Compiler Optimizations
Using Data Compression
To Decrease Address Reference Entropy

H. G. Dietz and T. I. Mattox
Electrical and Computer Engineering Department
University of Kentucky
Lexington, KY 40506-0046
`{hankd, tmattox}@engr.uky.edu`
`http://aggregate.org/`

In modern computers, a single "random" access to main memory often takes as much time as executing hundreds of instructions. Rather than using traditional compiler approaches to enhance locality by interchanging loops, reordering data structures, etc., this paper proposes the radical concept of using aggressive data compression technology to improve hierarchical memory performance by reducing memory address reference entropy.

In some cases, conventional compression technology can be adapted. However, where variable access patterns must be permitted, other compression techniques must be used. For the special case of random access to elements of sparse matrices, data structures and compiler technology already exist. Our approach is much more general, using compressive hash functions to implement random access lookup tables. Techniques that can be used to improve the effectiveness of any compression method in reducing memory access entropy also are discussed.

1. Introduction

Optimization of memory accesses is not a new idea, nor is it new that a compiler should perform the appropriate transformations. However, over the past few years, the natural evolution of computer hardware has yielded a qualitative change in how memory accesses affect processor performance.

1.1. Modern Computer Architecture

Logically, processors are more complex than memory, so one would expect them to be slower than memory. In fact, that was the case for much of the history of digital computing. However, through the relatively short history of digital computing, a surprisingly wide variety of different technologies have been used for constructing main memory and processors. Using different technologies, processors and memories have followed different performance curves... both getting faster, but processors increasing in speed at a much greater rate than memories. The result is what we all know: main memory is now much slower than a processor. But the relationship is much more complex than that suggests.

It is true that processor clock rates have been increasing at an impressive rate, but the processors running at these higher clock rates are not the same designs that were used at lower clock rates. Very little of the performance increase in modern processors comes from using the same design with faster gates. For example, the design of an Intel 468DX processor allowed it to run with the then-fast clock frequency of 33MHz and to complete execution of an instruction every few clock cycles. In contrast, the Pentium 4 uses "superscalar" instruction-level parallel execution to complete execution of several instructions every clock cycle — an order of magnitude more work per clock cycle, even ignoring the fact that the Pentium 4's clock ticks at a blazing 2.4GHz. Beyond that, the reason a Pentium 4 can run with a 2.4GHz clock frequency is not simply because it is built using better gates than a 486DX, but also because it carves long logic paths into many pipeline stages. For example, this is why a Pentium III cannot achieve the same clock rate as a Pentium 4 even when they are built with the same technology: a Pentium 4 has much deeper pipelines yielding shorter logic paths for each clock cycle. In summary, processor speed increases are largely enabled by extensive use of superscalar pipelining — all of which comes to a screeching halt when the processor has to wait for a memory read.

Computer architects are very aware of this problem. The solution is to tune the architecture for the kinds of reference patterns that are common and/or those for which performance easily can be improved. Traditionally, the primary hardware mechanism used is a memory hierarchy in which small, fast memories are placed within or near the processor and intended to be used to hold copies of memory blocks that will be referenced with good spatial and/or temporal locality.

The fastest such memory structure is a register file. Compiler writers have long understood register allocation... but there is a twist: the number of registers accessible to the compiler is a function of the instruction set design, so the compiler can only manage as many general-purpose registers on a Pentium 4 as it had on a 486DX. Fortunately, aggressive use of register renaming has allowed computer architects to build hardware that performs on-the-fly reallocation of registers to a much larger pool. For example, the 8 compiler-visible floating point registers of the Intel 486DX turn into 88 within the AMD Athlon. In many processors, special write buffer hardware even attempts to short-circuit-route data being stored from one register into another register which is loading from the address being stored into.

After registers, there are usually two or more levels of cache. Cache line sizes and replacement policies vary, but in general the line size gets bigger and access gets slower as caches get further from the processor. Across processor generations, cache line sizes tend to be increasing in general. Further, most caches now have special provisions for fetching the requested word within a cache line first, rather than fetching the words in sequence.

Even though your program might not use disk-based virtual memory, modern operating systems rely on a page table mechanism to allocate main memory space. Thus, all main memory addresses have to be translated from logical to physical addresses. In most modern machines, this is done by two levels of TLB (translation lookaside buffers) which serve as "caches for address translations." Caches typically are indexed by physical addresses, so that TLBs appear between the processor and L1

cache. The implication is that even if a particular address is in cache, it will be fast to access only if its address is also in the TLB. Although TLBs are often ignored by programmers, they are often very small (typically 32 to 128 entries), so TLB misses can seriously limit performance.

Further complicating all of this, hardware in the latest AMD Athlon and Intel Pentium 4 processors attempts to automatically recognize access patterns and issue prefetch operations. Thus, the old notions of temporal and spatial locality are only part of the story; performance depends on having a memory access pattern that differs only slightly from what the processor was designed to optimize — i.e., that has low entropy.

