7-1-1992

Coding Multiway Branches Using Customized Hash functions

H. G. Dietz
Purdue University, School of Electrical Engineering

CODING MULTIWAY BRANCHES USING CUSTOMIZED HASH FUNCTIONS

H. G. DIETZ

TR-EE 92-31
JULY 1992

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Coding Multiway Branches Using Customized Hash Functions[†]

H. G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47906
hankd@ecn.purdue.edu

Abstract

In most modem languages, there is a construct that allows the programmer to directly represent a multiway branch based on the value of an expression. In Pascal, it is the **case** statement; in C, it is the **switch** and in Fortran-90 the **SELECT.** However, it is quite common that the efficiency of these constructs is far worse than one might reasonably expect. This paper discusses the construction and use of customized hash functions to consistently improve execution speed and reduce memory usage for such constructs. Performance results are given, including some that lead to the suggestion that adding a population count instruction to the instruction set of a processor will greatly improve its hashing performance.

Keywords: multiway branches, hashing, compiler design, Pascal, C, Fortran-90, population count.

[†] This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013 and by the National Science Foundation (NSF) under Parallel Infrastructure Grant number CDA-9015696.

1. Introduction

Before discussing how compilers might code multiway branches, it is useful to review the constructs as they are defined in current high-level languages. In particular, we will consider the multiway branch constructs of Pascal [JeW75], C [KeR78] [ANS90], and Fortran-90 [ANS89]. As a simple example, we will consider code that selects and executes one of four subroutines determined by an integer baud, which has the value 110, 300, 1200, or 9600. The simplest — and most constrained — type of multiway branch is that specified by Pascal:

```
care baud of
    110:    a();
    300:    b();
    1200:   c();
    9600:   d();
end
```

Listing 1: Simple Pascal case

The above Pascal case statement is defined to perform *exactly* one of the subroutine calls. The definition of Pascal explicitly states that the effect of the above construct is ***undefined*** if the value of baud is not one of those listed[1]. Hence, the compiler is free to generate code based on the assumption that the value of baud must be one of the listed values.

In contrast, the C switch and Fortran-90 SELECT are defined to explicitly filter-out values that are not listed, and to make these values invoke a "default" action. In both C and Fortran-90, the default action is implicitly to skip over the entire construct, but a user-defined default action may be supplied:

```
switch (baud) {
    case 110:    a(); break;
    case 300:    b(); break;
    case 1200:   c(); break;
    case 9600:   d(); break;
    default:     error();
}
```

Listing 2: C switch

[1] The same semantics apply to the multiway branches generated by an optimizing compiler when fine-grain parallelization of a program merges multiple ordinary branch instructions into a single multiway branch [BrN90] [Die92]. In these compiler-generated multiway branches, the "case" values are bit vectors; each vector represents a possible set of true/false sequential branch decisions.

```
SELECT CASE (BAUD)
    CASE (110)
        CALL A()
    CASE (300)
        CALL B()
    CASE (1200)
        CALL C()
    CASE (9600)
        CALL D()
    CASE DEFAULT
        CALL ERROR()
END SELECT
```

Listing 3: Fortran-90 SELECT

Although this relatively subtle semantic extension beyond the Pascal construct significantly increases the cost of the construct, it is generally agreed to be worthwhile. Many dialects of Pascal have teen extended to allow default cases. However, this distinction is important, and this paper will distinguish between the treatment of unlisted values as undefined versus default.

Another important observation about the semantics of multiway branch constructs is that multiple values may select the same block of code. For example, suppose that 110 and 300 are to select the same function:

```
case baud of
    110:
    300:    e();
    1200:   c();
    9600:   d();
end
```

Listing 4: Shared cases in Pascal

```
switch (baud) {
    case 110:
    case 300:   e(); break;
    case 1200:  c(); break;
    case 9600:  d();
}
```

Listing 5: Shared cases in C

```
SELECT CASE (BAUD)
    CASE (110, 300)
        CALL E()
    CASE (1200)
        CALL C()
    CASE (9600)
        CALL D()
END SELECT
```

Listing 6: Shared cases in Fortran-90

Aside from being a notational convenience and helping to avoid replicating code, the mapping of multiple selection values into a single block of code yields an interesting additionad benefit when combinecl with the unlisted-undefined (Pascal) semantics. Simply stated, if 110 and 300 map into the same code address, then there is no need to distinguish between them. This effectively gives the search for a mapping another degree of freedom, and hence makes mappings easier to find.

A related semantic extension is supported by C, in which actions are allowed to "fall through" unless **break** statements are used to mark the end of each case:

```
switch (baud) {
    case 110:    a();
    case 300:    b();
    case 1200:   c(); break;
    case 9600:   d();
}
```

Listing 7: C **case** Fall-Through

would cause the value 110 to select execution of **a()**, **b()**, and c(),300 would select **b()** and c(), etc. However, this apparently dramatic difference has no impact on the mapping used to select the correct action, but merely omits some "jump" **(break)** instructions that would mark the ends of the actions. Thus, we can ignore this semantic difference without loss of generality.

A more significant extension of the multiway branch semantics appears in Fortran-90, in which intervals may be used to select actions. For example, all values between 110 and 300 (inclusive:) would select execution of F() in:

```
SELECT CASE (BAUD)
    CASE (110:300)
        CALL F()
    CASE (1200)
        CALL C()
    CASE (9600)
        CALL D()
END SELECT
```

Listing 8: Fortran-90 Interval CASE

Treatment of these range expressions was discussed briefly in [Sal81]. Given large ranges, it is clear that the range expression itself is the most compact way to specify the mapping;; given small ranges, we suggest that the range notation be used as a shorthand, and that the range's values should be enumerated to define the mapping. With this transformation, we may also ignore range operators without loss of generality.

Finally, it is useful to note that some languages, such as PL/I and Ada, have multiway branch constructs that do not require the selection values to be compile-time constants. Clearly, if the values are not constants, it is generally impossible to define an efficient mapping at compile time. However, it is possible to recognize when all values listed as selectors in a construct are compile-time constants, and then to treat that particular instance much like the C or Fortran-90 construct. The same detection algorithm can also be applied to treat a sequence of `if` statements as if it had been written using the C or Fortran-90 multiway branch. For example, assuming that `baud` is not modified by `a()`, `b()`, `c()`, and `d()`, any of the following C codes can be mechanically transformed into a single C `switch`:

```
if (baud == 110) a();
else if (baud == 300) b();
else if (baud == 1200) c();
else if (baud == 9600) d();
```

Listing 9: Nested `if`

```
if (baud == 110) a();
if (baud == 300) b();
if (baud == 1200) c();
if (baud == 9600) d();
```

Listing 10: `if` sequence

```
if (baud < 1200)
  if (baud == 110) a();
  else if (baud == 300) b();
else if (baud == 1200) c();
  else if (baud == 9600) d();
```

Listing 11: `if` Tree

2. Approach

Traditionally, multiway branches have been implemented by linear sequences or trees of comparisons and jumps or by jump tables. An excellent overview of the issues involved in using these encodings is given in [Sal81]. A clever combination of range checking and use of multiple jump tables is presented in [HeM82]. The selection between these various types of encodings in a compiler for PL.8 is discussed by [Ber85].

