those for chained-bucket hashing. Even with a load factor of 0.1, meaning that the table uses almost 10 times as much space as the splash table, the performance is still five times worse.

While we have not included performance graphs for other splash table configurations on the SPE, for completeness we mention that the $H = 3, B = 4$ table took 26 cycles/probe, the $H = 4, B = 4$ table took 31 cycles/probe, and the $H = 2, B = 8$ table took 27 cycles/probe.

The performance results shown so far used the `xlc` compiler with no special performance tuning beyond `inline` hints and using the maximum optimization level. We were able to obtain even better probe results by forcing the compiler to unroll eight copies of the inner probe loop rather than four. (Since the compiler did not seem to change the unrolling behavior in response to the `#pragma unroll(n)` directive, we explicitly coded the probe loop as two nested loops, with the inner of the two having eight iterations. The resulting code was only 2K larger than when unrolling by a factor of four.) We measured the performance for $H = 2, B = 4$ as 16.3 cycles per probe using the simulator, compared to 20.5 above, with each probe taking 23.3 instructions rather than 29. On an actual Cell processor, the performance was also 16.3 cycles per probe, compared with 21.1 above.[6]

4.4 Longer Keys and Payloads

To investigate the effects of key and payload size, we implemented a version of splash tables with 64-bit keys and 64-bit payloads. To hash a (32n)-bit key, we separately hash each 32-bit component of the key using multiplicative hashing (with different randomly chosen multipliers), and XOR the n hash values together.

On the Pentium with $H = 2$, two separate hash computations are required since the Pentium implementation performs full 32-bit multiplication to generate a 64-bit result. On the Cell SPE with $H = 2$, one hash computation suffices since 16-bit multiplication to 32-bit values allows four hash values in one SIMD vector. In both architectures, keys and payloads are explicitly represented as 64-bit quantities; two values fit in a 128-bit SIMD vector. We also implemented splash tables using 128-bit keys and 32-bit payloads, on both the Cell SPE and the Pentium 4.

To get reasonable performance numbers for the Pentium, we had to change the probe code slightly so that the probe keys were generated in SSE registers rather than standard registers. Our original code generated probes in standard registers and/or memory. There is no direct way to load a 64-bit value into an SSE register without going through memory. As a result, there were additional memory references in the inner loop competing for the small number (four) of allowed outstanding memory requests. Because of these additional requests, there was substantially less overlapping of memory latency, and there was a slowdown by more than a factor of three relative to the 32-bit case.

The performance results for longer keys and/or payloads are given in Figures 11 and 12. In Figure 11, the per-

[6]This improvement appears to be primarily due to better code generation by the compiler. For example, registers that were initialized with constants at the start of each loop and reused for other purposes later were each replaced with two registers: one of these was initialized to a constant outside the loop and not modified within the loop. It is unclear why the compiler found this rewrite when unrolling eight (or more) loops, but not four.

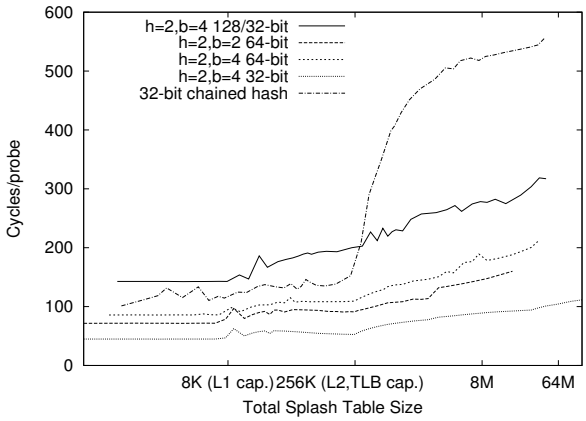

Figure 11. Longer keys/payloads: Pentium 4.

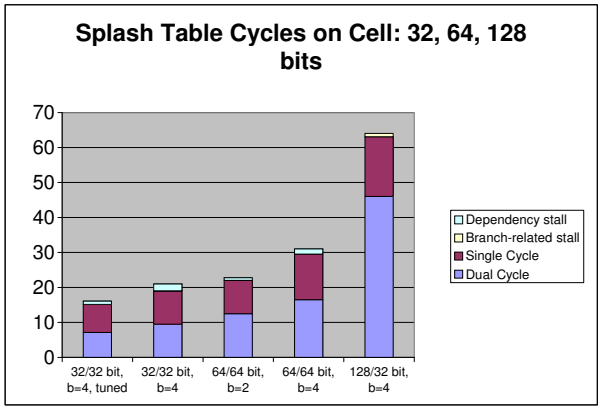

Figure 12. Longer keys/payloads: Cell.

formance of 32-bit chained-bucket hashing is included to aid comparison. Due to space limitations, we have not shown performance results for regular hashing techniques with larger keys/payloads. Nevertheless, it is clear simply by comparison with the 32-bit results for standard hashing methods (representing a lower bound for the 64-bit and 128-bit results) that (a) splash tables would outperform regular hash tables for 64-bit data for all table sizes, and (b) that splash tables would outperform hash tables for 128-bit keys on tables larger than the L2 cache.