1.2. Memory Access Performance Of Modern Architectures

How do all of the above architectural features change how code should be written? The best way to answer such a question is to make some performance measurements on real machines so that the cost of different coding constructs can be accurately estimated. To make the memory access trends more visible, we have restricted our benchmarks to processors that execute the basic IA32 (Intel Architecture, 32-bits) instruction set. This not only eliminates artifacts from use of different instruction sets, but also made it possible to literally use the exact same binary executable on all the machines. Consequently, the memory system effects are not convolved with differences between compilation systems; the one executable was produced using EGS 2.91.66 with the optimizations enabled by the -O1 command line option. An additional benefit in using this instruction set is that all the processors provide the same processor clock cycle timing mechanism.

Most of the architectural features listed above are aimed at improving performance of low-entropy memory access patterns: read sequences that have good spatial and temporal locality or are easily predicted by the hardware. One would hope that repeated references to the exact same word (temporal locality) would be optimized by the compiler to access the word from memory once, and thenceforth from a register. Thus, the lowest entropy memory reference pattern is generally assumed to be a stride-1 access pattern in the increasing address direction. Have these architectural changes achieved speed-up for this read access pattern? As Figure 1 clearly shows, the answer is yes; from the 100MHz Pentium to the most modern Athlon and Pentium 4 an order of magnitude speedup is seen.

It is important to note that, because processors are heavily pipelined, memory access latency can be partly overlapped with loop overhead. It is not possible to separate-out the test loop overhead; any memory access latency that is completely overlapped with loop overhead would appear to be zero and inefficient loop implementations would make memory seem faster. For this reason, all of the graphs in this paper include the loop overhead.

That good speedup is achieved for a low-entropy reference pattern is not surprising. To determine if good speedup is also achieved for high-entropy reference patterns, we selected a simple random number generator — RANQD1 [PrT88] — and used that to generate the address sequence. Ironically, a random number generator does not generate the highest entropy memory access sequence, but is a good model for the
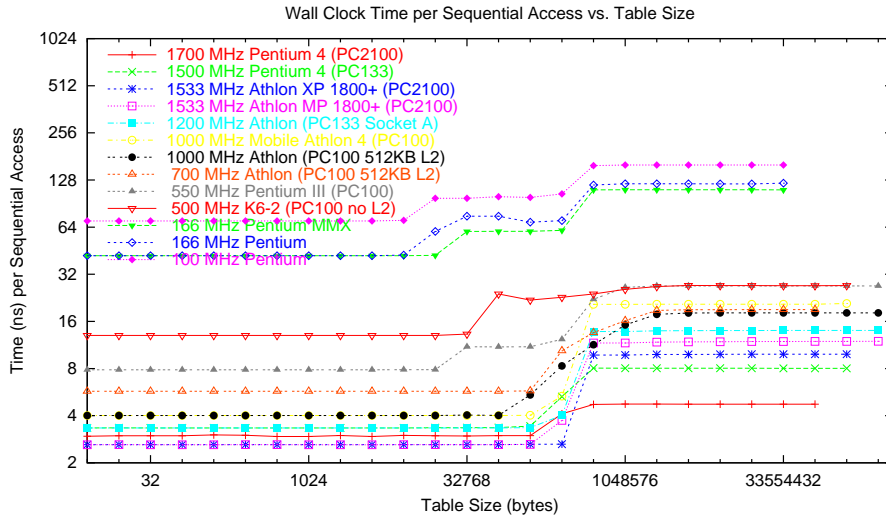
Wall Clock Time per Sequential Access vs. Table Size

1700 MHz Pentium 4 (PC2100)
1500 MHz Pentium 4 (PC133)
1533 MHz Athlon XP 1800+ (PC2100)
1533 MHz Athlon MP 1800+ (PC2100)
1200 MHz Athlon (PC133 Socket A)
1000 MHz Mobile Athlon 4 (PC100)
1000 MHz Athlon (PC100 512KB L2)
700 MHz Athlon (PC100 512KB L2)
550 MHz Pentium III (PC100)
500 MHz K6-2 (PC100 no L2)
166 MHz Pentium MMX
166 MHz Pentium
100 MHz Pentium

Fig. 1. Low-Entropy Memory Read Access Pattern Times

type of high-entropy memory reference pattern commonly seen in programs. The good news is that, as Figure 2 shows, good speedup is also achieved for this high-entropy pattern.
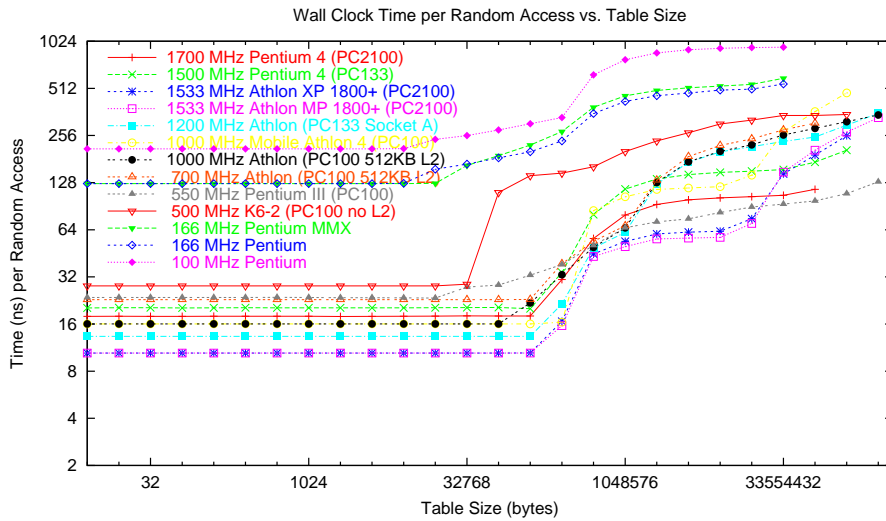
Wall Clock Time per Random Access vs. Table Size