However, although the comparison and jump encodings are familiar in that they represent how a multiway branch might be encoded using only `if` statements, there is no reason to restrict a multiway branch to be encoded in that way. A similar comment applies to the use of a jump table; just because early multiway branch constructs, such as Fortran's "computed `GOTO`," were designed to be implemented directly by jump tables does not imply that the more general modem multiway branch should be implemented in that way.

A more fundamental way to view the multiway branch constructs outlined in section 1 is that any multiway branch construct defines a mapping whose domain is the set of selector (`case`) values and whose range is the set of code addresses that are the jump targets. Thus, the best encoding of this mapping is the best encoding of the construct. In computer terminology, such a mapping is simply a hash function.

The basic concept of a hash function is very simple, but the practical matter of generating good, low-cost, functions implementing particular mappings is surprisingly complex. A number of techniques have appeared in the literature on finding hash functions, but these techniques are all based on the fundamental assumption that all the hash functions searched will have the same fonn, i.e., all mappings would be implemented by the same computation except for changes in a few constants. It is our claim that the assumption of a particular form will often result in a far more costly hash function than can be achieved if the form can be varied. Thus, our approach is based on searching for the minimum cost hash function among a wide range of forms that differ algorithmically, as well as by the values of constants.

2.1. Searching Multiple Forms

In spirit, our approach is most similar to that of the "Superoptimizer" [Mas87] — a system which attempts to find a functional equivalent to a given instruction sequence by searching all possibly useful instruction sequences in order of increasing cost (increasing length of instruction sequence:,. This is done using self-modifying code to construct each test coding. To determine if the same function is implemented by the reference encoding and the coding under test, each possible input is evaluated by both and the results are compared. If the outputs differ, the test function is immediately rejected and the search continues with the next test coding. If all outputs match, the coding under test is reported as the solution. Thus, the Superoptimizer may be viewed as searching for the least expensive hash function that will map each value in the domain into its corresponding value in the range. The primary difficulty is that such hash functions are very rare, hence the search can take a long time and can generate excessively expensive instruction sequences.

The fact that the Superoptimizer can generate excessively expensive instruction sequences seems to contradict the claim that it searches all possibly useful instruction sequences. In fact, this is not contradictory; ignoring any failings of a particular Superoptimizer's implementation, the cheaper instruction sequences missed involve the use of data — and the Superoptimizer excludes constructions like lookup tables. Consider the example function from [Mas87] that maps each domain value d0 ∈ {0, 1, 2, ..., 99) into trunc(d0 / 10). The Superoptimizer sequence is 7 instructions long, but a 100-element lookup table trivially implements the same function using just 2 instructions. In other words, we need not find a function that directly produces the correct range value for each domain value. It is sufficient to find any collision-free hash function (CFHF') — i.e., a function that never maps two domain values into the same range value unless they have the same range value for the reference function. Given such a hash function, the reference function values are simply looked-up in a table indexed by the hash function.

CFHFs are much more common than hash functions that implement mappings without the use of a lookup table. Thus, search time should be correspondingly lower. It is also possible that the search space can be reduced so that only a select group of forms is considered; this may also allow those forms to be built into the search program, rather than constructing them using self-modifying code.

2.2 Controlling Lookup Table Size

The key problem in using a CFHF with a lookup table is that the index values can be sparse, requiring an impractically-large table. Indeed, if the size of the lookup table is ignored, the minimum cost hash function, HASH(n), is always to use n-min(domain) to directly index a lookup table whose size is proportional to max(domain)-min(domain)+1. Using such a scheme, the table for any of the trivial multiway branch examples in Section 1 would have 9600-110, or 9490, elements.

To minimize the size of the lookup table, we wish to find a hash function which, for a particular domain containing |domain| items, maps the elements one-to-one and onto the integer range {0, 1, 2, ..., |domain|-1}. Such a function is called a (minimal) perfect hash function, and can be used to implement any mapping from that domain with a table of size |domain|.

Although our tool will attempt to find a (minimal) perfect hash function with a table of size N, a function to perform this mapping might be hard to find and expensive to evaluate. By sacrificing the requirement that the function be onto, we greatly increase the probability that an appropriate function can be found. Such a mapping takes the domain into {0, 1, 2, ..., |domain|+k), where k≥-1, and all range values that are not mapped into by any domain value are "don't care" states, harmless except in that they consume memory space. Of course, it is necessary to consider time/space tradeoffs, since the number of don't care states could become very large and available memory space is always limited. One could even argue that caches and paged memory systems probabilistically make lookup time proportional to table size for large tables, so that eventually the cost of indexing the table would outweight the cost savings for the simpler hash function. In any case, some cost must be associated with the size of the data.

All of the above refers to collision-free hash functions. In addition, our tool takes advantage of two properties of collisions in order to speed the search and create cheaper hash functions. The first is that if the |domain|>|range| for the reference function, then the lookup table might be as small as |range|. In other words, we will accept hash functions that have collisions provided that the domain values that collide have the same range values in the reference function. The second is the observation that disambiguating between domain values that collided only has a cost when the domain value being hashed is one involved in a collision. For example, if we assume that all domain elements are equally probable as input, |domain|=100, and only two domain values collide, then the cost of the comparisons (or of a secondary hash function) to distinguish between those values is encountered only about 2% of the time. Thus, the expected cost for a hash function with collisions might well be less than that of a more complex hash function which has no collisions; the expected cost of disambiguating collisions is incorporated into the cost estimate for each hash function considered.

3. Implementation

As a proof of concept, we have implemented a system that attempts to find the minimum cost hash function to perform any given mapping. This system consists primarily of a set of C and AWK programs that automatically generate a C program that will find appropriate hash functions to implement any mapping. The search program takes full account of the machine-dependent costs of different forms for the target architecture, and need not be executed on the target machine; for example, all the results presented in this paper were obtained by running the searches on a 16,384-processing element SIMD supercomputer (a MasPar MP-1 [Bla90]), since that allowed a much larger set of forms and parameter values to be considered.

3.1. The Forms Table

To simplify customizing the set of forms searched, the forms are given by a table specifying all the form-dependent information. Each line in this table specifies a form:

- The first field is the "Formula," i.e. actual C code that will compute `HASH(n)`. The formula also may be parameterized by one immediate (constant) value called i.
- The next two fields specify the "Min" and "Max" values for i in this formula; the form is considered for each integer value of i between Min and Max (inclusive). The Min and Max can be specified as absolute values or a functions of various attributes derived from the mapping, such as logmax, the $\log_2$ of the largest value in the domain. If the formula is not parameterized, then Min=Max=0.
- The fourth field is a symbolic C expression for the execution "Cost" of the form as a function of i. For pruning the search, it is assumed that as i is increased from Min to Max, the Cost is non-decreasing.
- The final field describes how to "Print" the form. In the current version, this is simply a set of arguments to `printf()` to output the C code for the form selected — and is somewhat redundant in that the first field provides essentially the same information. However, this last field is separated-out so that it would be easy to modify the search program to generate machine-specific assembly code.

For example, using the results reported in table 2, we can construct a forms table describing only those form!; which were selected as optimal in implementing at least one of the test **mappings** for the Sparc processor. The resulting forms table is given as Table 1.

