# 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. Payloads may be pointers (or record identifiers) to records, or they may represent values of an aggregate computed using hash aggregation.

We use an open addressing hash scheme based on a bucketized version of $d$-ary cuckoo hashing [4]. 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.

[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.)

Our main contribution is to enhance the probe phase of 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, and independent accesses can be overlapped. 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. [12] 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 [8] rather than [4], 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) [12] does not use SIMD instructions. (e) [12] does not use a universal hash function.

Dietzfelbinger and Weidling propose and analyze a different bucketized extension of cuckoo hashing [3]; the measured probe cost in their implementation on a Pentium 4 is

approximately 1900 cycles/probe for a large table, an order of magnitude more expensive than the results achieved here.

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 [6]. 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 3) 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. 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

We shall assume that both keys and payloads are 32 bit values. Longer keys and/or payloads are discussed in [10]. We assume that the hash table contains no duplicate keys[2] and that zero is not a valid payload value.

We use multiplicative hashing, which is universal [2], efficiently computable [11], and amenable to vectorization, so that we can compute multiple hash functions at once. We do not require that the table size be a power of 2; arbitrary tables sizes are handled efficiently, without the use of expensive division or modulus operations [10].

The build process is essentially the same as that described by Erlingsson et al [4]. 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.

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 [5]. 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*. One can typically achieve very high space utilization (95–99%) without encountering an insertion failure [4, 10]. Hash buckets that are not full are padded with records having a zero payload.

Given a hash table constructed as above, a probe 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. Because we always do the same number of comparisons, loop unrolling eliminates loop branches within the probe operation.

We can also avoid branch mispredictions in the test for a hash match 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 1 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.

## 3 Experimental Results

We focus on the performance of the probe algorithms. For a comparison of the build times, see [10]. We have implemented the various probe 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.

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

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

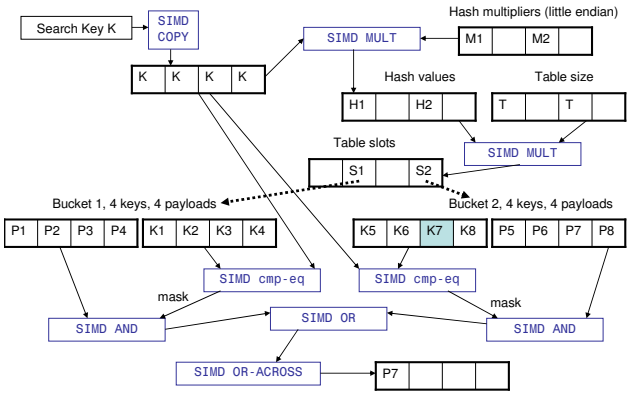


**Figure 1. Probe flowchart (K7 matches K).**

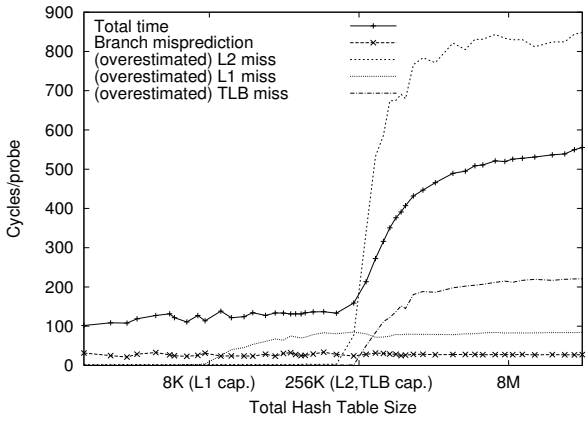


**Figure 2. Chained-bucket hashing: Pentium.**

We implemented an open addressing hash table with quadratic probing, and chained-bucket hashing 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. Unless otherwise mentioned, both hash table variants are populated with a load factor of 0.75. 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. 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 [7]; they were 273, 17, and 56 cycles respectively. During code execution we measured the values of hardware performance counters using the `perfctr` tool [9]. The branch misprediction penalty of the Pentium is assumed to be 20 cycles. (It is 18 cycles on the Cell SPE [6].)

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 [6]. 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 happen 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 [4]. Thus, comparing the two methods at a given data structure size is somewhat biased in favor of hash tables.

