Survey of General Area

R. Clint Whaley *

September 30, 2014

Abstract

This paper is a survey of memory hierarchy optimizations and bit-level optimization, and is submitted in partial fulfillment of the doctoral candidacy requirements.
9 Detecting and analyzing unused bits 28
  9.1 Static range analysis ......................................................... 28
  9.2 Static section analysis ....................................................... 29
  9.3 Static range and section analysis ......................................... 29

10 Exploiting unused bits 30
  10.1 Computational optimization ............................................... 30
  10.2 Memory optimization ....................................................... 30
    10.2.1 Compressing integers and pointers ................................... 31
    10.2.2 Structure transformation to support compression ................. 31
    10.2.3 Instruction set support ............................................... 32
    10.2.4 Results ........................................................................ 34
1 Introduction

This paper is a survey of the area of proposed research, in partial fulfillment of the doctoral candidacy requirements. As such, it is not concerned with explaining the proposed work in detail (such will be done in the prospectus), but some understanding of the general ideas we hope to investigate is required in order to identify the general area of the research.

We are interested in using profile information in order to reduce memory usage, where we use memory usage in a very broad sense. For instance, we are interested not only in reducing the total memory footprint of a program, but also in optimizing the the number of memory fetches along critical paths, and in effectively exploiting the memory hierarchy. We will therefore explore opportunities for both time and space optimizations.

At this stage, the details of the work are far from fully formulated. However, we are planning to investigate using profiling to enable bitwise compression of operands, and to combine this with memory hierarchy optimizations. At least initially, we will concentrate on dynamically-allocated data structures. We will therefore divide this survey into two gross classifications, those concerned with effectively exploiting the memory hierarchy, and those concerned with bit-level optimizations.

Part I

Exploiting the memory hierarchy

2 Memory hierarchy introduction

There are many techniques already in the literature on effectively exploiting the memory hierarchy. We hope to apply some of these techniques synergistically with our own work. For instance, we may be able to combine compression to enhance some memory hierarchy optimization, so that the effect of both optimizations together is greater than the cumulative benefit provided when they are applied individually. Even when our primary objective is reducing the total memory footprint, it may still make sense to employ related memory hierarchy optimizations in order to mitigate the costs of compression/decompression.

We will first overview what memory hierarchy is, and some basic terminology used to describe it, in order to get a common vocabulary for discussing the optimizations of interest. Then, due to its importance in achieving good performance for many operations, we will discuss blocking for both arrays and structures. We will illustrate this key optimization with an in-depth analysis on matrix-matrix multiply. We will then discuss several other array and structure specific optimizations in turn.

2.1 Notes on examples

We will use examples, including code snippets, to clarify how these optimizations are used and applied. For array-based optimizations, we will use Fortran77 as our example language. Historically, Fortran is the language of scientists, engineers, and mathematicians, and is overwhelmingly array oriented. Even today, a great deal of this kind of work is done in one of several variants of Fortran. Therefore, our arrays will be stored in column-major storage, and each column will be constantly strided by lda (the leading dimension of the array). For examples involving structures, our example language will be ANSI C.
3 Understanding the memory hierarchy

This section briefly overviews the basics of memory hierarchies, providing terminology and definitions that will be used in subsequent sections. The descriptions are relatively brief, and intended to provide insight into the problems we address later in the paper. For more rigorous and complete discussion, see [?].

Most modern machines possess a *memory hierarchy*. In a memory hierarchy, memory is segmented into various levels; levels that are closer to the processor are typically smaller and faster, and thus more expensive per unit storage, than the levels further away from the chip. Figure 1 gives a graphical representation of a typical memory hierarchy. On the left side of the figure, an approximate cost of a reference is given (in units of register access, registers being the fastest storage), while typical sizes of these levels are displayed on the right.

![Figure 1: Memory hierarchy](image)

Because of the wide range of memory hierarchies available, every aspect of this graphic can be shown to be incorrect for a particular machine (e.g., the IBM POWER2 chip has a 128K L1, the Itanium possesses three levels of cache rather than two, some embedded systems have no caches at all, etc). However, the figure is a good general indicator of the speed and capacity of different levels of the memory hierarchy, and can thus provide a useful high-level understanding of the importance of using the memory hierarchy. For instance, understanding from the figure that a disk access might be ten million times slower than a register access makes it clear how important it is to access the near levels of the hierarchy, rather than the far.

Memory hierarchies are used for both instructions (e.g., code), and for data. Some systems possess separate instruction and data caches, and some systems combine the two. Level 1 caches are often separate, while higher levels are often used for both instructions and data. In this survey,
we concentrate on memory hierarchy for data.

The levels of the memory hierarchy between the registers and the main memory are referred to as caches. Caches are usually inclusive, which means that a cache at a particular level of the hierarchy contains all of the data contained in all caches between it’s level and the registers (the contents of registers are not replicated in the caches). Some caches, however, are exclusive, such that their contents do not intersect. In this paper we will assume the caches are inclusive.

The data in registers is not replicated anywhere else in the memory hierarchy, and movement into and out of the registers is done explicitly by software (usually by a compiler, but also sometimes explicitly by a programmer). What memory is stored in caches, on the other hand, is usually controlled by hardware. The basic idea of a cache is that the first reference to a location in memory will involve a very costly load from main memory (or even, perhaps, from disk), but then a later reuse of that data will hit the cache, and thus have a correspondingly faster access time.

More specifically, caches make use of two properties of data access. Temporal locality is the property that recently accessed data will likely be used again, and is thus worth retaining in the cache for subsequent use. The second such property is spatial locality. Spatial locality says that data located close in memory to the present reference is also likely to be used soon. The cache line concept, explained below, is one of the main ways caches exploit spatial locality.

3.1 Cache basics

There are five basic properties which, taken together, provide a solid overview of how a cache operates. The first is whether the cache is inclusive or exclusive, as previously discussed. The second is Cache line (also known as cache block) size, followed by the associativity, replacement policy, and finally, the write policy of the cache. We discuss each of these new categories below, with a special emphasis on cache lines and associativity, which will be used extensively later.

3.1.1 Cache lines

When a data item is not found in a given cache, it must be loaded from a more distant level in hierarchy. When this load takes place, the cache may load more data than the memory reference requires. In particular, caches always load a cache line (also known as a cache block). For instance, a user might load a 32 bit number, but the cache line size might be 128 bits, so the next three elements are preloaded into the cache. If spatial locality holds true, this means the next several accesses will be very cheap. If the user does not use these extra elements before the line must be replaced (see Section 3.1.3 for information on line replacement), these loads will be wasted.

Note that the cache line size is usually heavily influenced by the bus width connecting the cache with the next further level of memory. As would be expected, then, different memory hierarchy levels in the same machine often possess different block sizes.

Cache lines simplify the logic required to keep the cache coherent with other layers of memory, by reducing the bookkeeping costs. If a byte was the smallest unit of access the ISA allowed, a 64KB cache would need a 64K-entry lookup table describing which memory location each byte in the cache mapped. Since many programs display spatial locality, cache lines typically provide performance benefits as well.

One important property of using cache lines is that they are usually filled in order, from lowest memory location to highest. Many architectures do not require the full cache line to be loaded before the processor is allowed to continue. This means that a stand-alone reference that maps to the beginning of a cache line may take less time than one that maps to the end of a cache line.
3.1.2 Associativity

The associativity, or more completely, set associativity of the cache is the key to understanding how memory addresses are mapped into the restricted size of the cache. For purposes of this discussion, define:

- \( N \) = set associativity
- \( N_L \) = number of lines in the cache
- \( A \) = address of the start of the cache line in memory

The easiest mapping to understand is probably the direct mapped cache, which is the same thing as a 1-way set associative cache. In a direct mapped cache, a given cache line size chunk of memory may be mapped to only one location in the cache. This location is usually computed by a simple function such as \( A \mod N_L \). This easy mapping is both the strength and the weakness of direct mapped caches. On the positive side, a search to see if the desired location is in the cache must only check one location. This easy mapping means that the hardware required to do the lookup will be comparatively straightforward and cheap to produce. On the negative side, such a rigid mapping means that cache line conflicts will be relatively common.

A cache line conflict occurs when two memory locations map to the same cache line, even though they are more than a cache line apart. In a direct mapped cache, the load of one line into the cache knocks the other one out of the cache.

The way to minimize cache line conflicts in a given size of cache is to make it fully associative. In a fully associative cache, a memory line may be placed anywhere in the cache. Cache line conflicts are minimized, but the entire cache must be searched to establish if a memory address is resident in the cache.