Formula	Min	Max	Cost	Print
`n>>i`	1	logmax	`N+SHR(i)+I`	`"n>>%d", i`
`n`	0	0	`N`	`"n"`
`(n>>i)^n`	1	logmax	`N+SHR(i)+I+XOR+N`	`"(n>>%d)^n", i`
`(n>>(n&i))`	1	logmax	`N+SHRN+N+AND+I`	`"(n>>(n&%d))", i`
`(n>>i)+n`	1	logmax	`N+SHR(i)+I+ADD+N`	`"(n>>%d)+n", i`
`(-n)>>i`	1	logmax	`NEG+N+SHR(i)+I`	`"(-n)>>%d", i`

Table 1: Sample Forms Table for Sparc

Notice that the forms given are missing a final step which ensures that the **table** bounds are not exceeded by using modulus. There are two reasons for this omission. First, this final step is common to all forms, hence, it can be assumed. Second, the operation used is only a true modulus if' the table size is not a power of two; power of two table sizes are **handled** by masking, and **generally** at lower cost than using modulus. This adjusrment is made **automatically** in the search **program.** Thus, the formula `n` is actually either `(n)%SIZE` with cost `N+MOD(SIZE)+I` or `(n)&MASK`[2] with cost `N+AND+I`, depending on whether `SIZE` is a power of two.

It is **useful** to further note that the Cost field expressions are not in terms of arbitrary names. Rather, a C program was constructed to experimentally determine approximate relative times for a variety of basic operations. When **run** on the target machine, the output of that C program is a set of `#define` directives that give **the** relative execution times for each of the **basic** operations. For **example, the** Sparc ratio between cost of `ADD` and cost of `MULN` was **measured** as **90:2323** **(i.e.,** multiply is about **26x** the cost of an add) — which explains why multiply operations are rarely **used** in the forms chosen as optimal for the Sparc.

33. The Search Code

A pure C program is constructed to efficiently search for the lowest-cost mapping. There are two basic algorithms involved in the search; one that guides the search **overall** and another that is applied to evaluate each form.

[2] Where `MASK=SIZE-1.`

33.1. Search Control Algorithm

There are several dimensions to the search for a hash function. Not only must the correct form be selected, but we also must find the correct parameter value and size of the hash table. There are many ways in which this could be done, however, some techniques prune the search space faster than others. We do not attempt to optimize the search order, but we do employ a few heuristics to improve it. We search power-of-two table sizes first because they use masking rather than modulus instructions, and the lower cost of masking tends to prune the search faster. In addition, searching smaller tables first tends to prune faster because there is less memory use cost. The overall algorithm is:

1. Set bestcost = cost of the best conventional encoding.
   (E.g., only want functions cheaper than optimal binary search.)
2. Set cursize = the least power of 2 ≥ |range|.
3. Set maxsize = (bestcost / cost per unit of memory use) - 1.
   (As big as the hash table can be before the table size itself makes some other function cheaper.)
4. If cursize > maxsize then goto 8.
5. For each form, use the form evaluation algorithm to find cheapest for hash table of size cursize.
   (If a new best is found, bestcost is updated, thus steps 3 and 4 may prune the search earlier.)
6. Set cursize = cursize * 2.
7. Goto 3.
8. Set cursize = |range|.
9. Set maxsize = (bestcost /cost per unit of memory use) - 1.
   (As big as the hash table can be before the table size itself makes some other function cheaper.)
10. If cursize > maxsize then done.
11. If cursize is a power of two then goto 13.
12. For each form, use the form evaluation algorithm to find cheapest for hash table of size cursize.
    (If a new best is found, bestcost is updated, thus steps 9 and 10 may prune the search earlier.)
13. Set cursize = cursize + 1.
14. Goto 9.

3.3.2. Evaluation of a Form

Two simple AWK scripts are used to convert the table of form descriptions into pure C code to evaluate the cost of using each form to implement the desired mapping. One script generates a C function that searches for the lowest cost form for a table size that is not a power of two; the other handles only power of two table sizes. In either case, the evaluation of each form is done by the same algorithm.

To evaluate each form, the hash function for each parameter value for that form is applied to hash all the domain items, and a hash table is used to detect conflicts. Rather than resetting (clearing) the table before each function is tested, we use a serial-numbering scheme to ensure that hash table entries made when trying the form for this value of i have values that do not overlap those made for any other value of i. The algorithm is:

1. Set i = Min for this form.
2. If i > Max then done.
   (The complete set of possible parameter values has been checked.)
3. Set cost = cost of memory use + execution time cost of this form for the parameter i.
   (Memory use cost accounts for table size.)
4. If cost ≥ bestcost then done.
   (This prunes based on the fact that the cost of a form is non-decreasing as i is increased.)
5. Set serial = serial + |range|.
   (Get a new base serial number for table entries.)
6. For each value d ∈ domain do:
   a) Set h = HASH(d), r = serial + position of mapping(d) in range.
      (HASH() is the computation given by the formula, with either the modulus or masking included.)
   b) If table[h] < serial then go to step f.
      (This table entry was empty.)
   c) If table[h] = r then continue with the next loop iteration.
      (This table entry already maps to the desired value.)
   d) Set cost = cost + cost of a collision.
      (The hash of d was the same as the hash value of some d' such that mapping(d) ≠ mapping(d').)
   e) If cost ≥ bestcost then exit the for loop.
   f) Set table[h] = r.
      (Reserve hash table entry h for mapping(d).)
7. If cost < bestcost then record this form as the new best and set bestcost = cost.

8. Increment i.
9. Go to step 2.

It is significant that the above algorithm is also trivially adapted for a SIMD (Single Instruction stream, Multiple Data stream) parallel search. Indeed, two more AWK scripts were created to generate MPL [Mas91] programs to perform the parallel search on a 16,384 processing element MasPar MP-1 supercomputer [Bla90]. The only changes to the form evaluation algorithm involve declaring a few variables as parallel (plural, in MPL) and changing a few algorithm steps:

1. Set i = Min for this form + the current processing element number.
7. If cost < bestcost and cost for this processing element is the lowest then record this processing element's form as the new best and set bestcost = cost for the processing element.
8. Set i = i + the number of processing elements.

Notice that since the MasPar MP-1 has 16,384 processing elements, there may be up to a 16,384-way tie for the lowest cost in step 7. Hence, the parallel version may select a different parameter value (i.e., value of i) from that found by the sequential search. To force consistent behavior, our MasPar code will always resolve such a tie by taking the lowest parameter value — exactly as the sequential search would.

How much speedup did we get? That depends greatly on the forms used. SIMD parallelism in the above is not across forms, but rather across groups of up to 16,384 different parameter values. For the forms listed in table 1, the maximum possible parallelism is the $\log_2$ of the maximum domain value — presumably, 32 or less. In such a case, the MasPar yields a speedup of less than two over a modem workstation (e.g., a Sparc). In contrast, some forms have parameters that span millions of values, in which case the MasPar often provides a speedup of a thousand or more. As discussed in section 4.2, it took many long runs on the MasPar to demonstrate that omission of most of the forms would have relatively little impact on the cost of the best form found. In other words, the speedup is irrelevant; we simply used a supercomputer to quickly determine the set of forms to use for a production version of the system that can easily run in reasonable lime on a workstation or PC.

