Improving Switch Statement Performance with Hashing Optimized at Compile Time

Jasper Neumann[1] and Jens Henrik Göbbert[2]

[1] `http://programming.sirrida.de`, `jn@sirrida.de`
[2] `jens.henrik.goebbert@rwth-aachen.de`

Abstract. Switch statements are elementary to most widely used high-level programming languages. If they consist of n sparse constant case labels contemporary compilers usually translate them into decision trees. This approach gives good $O(\ln n)$ performance, can easily cope with case label ranges, and can be tweaked to speed up the cases which are executed most frequently.
Often however, hashing performs much better:

– The main and most impressive virtue is its $O(1)$ performance, i.e. constant time independent of the number of cases.
– A big advantage over a decision tree is that the central dispatching point of hashing which is a rather expensive indirect jump even can be replaced by a table lookup if all jump targets only differ by constants.
– The consumption of the processor's branch prediction caches is reduced to the bare minimum.

We justify hashing by measurements and describe heuristics to determine fast and good-fitting hash functions at compile time.

Keywords: hashing, switch statement, sparse case labels, jump table, compiler, optimization

1 Introduction

Computers are good at computing — but not good at jumping, especially when they can not predict where to jump or whether to jump at all. Looking up a jump target is often faster than searching for it.

The main goal of this paper is to justify hashing and to point the direction for the incorporation of hashing into compilers. We focus on switch statements which dispatch on reasonable numbers (about 10 up to a few thousand) of sparse constant case labels, thus catching the cases where a jump table gets too big.

After giving a survey of the main methods to implement switch statements, we will present variants of hashing and their implementations and discuss their merits. We then present optimization techniques and discuss the measurements.

1.1 Motivation

Although known for about 60 years[21], almost all commonly used production quality compilers do not use compile time hashing for the implementation of switch statements. We wondered how hashing compares to the currently used techniques and why hashing has not yet found its way into most compilers. Switch statement consisting of many sparse case labels which is the area hashing is famous for are usually implemented with decision trees.

A *decision tree* is a quite effective method of "divide and conquer". The many needed conditional jump instructions naturally lead to a flood of entries in the branch optimization caches which might be needed elsewhere. An *else-if chain* is a degenerate kind of decision tree where all cases are tested one after another.

In the ideal case the decision is reduced to a single point. If the given range of the set of *case labels* is dense enough, we can uses the switch value as an index into a *jump table*. This is a well known and often applied method. However, for sparse case labels or large *case label ranges* this method usually leads to a much too large table.

1.2 Hashing

Hashing is a technique which inherits most features of the jump table approach but without its excessive memory consumption for n sparse *case labels*. This is done by compressing the set of case labels into a much smaller *hash range* of s values $\{0 \dots s-1\}$ with a *hash function* $h(x)$ such that the needed tables get a reasonable size.

Often, there are some *case labels* which are mapped to the same *hash value* and thus share the same *hash slot*; this circumstance is called a *hash collision*. To minimize the number of hash collisions, it is preferable that the hash function delivers many different values. The number of hash collisions can be reduced by enlarging the hash range, nevertheless however, the hash range should be kept reasonably small. The *load factor* $\alpha = {}^{n}\!/_{s}$ indicates the relative usage.

If there are any collisions, we have an *imperfect hash function* (section 4); a subsequent test for all matching case labels must follow.

In the absence of collisions we have achieved a *perfect hash function* (section 3); only one test for the case label is needed.

A *reversible hash function* (section 2) is a perfect hash function which can even be inverted (bijective) and thus no collision can ever occur. In this case a simple range check is sufficient to rule out non-matching case labels.[17] The plain jump table is the trivial case of a reversible hash function where the "hashing" (i.e. scrambling) is absent.

2 Reversible Hashing

We might be lucky and find a *reversible hash function*. In contrast to other hashing methods we only need to take care that the function value is a member of the hash range and there is no need for a table of case labels.[17] Unfortunately, finding a suitable and fast reversible function is difficult and only succeeds in some special cases such as the one described below.