The more general classification, which subsumes the previously described mapping strategies, is the \( N \)-way associative cache. For a direct mapped cache, \( N = 1 \), and for a fully associative cache, \( N = N_L \). In a \( N \)-way set associative cache, a given line may be mapped to a set of locations in the cache. For instance, in a 4-way associative cache, a line may be mapped to 4 different locations, and thus you will need four lines to map to the same place in order to force a conflict. The number of sets in a given cache (call this \( N_s \)) can be determined by \( N_s = N_L/N \). Thus, to find if a line is resident in the cache, an equation such as \( i = A \mod N_s \), yields the set to search, and then all \( N \) of the possible lines are checked for a match.

3.1.3 Replacement policy

Since caches can contain only a subsection of memory, at some point cache lines will need to be moved further away in the hierarchy to make room for new data. The replacement policy of the cache determines what data will be thrown away in order to make more room.

If the cache is direct mapped, there is no choice to be made: the line at the appropriate address must be flushed, and the new line will take its place. In an \( N \)-way associative cache, there will be \( N \) candidates for flushing, and thus a policy is required to decide which line will be flushed.

Because of temporal locality, it is usually considered beneficial to flush the least recently used (LRU) cache line. However, since LRU requires significant overhead to implement in caches with high degrees of associativity, such caches often use first in first out (FIFO) as an approximation to LRU, or they use a random or pseudo-random replacement algorithm. The random replacement schemes require the least bookkeeping, and are therefore fairly commonly used.
3.1.4 Write policy

The discussion so far has exclusively dealt with memory reads. Memory writes are harder to deal with in caches. Writes require extra steps in order to maintain coherency between levels of the memory hierarchy. Since memory is not altered by a read, the fact that a data block is replicated in multiple levels of the memory hierarchy poses no difficulties for read access.

For writes, however, there must be a mechanism in place to ensure that when a memory location is written, all levels of the hierarchy are updated. There are a multiplicity of techniques that are used to handle this problem in detail, but in general, much of the high level view can be understood simply by knowing the write policy of the cache.

The two basic classes of write policy are:

- **Write through**: A write immediately updates not only this level’s cache line, but also the cache line in the hierarchy one level further away than this one.

- **Write back**: The write immediately updates this cache level’s cache line, but is written to a further level only later, as needed.

While write through caches are relatively simple, there are a large variety of strategies used to optimize write back. This gross classification of caches should be adequate for these discussions, however, so we will leave the classification at this level.

3.2 TLB and paging

Most modern architectures support virtual memory. Thus, an address may be on disk, rather than in memory. Just as caches have cache lines, virtual memory has pages. A logical address must be translated into the appropriate page index and offset. The TLB (Translation Lookaside Buffer) is a special cache dedicated to this purpose.

4 Blocking

**Blocking** (also known as tiling) is a memory hierarchy optimization that can yield tremendous speedups for algorithms which can employ it. Blocking is a simple idea: when a data set is too large to fit into a given level of cache, and there is data reuse in the algorithm, the data set is subdivided into blocks of data which will fit into a given cache level, and then these cache-resident subsections are reused across multiple computations.

To put it a bit more formally, the following must be true in order for blocking to be successfully employed:

1. The data set used by the operation being blocked must exceed the cache size we are blocking for (otherwise, it will be cache-contained without blocking).

2. There must be an opportunity for data reuse (i.e. the number of accesses on a given data structure must be greater than the number of elements in the data structure).

3. There must be enough flexibility in computation order to allow the reorderings required to perform blocking.

Blocking has proven extremely effective for optimizing array-based codes, particularly linear algebra [?, ?, ?]. Blocking usually requires greater knowledge about the operation being performed than most compilers can extract from unelucidated source code, and the transformations required
to perform it are often extreme, and so blocking is often done by the programmer, rather than the compiler (there are exceptions, even for fairly complex blockings, as in [?, ?], but these techniques, due to their high cost and limited return, are still not deployed in any commercially available compiler).

For many algorithms, blocking can be done for multiple levels of cache, from the largest and slowest levels, down to and including even blocking for the registers [?].

Blocking has been most effective when applied to arrays, but it may be applied to more general data structures such as linked lists. However, since most dynamically allocated data structures do not display the high degree of spatial locality enjoyed by arrays, structure blocking is often less effective due to the increased associativity problems and under-use of cache lines (it is still an effective technique, just not as effective as when used on arrays).

4.1 Canonical blocking example: matrix-matrix multiply

This section gives a concrete example of blocking by demonstrating how it may be applied to matrix-matrix multiplication (hereafter abbreviated as matmul). We refer to matmul as the 'canonical blocking example', and for good reason. First, Linear Algebra is one of the disciplines whose performance can be most improved via blocking. Further, a great many matrix-matrix computations can be recast to use matmul as their compute engine [?], and many higher level Linear Algebra operations (eg. factorizations such as LU and Cholesky) can be formulated to call matmul directly [?], and thus matrix multiply is among the most studied and optimized of operations. Finally, blocking is the linchpin optimization for matmul. Without blocking, many other optimizations will not provide speedup. When blocking is applied in conjunction with computational optimizations and other memory optimizations, very large speedups are observed (i.e., speedups of a multiple orders of magnitude are not uncommon). As the gap between cache and memory performance grows, the benefits of blocking grow with it.

The most general form of matrix multiply is $C \leftarrow \alpha op(A) op(B) + \beta C$, where $op(X) = X$ or $X^T$, but we will examine the simplified case of where $\alpha = 1$, $\beta = 1$, $op(A) = op(B) = \text{No Transpose}$, so our operation is therefore: $C \leftarrow AB + C$. In order to maximize the clarity and simplicity of the examples, we will further assume that all matrices are square and of size $N$, and that this $N$ is an even multiple of the blocking factor $N_B$.

Figure 2 shows the loop nest and operand access for one step of an unblocked matmul. One important note about matmul is that it may be arbitrarily reordered: the three loops may all be interchanged, their iteration order varied, etc. This flexibility improves the ease of blocking considerably over more strongly ordered operations.

```
DO I = 1, N
  DO K = 1, N
    C(I,J) = C(I,J) + A(I,K) * B(K,J)
  END DO
END DO
```

(a) unblocked matrix multiply

```
          C
     ← A  × B
```

(b) Operand access in calculating one element of C

Figure 2: Unblocked matrix multiplication
From this figure, it can be seen that there are $2N^3$ flops (floating point operations) to be done (simply: there is an add and a multiply in the innermost loop, which is executed $N^3$ times). Further, the operation will require $N^3$ reads of $A$, $B$, and $C$, as well as $N^3$ writes of $C$. So, the operation contains $2N^3$ flops, $3N^3$ reads, and $N^3$ writes. This cannot be changed without changing the basic matmul algorithm, but blocking can transfer many of the references to nearer levels of the memory hierarchy.

The minimal access to the furthest level of the hierarchy (we will hereafter assume this level is main memory, though it may actually be disk due to virtual memory) is $3N^2$ reads and $N^2$ writes (obviously, no algorithm can avoid accessing each element at least once). The gap between $4N^3$ memory references and $4N^2$ is then the possible win this algorithm can gain from exploiting the memory hierarchy. A little more precisely, no optimization can reduce the access at any level of the hierarchy to less than $4N^2$, and all accesses at all levels of the hierarchy must add up to at least $4N^3$. As previously mentioned, blocking is a strategy aimed at minimizing the number of references requiring far access, and thus increasing the number satisfied by nearer levels of the hierarchy.

Subsequent sections will discuss blocking at various levels in turn, with special attention being paid for level 1 cache blocking (usually the most important), and register blocking. These two optimizations not only provide a bulk of the win, but are also present in most machines. Blocking for additional levels of the memory hierarchy is then discussed, but not in as great a detail.

4.1.1 Blocking for the first level of cache

In this section we discuss blocking for the first level of cache. This is usually the level 1 data cache, but it need not be. For instance, the SGI R8000, Intel Itanium, and Intel Pentium 4 are all examples of architectures where the floating point unit (FPU) does not access the level 1 cache, so that the level 2 cache becomes the first cache that can be blocked for. Despite this, we will refer to this first cache blocking as blocking for the L1 cache: this is not only common usage, but is often less unwieldy than more accurate constructions.

Figure 3(a) shows the loop nest of a blocked matrix multiply. Here we have specified the block size as $N_B$. Subsequent discussion will clarify how this value is chosen. Figure 3(b) shows the operand access necessary to compute one block of $C$. Note that we have chosen the “square” case of blocking, in that all dimensions of the blocking factor are the same. In practice, they can all be different, and there are sometimes system-specific reasons that increasing the length of the KB loop, in particular, would be a good idea. However, in general the square case is the best or close to it, and it simplifies the analysis considerably, so we use it exclusively in our discussions here.

In a matmul designed for L1 cache reuse, one of the input matrices (this is not the full input matrix, but rather the $N_B \times N_B$ blocked matrix) is brought completely into the L1 cache, and is then reused in looping over the rows or columns of the other input block. The example code given here brings in the matrix $A$, and loops over the columns of $B$; this was an arbitrary choice, and there is no theoretical reason it would be superior to bringing in $B$ and looping over the rows of $A$.