1700 MHz Pentium 4 (PC2100)
1500 MHz Pentium 4 (PC133)
1533 MHz Athlon XP 1800+ (PC2100)
1533 MHz Athlon MP 1800+ (PC2100)
1200 MHz Athlon (PC133 Socket A)
1000 MHz Mobile Athlon 4 (PC100)
1000 MHz Athlon (PC100 512KB L2)
700 MHz Athlon (PC100 512KB L2)
550 MHz Pentium III (PC100)
500 MHz K6-2 (PC100 no L2)
166 MHz Pentium MMX
166 MHz Pentium
100 MHz Pentium

Fig. 2. High-Entropy Memory Read Access Pattern Times

However, viewing Figures 1 and 2 together reveals a disturbing fact: newer processors generally have larger differences between the best sequential time and the worst random time. The 100MHz Pentium had only a time factor of 13.3 penalty for a bad reference pattern, whereas an Athlon MP had a time factor of 127.5 penalty.

Of course, some differences are due to differing clock rates; looking at raw counts of clock cycles is an arguably purer measure. These results, respectively for the sequential access pattern and for the random access pattern, are in Figures 3 and 4.
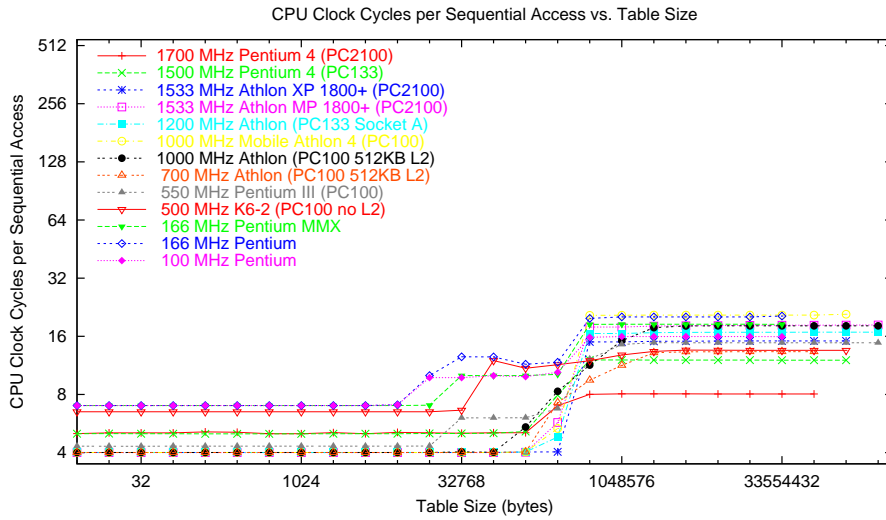
CPU Clock Cycles per Sequential Access vs. Table Size

Fig. 3. Low-Entropy Memory Read Access Pattern Clock Cycles

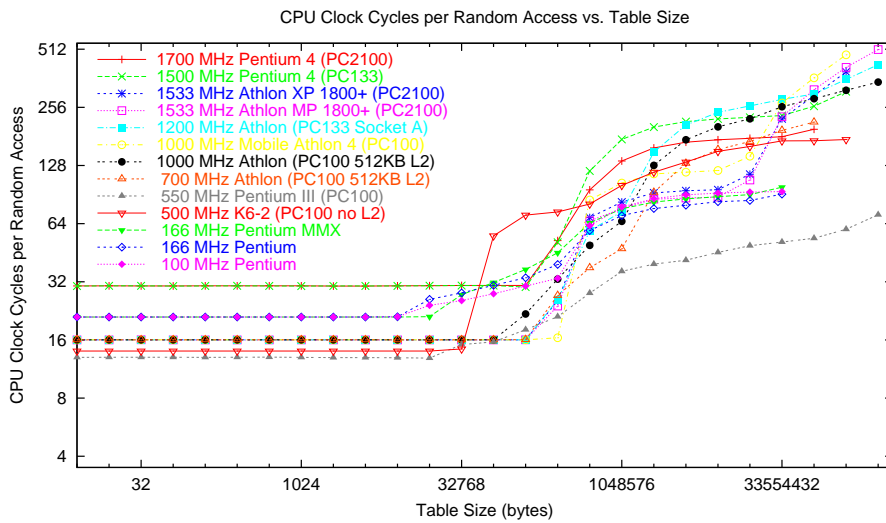CPU Clock Cycles per Random Access vs. Table Size

Fig. 4. High-Entropy Memory Read Access Pattern Clock Cycles

In summary, the cost of memory references is getting further from constant; access times are a complex function of the access pattern with costs currently ranging over at least two orders of magnitude. High entropy memory access patterns can take hundreds of clock cycles per read — and many operations can be executed per clock

cycle. Executing as many as a thousand instructions to avoid a single high-entropy memory reference can yield speedup! This huge payoff makes it practical to consider very complex mechanisms for reducing address reference entropy. Throughout this paper, our focus is using compression to decrease address reference entropy — in some cases, the total size of the compressed data structures is actually larger than the original data.