2.1 Implementation

The probably most useful reversible instructions are subtract, rotate, and multiply by an odd number, because these operations can be combined to map equidistant numbers x_i to i:

$$x_i = d \cdot i + c\,. \tag{2.1}$$

The intuitive way is to subtract c and then divide by d, unfortunately however, a division is neither fast nor reversible. Our proposed alternative is to replace the division by a variant of the so-called *exact division*[7]. First of all we split our divisor into a product of an odd number a and a power of two (2^b) such that

$$d = a \cdot 2^b\,. \tag{2.2}$$

The division by 2^b is replaced by a rotate right operation by b.[17] To cope with the odd factor a, we calculate $\bar{a}$, its *multiplicative inverse*[19] which is only defined for odd numbers in this context (every odd number is relatively prime to a power of 2), hence the separation. The resulting formulas are

$$i = ((x_i - c) \overset{\text{rot}}{\gg} b) \cdot \bar{a}\,, \tag{2.3}$$

$$i = ((x_i - c) \cdot \bar{a}) \overset{\text{rot}}{\gg} b\,. \tag{2.4}$$

For "well-formed" x_i (as of eq. 2.1) both formulas give the same result i; for other values the results are not necessarily equal.

To implement this (e.g. in a compiler), one determines the smallest case label c and calculates the greatest common divisor d of the differences of the case labels and c. It does not matter whether the case labels are signed or unsigned. Any overflows have to be discarded. Needless to say, a subtraction of 0, a rotation by 0, and a multiplication by 1 should not produce any code. If c is positive, divisible by d and sufficiently small, we might omit the subtraction and instead insert default labels into the jump table.[15]

2.2 Example

For example, the case labels {100, 106, 112, 118, 124} can be mapped to {0, 6, 12, 18, 24} by subtracting 100, then to {0, 3, 6, 9, 12} by rotating right by 1, and finally to {0, 1, 2, 3, 4} by multiplying with the inverse of 3. All these operations are reversible. In this case we even achieved a minimal reversible hash function since the series has no holes, and hence the jump table need not contain default entries. The C source of fig. 2.1 can be assembled to the 32 bit x86 code of fig. 2.2.

Please note the amazingly compact code which does a subtraction, a divisibility check, a division, and a range check with only few instructions!

```
switch (x) {
  case 100: goto L100;
  case 106: goto L106;
  case 112: goto L112;
  case 118: goto L118;
  case 124: goto L124;
  default:  goto Ldefault;
}
```

Fig. 2.1. C source for switch on {100, 106, 112, 118, 124}

```
action:                         ; table of jump targets
  dd L100, L106, L112, T118, L124

switch:                         ; x = eax
  sub eax, 100                  ; subtract low bound
  ror eax, 1                    ; replacement of division by 2
  imul eax, eax, 0xaaaaaaab     ; multiply with inverse of 3
  cmp eax, 4                    ; compare with highest index
  ja  Ldefault                  ; in range? no: Ldefault
  jmp [action+eax*4]            ; yes: found; indirect jump
```

Fig. 2.2. x86 assembler source for fig. 2.1 using a reversible hash function

3 Perfect Hashing

Usually hash functions simply force all values to the hash range. We might for e.g. mask out the lower b bits to achieve the hash range $\{0 \dots 2^b - 1\}$. Since such a reducing mapping function is not reversible, we need to check whether the switch value is among the case labels which map to the same hash value. If there is at most one case label for each of the possible values of the hash range, we have achieved a *perfect hash function.* This is fortunate, since the check is thereby reduced to a single comparison against a case label fetched from a table.

Obviously, a hashing function must be fast enough to be of use. Therefore it makes sense to put some effort into finding a good perfect hash function.[4, 5]

There are algorithms which always achieve a perfect hash function by using additional tables. There are freely available code generator programs[6, 10] or libraries[3] which implement them. They concentrate on hashing strings but only a few of them can deal with integers[10]. The evaluation of this software is out of scope of this paper.

3.1 Probability for Finding a Perfect Hash Function

The probability that a randomly chosen hash function achieves perfect hashing for n case labels into s slots is[13]

$$P_s(n) = \prod_{i=0}^{n-1} 1 - \frac{i}{s} = \prod_{i=0}^{n-1} \frac{s-i}{s} = \frac{s!}{(s-n)!\, s^n}\,. \qquad (3.1)$$

The related diagram fig. 3.1 where $(1 - P_s(n))^k = 0.5$ and k is the number of tries describes the realistically achievable load factor $\alpha = {}^n\!/_s$ which gets fairly small for high case label counts, even if we try a vast number of hash functions; it is plotted for a 50 % chance to find a perfect hash function. Hence for large sets we will need to use a different approach to achieve perfect hashing[10] or to utilize imperfect hashing as described in section 4. Nevertheless at least for relatively small sets we should try to find a simple and fast perfect hash function as sketched in the next sections.