To achieve good probe performance, we have optimized the probe phase of one hash algorithm so that (a) it does not

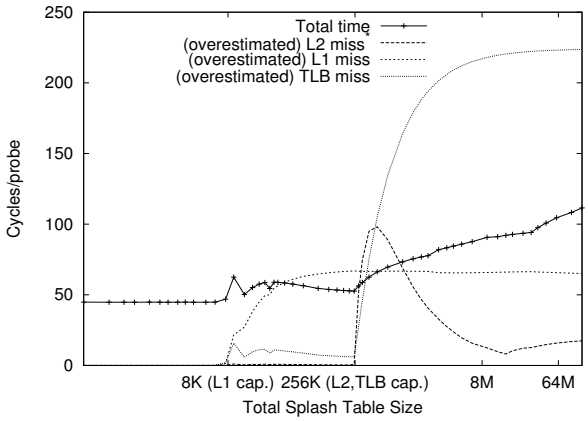


**Figure 3. Splash tables: Pentium.**

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. When we attempted these transformations on conventional hash algorithms, the performance worsened [10].

### 3.1 Pentium

Figure 2 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 is also the TLB capacity) 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. The performance of quadratic probing is slightly better than chained-bucket hashing for small tables, but substantially worse for large tables [10].

Figure 3 shows the performance of a splash table with $B = 4$ and $H = 2$ on the Pentium 4. (See [10] for other values of $B$ and $H$.) The L2 cache miss measurement is anomalous, and should be ignored.[4] For 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.

[4]It appears that the Pentium 4 cache-miss performance counter has a design flaw that causes it to ignore certain kinds of L2 misses [1], including the kind encountered in this code.

The difference between Figures 2 and 3 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.

For tables smaller than the L2 cache, the time performance of splash tables is comparable to the performance of a 10% full hash table, while the hash table uses 9.5 times as much space [10]. However, for tables larger than the L2 cache, splash tables perform about three times better [10].

The number of cycles per probe in [12] appears smaller than what is presented here for L1-cache resident tables. However, [12] is measuring a probe method that does less work. In particular, the probe returns when it has the index of the matching record rather than the value of the payload itself, and a non-universal hash function is used.

### 3.2 Cell

Figure 4 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 execute memory operations in parallel with computation. About half of the useful cycles were spent executing two instructions ("dual cycle" in the figure). The 20.5 cycles for the splash table is an order of magnitude better than the 250 cycles needed for the hash table.

The chained-bucket hash table needed 125 cycles per probe, 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 4. 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. In contrast, the splash table implementation allows the compiler to interleave instructions from several consecutive probes. Because each probe is independent, there are

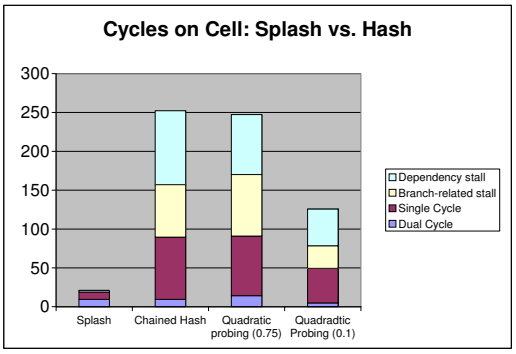


**Figure 4. Simulated Cell SPE Performance.**

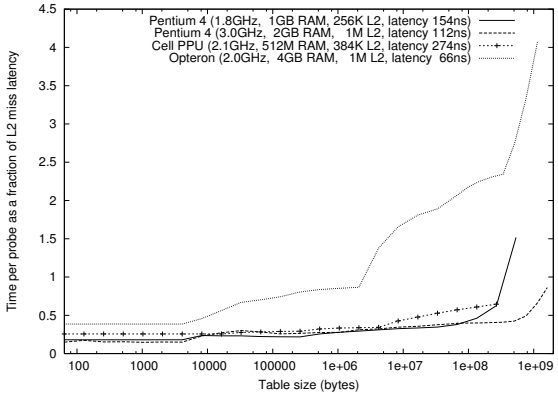


**Figure 5. Probe cost ÷ memory latency.**

far fewer dependency-induced stalls. The quadratic probing results are similar to those for chained-bucket hashing.

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. For an explanation of why these architectures have such contrasting performance characteristics, see [10].

### 3.3 Other Architectures

We examine whether the nice scaling results demonstrated for a 1.8 GHz Pentium 4 in Section 3.1 hold for other architectures. Figure 5 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 [7].) 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. 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.

## 4 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] 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.

[3] 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`.

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

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

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

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

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

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

[10] K. A. Ross. Efficient hash probes on modern processors. IBM Research Report RC24100, 2006. Available at http://domino.watson.ibm.com/library/CyberDig.nsf.

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

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