33. Use of Cost Information

Although the search and form evaluation algorithms do not seem to directly manage the various semantic differences noted in section 1, the system actually does take these differences into account. All of these variations are handled simply by adjusting the cost computations.

33.1. Default Vs. Undefined Semantics

Given a CFHF, we have the complete implementation of the undefined semantics for values not $\in$ domain, All the hash functions generated by the above scheme will take any input and map it into some table entry; to implement default semantics, we must simply test if the value that mapped into this table entry is actually the domain member that was intended to map there. For example, for the Sparc the cheapest CFHF for the domain (110,300, 1200,9600) was the function ((n>>7)&3). This maps 110 into 0, 300 into 2, 1200 into 1, and 9600 into 3; however, it would also map 2400 into 2. Thus, we insert code for each hash value to check: if the value hashed was the domain value intended. I.e., code something like:

```
table:  data r110, r1200, r300, r9600

        r <- ((n>>7)&3)   ;compute hash(n)
        jump table[r]
r110:   if (n != 110) goto default
        { case for 110 }
r300:   if (n != 300) goto default
        { case for 300 )
r1200:  if (n != 1200) goto default
        { case for 1200 }
r9600:  if (n != 9600) goto default
        { case for 9600 }
```

Listing 12: Handling Default Semantics

The cost overhead for this treatment is easily predicted, since it is simply the execution time of one compare and jump plus the memory use penalty associated with the code implementing these compares and jumps (usually, a few words for each domain element). Adding these costs, no other changes need be made to the search algorithm.

The CFHF above used all hash values. If there were some hash values that no domain element mapped into, the additional compares and jumps would be omitted for those hash values and the jump table entry would lead directly to the default. Thus, this can also be effectively modeled by simply adding the appropriate cost, which is always the compare and jump cost times |domain|.

If the HF found is not a CFHF, there will be additional overhead in handling the collisions as discussed in the next section, but nothing else is different.

33.2. Handling Collisions

As written above, the algorithm seems to find an HF which is not necessarily a CFHF. Surprisingly, all the variations on the handling of collisions are implemented by simply changing the cost associated with a collision, as applied in step 6d of the form evaluation algorithm.

If we wish to obtain an HF which minimizes expected runtime, it is somewhat surprising that disambiguating conflicts caused by a simple hash function is often cheaper than using a more sophisticated hash function that has no conflicts. The reason is simple. Suppose that K of the |domain| values in the domain of the mapping cause conflicts. Whenever one of those K values is hashed, we will need to perform some additional tests — linear search, an optimized binary search, or even a secondary hash function. Whichever we choose, the memory use cost is trivially computed and the execution cost is simply the expected cost of executing the additional instruction sequence. Note that even a high cost method (e.g., linear search) tends to have a very low expected execution cost because the number of items searched is rarely more than two and the probability of executing the additional search at all is only K/|domain|, which is typically less than 10%. The current version assumes that an optimal binary search will be used. to disambiguate collisions; the probability used to weight each execution cost is 2/|domain| for each collision.

Further, suppose that we wish to ensure that only a CFHF will be selected. All we need do is make the weighted cost for a collision 2 the cost for the best CFHF found thus far.

4. Results

In order to test the effectiveness of the above approach, we conducted two separate types of tests. The first involved taking the example cases presented for previous coding techniques, determining the optimal hash encoding, and then comparing the previously published result with our result. The second involved taking a large, hopefully statistically significant, set of mechanically generated test problems and observing the performance of the search algorithm and the solutions found.

4.1. Example Cases

In order to support a direct comparison with previous work, in this section we present the lowest-cost encodings our system found for the same example cases that appeared in [HeM82]. The costs we used are those reflecting the measured performance of a Sparc processor and the unspecified case values are treated as undefined (the standard Pascal interpretation).

Our first example, shown in listing 13, is the one used by [HeM82] to illustrate a Pascal `case` statement with case values that are too sparse to make a jump table implementation effective. As per Sale's paper [Sal81], it is suggested that the best encoding is an optimal binary search. However, our system easily found hash functions with lower time and space wst. The CFHF found by our system if all domain elements must be mapped to unique hash values is given in table 2. However, our system can find even cheaper solutions by allowing domain elements that map into the same range value to "share" the same hash value. The cheapest such function also is given in table 2.

```
case J of
    3, 5, 4:   stmt1;
    100:       stmt2;
    200:       stmt3
end
```

Listing 13: Pascal `case` from [HeM82]

HASH(J)≡(((J>>3)^J)&7)		HASH(J)≡((J>>5)&3)	
HASH(J)=0	{100:stmt3}	HASH(J)=0	{3:stmt1, 4:stmt1, 5:stmt1}
HASH(J)=1	{200:stmt4}	HASH(J)=1	-
HASH(J)=2	-	HASH(J)=2	{200:stmt3}
HASH(J)=3	{3:stmt1}	HASH(J)=3	{100:stmt2}
HASH(J)=4	{4:stmt1}		
HASH(J)=5	{5:stmt1}		
HASH(J)=6			
HASH(J)=7			

Table 2: CFHF and CFHF with Sharing for Listing 13

The other example from [HeM82], as shown in listing 14, involves a **case** statement whose case values occur in a series of "runs." This is precisely the type of construct Hennessy's paper attempts to optimize, by using a set of range tests and multiple jump tables (one for each run).

```
case K of
    1:      stmt1;
    2:      stmt2;
    3:      stmt3;
    4:      stmt4;
    5:      stmt5;
    6:      stmt6;
    7:      stmt7;
    8:      stmt8;
    1001:   stmt9;
    1002:   stmt10;
    1003:   stmt11;
    1004:   stmt12;
    2001:   stmt13;
    2002:   stmt14;
    2003:   stmt15;
    2004:   stmt16
end
```

Listing 14: Pascal **case** with Runs from [HeM82]

HASH(K)=(((K>>8)+K)615)	
HASH(K)=0	-
HASH(K)=1	{1:stmt1}
HASH(K)=2	{2:stmt2}
HASH(K)=3	{3:stmt3}
HASH(K)=4	{4:stmt4}
HASH(K)=5	{5:stmt5}
HASH(K)=6	{6:stmt6}
HASH(K)=7	{7:stmt7}
HASH(K)=8	{8:stmt8, 2001:stmt13}
HASH(K)=9	{2002:stmt14}
HASH(K)=10	{2003:stmt15}
HASH(K)=11	{2004:stmt16}
HASH(K)=12	{1001:stmt9}
HASH(K)=13	{1002:stmt10}
HASH(K)=14	{1003:stmt11}
HASH(K)=15	{1004:stmt12}