2. Compression To Reduce Access Entropy

Although we believe the fully general concept of using compiler technology to employ compression for the purpose of reducing address reference entropy to be entirely new, there are a few special cases in which compression has been used to improve memory system performance.

Although our focus is using compiler technology to apply compression to reduce entropy of data references, the work most similar in concept involves hardware technology to operate on compressed code. Shortly after the invention of VLIW (Very Long Instruction Word) architecture, it was recognized that VLIW instructions often contained redundant or empty fields. Although the fact was not widely published, the Multiflow Trace architecture took advantage of this fact by having processor hardware fetch compressed blocks of VLIW instructions and decompress them on the fly. An even more aggressive compression scheme was used for encoding instructions for the complex instruction set of the Intel 432 [ARM81]: instructions were Huffman encoded as bit sequences that were extracted directly from the code stream by the processor hardware. Although modern processor architecture implementations could benefit from such a hardware-driven approach, the benefit is not as great as one might expect because code stream address reference entropy is relatively low — spatial locality is very good.

Very recent work [ZhG02] attempts to achieve modest compression for dynamically-allocated data structures, but the majority of compiler techniques have been developed to translate code written as "dense" matrix operations to use "sparse" data structures [BiW95]. The sparse representations assume that the majority of data elements have the same value (most often, zero). Despite this constraint, these compiler code and data transformations, and the associated analyses, are very closely related to our more general notion of using compression as a memory address entropy-reducing transformation. In particular, the analysis that determines what code would be impacted if the representation of a particular data structure were to be changed is directly applicable. In fact, the analysis we presented in [JuD92] also would suffice for that purpose.

The generalized problem of using compressed data structures with non-sparse data can be subdivided into four classes based on two simple attributes:

1. **Is the data structure read-only?** Compression algorithms for read-only data structures, especially those with compile-time constant values, can be very computationally expensive provided that the decompression algorithm is inexpensive. If the data can be changed during program execution, the efficiency of the compression algorithm is also critical.

2. **Are elements of the data structure accessed in a fixed pattern — i.e., are they ordered?** Given a fixed access pattern, transmitting the data structure from memory to the processor in that order is nearly the same problem as transmitting the data structure through a communications network — the classical application of compression technology. Note that the access order need not access each element precisely once; a structure containing "a,b,c" accessed with the fixed order "c,a,c,c" is essentially the same as sequential access of the structure "c,a,c,c". If a variable access pattern must be supported, compression methods that make decompression of an element dependent on decompression of previous elements are generally inappropriate.

Techniques for fixed access pattern compression are very well developed; thus, the primary contribution here is the concept of using these techniques as a compiler technology. This is discussed in the following section. Given a variable access pattern and read-only data, new compression techniques are needed. Section 2.2 outlines a very aggressive technique for this type of compile-time compression, which is most useful for increasing the efficiency of lookup tables. To efficiently compress given a variable access pattern and changeable data, the compression scheme must have a relatively efficient method for incremental update of the compressed form. Very few such schemes exist; a very brief discussion is given in section 2.3.

2.1. Compression with Ordered Access

Compiler technology for recognizing everything necessary to improve ordered access is very well developed. The required information is essentially accumulated as a side-effect of performing traditional loop parallelization dependence analysis. For example, consider the simple loop nest:

```
      DO 10 J=1,100
        DO 10 I=1,100
10        A(I,J) = A(I,J) * B(I, J)
```

Within this example loop nest, the elements of B are only read; let us further assume that B is in fact an array of constant values known at compile time. The elements of the array A are both read and written. Thus, the example contains both read-only and read-write data structures with a known access order.

For B, because both the element values and the access order are known at compile time, we can apply a traditional communications-oriented compression scheme at compile time. For example, a variant of Huffman encoding, LZW (Lempel-Ziv Welch), or even fractals and wavelets can be used to compress B. Simple type-dependent compression techniques may be particularly appropriate; for example, although mantissa bits vary, it is very likely that the exponent and sign are the same (or differ little) from one floating point value to the next. Further, because the compression is done at compile time, it is feasible to try several alternative compression techniques and pick the most effective.