There is a common misconception that cache reuse is optimized when both input matrices, or all three matrices, fit into L1 cache. In fact, there is no win in fitting both input operands in the cache, and the only win in fitting all three matrices into the L1 cache is that it is possible, assuming the cache is not write-through, to reuse a block of $C$ across the iterations of the $K$ loop. Often, however, the L1 cache is write-through, while higher levels are not. If this is the case, there is no way to minimize the write cost, so keeping all three matrices in L1 helps only the read of $C$, at best. In these case, the advantages of using a large $N_B$ usually outweigh the read savings on $C$.

Therefore, often the best case is to fit all of $A$ into the cache, with room for at least two columns of $B$ and 1 cache line of $C$. Only one column of $B$ is actually accessed at a time in this scenario;
DO J = 1, N, NB
  DO I = 1, N, NB
    DO K = 1, N, NB
      DO IB = 0, NB-1
        DO JB = 0, NB-1
          DO KB = 0, NB-1
            C(I+IB,J+JB) = C(I+IB,J+JB) + A(I+IB,K+KB) * B(K+KB,J+JB)
          END DO
        END DO
      END DO
    END DO
  END DO
END DO

(a) Loop nest for blocked matrix multiply

(b) Operand access in calculating one block of C

Figure 3: Blocked matrix multiplication
having enough storage for two columns assures that the old column will be the least recently used data when the cache overflows, thus making it likely that all of A is kept in place (this obviously assumes the cache replacement policy is least recently used). In subsequent discussion, we will ignore the extra columns of B, in order to simplify the analysis.

With this information in hand, a good blocking factor may now be determined. There are two cases of interest, fitting all three blocks into the cache, and fitting only one. Both might be of interest on a system with a write-back cache, but, as previously mentioned, fitting all three in cache is often not worthwhile if the cache is write-through.

Assuming the size of the cache in data words is given by \( L_1 \), and that the blocking factor to be determined is \( N_B \), we can now provide equations which suggest good blocking factors. If all three operands are to be retained in the cache (usually of interest only with write-back caches), it is:

\[
N_B \approx \sqrt{\frac{L_1}{3}} \tag{1}
\]

If only one block is retained in the cache, a good value might be:

\[
N_B \approx \sqrt[3]{L_1} \tag{2}
\]

Equation 2 is often best even on machines with write-back caches. As we will see in the following analysis, the benefit of blocking is directly related to the size of \( N_B \), and thus making it large is of benefit. Usually, only machines with large write-back caches make use of Equation 1.

The inner three loops of Figure 3(a) represent a matmul of three \( N_B \times N_B \) matrices, containing \( 2N_B^3 \) computations and \( 4N_B^3 \) references. The total number of computations for the full matmul remain the same, as we can see by adding up the blocked multiplies:

\[
\sum_{i=1}^{N/N_B} \sum_{j=1}^{N/N_B} \sum_{k=1}^{N/N_B} 2N_B^3 = \left(\frac{N}{N_B}\right)^3(2N_B^3) = 2N^3 \tag{3}
\]

The same is true for the number of memory references, but where they hit in the hierarchy has changed. Assuming that the cache was not large enough to contain an entire row of A and a column of B, the unblocked matmul accessed main memory for all input operand accesses. Only \( C \) was optimally accessed, at a cost of \( N^2 \) main memory reads and writes.

In our block reordering, we may actually increase the \( C \) accesses that hit further cache levels, but we will keep them below the full \( N^3 \) term, while simultaneously reducing \( A \) and \( B \) as well.

The entire matrix block of \( A \) is loaded by the innermost two loops of Figure 3, and is retained for all accesses of the JB loop. Thus, although we still have \( 2N_B^3 \) flops being done by these loops, the load cost of \( A \) from further caches is merely \( N_B^2 \). Similarly, only one column is loaded for each iteration of the JB loop, and thus its access is also \( N_B^2 \).

Therefore, total accesses of \( A \) and \( B \) are now given by

\[
\sum_{i=1}^{N/N_B} \sum_{j=1}^{N/N_B} \sum_{k=1}^{N/N_B} N_B^2 = \left(\frac{N}{N_B}\right)^3(2N_B^2) = 2\frac{N^3}{N_B} \tag{4}
\]

The access of \( C \) varies depending on whether we kept all matrices in the cache, or only one. If we kept only one matrix in the cache, the access cost of \( C \) is \( 2\frac{N^3}{N_B} \), whereas if all three matrices were retained it is still optimal at \( 2N^2 \).

It may be unclear why this optimization is worth doing from this analysis. In particular, all terms are still the same order (\( N^3 \)), with merely a smaller coefficient, and this may appear to be
a fairly small savings. This is untrue for for several reasons. First, with the high cost of memory
accesses, any savings at all is significant. More importantly, as long as \( N_B \) is close in order of
magnitude to \( N \), the term is pretty much low-order (\( N^2 \)). Finally, this L1-blocked format also
allows for greater use of the further levels of the cache, both implicitly, and as we will later discuss,
explicitly through blocking.

To get an idea of how this might be, consider the case where only one input matrix is retained
in the cache, and thus we increase the number of further cache references on \( C \) from the optimal
\( 2N^2 \) to \( 2\frac{N^3}{N_B} \). As long as a level of cache between this one and main memory is large enough to
contain all \( 3N_B \) elements of the three blocks (almost always true), the main memory references on
\( C \) is still only \( 2N^2 \).

Similarly, if the a further cache level is large enough to contain a block of \( C \), and the row and
column panels of \( A \) and \( B \) (e.g. \( S_2 > 2N_BN + N_B^2 \)), then the references given in Equation 4 hit
that level of the cache, but the main memory references are optimal at \( 2N^2 \).

The unblocked algorithm can also get this implicit further cache blocking; in particular, it will
also have only \( 2N^2 \) main memory accesses if \( S_2 > 2N + 1 \), but it will require \( 2N^3 \) access of the
level 2 cache, rather than the \( 2\frac{N^3}{N_B} \) enjoyed by the blocked algorithm.

### 4.1.2 Blocking for the registers

As previously discussed, the three innermost loops of the block matmul given in Figure 3 are
themselves simply an \( N_B \times N_B \) matmul. Since registers represent a ‘level 0 cache’, closer to the
processor than the level 1 cache, we look at applying register blocking to this block multiply, rather
than the complete matmul. Therefore, register blocking will not reduce the number of memory
references hitting levels further out than the level 1, but it will reduce the number of references the
level 1 cache must satisfy.

In order to discuss this easily, Figure 4 shows the innermost three loops in isolation, ignoring the
effects of the outer three loops. As expected, we are back to an unblocked matmul, equivalent to
that shown in Figure 2.

```plaintext
DO IB = 1, NB
   DO JB = 1, NB
      DO KB = 1, NB
         C(IB,JB) = C(IB,JB) + A(IB,KB) * B(KB,JB)
      END DO
   END DO
END DO
```

Figure 4: L1 block matmul

The number of floating point registers available in a system is usually quite limited when com-
pared to typical cache sizes. The most common number is 32, though this number can be both
larger (as many as the 128 user-addressable registers enjoyed by the Intel Itanium) and smaller
(the 8-register stack inflicted on the programmer by the Intel x86 ISA comes to mind). With such
limited resources, we will obviously not be able to contain large amounts of data in the registers.
However, for the loops given in Figure 4, \( C \) may be maintained in registers throughout the inner-
most loop computation, resulting in the theoretical minimal access of \( 2N_B^2 \). In fact, by varying
which loop is innermost, we could minimize \( A \) or \( B \), but only one block may be thus minimized via
register blocking.
Since the cost of $C$ is at least twice that of either $A$ or $B$ (writes are usually at least as expensive as reads, and if the L1 cache is write-through, often much more expensive), the innermost loop should be the one given in the figure, while there should be no real difference in interchanging the outermost two loops. Note that even if a sophisticated write-back policy is in place, coherency upkeep will require additional bus traffic, so it is almost always a good idea to minimize the writes at this level rather than the reads (register/memory coherency is not generally hardware controlled, so this is the right level for this optimization).

Having decided to keep $C$ in the registers throughout the inner loop, we must stream $A$ and $B$ through the remaining registers. To do so, we dedicate $R_A$ registers to holding elements of $A$, and $R_B$ registers to holding the elements of $B$. We then must use $R_A \times R_B$ registers to hold the elements of $C$. Thus, the total number of registers used is given by $(R_A \times R_B) + R_A + R_B$. That means there are many potential ways to perform register blocking, but the square (or nearest to square that can be done with a given number of registers) provides the best theoretical performance. Figure 5 shows a $R_A = R_B = 4$ register blocking.