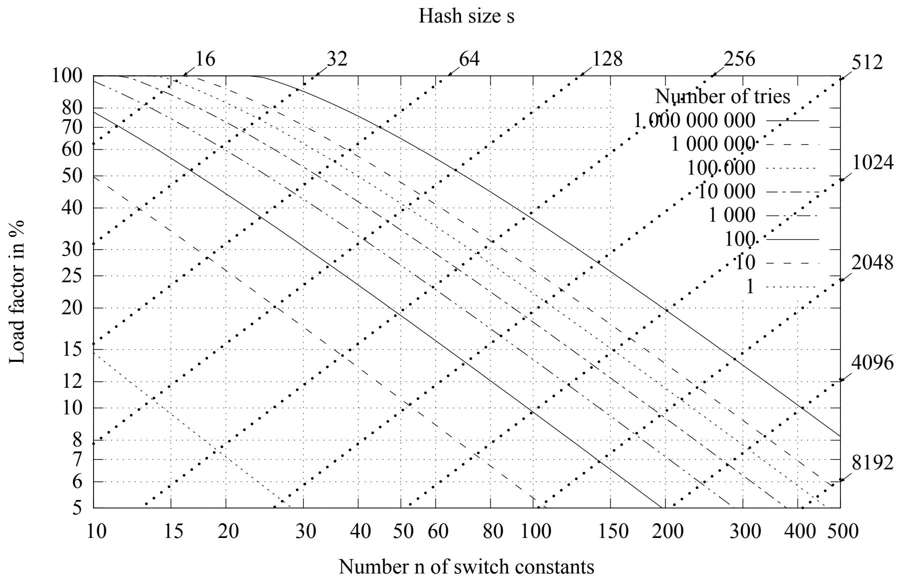

Fig. 3.1. Average achievable load factor α for perfect hashing at 50 % chance

3.2 Good Hash Functions

Good candidates for hash functions are listed below. M and S are used to map the switch value to the hash range and must be set accordingly. Q can be set to an arbitrary (valid) value.

$x \ \& \ M$ — mask out the lower bits

$(x \overset{\text{u}}{\gg} S)$ — extract the upper bits

$(x \overset{\text{rot}}{\gg} Q) \ \& \ M$ — extract a bit group using rotate

$((x \overset{\text{rot}}{\gg} Q) + x) \ \& \ M$

$((x \overset{\text{rot}}{\gg} Q) - x) \ \& \ M$

$((x \overset{\text{rot}}{\gg} Q) \oplus x) \ \& \ M$ — $\oplus$ is the *xor* operator

$(x \cdot Q) \overset{\text{u}}{\gg} S$ — Knuth's multiplicative hashing[11]

The last mentioned candidate (Knuth's multiplicative hashing) has become very interesting since most contemporary processors multiply quite fast and we achieve a good scrambling hash function with just two operations. Multiplications have once been very slow operations.

There are several other candidates as well[5] but most of them will give no better scrambling and/or take more time to calculate.

The hash size of all these functions is a power of two. Other sizes are possible but instead of shifting or bit masking typically require a *division* or a *modulo operation* which are often too slow to justify the savings. To visualize this the diagram fig. 3.1 also sports hash sizes s which are a power of 2.

3.3 Finding a Perfect Hash Function

No general formula has yet been found to calculate values for Q which achieve perfect hashing, hence we apply a brute-force algorithm; this is absolutely not elegant but it can be very effective.[4, 13]

For the first 6 candidate functions we should simply try all possible values of Q. For the last one however ($(x \cdot Q) \overset{u}{\gg} S$) this is too costly because there are way too many to choose from, instead we just can test some of them. We do not want to implement a superoptimizer[12, 17] since a compiler should be sufficiently fast.

At least for up to about 200 case labels we can usually afford to let a compiler test several thousand multipliers and achieve a reasonable load factor α. The usage of index mapping described in section 5 helps to tolerate fairly low load factors.

Assuming 32 bit calculations, we suggest trying values in a more or less randomized order such as starting from 0x 04d7 651f (a *de Bruijn cycle*[2, 19]) and incrementing by 0x 61c8 8647 (Golden ratio[11]).