The compression of **A** is much more difficult to make effective. In part, the complexity comes from the fact that the compression algorithm must be incrementally

applied (e.g., wavelets cannot be used because they require examining the complete data structure) and must be computationally cheap enough to be applied at every point where the data are changed. However, the fact that compression is applied at run time also makes it infeasible to try several alternatives and pick the most effective. For many incremental compression techniques, it is quite possible that the result of applying compression would be a data structure larger than the original — with the slowdown aggravated by the higher overhead of processing compressed accesses.

2.2. Compression with Variable Access, Read-Only Data

With the exception of some of the sparse compression techniques discussed in section 2, virtually all compression techniques in the literature are incapable of supporting a variable access pattern. However, if the elements of the data structure are read-only and known at compile time, there are a variety of techniques that can be used to compress the lookup table without compromising random access. The basic technology is creation of a **compressive hash function**: a hash function that implements a given mapping using a lookup table that contains fewer entries than there are domain elements.

A **hash function** is a mapping of domain (input or key) values into range (function or return) values. Normally, the ideal is to find a hash function that is minimal and "perfect" — i.e., that implements a domain-to-range mapping by *onto* and *1:1* indexing of a lookup table. However, a perfect hash function only provides rapid indexing: it does not provide compression of the data. In order to provide compression, a compressive hash function maps multiple domain elements, whose range values were equivalent, to the same lookup table entry. A **minimal perfect compressive hash** would have precisely one hash table entry for each unique range value, but any compressive hash function will provide some compression while supporting fully variable random access patterns.

Let $L(k)=v_k$ be the original table lookup function implemented by indexing an array of values, $a[]$, such that $v_k=a[k]$. If there exist two values of k, k_i and k_j such that $k_i \neq k_j$ and $L(k_i) \equiv L(k_j)$, then $a[k_i]$ and $a[k_j]$ are essentially copies of the same range value and one might be able to be eliminated from storage as redundant. It is common that lookup functions have many such redundant entries; further, there are techniques that can be used to transform the lookup problem to create such redundancies (see sections 3 and 4). The problem of finding a compressive hash function, $L'(k)=v_k$, is thus the problem of finding an index transformation function, $H(k)=x_k$, which maps into a table with fewer entries than $a[]$, such that for all pairs of values k_i and k_j, if $H(k_i) \equiv H(k_j)$, then $L(k_i) \equiv L(k_j)$. Notice that $L(k_i) \equiv L(k_j)$ does not imply $H(k_i) \equiv H(k_j)$; duplicate entries can also exist in the compressed form, provided that the total array size is still reduced. Similarly, having the compressed array contain entries that are not targeted by any value of k also merely reduces the compression factor achieved. Of course, optimizing the compression factor is not our goal; minimizing average access cost by taking advantage of lower memory access entropy is.

There are many approaches that can be used to search for a good hash function $H(k)$ and the array contents that it requires in order to perform the correct mapping.

Fundamentally, the problem of reverse-engineering an efficient hash function from the array contents becomes exponentially more difficult as larger domains and ranges are considered. Achieving higher compression generally has the same impact on complexity of the search, or, equivalently, generates hash functions that are computationally too complex to be useful. Our approach can be summarized as:

(1) Compute the minimum possible size of the hash table, s, by counting redundant entries in $L(k)$. If modulus operations are expensive, round s up to the next largest power of two. Also initialize a hash table, $e[s]$ to all "empty" entries.

(2) Generate a potential hash function, $H(k)$, which ensures that, for all values of k, $0 \leq H(k) < s$. If s is a power of two, this can be accomplished using bitwise AND $(s\text{-}1)$ in $H(k)$.

(3) Evaluate $H(k)$ for all values of k. In essence, this is done by evaluating $H(k_i)=x_i$ and then examining $e[x_i]$ for either of two conditions:

- If $e[x_i]$ is empty, set $e[x_i]=L(k_i)$ and mark the entry as full.
  (Serial numbering is often a good way to handle empty/full marking.)

- If $e[x_i]$ is full and $e[x_i] \neq L(k_i)$, record the conflict.
  If the hash function must be perfect, goto step 5;
  otherwise, continue with lossy compression (sections 5 and 6).

(4) Combine evaluations of conflicts and the computational cost for $H(k)$;
  record it as the new "best found so far" if appropriate.

(5) Increase s if the array size seems too small to afford a computationally efficient hash function.

(6) Exit if available search time has elapsed, sufficiently good solution has been found, or s has become too large.
  Otherwise, go to step 2.

Notice that it was not specified how one generates the potential hash function in step 2. There are many viable alternatives. Techniques we have used include:

- Searches of fixed collections of known-effective forms

- Enumerative searches (as per the Superoptimizer [Mas87])

- Genetic programming (GP) [Koz92]

- Adaptive methods that attempt to correct specific conflict(s) from previous hash functions

- Various curve-fitting techniques

Of these, the fixed-collection and GP methods have thus far proven to be most effective. However, further research is needed to find more efficient ways to handle very hard hash compression problems. Currently, overnight or longer runs are often needed to find appropriate hash functions.

2.3. Compression with Variable Access, Changeable Data

As discussed above, it is very difficult to find an appropriate compressing hash function for an arbitrary mapping... and the creation process is not incremental.