The access of $C$ has been reduced to its theoretical minimum ($2N^2$) with this register blocking. Each element of $A$ has been reused against each register $R_B$, and so its accesses of the level 1 cache is given by $N_B^3 \frac{R_C}{R_B}$, and the access of $B$ is similarly $N_B^3 \frac{R_A}{R_B}$. Assuming that $R = R_A = R_B$, then the access of $A$ and $B$ together is $2 \frac{N_B^3}{R}$.

### 4.1.3 Blocking for more distant cache levels

So far we have discussed blocking for the registers and the level 1 cache. As we have seen, this often leads to implicit blocking for the further cache levels. However, on systems with three levels of usable cache (remember that on some three level architectures, such as the Intel Itanium, only two levels are accessible to the FPU), this can be particularly suboptimal. Even on machines with only two levels of cache, explicit cache blocking can yield a benefit for problems too large to get the aforementioned implicit blocking.

In matmul, there is really no need to confine the algorithm to blocking for only a particular level of cache. One way to block for arbitrary levels of cache is to utilize recursion [?, ?, ?]. The basic idea is to halve the largest loop dimension at each step of the recurrence. At some level of the recurrence, each cache level will be “found”, in that the matrix will be appropriately sized to use that cache level. There are some extra tricks which can be applied to account for the fact that not all dimensions are the same (since some involve writes, while others involve only reads), but the description given above provides an adequate understanding of the concept.

The strength of the recursive method is that it is not machine dependent. The cache sizes are not explicitly specified, but are utilized automatically due to the algorithm. The main drawback is probably that it cannot guarantee maximal use of the hierarchy, since its best blocking may not fully utilize the available cache (for cache levels fairly close in size, it is even possible to miss a level). The cost of recursion on many machines also usually requires the algorithm to be somewhat modified from the clean description given above. Finally, the recursion must be halted at some point in order to apply register blocking, and if halted too early, level 1 blocking may be lost. Many applications thus use both methods: explicit blocking for the most fundamental algorithms, and recursion for others.

Since the cache sizes are usually large for the third level of usable caches, and thus implicit blocking is very likely to occur, it is fairly common to block only the first two levels of cache explicitly. This is what is done in [?]. In order to explicitly block for the second level of cache the $K$ loop is partitioned so that the matmul is transformed into $[K/K_m]$ rank-$K_m$ updates.

It is easily shown that the footprint of the algorithm computing a $N_B \times N_B$ section of $C$ in cache
DO IB = 1, NB, 4
  DO JB = 1, NB, 4
    c00 = C(IB ,JB ); c10 = C(IB+1,JB )
    c20 = C(IB+2,JB ); c30 = C(IB+3,JB )
    c01 = C(IB ,JB+1); c11 = C(IB+1,JB+1)
    c21 = C(IB+2,JB+1); c31 = C(IB+3,JB+1)
    c02 = C(IB ,JB+2); c12 = C(IB+1,JB+2)
    c22 = C(IB+2,JB+2); c32 = C(IB+3,JB+2)
    c03 = C(IB ,JB+3); c13 = C(IB+1,JB+3)
    c23 = C(IB+2,JB+3); c33 = C(IB+3,JB+3)
  DO KB = 1, NB
    a0 = A(IB ,KB); a1 = A(IB+1,KB)
    a2 = A(IB+2,KB); a3 = A(IB+3,KB)
    b0 = B(KB, JB); b1 = B(KB, JB+1)
    b2 = B(KB, JB+2); b3 = B(KB, JB+3)
    c00 = c00 + a0 * b0
    c10 = c10 + a1 * b0
    c20 = c20 + a2 * b0
    c30 = c30 + a3 * b0
    c01 = c01 + a0 * b1
    c11 = c11 + a1 * b1
    c21 = c21 + a2 * b1
    c31 = c31 + a3 * b1
    c02 = c02 + a0 * b2
    c12 = c12 + a1 * b2
    c22 = c22 + a2 * b2
    c32 = c32 + a3 * b2
    c03 = c03 + a0 * b3
    c13 = c13 + a1 * b3
    c23 = c23 + a2 * b3
    c33 = c33 + a3 * b3
  END DO
  C(IB ,JB ) = c00; C(IB+1,JB ) = c10
  C(IB+2,JB ) = c20; C(IB+3,JB ) = c30
  C(IB ,JB+1) = c01; C(IB+1,JB+1) = c11
  C(IB+2,JB+1) = c21; C(IB+3,JB+1) = c31
  C(IB ,JB+2) = c02; C(IB+1,JB+2) = c12
  C(IB+2,JB+2) = c22; C(IB+3,JB+2) = c32
  C(IB ,JB+3) = c03; C(IB+1,JB+3) = c13
  C(IB+2,JB+3) = c23; C(IB+3,JB+3) = c33
END DO
END DO

Figure 5: Block multiply after 4x4 register blocking applied
is roughly $2NN_B + N_B^2$, where $2NN_B$ stores the panels from $A$ and $B$, and the section of $C$ is of size $N_B^2$. If the above expression is set equal to the size of the usable L2 cache in words (call this value $L_2$), we can roughly solve for the maximal allowable $K$ loop:

$$K_m = \frac{L_2 - N_B^2}{2N_B}$$ (5)

Note that the $L_2$ in the equation above is difficult to determine a priori, since most level 2 caches are shared instruction and data, and only very rarely use true LRU replacement. Therefore, this value is best determined empirically, with the above equation providing an upper bound, where $L_2$ is set to the true size of the given cache.

### 4.2 Blocking for more general data structures

As previously mentioned, blocking may be performed for less regular structures such as trees or queues of dynamically allocated structures. With such structures, however, the application may not access all of the fields in a structure, which can lead to incompletely used cache lines. Further, since the structures are often not contiguous, cache line conflicts are more common, and in the extreme cases, TLB problems arise as well. In subsequent sections, we will discuss some techniques to ameliorate these costs, and these can be applied to help boost the effectiveness of blocking, but in general these issues make blocking less effective for these irregular structures.

One relatively common technique for such structures is to purposely under-utilize the cache, so that cache line conflicts are less likely to occur.

With these caveats, blocking for more general data structures is not much different than blocking for arrays. An example should illustrate the basic technique.

As an example, take modeling memory access on dynamic data structures. Each allocation of memory has an associated data structure giving details such as when it was allocated, the memory address where the allocation begins, the memory address where it ends, and the time it was deallocated. Refer to the data structure holding this information as the range structure. Deallocation and reallocation may allow differing data structures to be mapped to the same range, so these ranges are not unique.

As a memory access is recorded, these ranges must be searched in order to discover which data structure the access hit. The record of the access is similarly recorded in a IOREC data structure (containing such information as address and length referenced, the time when it occurred, what kind of access it was, etc.).

The simplest implementation results in a linear search of the range structures for each memory reference, which quickly becomes prohibitively expensive as the ranges exceed cache size, and the number of accesses goes up. Since we are dealing with non-unique ranges (as opposed to unique integral values), hashing is not as effective or easy to apply as usual.

Blocking, however, can be successfully employed. Instead of processing only one IOREC at a time, one can batch process a level 2 cache-size block of them (call this number $N_2$). Then, the sequential arrays of ranges can be blocked into roughly L1-size chunks (containing $N_1$ range structures). We can now search for matches in a $N_1$ chunk ranges along the entire $N_2$ block of accesses. The $N_2$ block of accesses are reused across all range blocks, and each range block is reused across this $N_2$ chunk of accesses.

There are many such blocking strategies available: which structure is in the L1 and which is in the L2 can be inverted, or both may be retained in a given level of cache, etc. In any case, total main memory accesses are dramatically reduced.
5 Array-specific optimizations

5.1 Cache line usage optimization for arrays

In order to maximally use the cache, we need to ensure that the spatial locality assumption holds true. For multidimensional arrays, this is most easily done by ensuring that access is along the contiguously-stored dimension (i.e., columns for Fortran, and rows for C). This will ensure that loaded cache lines are used in their entirety (with the possible exception of the first and last line in the column, obviously).

Even in the case where a Fortran array must be access row-wise (i.e., strided access), correct coding can ensure that a full cache line is used. For example, we accessed \( A \) row-wise in our matmul example. If there are 4 words in a cache line, the register blocking we showed in Figure 5 will utilize the full cache line even though the access is strided. Therefore, in general one looks to unroll the outer loop by the number of words in a cache line when row-wise access must be the inner loop.

Unrolling an outer loop by the cacheline is effective only in some arrays. First, the row stride must be a multiple of the cacheline length in words (so that all columns start at the same offset in the cache line). Then, if the row of the sub-matrix of interest is not at the start of a cache line, loop peeling must be employed to make it so.

If \( lda \) is not a multiple of the cache line length, the same technique may still be employed, but not with adjacent columns. Instead, if there are four words in a cache line, every fourth column will have the same alignment, and so they must be accessed together instead of adjacent columns. Then, as with the adjacent column technique discussed previously, loop peeling may need to be employed.