3.4 Code Using a Perfect Hash Function

As an example for the realization of perfect hashing let us switch on the first 32 powers of 2. In this case we can even find a minimal perfect hash function by utilizing the aforementioned "magic" constant.

The C source of fig. 3.2 can be assembled to the 32 bit x86 code of fig. 3.3.

```c
switch (x) {
  case 1 << 0: goto L0;
  case 1 << 1: goto L1;
  case 1 << 2: goto L2;
  // ...
  default:     goto Ldefault;
}
```

Fig. 3.2. C source for switch on powers of 2

```
labels:                         ; table of case labels
  dd 0x1, 0x2, 0x4, 0x1000000, 0x8, 0x80000, 0x40, ...
action:                         ; table of jump targets
  dd L0, L1, L2, L24, L3, L19, L6, ...

switch:                         ; x = eax
  imul edx, eax, 0x04d7651f     ; hash / scramble
  shr  edx, 32-5                ; reduce to 5 bit
  cmp  eax, [labels+edx*4]      ; lookup case label
  jne  Ldefault                 ; match? no: Ldefault
  jmp  [action+edx*4]           ; yes: found; indirect jump
```

Fig. 3.3. x86 assembler source for fig. 3.2 using a perfect hash function

4 Imperfect Hashing

Even if we could not achieve a perfect hash function e.g. with the techniques described above (section 3.3), we can still use *imperfect hash functions.* This means that we must deal with hash collisions and therefore have to scan a list of case labels by using a small loop. The alternative of using a jump table to indirectly branch to a location where the case labels which match one hash value are checked is possible but forfeits the advantage of saving processor resources.

For a randomly chosen hash function and a given load factor α the expected number of comparisons is approximately $1 + {}^{\alpha}/_{2}$.

4.1 Finding a Good Imperfect Hash Function

Similar to finding a perfect hash function, we can look for a good imperfect hash function by taking the cost of the hash function and of the subsequent search for the correct case label into account. Thus we can reduce the already small average value of the number of comparisons. We suggest to utilize the hash functions mentioned in section 3.2, however it makes little sense to put too much effort into optimizing such an imperfect hash function since we can easily tolerate few collisions.

4.2 Data Structure for Imperfect Hash Function

The natural data structure for imperfect hashing is a linear linked list of case labels attached to each hash slot. This is called chaining. The straight-forward realization of using pointers is practicable if we want to dynamically add or remove elements. But as we have a static set of case labels a faster and more compact solution is shown in fig. 4.1. The principle is similar: For each hash slot `h` we supply an index into the case label table `v` which is sorted by their hash value; most often utilized cases should come first. To find the end of the list of case labels hashed to the same value, we subsequently compare the index with the one given by the next hash slot; the difference between two adjacent indexes is the length of the list. To get this running for the last hash slot too, we simply append an extra value to the index table `t`.[1] If all indexes are below 256, this array `t` can consist of bytes.

```
h = hash(x);
h1 = t[h];                // first index
h2 = t[h+1];              // last index (excl.)
while (h1 != h2) {        // end reached?
  if (v[h1] == x)         // found?
    goto action[h1];      // yes
  ++h1;                   // try next
}
goto Ldefault;
```

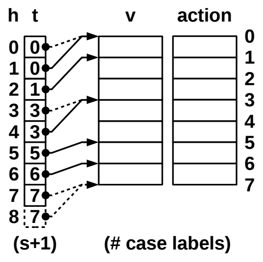

Fig. 4.1. C source for imperfect hashing

5 Index Mapping / Compressing Tables

For perfect hashing we can compress our case label table `v` and our jump table `action` as we did for imperfect hashing (fig. 4.1). However, as we have at most one case label to match, we can get rid of the comparison loop. We can also get rid of the extra entry at the end of the index table `t` because we also do not need to compare indexes but instead accept a comparison with any (non-matching) case label. See fig. 5.1 for an implementation.

```
h = hash(x);
h1 = t[h];
if (v[h1] != x)
  goto Ldefault;
else
  goto action[h1];
```

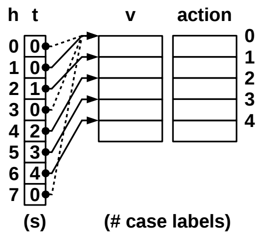

Fig. 5.1. C source for index mapping to compress hash range

If many jump targets are the same, we can similarly compress the jump table[17] via `t2` (fig. 5.2). For imperfect hashing this can also be done analogously. The principle of using reduced index sizes may be extended e.g. single bits if there are only 2 targets; bit tests[16, 17] are a variant thereof. We do not need any table at all if all targets are equal. This applies to all kinds of hashing and might even find other uses.