HASH(K)=(K&31)	
HASH(K)=0	-
HASH(K)=1	{1:stmt1}
HASH(K)=2	{2:stmt2}
HASH(K)=3	{3:stmt3}
HASH(K)=4	{4:stmt4}
HASH(K)=5	{5:stmt5}
HASH(K)=6	{6:stmt6}
HASH(K)=7	{7:stmt7}
HASH(K)=8	{8:stmt8}
HASH(K)=9	{1001:stmt9}
HASH(K)=10	{1002:stmt10}
HASH(K)=11	{1003:stmt11}
HASH(K)=12	{1004:stmt12}
HASH(K)=13	-
HASH(K)=14	-
HASH(K)=15	-
HASH(K)=16	-
HASH(K)=17	{2001:stmt13}
HASH(K)=18	{2002:stmt14}
HASH(K)=19	{2003:stmt15}
HASH(K)=20	{2004:stmt16}
HASH(K)=21	-
HASH(K)=22	-
HASH(K)=23	-
HASH(K)=24	-
HASH(K)=25	-
HASH(K)=26	-
HASH(K)=27	-
HASH(K)=28	-
HASH(K)=29	-
HASH(K)=30	-
HASH(K)=31	-

Table 3: HF and CFHF for Listing 14

As one might suspect, it is relatively difficult to find CFHFs when there are multiple runs in the case values[3]. The cheapest HF found for this particular data set was not a CFHF, but the HF given in table 3. Case values of 8 and 2001 both hash to the same table entry (hash value 8). Thus, although hash values other than 8 can directly jump to the appropriate statement, the hash value 8 case must jump to an instruction sequence that tests to see whether the value hashed was 8 or 2001, and then jumps to the appropriate location. Fortunately, the additional execution time of this comparison only has a 2/16 probability of occurring, so it is easy to see that the HF is both faster and smaller than an implementation using range tests to choose between multiple jump tables.

By changing the apparent cost of a collision (as discussed in section 3.3.2), we forced our system to find the lowest cost CFHF for the `case` statement of listing 14. The result is the remarkably simple CFHF given in table 3. In fact, the CFHF in table 3 is so simple compared to the HF in table 3 that it is difficult to see how the HF could have lower cost. The answer is

[3] In fact, this observation inspired the additional set of tests presented in section 4.2.2.

simply that despite a more complex hash function and one collision, the HF is cheaper than the **CFHF** because the HF uses a much smaller hash table — **i.e.,** ***it*** uses ***far less memory space.***

Although our system will not always find a cheaper implementation, it did find one for every **non-trivial** example in the papers cited. The success rate would be reduced if we applied the **unspecified→default** semantics, as used in C and Fortran-90, but that effect. is relatively small. **Thus,** the system was given a "stress test" to determine what its limits are.

43. **Performance** Statistics

The MPL **(SIMD** parallel) version of our system was used to find hash functions for a wide variety of test cases. The MPL version was used because it runs on a 16,384-processingelement **MasPar MP-1** supercomputer, hence, we were able to consider an abnormally **wide** variety of forms.

Since the pruning of the search depends heavily on the relative costs of **forms,** it is also important to **determine** how the system performs for a variety of target machines. Three common **processors** and one idealized processor were selected as targets:

- Intel **386sx.** Selected for its popularity as a CISC microprocessor, used primarily in **PC's.**
- Sun **Sparc.** This was selected as an example of a RISC processor, typically used in UNIX workstations.
- MIPS **R3000.** Selected as a second example of a RISC processor, also used in UNIX workstations.
- Super. An idealized processor whose characteristics approximate those: of processors used in supercomputers.

For each **real** target machine, the relative costs for different types of instructions were empirically **determined** using the C program described in section 3.

4.2.1. HF/CFHF Search for Random Mappings

To determine how the system performs, a set of 1,280 random mappings were created. The **|domain|** was from 2 to 128 values, each of which was between **0** and 65,535 **inclusive.** Range values were randomly selected between **0** and **|domain|-1,** inclusive. Each mapping was considered both with and without sharing of hash values, for a total of 2,560 input **mappings.** All 2560 input sets were used to find the cheapest HF and the cheapest **CFHF** for each of the real target **architectures.** In each case, 142 forms and millions of parameter values were considered.

Figure 1 shows the average relative cost for the cheapest HF (not necessarily a CFHF) found **versus** a traditional optimal binary search. The rather surprising results show approximately a **4x** improvement (reduction to about 0.25 cost). The cost is low even for **very** small data sets **primarily** because the data sets with sharing allowed often have near zero **cost** for small data sets; even without sharing the cost is often low because the table uses less **memory** than the instructions implementing binary search. The jagged waveform pattern is due **to** the fact that power-of-two table sizes have a significant cost advantage in that they use **masking** rather than modulus to confine hash values to the table size.

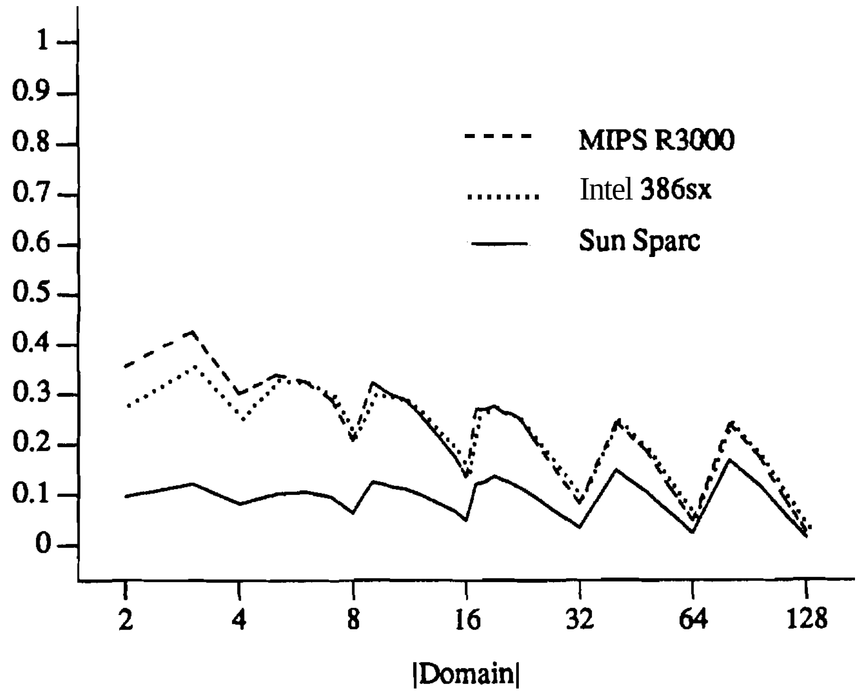

Figure 1: HF Relative Cost for **386sx, Sparc,** and R3000

Clearly, a significant **performance** increase was obtained. However, the use of **HFs** instead of **CFHFs** requires insertion of comparisons to disambiguate where collisions occur, and this **complicates** the coding. Thus, it is useful to consider what the performance **would** be if we required the **HFs** to all be **CFHFs.** The results of this **are** shown in figure 2. **The** remarkable similarity between figures 1 and 2 is due to the fact that the cheapest HF found is often a **CFHF.**