Once we are accessing contiguous storage (unidimensional arrays or a particular column of a multidimensional array), we can choose to traverse it from the smallest-to-largest (by memory address) order, or in largest-to-smallest order. Because cache lines are filled in smallest-to-largest order, we want to access the array in this order if possible. As we will see in Section 5.2, this order is also generally superior for hardware prefetch.

5.2 Hardware prefetch optimization for arrays

Many architectures attempt to minimize memory stalls by performing hardware-directed prefetch. In this scheme, the memory subsystem attempts to find a pattern in the memory references being made by a program, and then while the processor is busy with the computation, additional lines are fetched into the cache before they are actually requested. Some systems have several prefetch units (thus requiring several streams of input for best utilization), and some have only one. Some have fairly sophisticated pattern-matching schemes, but quite often it looks for sequential, lowest-to-highest access only. For this reason, on some systems, simply accessing contiguous storage in reverse order may lose any benefit coming from hardware prefetch.

5.3 Cache line conflict minimization for arrays

For sequential unidimensional or column-wise multidimensional array access, cache line conflicts are generally about as minimal as they can get (since we are accessing a contiguous chunk of memory). However, if multiple columns are accessed at a time (as is quite common in optimized code), the columns may easily cause unnecessary cache line conflicts. Recall that our columns are constantly strided by \( lda \). If \( lda \) is a multiple of the cache line, it is possible for the same row index of two consecutive columns to map to the same place in cache.
The most common fix is to choose \( \text{lda} \) such that this is not true. Since most caches are powers of two in size, \( \text{lda} \) should never be chosen as a power of two.

### 5.4 TLB optimization for arrays

TLB optimization is often applied as part of the blocking discussed earlier. To see why we might need optimizations related to the TLB, recall that our arrays are constantly strided by \( \text{lda} \). When \( \text{lda} \) grows larger than the page size, each matrix column requires at least one TLB entry. If we are applying blocking, we are accessing only \( N_B \) rows and columns in a sub-matrix of \( A \). Therefore, only when our sub-matrix spans two pages (happens very rarely overall), will we need more than one TLB for each column in our blocked access. Therefore, the most common solution in this case is to simply set \( N_B \times N_M \) (where \( N_M \) is the number of matrices being accessed by the blocked application in question) to be less than the number of TLB entries, in order to guarantee that we are not constantly loading and flushing TLB entries.

Another optimization applied when blocking large matrices is to copy them to \emph{block-major format}, so that each block is contiguous in memory, so that at most \( N_M \) TLBs are required for each blocked operation.

If an operation is not already being blocked for cache reasons (e.g., there is no reuse, so blocking was not applied), TLB-blocking may still be applied. An example of this can be seen in the LAPACK operation \emph{laswp}. This routine applies a series of row interchanges to a matrix. In LAPACK 2.0, the relevant loop section was that shown in Figure 6(a).

Since \( N \) is can be quite large, this routine displayed TLB problems on many platforms. What is needed is to block the \( K2 - K1 + 1 \) rows to be swapped, so that that column-access is achieved. Therefore, for LAPACK 3.0 I rewrote this routine for TLB optimization, as shown in Figure 6(b). Since this code needs to be kept as system-independent as possible, it assumes 32 is a safe minimum on the number of TLBs a system would have available for this operation. This simple example demonstrates how these kinds of optimizations add complexity to code, both by making the main operation less straightforward, and through the introduction of required cleanup code.

### 5.5 Putting these optimizations together with blocking for matmul

When level 1 cache blocking is being performed, and the arrays are large enough to tolerate the additional cost, most of these optimizations can be achieved by simply copying the input matrices into block-major format before starting the matmul. In block-major format, \( N_B \times N_B \) block is contiguous storage, and the matrix is transposed so that the inner loop accesses it contiguously. With this operation done, pretty much all the optimizations we have been discussing are achieved. Since matmul is a \( O(N^3) \) algorithm, it can easily tolerate this additional \( O(N^2) \) cost for large matrices.

### 6 Structure-specific optimizations

As mentioned in our blocking discussion, dynamically allocated data structures often provide poor cache utilization. In particular, there are increased problems with cache line underutilization and cache line conflicts.

There are a number of optimization techniques which may be applied to dynamically allocated structures. In order to more fully use cache lines, intra-instance (e.g., within one allocation of a given structure) field reordering can be performed, after analysis (either static or dynamic) has shown which fields are accessed in close temporal proximity, as in [?, ?, ?]. Another popular
DO 10 I = K1, K2
  IP = IPIV( I )
  IF( IP.NE.I ) then
    DO 10 K = 1, N
      TEMP = A( I, K )
      A( I, K ) = A( IP, K )
      A( IP, K ) = TEMP
    END DO
  END IF
10 CONTINUE
ENDIF
20 CONTINUE

(a) Loop nest for xLASWP in LAPACK 2.0

N32 = ( N / 32 )*32
IF( N32.NE.0 ) THEN
  DO 30 J = 1, N32, 32
    IX = IX0
    DO 20 I = I1, I2, INC
      IP = IPIV( IX )
      IF( IP.NE.I ) THEN
        DO 10 K = J, J + 31
          TEMP = A( I, K )
          A( I, K ) = A( IP, K )
          A( IP, K ) = TEMP
        END DO
      END IF
      IX = IX + INCX
    20 CONTINUE
  30 CONTINUE
ENDIF
IF( N32.NE.N ) THEN
  N32 = N32 + 1
  IX = IX0
  DO 50 I = I1, I2, INC
    IP = IPIV( IX )
    IF( IP.NE.I ) THEN
      DO 40 K = N32, N
        TEMP = A( I, K )
        A( I, K ) = A( IP, K )
        A( IP, K ) = TEMP
      END DO
    END IF
    IX = IX + INCX
  50 CONTINUE
ENDIF

(b) Loop nest for xLASWP in LAPACK 3.0

Figure 6: LAPACK’s xLASWP before and after TLB optimization
technique splits the structures so that the frequently-accessed fields are in a separate, smaller and more dense data structure, as in [?].

6.1 Basic information on structure allocation

In order to understand the forthcoming discussion, we need a basic understanding of how heterogeneous structures are arranged in memory. Examples will be used illustrate how this is done, and for all such discussions we will assume that characters require one byte, shorts two, integers and floats four, and doubles eight bytes. These values are typical on many machines, but certainly not true for all. However, assuming particular values should allow for more informative examples.

Most machines have a concept of native alignment for a given data type, where access of each data type is optimized if its storage begins on a multiple of its length. Therefore, a 4-byte integer should have a starting address in memory that is a multiple of four, a double should have a starting address that is a multiple of eight, and so on.

This means that padding is often used in heterogeneous structures (structures containing multiple data types). This padding will often be both internal to the structure, and also after the last structure element.

Here’s a simple example to illustrate the use of internal padding:

```c
struct internpad
{
    char c;
    int  i;
}
```

In this structure, given our assumptions, c will be followed by three bytes of padding, in order to guarantee that i begins on an address that is a multiple of four (obviously, the structure itself will need to be allocated on a multiple of four as well).

Since programmers sometimes wish to use arrays of structures, most compilers perform post-padding when required to ensure that each instance in such an array will start on an appropriate boundary. So, the structure

```c
struct externpad
{
    int  i;
    char c;
}
```

Will very likely have three bytes of storage suffixed to c.

6.2 Optimization caveats

All of the optimizations we will be discussing involve moving fields around, both inside the structure, and fission or fusion of structure instances. Some, and often all, of these optimizations are illegal in some languages, particular in our example language, ANSI C. Therefore, if they are done automatically by the compiler, they must only be applied only when specifically invoked by the programmer. When applied by hand, the programmer must guarantee that these optimizations do not change the semantics of the program.
6.3 Intra-structure reorderings for cache line usage optimization

One relatively simple technique with multiple uses is field reordering. At its most basic, this technique can be very straightforward indeed. In order to illustrate, say we have the following structure:

```c
struct example
{
    char c;
    int i;
    short s;
}
```

For the sake of simplicity, assume that our cache line length is four bytes, and type lengths are as previously discussed. Most compilers will take the above structure, and place inter-element padding after the character, and post-padding after s. As it is then, the load of c will waste cache line space with padding, and the same will be true of the short. Therefore, the above structure will likely take up three cache lines, and thus waste five bytes for every twelve loaded.

A very simple reordering to:

```c
struct example
{
    int i;
    short s;
    char c;
}
```

eliminates a great deal of this waste. Here, i takes up one entire cache line, and s and c share one, leaving only one byte of wasted cache line load, and only two cache lines required per structure.

This example illustrates a general rule to apply when no other information about data structure layout is available for performing a more complete analysis: declare the structure fields in descending order of base-type length. When the native alignment requirements are known, further optimize by putting types whose lengths add up to require no padding together.