This mapping can optionally be combined with the mapping above as can be seen in fig. 5.3.

```
h = hash(x);
if (v[h] != x)
  goto Ldefault;
else
  goto action[t2[h]];
```

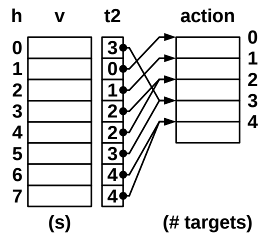

Fig. 5.2. C source for index mapping to compress jump tables

```
h = hash(x);
h1 = t[h];
if (v[h1] != x)
  else Ldefault;
else
  goto action[t2[h1]];
```

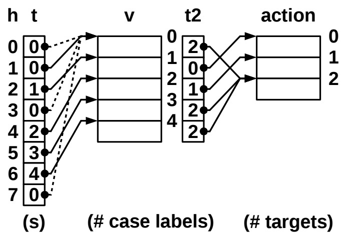

Fig. 5.3. C source for index mapping to compress hash range and jump tables

Furthermore it is possible to compress the jump table itself, e.g. by storing relative offsets instead of addresses.[18] Especially on 64 bit systems this is a win.

6 Circumventing Jumps by Lookup of Constants

Using hashing instead of decision trees can offer a way to get rid of the time consuming indirect jump, provided that all jump targets contain the same code which may differ by constants.[9] In this case we can achieve a colossal speedup as can be seen in fig. 7.2!

Consider the case that we have obtained an hash value that we are about to use as an index into a jump table. Instead of jumping to the corresponding target we can instead use this index to fetch the mentioned constant from a table and run the target code with this fetched value instead of a compiled-in constant. The realization thereof is shown in fig. 6.1 (C source), fig. 6.2 (using a jump table), and fig. 6.3 (using table lookup).

If also the default case has similar code, we can even get rid of the conditional jump as shown in fig. 6.4. For processors which do not support a conditional move instruction (`cmov`) which is used in fig. 6.4, there are alternative instruction sequences which do not need a conditional jump.[19] Please note that this technique where a conditional jump is replaced (by `mov` and `cmova`) can be much more generally used.

```
switch (x) {
  case 0:  y = X0;
  case 1:  y = X1;
  case 2:  y = X2;
  default: y = Xdefault;
}
```

Fig. 6.1. C source implementable with a lookup table

```
action:                              ; table of jump targets
  dd L0, L1, L2

switch:                              ; x = eax
  cmp eax, 2                         ; compare with highest index
  ja  Ldefault                       ; in range? no: Ldefault
  jmp [action+eax*4]                 ; yes: found; indirect jump
L0:
  mov edx, X0
  jmp Lend
L1:
  mov edx, X1
  jmp Lend
L2:
  mov edx, X2
  jmp Lend
Ldefault:
  mov edx, Xdefault
Lend:                                ; y = edx
```

Fig. 6.2. x86 assembler source for fig. 6.1 using a jump table

```
consts:                              ; table of constants
  dd X0, X1, X2

switch:                              ; x = eax
  mov edx, Xdefault                  ; preload the default constant
  cmp eax, 2                         ; compare with highest index
  ja  Lend                           ; in range? no: Xdefault
  mov edx, [consts+eax*4]            ; yes: found; load constant
Lend:                                ; y = edx
```

Fig. 6.3. x86 assembler source for fig. 6.1 using a lookup table

```
consts:                         ; table of constants
  dd X0, X1, X2, Xdefault

switch:                         ; x = eax
  cmp eax, 2                    ; compare with highest index
  mov edx, 3                    ; load index of Xdefault
  cmova eax, edx                ; in range? yes: eax, no: edx
  mov edx, [consts+eax*4]       ; yes: found; load constant
                                ; y = edx
```

Fig. 6.4. x86 assembler source for fig. 6.1 using a lookup table incl. default

7 Measurements

We have benchmarked the methods on an Intel® Core™ i7 920 using GCC 4.7.2 on 32 bit Windows™ with the options `-march=native` and `-O3`.