Except when the rate of change of entries is low enough to permit use of a fixed hash compression augmented by a conventional hash table with linear rehash used to identify changed entries, we currently know of no effective approach.

3. Accuracy and Range Precision Filtering

Although programmers often take the position that every value computed within their program should be computed with as much precision as possible, what really matters is the accuracy of the results. Precision simply indicates how many bits are used to represent a value; accuracy describes how many of the bits carry correct and useful information. Because various savings are possible in operating on lower precision values, it is generally desirable to make the storage precision of values equal to or slightly greater than the accuracy of those values. The only benefit in maintaining precision much higher than accuracy is that it saves the programmer from having to be aware of what the accuracy of their computations truly is — in other words, it facilitates bad programming practice.

Although integer values are absolutely accurate, the precision required for integer values is determined by the range of values. For example, an integer variable that ranges from 0 to 100 does not require storage with 32-bit precision; 7 bits would suffice. A value that ranges from 10000 to 10100 also can be stored in just 7 bits. In fact, a value that ranges from 10000 to 10200 and is always a multiple of 2 also can be stored in just 7 bits. Range compression also can be applied to floating point values that have a very limited range of exponent values.

Thus, when compression techniques are being applied, the compression techniques should not be constrained to produce values that are identical to the full precision, but only to preserve the accuracy and range of the original values.

For example, consider a typical lookup table. Each entry is usually either the result of a very complex computation or an empirically measured quantity — after all, if entries were determined by a cheap formula, few programmers would bother constructing a lookup table. However, even if complex computations were carried out using very high precision, the accuracy of the results placed in the table is likely to be far lower than the precision of the intermediate calculations used to compute them. Low accuracy also is common for empirical data. Thus, even if subsequent calculations using values from the lookup table require high precision arithmetic, storage of the table entries need not. More generally, a lossy compression scheme that recovers the table entries only approximately is acceptable provided that accuracy is not compromised. Alternatively, accuracy information can be used to filter the table before compression, reducing entropy by changing values to conform with other values in the table when the change does not compromise accuracy.

It is useful to further note that, if the accuracy and range values vary widely over portions of a lookup table, it may be appropriate to subdivide the table on this basis.

Although static accuracy analysis is not particularly difficult for a compiler to implement, an informal survey conducted by Dietz in the early 1990s of scientific Fortran codes then in use at Purdue University revealed that few, if any, results printed by these programs had any significant digits as determined by the standard static

analysis. Despite this, the codes seem to produce reasonably accurate answers, apparently with several significant digits. The discrepancy lies in the fact that compensating errors are common and worst-case loss of accuracy is very rare, so static analysis was far too conservative. For this reason, we suggest that the programmer should use a **pragma** to explicitly state the accuracy that should be preserved.

4. Synthetic Range Filtering

In some cases, accuracy and range precision filtering are not very helpful. For example, a table of floating point numbers often will have relatively random bit patterns in the mantissas. It may be exceedingly difficult to compress such data. However, an interesting trick can be used to simplify the search.

Let $L(k)=v_k$ be the original table lookup function. If the return value has b bits, then v_k is really the bit vector $v_k[0..b-1]$. Instead of searching for a single compressed lookup function, $L'(k)=v_k$, we can search for a set of compressed lookup functions $L'_0=v_k[0..b_0-1]$, $L'_1=v_k[b_0..b_1-1]$, ..., $L'_m=v_k[b_{m-1}..b-1]$. This effectively synthetically restricts the range for each compressed lookup function, significantly reducing the apparent entropy of the values and consequently making appropriate functions easier to create. Because the bit vectors can be stored as packed fields within a table, there is little or no additional storage overhead associated with the decomposition into bit vectors.

If the compression achieved for the decomposed bit vectors is comparable to the compression achieved without decomposition, having m lookup table references instead of 1 will introduce enough overhead to make decomposition inappropriate. However, the reduced ranges often yield significantly higher compression for some of the m compressed lookup functions. Thus, decomposition into m lookup tables may significantly reduce the total space needed for lookup tables. If this reduction allows the tables to reside in a higher level of memory (e.g., L2 cache rather than main memory), computing m decomposed lookup functions can be significantly faster than performing a single compressed lookup.

Another way to synthetically reduce the range is to convert bit positions that are constant across all lookup values into "don't care" bit positions. The bit positions that are constant ("stuck" at 0 or 1) can be obtained straightforwardly. Let O be the bitwise OR of all the values and A be the bitwise AND of all the values. The *active* bit positions are then those in O AND NOTA. The inactive bit positions can thus be treated as "don't care" values within the lookup function(s) and the correct bit position values can be inserted by bitwise ANDing with the *active* set (computed above) followed by bitwise ORing with A.

5. Individual Exceptions

Suppose that a particular table lookup operation, $L(k)=v_k$, is equivalent to a cheaper lookup operation $L'(k)=v_k$ for all $k \neq x$. The single exception can be corrected by code like:

```
if (k≡x) {
    return(v_x);
} else {
    return(L'(k));
}
```

This correction method can be generalized to correct multiple flaws in *L'* by coding either a binary tree or a linear nest of `if` tests.