The idea of rearranging a structure so that as many fields as possible share a cache line is known as cache line packing, and once the cache line size is known, it can be applied to many structures, though candidates must obviously have enough members of the right length.

With further analysis, more beneficial reorderings can be found. In particular, both static and dynamic analysis can discover which fields are accessed in close temporal proximity within a program. These fields can then be moved to share a cache line so that the load of one implicitly loads the other. If there is a typical ordering to the field access, they can be ordered this way in the structure so non-blocking cache line fills are fully exploited. Rarely accessed fields can similarly be grouped together on cache lines, in the hopes of avoiding the line load altogether.

6.4 Structure fission for increased locality

The same analysis that was previously applied to reorder fields within a given instance of the structure can now be applied in order to perform more fundamental transformations. We can split a structure into hot and cold fields. The hot fields are those fields which are frequently accessed, and the cold are rarely accessed. The original structure is then split into two separate data structures, with the much-reduced hot structure containing a pointer to its correlated cold structure. Access
to the cold structure will actually be more expensive in this scheme, but the idea is that the faster
hot accesses (which make up the bulk of the accesses) will make it worthwhile. Figure 7(a) shows
an example of a structure where only the i, j, and next fields are frequently accessed. Figure 7(b)
then shows the resulting structures after fission.

```c
struct example_hot
{
    int i, j;
    struct example_cold *cold;
    struct example_hot *next;
};

struct example
{
    int i, j, k, h;
    double f, d;
    struct example *next;
};

struct example_hot
{
    int i, j;
    struct example_cold *cold;
    struct example_hot *next;
};

struct example_cold
{
    int k, h;
    double f, d;
};
```

(a) Original structure           (b) Split hot and cold structures

Figure 7: Structure before (a) and after (b) structure fission

The essential advantage of the hot structure is that spatial locality will be much improved,
boosting performance in several ways. First, since the cold fields have been removed, hardware
prefetch will more often be correct within the structure. Since all the fields left in the structure
are frequently accessed, cache line packing is already to some extent done, though the temporal
analysis will improve it further. Finally, as a preview of TLB optimization, if we take care to
allocate the hot and cold structures on different pages, we can pack the hot fields onto just a few
virtual memory pages.

6.5 Structure fusion to enable cache line packing

In addition to structure fission, [?] also suggests structure fusion for small data structures. If the
data structure has insufficient hot fields to fill a cache line, the hot fields of several instances are
combined. An example of this is given in Figure 8. In this example, the fields a and c are hot,
while the fields b and d are cold. Further, it is assumed that a cache line is the same length as
eight integers. In this case, we fuse four structure instances into one. Now, in the fused structure
given in Figure 8(b), each structure instance has one cache-line full of frequently accessed items,
and a cache-line full of cold items. The cold cache-line will often not be referenced at all; in the
absence of hardware prefetch, a traversal of this modified structure will result in loading roughly
half as many cache lines.

So, this is very like normal cache line packing, but we are amalgamating multiple instances in
order to enable it. The paper discusses applying this technique only to array-accessed structures
(as opposed to linked lists, queues, etc), where these fused structures may be easily accessed by
modifying the index appropriately. It is possible to perform at least some limited structure fusion
and still utilize general pointer techniques, but the paper does not discuss any such methods.
struct example
{
  int a, b, c, d;
} s0, s1, s2, s3;

struct example_4fused
{
  int a0, c0, a1, c1, a2, c2, a3, c3;
  int b0, d0, b1, d1, b2, d2, b3, d3;
} s0_3;

(a) Original structure                  (b) Fused structures

Figure 8: Structure before (a) and after (b) structure fusion

6.6 Cache line conflict minimization for structures

The techniques discussed so far help with the problem of poor cache line utilization, but [?] offers an optimization for reducing cache line conflicts as well. This optimization requires significant modification of memory handling, as well as detailed usage analysis.

In this approach, profiling is used to discover typical memory access patterns (including global, heap, and stack data). A Temporal Relationship Graph (TRG) is developed from this profiling. A TRG edge between two objects has a weight that corresponds to the estimated number of cache misses that would occur if the two objects were mapped to the same associativity set (but not the same cache line, obviously). Therefore, an edge does not appear if they are never live at the same time (variables that are not live at the same time would have a zero cost if they mapped to the same cache line).

Memory objects are then split into ‘popular’ (extremely high degree of TRG connectivity) and ‘unpopular’ sets. The popularity of an object is a sum of the weights of the TRG edges that reference it. All objects that account for 99% of the total popularity are classified as popular, and the rest are unpopular.

Only the objects in the popular set are then optimized to reduce cache line conflicts. Initially, a compound node structure is created for each popular object. A compound node structure is a set of objects that have been grouped together in order to optimize cache usage.

Later, popular objects connected by TRG edges will be brought into the same compound node structure, which will dictate their relative cache offset. The first object in the compound node might be chosen to be allocated to an address mapping to the first line of the cache (for simplicity assume that it requires exactly one cache line). The second object added to the node might then be assigned to start at the second line of the cache, and so on. For static data, this memory layout can be enforced at compile-time, but for dynamic data, it must be supplied by changing the allocation/deallocation routines.

Obviously, this algorithm works better for static data than it does dynamic. Since this analysis is done on a per-instance basis (as opposed to being performed for all allocated instances of a given structure, for instance), some method of naming the allocated objects must be employed. The naming method used does not guarantee uniqueness, so this complicates the process. The number of dynamically allocated structure instances may vary with differing inputs, and so if profiling has not generated this name before, the allocation does not benefit from the technique (i.e., it is declared ‘unpopular’).

6.7 TLB optimization for structures

There is an obvious extension of cache line packing in caching optimization to page-packing for TLB optimization. From our previous discussion, it is clear that when structure fission has been performed, the hot split of the structure can be packed onto contiguous pages to ensure good TLB
usage. However, [?] goes further. They use dynamic analysis, and perform most of their analysis on whole structures. They classify each structure into one of three classes:

1. **Hot structures**: packed together on small range of pages
2. **Short-lived structures**: packed together on separate small range of pages
3. **Cold structures**: put elsewhere

The idea is that the hot and short-lived pages together will hopefully require less pages than the machine has TLB entries. By segregating the short-lived structures, they hope to reduce memory fragmentation due to the allocation/deallocation cycle, which should result in the long-term hot structures being even more densely packed.

### 7 Memory hierarchy optimization for irregular applications

So far, we have discussed memory optimizations for regular applications, where the pattern of data access is known at compile time. In irregular applications, data and computation order are unknown until runtime, which serves as a significant barrier to applying the techniques that we have been discussing.

In regular applications such as we have previously discussed, an optimizer needs to know very little information about the actual problem being solved. In such applications is it usually sufficient to understand the main data structures being used (arrays, vectors, dynamic structs, etc.), and some rough inter- and intra-dependence information of these structures. In irregular applications, on the other hand, knowledge of the physical properties of the system being modeled must almost always be taken into account in order to apply memory hierarchy optimizations. Since irregular applications embody many different physical systems, each type may require significantly different optimizations techniques. This in turn implies that studying irregular applications in detail would require much more space than we can possibly include in this survey. Therefore, bearing in mind that irregular applications per se are unlikely to be the main topic of our research, we will confine ourselves to a brief overview how memory optimizations have been applied to irregular applications in [?].

This paper concentrates on the applications involving particles or mesh elements in spatial neighborhoods. The primary example here is an n-body simulation, where particles within a given radius are said to mutually interact. Figure 9 shows an example of such a n-body problem in two dimensions, where two interaction radii for particles $P_i$ and $P_j$ are shown. Notice that this problem is irregular, since it is impossible to predict in advance of seeing the data how many and which particles will be within a given interaction radius.

As with regular applications, we want to increase spatial and temporal locality, which we can do via data and computational reorderings, respectively. The key insight for increasing both spatial and temporal locality for this class of problems is that elements that are close together in physical space interact.

In order to enhance spatial locality, we employ **data reordering**. A data reordering consists of changing the layout of particle information in memory. Since we are applying this optimization at runtime, data reordering requires memory shuffling, so the performance benefit during the computation must outweigh the overhead of the memory copy. Since the data is $O(N)$, while the computations are $O(N^2)$, this overhead will not be intrusive for any but the smallest of problems.

In the n-body problems studied, the particles move around, so every so many time steps the radius of interaction must be recalculated. If a space-aware reordering is being employed, data ordering therefore will need to be reshuffled to match the new particle positions.
So, we need a way to retain the concept of locality in a multidimensional space when the particles are mapped to unidimensional memory. One way of roughly retaining higher-dimensional locality when mapped to a one dimensional space is to employ space filling curves.

The definition of a space filling curve from [?] is A space-filling curve for some finite space of d dimensions (d ≥ 2) is a continuous, non-smooth curve that passes arbitrarily close to every point. Each point in a d-dimensional space can be mapped to the nearest position along a 1-dimensional space-filling curve by applying a sequence of bit-level logical operations to it’s d-dimensional coordinates.