The test routines mapped an array of 1 000 random values $\{0, 100, 200, \ldots 100 \cdot (n-1)\}$ to other values as in fig. 7.1. The default case was never triggered. The result values were stored in an other array. For each of the hashing methods one multiplication and one shift or rotate was used.

```
void test_switch(const int* x, int* y) {
  int j;
  for (j=0; j<1000; ++j) {
    switch (x[j]) {
      case 0:   y[j] =  0; break;
      case 100: y[j] =  1; break;
      case 200: y[j] =  2; break;
      // ...
      default:  y[j] = -1; break;
    }
  }
}
```

Fig. 7.1. C source to benchmark a switch statement

Here is a short summary of the measured switch implementations, ordered from slow to fast. The results are summarized in fig. 7.2.

Else-if chain This is the trivial implementation of a switch statement with $O(n)$ behavior. It is the fastest method for very few case labels but also by far the slowest method for many. Please observe the $O(n)$ characteristic.

Decision tree (C switch by GCC) This is the switch implementation chosen by GCC, the compiler we used. Please observe the $O(\ln n)$ characteristic.

Imperfect hashing + jump table See section 4. This method has its application for more than about 20 case labels and is the most universal hash method. Here the test routine was configured to always map 2 case labels to the same hash value in order to simulate 1.5 comparisons per switch on average in case of success. In reality we would use an optimized hash function and get much better times, especially for small numbers of case labels.

Perfect hashing + jump table See section 3. This method is usually applicable for up to about 30 case labels if the suggested hash functions are used. Index mapping (section 5) helps to keep this method usable even for fairly low load factors α.

Reversible hashing + jump table See section 2. This method will only find its application in special cases where all case labels are equidistantly spaced.

Imperfect hashing + constant table See sections 4 and 6. Imperfect hashing is combined with a constant table thereby circumventing the indirect jump of the jump table methods.

Perfect hashing + constant table See sections 3 and 6. Perfect hashing is combined with a constant table. Mispredicted jumps could be avoided.

Reversible hashing + constant table See sections 2 and 6. This method is similar to the last item but with reversible hashing.

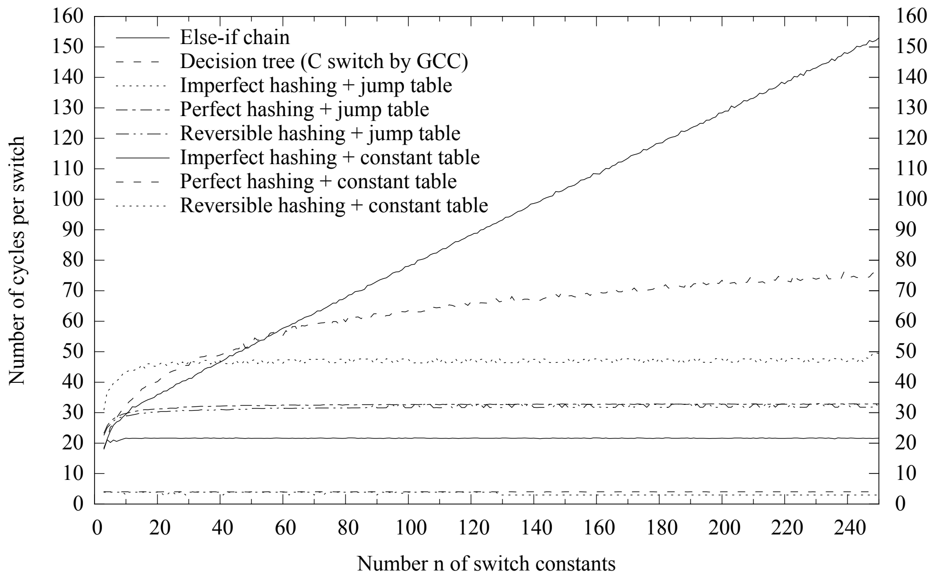

Fig. 7.2. Comparison of different switch implementations for random values

The diagram clearly shows the $O(1)$ characteristic for all hashing methods and demonstrates that when mispredicted jumps can be completely avoided (by perfect hashing and constant table), the processor can demonstrate its true power since nothing destroys its pipelining as described in section 6.

The cycles measured for reversible hashing were almost identical to the ones for perfect hashing.

For this processor the cross-over point of a decision tree or an else-if chain to perfect hashing using a jump table appears to be at about 10 case labels.