Unfortunately, as discussed in the introduction, modern processors are heavily pipelined; thus, performance depends critically on the processor correctly guessing whether to take or not to take each conditional branch. One implication is that the binary tree can be slower than the linear nest because the branch directions are less predictable. In any case, branches often will be mispredicted. We can avoid branch misprediction by converting each `if` statement into a masking operation like the following C code:

```
t = k^x;
m = ((t | -t) >> (WORDBITS-1));
return((m & (v_x ^ L'(k))) ^ v_x);
```

In this code, assume that K, x, `t`, and `m` are 2's complement signed integers. The value of `t` will be non-zero *iff* $k \neq x$. For any non-zero value of `t`, the expression `(t | -t)` will yield a negative integer value. A signed shift right of a negative value by the number of bits in a word minus one essentially replicates the sign bit, making `m` have the value -1. The same process gives `m` 0 if `t` is 0. Thus, `m` can be used as a bitmask to conditionally enable part of the computation. The returned result is v_x if `m` is 0 (i.e., $k \equiv x$). Otherwise, because $v_x \hat{\ } v_x$ is 0, the result is just *L'(k)*. We can further optimize this code to:

```
t = k^x;
m = ((t | -t) >> (WORDBITS-1));
return((m & L''(k)) ^ v_x);
```

By replacing the table entries of *L'(k)* with *L''(k)=(L'(k) ^ v_x);*, we can avoid the overhead of one of the exclusive-OR operations.

6. "Lossy" Compression

A "lossy" compression scheme is one in which the values recovered from the compressed form are not identical to the original, but have similar properties. In many cases, a lossy compression scheme can yield significantly higher compression than a lossless scheme. For example, JPEG image encoding achieves high compression using a lossy scheme, but the compression technique is carefully engineered so that the lost information is usually visually unimportant. Thus, the question is: how can a lossy compression scheme be engineered to provide similar benefits for reducing memory access entropy?

6.1. The Basic Approach

The surprising answer is that a compression scheme that only yields a correct value for *some* inputs can dramatically decrease access entropy. Suppose that a particular table lookup operation, $L(k)=v_k$, is approximated by a lossy compressed lookup operation $L'(k)=v'_k$. It is possible to construct $L'(k)$ such that, for some values of k, $v \equiv v'$; i.e., the lossy scheme returns the correct value. Let p be the probability that k is selected such that $L'(k)$ is correct. By using $L'(k)$ rather than $L(k)$ for those values of k that yield correct results, we can reduce the memory access entropy by an amount proportional to p.

The only remaining problem is how to select when to use $L'(k)$ and when to use $L(k)$. This can be solved by creating an auxiliary correctness-check function, $C(k)$ that returns *true* only for values of k for which $L'(k)$ yields the correct answer. An implementation of $C(k)$ can be created trivially by using a lookup table with a single bit for each possible value of k. However, lossy compression of $C(k)$ also can be applied to create a lookup function $C'(k)$. The only constraint is that for all k such that $C'(k) \equiv true$, $C(k) \equiv true$. If there exists at least one value of k such that $C'(k) \equiv false$ and $C(k) \equiv true$, then the effect is that the probability of using $L'(k)$ is reduced by the sum of the probabilities of those values of k incorrectly classified by $C'(k)$.

One further optimization is possible. Since $L(k)$ will not be evaluated for values where the correctness-check function returns *true*, it is possible to create a residual lookup function, $R(k)$, such that $R(k) \equiv L(k)$ for all k where the correctness-check function returns *false*. There are several different ways to produce $R(k)$.

An obvious approach is to treat $R(k)$ as a new $L(k)$, and to recursively apply the search for a possibly lossy, but cheaper, lookup function $L'(k)$. It should be noted, however, that the recursive application is slightly more complex because $R(k)$ is only defined for certain values of k, not for all values between a minimum and maximum. This complication is easily accounted for in the search.

Alternatively, a valid $R(k)$ always can be produced by using an arbitrary (imperfect) hash function with linear rehashing. Each hash bucket in R would contain an input/output value pair; if the input does not match, the sequentially next hash bucket is examined, and so on, until the the value is found. The sequential re-hash is very friendly to both caches and TLBs, so even performing several probes can take only a small fraction of the time required for a random lookup using $L(k)$. Of course, this last optimization applies only when p is sufficiently large; for small values of p, directly using $L(k)$ is faster because the lookup table for $L(k)$ is comparably sized or smaller than the one for $R(k)$ — the table for $L(k)$ does not need to hold values of k.

6.2. A Simple Example

For example, one test case that we have examined is a lookup table taken from a weather prediction code. This table can be viewed as a lookup function $L(k)$, $0 \leq k < 742,600$, which returns a 32-bit floating point value.

It happens that many of the entries are 0, so the table is somewhat sparse — although not sparse enough for the usual sparse data structure methods to be directly useful. It is trivially easy to recognize that a very good choice for a lossy compressed $L'(k)$ is

literally the function *L'(k)=0*. There are 297,613 entries computed incorrectly by *L'(k)*=0 (40%). If all values of *k* are equiprobable, *p*=0.6.