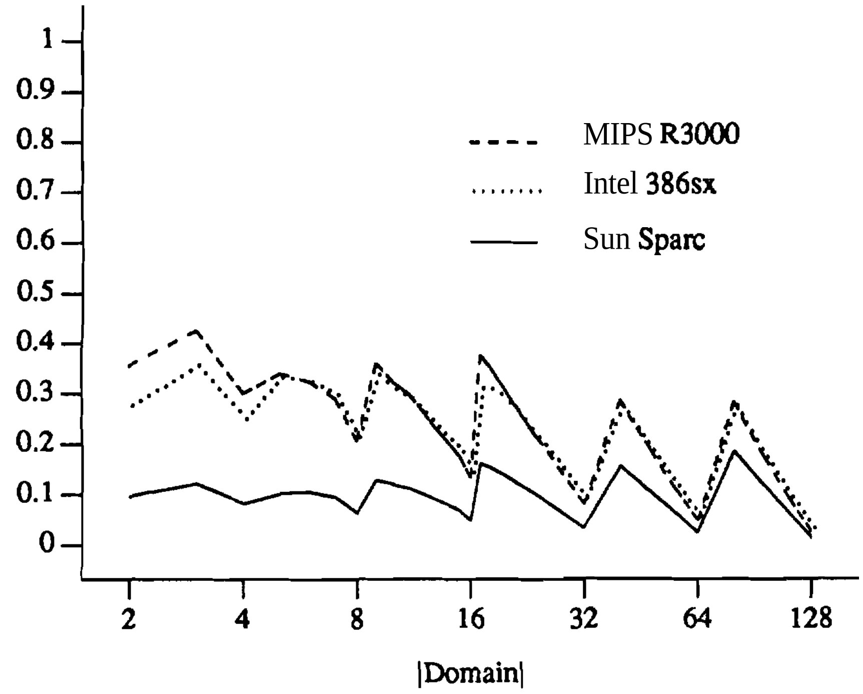

Figure 2: CFHF Relative Cost for 386sx, Sparc, and R3000

The next obvious question is how long did it take to find these functions? This is a very difficult question to answer, since it depends greatly on how many forms are considered and might depend heavily on the particular mappings searched for. The absolute search times for a large set of forms using the MasPar MP-1 supercomputer are relatively meaningless, however, the trends are significant. Figures 3 and 4 show, respectively, the average time taken (in seconds) to find the cheapest HF and to find the cheapest CFHF.

As expected, search times grow slowly as larger domains are considered and searching for a CFHF generally takes longer than searching for an HF. For the HF search, there is a dip in search time when |domain| is slightly greater than a power of two — presumably because a very cheap solution is likely to be found very early in the search when the next largest power-of-two table size is considered. There is a similar jagged pattern in the CFHF search times, but the pattern is obscured by the fact that there is more variation in CFHF search times, especially as |domain| gets large.

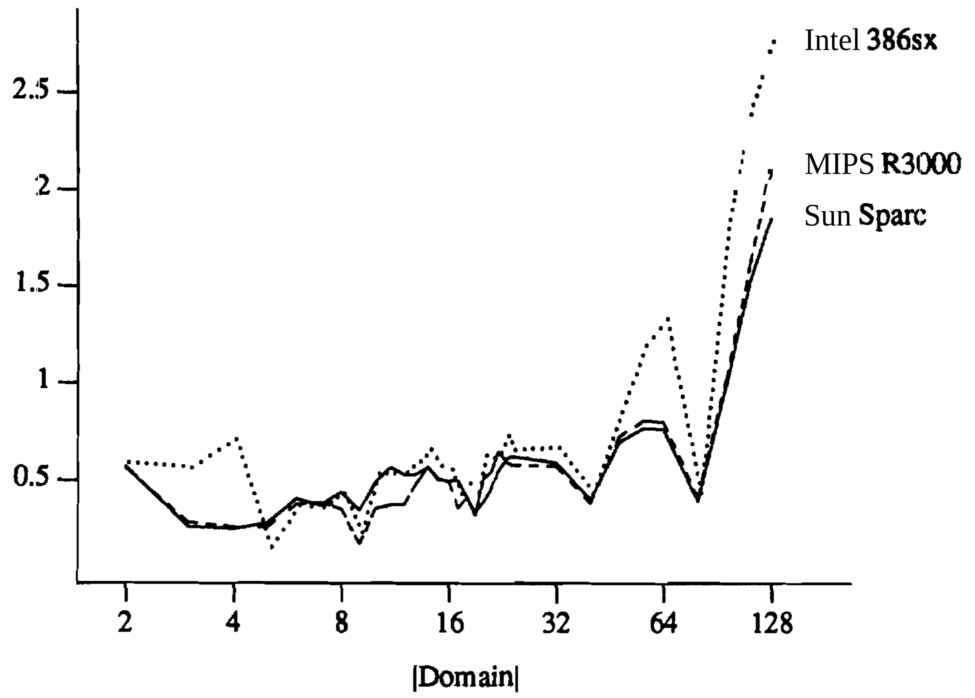

Figure 3: HF Search Times for 386sx, Sparc, and R3000

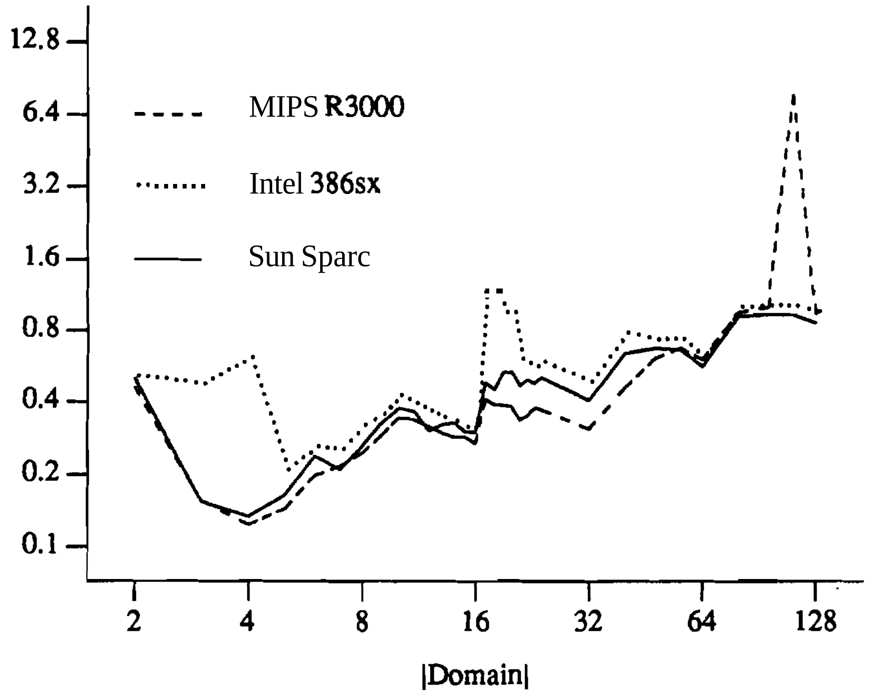

Figure 4: CFHF Search Times for 386sx, Sparc, and R3000

Although the MasPar MP-1 times are not directly useful, there is an additional benefit: by trying so many (142) forms and mappings, we can be relatively certain that all useful forms will be used at least once. Thus, to create a "production" case-statement coder, we need only search the forms that the massive tests on the MasPar MP-1 have shown to be useful for that machine. Over all 7,680 hash functions found for these three target machines, only 13 of the 142 forms tried, were selected as optimal. These are listed in table 4. For the 386sx, 13 forms must be searched. For the R3000, only 7 f o n s are needed; for the Sparc, it is just 6. Most of these forms also have small parameter search spaces.