4.5 Cell SPE vs. Pentium

It appears that on conventional code such as chained-bucket hashing, the Pentium 4 outperforms a Cell SPE by a factor of two (assuming the same clock frequency). Yet for specialized code such as the splash table code that is free of branches, the SPE outperforms the Pentium 4 by a factor of 3.5. How is it that a Cell SPE can execute 29 instructions in 21.1 cycles, while the Pentium 4 executes 27 instructions in 54 cycles for a table of the same size?

L1 cache misses cost 17 cycles on the Pentium, while the SPE's local memory latency is just 6 cycles. However even for L1-cache resident data sets, the Pentium takes 45 cycles per probe for 27 instructions.

The Cell SPE can sometimes execute two instructions in a single cycle, and can issue at least one SIMD instruction on each cycle. On the Pentium, in contrast, SSE2 instructions other than a load/store require at least two cycles before another SSE2 instruction can be executed.

A key difference between the two architectures is the issue of dependency stalls. By examining the assembly code for the two architectures, one can verify that in both cases the compiler has unrolled a number of probe operations (four, in this case) within the inner loop. These probe operations can be interleaved with one another. On the Cell SPE, this interleaving is effective at reducing dependency stalls. Because the SPE has 128 SIMD registers available, intermediate results can be kept in registers while other probes are being processed, making each probe independent of the others. In the actual splash-probe tests, between 57 and 78 registers were used, depending on B and H.

The Pentium 4, in contrast, has only eight SIMD registers available in 32-bit mode. Unlike regular registers, SSE registers are not renamed into a larger internal register file. As a result, interleaving of probe operations is not as effective. The compiler has two choices, neither of which is ideal. Operations could store their results in memory to avoid a register conflict between probes, but memory operations are more expensive than register operations. Alternatively, operations could use registers, but different probes within the loop become dependent because they need to use the same small set of registers.

One way to address this stalling behavior of the Pentium 4 might be to utilize its hyperthreading capabilities. Two logical threads would independently probe a shared hash table, and one thread could sometimes make progress when the other is stalled. One might anticipate a speedup by a factor of 1.5 for hashing with hyperthreading [16, 8].

4.6 Other Architectures

For our final experiment, we examine whether the nice scaling results demonstrated for a 1.8 GHz Pentium 4 in Section 4.2 hold for other architectures. Figure 13 shows the results for two Pentium 4 machines, the Cell Power processing element (PPE), and an AMD Opteron. The PPE code was compiled using the IBM `xlc` compiler version 1.0, while the Opteron runs the same code as the Pentiums, generated using `icc`. The vertical scale shows the probe cost as a fraction of the memory latency. (Memory latencies are measured using the Calibrator tool [11].) If this fraction is less than 1, it means that the amortized time for a probe is less than the latency of an L2 cache miss. Clearly,

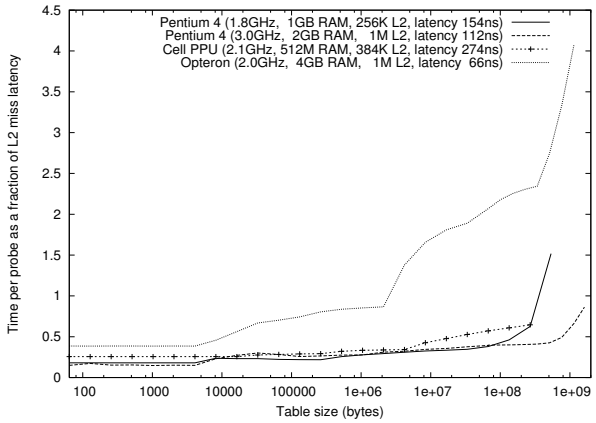

Figure 13. Probe cost ÷ memory latency.

for tables much larger than the L2 cache, this number can be less than 1 only if there is significant overlap of memory latency with other work/latency.