The space filling curve used in this paper is a Hilbert curve, which is essentially recursive in nature. This recursive structure helps to ensure that points nearby in space are relatively close on the Hilbert curve.

There are many different data reorderings that could be employed, but since we are applying them during execution, they cannot be prohibitively expensive to compute. Unlike many other locality-retaining data reorderings, Hilbert curves are inexpensive to compute. An even less expensive order to compute is first touch reordering, where particle data is reordered so that it is laid out in memory in the order in which it is accessed on the first pass through the data. The only cost of this ordering is storage in order to keep an access history on the first pass. Therefore, the data re orderings compared in the paper are Hilbert, first touch, and none.

Data re orderings help only spatial locality. In order to boost temporal locality, computational reordering is necessary. I.e., instead of changing the order of the particles in memory, change the order in which they are referenced. Note that changing of the computational order to encourage temporal locality is what we have been referring to as blocking in this survey paper.

The authors study two types of computational reorderings. One is a Hilbert reordering, and the second is contiguous memory blocking.

All of the computations involve interactions between pairs of particles. In Hilbert reordering, the computations are sorted along the Hilbert curve according to the first of the pair, and ties are broken by a Hilbert sorting on the second of the pair.

For contiguous memory blocking, the blocking is applied to all levels of the memory hierarchy except registers: in the studied architecture, level 1, TLB, and level 2 caches. The blocking used here is always on contiguous memory (unlike our earlier examples, where non-contiguous chunks of rows or columns of an array are lumped together in a block), so the effectiveness of this computational reordering will strongly depend on the data reordering that has been applied.

They then compare these various combinations of data and computational reordering. The best performance is achieved via a Hilbert data sort and a Hilbert computational reordering, which achieves almost a four-fold speedup (remember that this includes the overhead of computational and data reordering).
Hilbert data and contiguous blocking are almost as good. The most likely explanation for the slowdown over Hilbert-Hilbert is that contiguous blocking computational reordering requires more overhead to apply.

Part II

Bit-level optimization

In this section we survey work being done to support bit-level optimization. This field is far less well defined than memory hierarchy optimizations (which have received intense and deserved scrutiny since the advent of the first cache-based machine), and so our survey of this field will be correspondingly less complete.

In the early days of computing, bit-level optimizations were very important due to the extreme resource limitations inherent in the relatively primitive machines and tools that were then available. At this time, the programmer was responsible for applying these bit-level optimizations, as indeed he/she was for most optimization. The various sized integers in C are to a large extent a reflection of this early emphasis on bit-level optimization.

As machines progressed to have faster processors, more functional units, wider busses and larger memories, bit-level optimization was de-emphasized in aid of portability, maintainability, and programmer time. At the present time, continuing architecture evolution and the rise of the embedded market are renewing interest in bit-level optimization.

Bit level optimizations help in at least three relatively distinct areas: time (performance), power, and memory. The introduction of narrow-width oriented SIMD instructions (mainly used for today’s multimedia applications) to commodity processors has greatly expanded the possibilities of achieving performance improvements from bit level optimizations. The expansion of the embedded market, and the growing problems with heat dissipation in increasingly dense microprocessors, has renewed interest in power savings. As processor speeds have continued to increase much more rapidly than memory speeds, techniques for reducing memory access (as bit level compression can do) have become increasingly important. Finally, embedded systems have reinvigorated even the original interest in methods to reduce total memory footprint.

The basic idea behind bit-level optimization is easy enough to understand. In many operations, only a small subset of the bits of the operands are actually used or set, and so optimization may be accomplished by only operating on or storing these relevant bits. For our own work, we intend to concentrate on the storage savings (although of course we will exploit computational optimizations if warranted), and thus we use the term bit-level compression. Optimization for storage uses similar analysis and detection methodologies as those employed in performing bit-level computational optimizations, and so this section deals with bit-level optimizations in general.

Our bit-level optimization discussion will be organized in the following way: We will first discuss various ways in which bits may be classified as unneeded or useless, and then survey both dynamic and static methods of finding and analyzing them. Finally, we will discuss the ways in which these useless bits have been exploited. Useless bits are exploited for both computational and memory optimizations; since our own work emphasizes memory optimizations, we will cover these in more detail.
char a, b;
...

b = a & 0x0f;

Figure 10: Never useful and constant bits

8 Classifications of unused bits

In order to apply bit level optimizations, you must first define a scheme for classifying which bits will be compressed, elided or ignored. If analysis were perfect, and machines infinitely adaptable, this would mean that only the exact bits required by a given operation would be accessed. In the real world, this is rarely practical, and so it is important to understand if savings on more restricted sets of bits can still provide good opportunities for optimization.

One of the more comprehensive studies of the classifications of useless bits is given in [?]. They classify the bits themselves into three categories:

1. **N**, *Never useful*: bits that are provably never accessed.

2. **C**, *Constant*: bits that are sometimes used, but are known to have a particular value.

3. **V**, *Variable*: all bits not in **N** or **C**.

To understand the distinction between never useful and constant bits, examine Figure 10. Assuming this is the only set of b, the four most significant bits are known to have a constant value of 0. On the other hand, if this is the only read of a, we know that the four most significant bits of a are never useful.

The authors’ main interest in making this study was to perform computational optimizations, and thus they distinguished between **N** and **C**. If bits were **N**, then any output was acceptable, but if it were **C**, specialized hardware would need to output the known quantities. In general, the distinction between **N** and **C** is a narrow one, and we will often lump them together.

With these bit classifications, the authors then studied five classifications of useless bits based on them. These five classifications were built from a base of three bit-range categorizations, coupled with two modifier classes. The range classification determines the ranges within the word that are being examined, and the modifiers deal with which bits within that range are classified as useless.

If both **N** and **C** bits are declared useless, the class is modified as optimistic, while less-optimistic describes declaring the **N** bits only as useless.

The three range classes are then:

- **bit-wise**: Useless bits may be found anywhere in the word.
- **prefix**: Useless bits are only searched for in contiguous high-order bits of the word
- **data range**: All leading sign bits except one are ignored. For negative numbers, assuming two’s complement storage, all leading ones are assumed, as are the leading zeros of a positive number. In this case, all these leading bits are elided (compressed) to one bit.

Since the data range category has a built in assumption about which bits are useless, only the first two categories were subdivided into optimistic and less optimistic, and so these classes and modifiers yield the five classifications covered in the study.

These categories were then explored against various ways of detecting unused bits, including profiling for perfect analysis. All of these combinations were then tried across several well known
benchmark suites. The central conclusion was that the prefix category captures the bulk of the possible win regardless of detection strategy. Bit-wise provides much too small of an improvement to be worthwhile, and data range is mostly a weaker form of prefix. As expected, considering \( N \) and \( C \) as useless provided substantial savings over considering only \( N \).

One caveat to this analysis is that only integer operations were considered. Since their main interest was in ALU optimization, it appears that pointer usage was not considered, and floating point usage was explicitly disallowed.

9 Detecting and analyzing unused bits

In order to exploit unused bits, they must first be identified. Most surveyed papers have slightly different approaches. The easiest to understand is undoubtedly that given in [?], where the problem is handed off to proposed hardware. In this scheme, each operand to the ALU is examined by the hardware, and data range eliding of the sign bit is done in order to automatically save these leading bits.

The remaining surveyed papers all employ varying types of static analysis in order to extract bitwidth information from standard code. While each paper approaches this analysis slightly differently, there are features shared by all. All approaches have a local phase, where varying methodologies are employed in order to discover the bitwidth requirements of each instruction. This local bitwidth information is then related using differing types of control flow analysis in order to get a global view of the data.

Each paper does this static analysis differently, and so we will summarize each in turn in the following subsection. Our own work will likely utilize profiling, and so our analysis techniques will undoubtedly be considerably different.

9.1 Static range analysis

In [?], the authors concern themselves with data range transformations on integral data (integers and pointers). They employ a static analysis, where the data range problem is generalized to a value range propagation problem. A suite of bitwidth extraction techniques are employed to perform bi-directional propagation of data range values. Finally, an algorithm is given which calculates closed form solutions in order to discover bitwidth information in the presence of loops.

A data range is a single connected subrange of integers, from a lower bound to an upper bound (e.g., \([1 \ldots 100]\) or \([-50 \ldots 50]\)). In the local step, operands are examined to find implicit bit ranges. Bitwise and, divides, right shifts, and type promotions are all taken into account in restricting ranges. Further analysis is done by assuming the code is always correct. For instance, if we see the statement: \( s = s + 1 \), we can be sure that before this statement, the range of \( s \) is at most \([INT_{\text{min}}, INT_{\text{max}} - 1]\). Similarly, if an array’s size is known, and an integer is used to index it, that integer’s range is implicitly known to be within the array bounds. Comparisons are taken into account as well (i.e., if an if statements tests that a value is positive, for instance, statements within the if body can use the restricted range).