8 Conclusions

In this paper we gave reasons to use hashing as a means to increase runtime performance. We presented methods to find fast and good fitting hash functions and analyzed the applicability and needed effort at compile time and at run time. This was demonstrated by an artificial benchmark.

According to our measurements, — in situations where a value is tested against many sparsely distributed case labels — hashing is generally superior to a decision tree. This is especially true if jumping can be suppressed, i.e. by using perfect hashing and lookup of constants.

9 Future Work

There is still room for improvement since hashing is only one of several techniques a compiler can apply.[14] The tricky part comes when these techniques are combined.

If there are case labels or case label ranges with a very high probability these can be caught using a decision tree or a simple else-if-chain before treating the remaining ones with hashing (a jump table being a special case thereof). On the other hand case label ranges with a low probability can easily be checked last. This reordering can speed up the run time.[20] Without a suitable hint however the compiler usually assumes that all cases have equal probability.[17] We therefore strongly recommend providing a language feature describing the relative probability of case labels.[8] These values can be estimated by the programmer or derived from usage statistics (profiling information[17]).

Hashing can often also be used to switch on other types as well, especially strings.[4, 3, 6, 10] This comes very handy for applications which e.g. interpret XML streams or within a compiler.

References

1. Alcantara, D.: Efficient Hash Tables on the GPU. BiblioBazaar (2012)
2. Arndt, J.: Matters Computational: Ideas, Algorithms, Source Code. Springer (2010)
3. Botelho, F., Reis, D.d.C., Belazzougui, D., Ziviani, N.: CMPH – C minimal perfect hashing library, `http://cmph.sourceforge.net/`, 2013-08-28
4. Cichelli, R.J.: Minimal perfect hash functions made simple. Commun. ACM 23(1), 17–19 (1980), `http://doi.acm.org/10.1145/358808.358813`, 2013-08-28
5. Dietz, H.G.: Coding multiway branches using customized hash functions. ECE Technical Reports 308 (1992)

6. GNU: GPERF – GNU perfect hash function generator (2010), `https://www.gnu.org/software/gperf/`, 2013-08-28
7. Granlund, T., Montgomery, P.L.: Division by invariant integers using multiplication. ACM SIGPLAN Notices 29(6), 61–72 (1994)
8. IBM: C for AIX (1999), `http://sc.tamu.edu/IBM.Tutorial/docs/CforAIX/CforAIX_html/compiler/ref/rupraexf.htm`, 2013-08-28
9. Jambor, M.: Switch initializations conversion (2007), `http://gcc.1065356.n5.nabble.com/PATCH-middle-end-Switch-initializations-conversion-td531428.html`, 2013-08-28
10. Jenkins, B.: Minimal perfect hashing (1996), `http://www.burtleburtle.net/bob/hash/perfect.html`, 2013-09-15
11. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, vol. 3. Addison-Wesley Professional, second edn. (1998)
12. Massalin, H.: Superoptimizer: a look at the smallest program. SIGPLAN Not. 22, 122–126 (1987)
13. Ramakrishna, M.V.: A simple perfect hashing method for static sets. In: Koczkodaj, W.W., Lauer, P.E., Toptsis, A.A. (eds.) ICCI. pp. 401–404. IEEE Computer Society (1992)
14. Sale, A.H.J.: The implementation of case statements in Pascal. Softw., Pract. Exper. 11(9), 929–942 (1981)
15. Sayle, R.A.: Optimize tablejumps for switch statements (2001), `http://gcc.gnu.org/ml/gcc-patches/2001-10/msg01234.html`, 2013-08-28
16. Sayle, R.A.: Implement switch statements with bit tests (2003), `http://gcc.gnu.org/ml/gcc-patches/2003-01/msg01733.html`, 2013-08-28
17. Sayle, R.A.: A superoptimizer analysis of multiway branch code generation. In: GCC Developers' Summit. vol. 103 (2008)
18. Uh, G.R., Whalley, D.B.: Coalescing conditional branches into efficient indirect jumps. In: Proceedings of the International Static Analysis Symposium. pp. 315–329 (1997)
19. Warren, Henry S., j.: Hacker's Delight. Addison-Wesley Professional, second edn. (2012), `http://www.hackersdelight.org/`, 2013-08-28
20. Wienskoski, E.: Switch statement case reordering FDO. In: GCC Developers' Summit (2006)
21. Wikipedia: Hash table, `http://en.wikipedia.org/wiki/Hash_table`, 2013-08-28