Total	Times Form Selected						Formula
	Lowest Cost HF			Lowest Cost CFHF			
	386sx	r3000	Sparc	386sx	r3000	Sparc	
6916	1049	1161	1206	1084	1207	1209	`(n>>i)&MASK`
573	193	100	56	112	56	56	`(n)&MASK`
40	12	3	7	8	5	5	`((n>>i)^n)&MASK`
38	10	0	0	28	0	0	`(n&i)%SIZE`
30	3	7	6	2	6	6	`((n>>(n&i)))&MASK`
27	0	0	0	27	0	0	`(n^i)%SIZE`
17	3	1	3	4	3	3	`((n>>i)+n)&MASK`
16	8	4	0	2	2	0	`((n-(n>i)))&MASK`
10	1	4	2	1	1	1	`((-n)>>i)&MASK`
7	1	0	0	6	0	0	`((-n)^i)%SIZE`
3	0	0	0	3	0	0	`((n+i)^n)%SIZE`
2	0	0	0	2	0	0	`((n>>i)*n)&MASK`
1	0	0	0	1	0	0	`((~n)/i)&MASK`

Table 4: Lowest-Cost Forms Selected for Random Mappings

Note that in every case our system found a hash function whose cost was lower than that of an optimal binary search.

42.2. HF/CFHF Search for Mappings with Runs

Preliminary experiments, such as the c a s e statement shown in listing 14 of section 4.1, led to the observation that good hash functions are more difficult to find for mappings in which there are several runs in the domain. To further investigate this notion, the program used to generate random mappings was modified so that the probability of any two adjacent domain values having their values differ by 1 (i.e., be part of the same run) was 1/2. This tends to generate domains with many relatively short runs irregularly spaced, which we suspected would be nearly the worst,-casescenario.

Except for the above difference in how the mappings were selected, the same tests described.in section 4.2.1 were performed. The general performance of the search was very similar to that reported for random mappings. However, there were several very important differences between the f o n s selected as optimal for the random mappings (table 2) and those selected for the mappings with runs (table 5):

Total	Times Form Selected						Formula
	Lowest Cost HF			Lowest Cost CFHF			
	386sx	r3000	Sparc	386sx	r3000	Sparc	
2216	505	717	397	167	193	237	(n)&MASK
1672	57	275	487	70	251	532	((n>>i)^n)&MASK
970	413	0	0	445	112	0	(n^i)%SIZE
906	34	183	290	16	147	236	((n>>i)+n)&MASK
454	0	0	0	154	171	129	*No hash found*
332	121	0	0	146	65	0	((-n)^i)%SIZE
227	7	18	0	51	137	14	((n-(n<i)))&MASK
206	94	0	0	106	6	0	(n&i)%SIZE
179	1	14	64	2	26	72	((n>>(n&i)))&MASK
152	16	40	32	16	24	24	(n>>i)&MASK
149	1	10	0	32	106	0	((n-(n>i)))&MASK
82	8	20	10	16	17	11	((-n)>>i)&MASK
28	10	0	0	14	4	0	(n)%SIZE
23	7	0	0	14	2	0	(i-n)%SIZE
21	1	3	0	1	16	0	((n^(n>i)))&MASK
9	0	0	0	9	0	0	((n*i)^n)%SIZE
8	0	0	0	0	1	7	((n>>i)*n)&MASK
5	0	0	0	0	0	5	((n+(n<i)))&MASK
4	3	0	0	1	0	0	(n%i)&MASK
4	0	0	0	4	0	0	((n*i)*n)%SIZE
4	0	0	0	2	2	0	((n>>i)^n)%SIZE
4	0	0	0	0	0	4	((pop(n|i)^n))&MASK
3	1	0	0	2	0	0	((n+i)^n)%SIZE
3	0	0	0	3	0	0	((n^i)%n)%SIZE
3	0	0	0	3	0	0	((n%i)+n)%SIZE
3	0	0	0	0	0	3	((n^(n<i)))&MASK
2	0	0	0	2	0	0	(-(n^i))%SIZE
2	0	0	0	2	0	0	((~n)%i)%SIZE
2	0	0	0	0	0	2	((pop(n^i)+n))&MASK
2	0	0	0	0	0	2	((pop(n>>i)^n))&MASK
2	0	0	0	0	0	2	((pop(n>>i)-n))&MASK
1	1	0	0	0	0	0	((~n)%i)&MASK
1	0	0	0	1	0	0	((n/i)^n)&MASK
1	0	0	0	1	0	0	((n/i)^n)%SIZE

Table 5: Lowest-Cost Forms Selected Mappings with Runs

- The number of different forms selected was much larger. Instead of 13, there were 33 different forms. The 386sx needed 26 vs. 13, the R3000 needed 16 vs. 7, and the Sparc needed 14 vs. 6; in summary, twice as many fotms as for random mappings.
- There were some mappings for which no CFHF was cheaper than binary search. The fraction of failures ranged from 10% for the Sparc to 13% for the R3000. However disappointing this may be, notice that the system never failed to find an HF that was cheaper. Thus, a production system should allow HFs that are not CFHFs.
- Some of the fotmulas selected used pop — a function to compute the population count (number of 1 bits). This is interesting because pop is not an instruction on any of these processors, but rather a subroutine call. Thus, the cost of pop was much more than it would be for machines that have that instruction — as most

supercomputers do (e.g, all Cray machines have a population count instruction).

This last observation led us to define the generic supercomputer instruction set costs (called Super in section 4.2), including the treatment of pop as an instruction rather than a subroutine. In the three real target machines, the timing for population count was obtained using a subroutine call to the C function given in listing 15.

```
int
pop(register unsigned n)
{
    /* Compute population count by SIMD summation
       of the bits within the 32-bit word n.
    */
    register unsigned mask = 0x55555555;

    n = (n & mask) + ((n >> 1) & mask);
    mask = 0x33333333;
    n = (n & mask) + ((n >> 2) & mask);
    mask = 0x0f0f0f0f;
    n = (n & mask) + ((n >> 4) & mask);
    mask = 0x00ff00ff;
    n = (n & mask) + ((n >> 8) & mask);
    mask = 0x0000ffff;
    return((n & mask) + ((n >> 16) & mask));
}
```

Listing 15: C Function to Compute Population Count

433. A Target Machine with Population Count

The "Super" target machine cost model was designed to reflect relative costs for various instructions in an idealized supercomputer instruction set. Most operations, including population count, are assumed to take a single cycle; multiplicative operations are assumed to take 8 cycles[4].

The exact same test mappings used in sections 4.2.1 and 4.2.2, both the random and multiple run mappings, were submitted to the system to find optimal hash functions for the Super costs. The forms selected for the random mappings are listed in table 6; table 7 lists the forms selected for the mappings with runs.

[4] Although many supercomputers have faster multiplicative operation times, that ability is often restricted to floating point operations while the operations performed here are on integers. The 8-cycle cost represents an approximation over hardware, convert to float, and multiply-step implementations. This cost is not critical to the point being made with the data for the Super target.