The obvious implementation of *C(k)* is a lookup table containing 742,600 bits — a mere 92,825 bytes compared to 2,970,400 in the original data structure. This is small enough that both *L'(k)* and *C(k)* fit within the L2 cache of most modern processors. However, it is possible to achieve a still smaller cache footprint by lossy compression of *C(k)*. In this case, one of our hash search codes was able to create a 32,768-byte table that can be used to implement *C'(k)* such that *C'(k)* is overly conservative in estimating *C(k)* for less than 0.01% of the values of *K*, essentially leaving *p* unaffected. However, the hash function for *C'(k)* is a degree-3 polynomial requiring three multiplies and two adds to be evaluated to index the appropriate byte, which would take significantly longer than the L2-cache access for *C(k)* — so use of a compressed *C'(k)* is not worthwhile in this case. If 32,768 bytes fit in L2 cache and 92,825 bytes did not, use of *C'(k)* may have been justified. In general, the choice is made by plugging-in the cost metrics for the particular target machine's memory access structure; further, it is not necessary to search hash function forms that exceed the cost that the target machine would have for *C(k)*.

Continuing our example, is it appropriate to replace *L(k)* with *R(k)*? As discussed above, *C(k)* finds that there are 297,613 values of *k* that are incorrectly evaluated by *L'(k)*. For simplicity, assume that the recursive approach is ignored and we instead accept an imperfect hash function with linear rehash. For virtually any data, it is easy to find such a hash function that has an average of less than 1 linear rehash per lookup. However, the imperfect hash function must not only store the 297,613 result values, but also the value of *k* that each result is produced by. Because there are 742,600 possible values of *k*, storing each *k* value would require a minimum of 23 bits. For alignment reasons, one would certainly round that up to at least 24 bits, and perhaps to 32 bits per *k* value. At 32+32 bits per table entry, the table for *R(k)* is 2,380,904 bytes — whereas the original table for *L(k)* was 2,970,400 bytes. This constitutes a savings of just under 20%, which is probably not sufficient to justify using *R(k)*, because *R(k)* will be slower for the values of *k* that require linear rehashes. Of course, if this size difference would allow *R(k)* to fit in cache where *L(k)* does not, it would be worthwhile; our example just happens to be too large to fit *R(k)* in L2 cache on most modern processors.

On a 1GHz Athlon 4 laptop, the use of *L'(k)*, *C(k)*, and *L(k)* as described above gave a speedup of 1.4x to 2.1x over use of *L(k)* alone. The variability reflects changes in the reference pattern; clearly, for some reference patterns, the use of compression would yield slowdown due to the extra overhead of evaluating *C(k)*. It is likely that a better compressive hash would yield significantly greater average speedup, but the interesting fact is that our existing software was able to create the above compression scheme quickly enough so that integration of the technique in a compiler could yield acceptably short compilation times.

Generally, read-only data structures that are accessed in fixed patterns are even easier to compress. If the same 742,600-entry table used for the above example is accessed in a fixed order, the lossless compression scheme used in `gzip` reduces the binary data structure to less than 14% of its original size. Optimal use of compression to reduce memory access entropy requires much more research, but obvious cases are worth compressing now.

7. Conclusion

In this paper, we have outlined a family of new methods for achieving higher performance from the complex memory access mechanisms used in superscalar pipelined processors. Speedup is obtained by using very aggressive compression technology, especially lossy compressive hash functions, to reduce the entropy of memory access patterns. Decreasing entropy of references is all that matters; adding lossy compressed data structures can simultaneously increase total memory footprint and decrease entropy.

Accepting that some compression problems are unsolvable or would take to long to solve, the new techniques easily could be integrated into a compiler using existing compiler analysis combined with directives or pragmas to help identify appropriate data structures. Our ongoing research centers on more efficient methods for creating compressive hash functions.

This paper represents neither a completed study nor a final answer as to how compression should be used. Rather, it was written because we had long been applying some of these techniques in obscure special cases, but only recently discovered that they have been rendered important and common by modern processor architecture. The evolution of memory systems will no doubt necessitate far more research into exotic methods for improving access pattern entropy.

References

[ARM81]   *iAPX 432 General Data Processor Architecture Reference Manual*, Intel, January 1981, Appendix A.2, pp. A-13 - A-22.

[BiW95]   A. J. C. Bik, H. A. G. Wijshoff, "Advanced Compiler Optimizations for Sparse Computations," *Journal of Parallel and Distributed Computing*, Vol. 31, No. 1, 1995, pp. 14-24.

[JuD92]   Y-J. Ju and H. G. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," *Languages and Compilers for Parallel Computing*, edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Springer-Verlag, New York, New York, 1992, pp. 344-358.

[Koz92]   J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.*

[Mas87]   H. Massalin, "Superoptimizer — a look at the smallest program," ASPLOS II, 1987, pp. 122-126.

[PrT88]   W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 2nd edition, 1988, p. 284.

[ZhG02]   Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction,* April 2002, pp. 14-28.