For both Pentiums and the Cell PPE, the ratio is about 0.5 for almost all of the memory range. Each probe incurs two cache misses, meaning that the system typically has at least four memory references in flight at the same time. The ratio for the Opteron is somewhat higher, due in part to its low memory latency that means that other parts of the computation have a larger relative impact on the overall time.

5 Related Work

Multiplicative hashing is universal, efficient, and it appears to work well empirically. However there is no theoretical guarantee that two or more randomly chosen multiplicative hash functions are independent. For a construction of a collection independent hash functions, see [5].

Cuckoo hashing [12] allocates two hash tables and inserts a key in the first table, whether or not the slot was occupied. If the slot had a previous occupant k, then k is removed from the first table and reinserted into the second hash table. A cascade of reinsertions may follow. Cuckoo hashing targets dictionary applications, in which load factors are typically small. In fact, the data structure has a space utilization of less than 50%. Pagh and Rodler provide both a theoretical analysis and an evaluation of an implementation of their methods [12]. Their implementation shows that (for lightly loaded hash tables with table size being a power of 2) Cuckoo hashing has a probe cost that is comparable to other hashing methods.

Fotakis et. al. define a variant of Cuckoo hashing called d-ary Cuckoo hashing [7]. When $B = 1$, splash tables reduce to d-ary Cuckoo hashing. Pagh and Rodler reported that Cuckoo hashing was sensitive to the hash function chosen, and that multiplicative hashing sometimes led to premature build failures [12]. When we used multiplicative hashing with $B = 1$, we observed the same behavior. We did not see this behavior for $B \geq 4$. Presumably there are still some degenerate key placements, but the splash table can tolerate them because of the extra capacity per slot.

Panigrahy studies a variant of cuckoo hashing with a bucket size B equal to two [13]. Theoretical results on space utilization are derived, but probe cost is not discussed. Erlingsson et al. provide some space utilization results similar to those of Table 1 [6], but they do not discuss probe costs. Dietzfelbinger and Weidling propose and analyze a different bucketized extension of cuckoo hashing [4]; the measured probe cost in their implementation on a Pentium 4 is approximately 1900 cycles/probe for a large table, an order of magnitude larger than the probe cost for splash tables.

Zhou and Ross have used SIMD operations to speed up various database operators [17]. The two main benefits identified in [17] are increased parallelism, and the reduction in branch mispredictions achieved by converting control dependencies into data dependencies.

6 Conclusions

Past work has shown that extensions of cuckoo hashing can achieve good space utilization. However, it has typically been assumed that these schemes perform no better than (and probably worse than) conventional hash tables since they require additional memory references and hash evaluations. The main contribution of the present work is to show that one can achieve *both* superior space utilization and superior probe time for bulk probes of small keys and payloads, for small or large tables.

References

[1] *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, September 2005.

[2] T. H. Cormen et al. *Introduction to Algorithms*. McGraw-Hill, 2001.

[3] M. Dietzfelbinger, T. Hagerup, J. K. Jainen, and M. Penttonen. A reliable randomized algorithm for the closest pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[4] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *ICALP*, pages 166–178, 2005. Extended version available at `http://www.tu-ilmenau.de/fakia/md-papers.html`.

[5] M. Dietzfelbinger and P. Woelfel. Almost random graphs with simple hash functions. In *STOC*, pages 629–638, 2003.

[6] U. Erlingsson et al. A cool and practical alternative to traditonal hash tables. In *Workshop on Distributed Data and Structures*, 2006.

[7] D. Fotakis et al. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005.

[8] P. Garcia and H. F. Korth. Hash-join algorithms on modern multithreaded computer architectures. Technical Report LU-CSE-05-001, Lehigh University, 2005.

[9] J. A. Kahle et al. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[10] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman, Redwood City, 1998.

[11] S. Manegold. The calibrator: a cache-memory and TLB calibration tool. (version 0.9e). Available from `http://homepages.cwi.nl/~manegold`, 2004.

[12] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[13] R. Panigrahy. Efficient hashing with lookups in two memory accesses. In *SODA*, pages 830–839, 2005.

[14] M. Pettersson. Perfctr (version 2.6.18). `http://user.it.uu.se/~mikpe/linux/perfctr/`.

[15] M. Thorup. Even strongly universal hashing is pretty fast. In *SODA*, pages 496–497, 2000.

[16] J. Zhou et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, pages 49–60, 2005.

[17] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[18] M. Zukowski et al. Architecture-conscious hashing. In *Workshop on Data Management on New Hardware*, 2006.