	Times Form Selected		
Total	Lowest Cost HF	Lowest Cost CFHF	Formula
727	297	430	((pop(n|i)+n))&MASK
502	294	208	(n)&MASK
333	138	195	(n>>i)&MASK
155	86	69	(pop(n|i))&MASK
150	97	53	(pop(n+i))&MASK
119	31	88	((n+(n>i)))&MASK
100	33	67	(n&i)%SIZE
95	66	29	((pop(n+i)+n))&MASK
77	48	29	(pop(n^i))&MASK
67	33	34	(pop(n))&MASK
65	43	22	((pop(n|i)^n))&MASK
42	27	15	((pop(n^i)+n))&MASK
34	20	14	((pop(n|i)-n))&MASK
28	24	4	((pop(n+i)^n))&MASK
22	15	7	((pop(n+i)-n))&MASK
12	10	2	((pop(n^i)^n))&MASK
11	7	4	(n^pop(n))&MASK
9	6	3	(n+pop(n))&MASK
6	0	6	(n^i)%SIZE
4	4	0	(n-pop(n))&MASK
2	1	1	((pop(n^i)-n))&MASK

Table 6: Lowest-Cost Forms Selected for Random Mappings

	Times Form Selected		
Total	Lowest Cost HF	Lowest Cost CFHF	Formula
680	517	163	`(n)&MASK`
451	172	279	`(n^i)%SIZE`
229	123	106	`((pop(n+i)+n))&MASK`
205	63	142	`((pop(n|i)+n))&MASK`
160	104	56	`(pop(n+i))&MASK`
118	54	64	`(pop(n|i))&MASK`
104	36	68	`((pop(n^i)+n))&MASK`
95	0	95	*No hash found*
78	40	38	`(pop(n^i))&MASK`
74	41	33	`((pop(n+i)^n))&MASK`
69	11	58	`((-n)^i)%SIZE`
64	40	24	`((pop(n|i)^n))&MASK`
60	17	43	`(n&i)%SIZE`
49	26	23	`(pop(n))&MASK`
27	17	10	`((pop(n|i)-n))&MASK`
22	0	22	`((n+(n>i)))&MASK`
14	7	7	`(n)%SIZE`
13	6	7	`(n^pop(n))&MASK`
12	2	10	`(n+i)%SIZE`
10	0	10	`((n+(n<i)))&MASK`
6	1	5	`(n|pop(n))&MASK`
4	2	2	`((pop(n^i)^n))&MASK`
3	0	3	`(n+pop(n))&MASK`
3	0	3	`((n^(n>i)))&MASK`
2	1	1	`(n%i)&MASK`
2	0	2	`((n%i)+n)%SIZE`
2	0	2	`((-n)%i)%SIZE`
1	0	1	`(n-pop(n))&MASK`
1	0	1	`((pop(n|i)^n))%SIZE`
1	0	1	`((pop(n^i)+n))%SIZE`
1	0	1	`((pop(n+i)|n))&MASK`

Table 7: Lowest-Cost Forms Selected Mappings with Runs

The significant observation is that population count was used in 16/21 formulas from table 6 and in 18/30 from table 7. These results are so strong that they may be seen as sufficient justification for adding a population count instruction to a processor's instruction set.

Why is use of `pop` so effective for hash functions? — because `pop` uses the binary representation to efficiently distinguish values in a highly non-linear way. It is useful to recall that the hamming distance between two values can be obtained by exclusive-oring the two values and taking the population count of the result, so the use of `pop` is really incorporating a hamming distance into the hash computation.

5. Conclusions

In this paper, we first discussed the semantics associated with multiway branch statements in C, Pascal, and Fortran-90. This led to a brief review of the traditional codings used, and to the observation that careful generation and use of a hash function to encode the mapping should yield consistently good results.

Serial and massively parallel implementations of a system to automatically generate machine-specific hash functions are detailed in section 3. The performance of the system is studied in section 4. Although restricting the hash to be conflict-free will cause the system to occasionally fail to find a cheap hash function, the results clearly show that an average improvement of about 4x can be expected by letting the system find an appropriate hash function that may have a few conflicts. Multiway branch statements in which the case values have several runs were shown to be more difficult than random case values, but were still handled efficiently by searching only a small number of forms.

Finally, driven by the observation that population count was used a few times in the cheapest formulas selected for machines in which population count was very expensive, we studied the effect of having a relatively cheap population count instruction. As shown in section 4.2.3, if a population count instruction is available, it is used in most of the fonns selected as optimal. Thus, we suggest that population count should be considered as an instruction providing efficient hardware support for computing hash functions.

The: software described in this paper is available as a public domain software package; send email to `hankd@ecn.purdue.edu` for more information. In the future, we hope to integrate these techniques into the public domain Purdue Compiler-Construction Tool Set (PCCTS) [PaD92], so that all compilers generated by PCCTS may incorporate a target machine specific phase that will generate optimized hash codings for multiway branches.

Thanks are given to W. E. Cohen for his help in proofreading and correcting this paper.

Reference;

[ANS89] *American National Standard for Information Systems — Programming Language Fortran,* **X3J3/S8.112,** June 1989.

[ANS90] *American National Standard for Information System — Programming .Language C,* **X3J11/90-013,** Feb. **1990.**

[Ber85] R.T. Bemstein, "Producing Good Code for the Case Statement," Software Practice and Experience, Vol. **15(10),** pp. **1021-1024,** Oct. 1985.

[Bla90] T. Blank, "The **MasPar** MP-1 Architecture," 35th IEEE Computer Society International Conference (COMPCON), February 1990, pp. 20-24.

[BrN90] C.J. **Brownhill** and A. Nicolau, "**Percolation** Scheduling for **Non-VLIW** Machines," University of California **Irvine,** Technical Report 90-02, Jan. 1990.

[Die92] H.G. Dietz, "Meta-State Conversion," Parallel Processing Laboratory, School of Electrical Engineering, **Purdue** University, Indiana, Technical Report in preparation.

[HeM82] J.L. Hennessy and N. Mendelsohn, "Compilation of the Pascal Case Statement," Software Practice and Experience, Vol. 12, pp. 879-882, 1982.

[JeW75] K. Jensen and N. **Wirth,** *Pascal User Manual and Report* (2nd *edition),* Springer-Vedag, 1975.

[KeR78] B.W. Kemighan and D.M. Ritchie, *The* C *Programming Language,* **Prentice-Hall,** New Jersey, 1978.

[Mas91] **MasPar** Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version* 2.2, Document **Number** 9302-0001, Sunnyvale, California, November 1991.

[Mas87] H. **Massalin, "Superoptimizer —** a Look at the Smallest Program," **IEEE** Proceedings of ASPLOS **II,** pp.122-126, Oct. 1987.

[PaD92] T.J. **Parr,** H.G. Dietz, and W.E. Cohen, "PCCTS Reference Manual (version 1.00)," *ACM SIGPLAN Notices,* Feb. 1992, pp. **88-165.**

[Sal81] A. Sale, "The Implementation of Case Statements in Pascal," Software Practice and Experience, Vol. 11, pp. 929-942, 1981.

Table of Contents