When global analysis is performed, we need a technique to group and subdivide ranges. Ranges are combined using the specialized union operator \( \sqcup \), and subdivided using the specialized intersection operator \( \sqcap \). These operations are defined as:

- \([a_l, a_h] \sqcup [b_l, b_h] = [\min(a_l, b_l), \max(a_h, b_h)]\)
- \([a_l, a_h] \sqcap [b_l, b_h] = [\max(a_l, b_l), \min(a_h, b_h)]\)
These union and intersection operators are then used to propagate range information in both forward and backward traversals of the control flow. An example of forward propagation can be seen in transferring knowledge from an if to its body (eg., \(\text{if } (s > 0)\)) means the true body can assume a range of \([1, \text{INT}_{\text{max}}]\). A simple example of backward propagation can be seen when a integer is found to index an array of known size. The array’s limits can then be back propagated as limitation on the integer.

Since loops usually contain the bulk of the dynamically executed instructions, the authors make a special point to handle bit range analysis on loop-carried expressions. If knowledge of static loop bounds is not taken into account, most loop-carried expressions will be required to use the full type range for safety.

In order to tackle this problem, the authors’ compiler identifies loop-carried expressions and finds closed-form solutions where possible. They use the techniques discussed in \([?]\) for detecting such sequences, and finding their closed form. If such a closed form can be found, an upper bound on the bitwidth requirements may then be found, assuming the number of loop iterations is known.

### 9.2 Static section analysis

In \([?]\), the authors propose a methodology for performing bit-section analysis. In the local phase, individual statements are analyzed to determine which bit sections (of the LHS and RHS) are actually required for a given operation. In a global phase, control flow information is used to propagate this information. The idea is that once this is done, traditional optimizations (constant folding, dead code elimination, common sub-expression elimination, etc.) may then be applied to bit sections. Algorithms are provided to perform the required local and global analysis.

The local phase consists of two main parts. The first step is to find the bit sections of the LHS variable. A bit section is a series of consecutive bits that are set with/involved in the same operation. For instance, assuming \(a\) is a 32 bit variable,

\[
a = (b >> 4) \& 0xf;
\]

it requires two bit sections. The upper twenty-eight bits comprise a bit section which is zeroed, while the lower four bits are assigned bits from \(b\). In order to find these bit sections, the RHS expression tree is traversed in bottom up fashion.

In the second phase of the local analysis, the expression tree is again traversed in bottom up order, but this time to discover the bit sections of the RHS operands required by the assignment (i.e., in our previous example, we will need the bits 4-7 of \(b\), but nothing else from the RHS).

In the global phase, the bit sections introduced in the local analysis are combined (resulting in eliminating a bit section) or split (producing an additional bit section). In order to find when a section can be eliminated (via a combine), we must discover it’s last use, leading to a backward traversal of the control flow. Finding when a new section is required (eg., a split) is accomplished via a forward analysis. Algorithms are given to perform these analyses.

In order to maximize the opportunity for optimizations, the authors provide an algorithm that combines sections at the earliest opportunity in the control flow, and splits them as late as possible.

### 9.3 Static range and section analysis

In \([?]\), both section and range analysis are performed. As before, bits are classified into the three main categories \((N, C, \text{ or } V)\) in a local phase. Propagating \(C\) bits is seen as constant folding applied to bits, whereas \(N\) propagation is described as bit-wise extension of dead code elimination. The section analysis appears to be functionally similar to the analysis discussed in Section 9.2,
though it is specified in a different manner. As before, both forward and backwards control flow analysis is performed.

Loops are handled via iteration, rather than closed form. The authors also add a special case of range analysis which is applied to loop carried dependencies. Because it is not handled by section analysis, they implement the loop carried optimizations from [376], as discussed in Section 9.1.

10 Exploiting unused bits

Once a classification of unused bits is selected, and a method of identifying them has been implemented, these unused bits must be exploited for some type of optimization. We split this discussion into two broad categories: computation and memory optimizations.

10.1 Computational optimization

There are several ways in which computational optimization can result from bitwise analysis. One fairly straightforward way (mentioned in [267]) to exploit narrow width data is to utilize SIMD (single instruction, multiple data) operations. This can be done using the recently popularized hardware specific SIMD extensions. Examples include Intel’s MMX and SSE, AMD’s 3DNow!, and Motorola/Apple’s AltiVec. All of these SIMD add-ons incorporate ISA extensions coupled with specialized hardware that performs the same instruction on contiguous sections of bits. For instance, MMX allows 8 8-bit values to be packed into a 64-bit register, and then the same operation will be applied to all values (eg., sum, shift, etc). Since all computations are performed in parallel, a perfect implementation would reduce the computational costs to an eighth. This rarely occurs in practice, however, as there are usually pack/unpack costs, alignment issues, etc., to be considered.

Another use for bitwise optimization is in reconfigurable architectures, where the reduced bit demands can result in specialized ALUs for the width in question. This allows for either power savings (because circuits are not activated for the unneeded bits) or additional compute power (because the circuits that would have accessed the unneeded bits may instead be used to perform a second computation).

The authors of [376] point out that it is sometimes possible to achieve SIMD optimizations even in the absence of explicit SIMD extensions. For instance, four 8-bit bit-wise ands can be performed in parallel using a standard 32-bit ALU.

Also discussed in this paper is a power-consumption optimization. In this case unneeded bit detection is used to turn off sections of the integer arithmetic units in order to save power.

10.2 Memory optimization

In [376], both prefix and data range optimizations are utilized to reduce dynamic memory usage. In order to understand how this may be accomplished, we will discuss how integers and pointers may be compressed, the way in which dynamic structures must be transformed in order to support such compression, and finally, the general results of applying the discussed techniques.

These techniques are closely related to the structure-specific optimizations discussed in Section 6, and they involve a similar shuffling around of structure elements. Therefore, the optimization caveats discussed in Section 6.2 apply to these transformations as well.
10.2.1 Compressing integers and pointers

Data range optimization is used to compress integers, assumed to be 32-bits long. The leading 18 bits are examined to determine if they are all 1 or all 0 (narrow width negative or positive numbers, respectively). If they are, these leading 18 sign bits may be elided to 1 sign bit, leaving a compressed operand of 15 bits \((32 - 18 + 1 = 15)\). Now, two such integers, for instance, could be packed into the storage of one (with two extra bits unused; we will see why later).

For pointers, common prefix analysis is used instead of range analysis. What we look for here is two addresses that share a common 17 bit prefix (again leaving 15 bits to be stored). If this is the case, only the first address need be stored in full, and the upper 17 bits of the second address may be recovered with a simple mask. One important note is that the first, uncompressed address, may be implicitly supplied, as well as explicitly. For instance, assume we have the structure:

```c
struct example
{
    ....
    int ival;
    struct example *next;
    ....
};
```

If the `next` pointer shares a common prefix with the address of the structure it appears in, then the address of this structure instance may form the mask required to compress `next`.

So, if pointers or integers are compressible by our scheme, they are both reduced to 15 bits in size. The compression scheme then looks for pairs of compressible operands, which can be compressed together into one 32 bit value. Figure 11 shows the format of such a compressed item; the details of this compression will be further explained by the example given in Section 10.2.2.

10.2.2 Structure transformation to support compression

Figure 12(a) shows a simple example structure that we want to apply compression to. Figure 12(b) shows the structures we will need in order to perform this compression. When compression can be performed, each uncompressed `example` instance generates only the corresponding compressed `example` allocation, thus halving the required space. For each instance which cannot be compressed, however, an allocation of both the compressed `example` and the new structure `example_overflow` is required.

The compressed `ival_next` is now split up as in Figure 11, where operand 0 is `ival` and operand 1 is `next`. Thus, to uncompressed the `next` pointer, the fifteen low-order bits from `ival_next` are
concatenated with the seventeen high order bits of the current structure instance address. When \texttt{ival} is required, bits \([16 − 30]\) are extracted as the first 15 low-order bits of the integer, with bit 30 of the compressed value being replicated throughout the uncompressed operands’ \([16 − 32]\) bit range.

\begin{verbatim}
struct example
{
    int ival_next;
};

struct example
{
    int ival;
    struct example *next;
};

struct example_overflow
{
    int ival;
    struct example *next;
};
\end{verbatim}

(a) Uncompressed structure

\begin{verbatim}
struct example
{
    int ival_next;
};

struct example
{
    int ival;
    struct example *next;
};

struct example_overflow
{
    int ival;
    struct example *next;
};
\end{verbatim}

(b) Structures required for compression

Figure 12: Structure before (a) and after (b) compression

So far, bit 31 and bit 15 of the compressed operand are unused. Bit 15 is actually unused, but bit 31 is used as a flag. If bit 31 of Figure 12(b)’s \texttt{ival_next} is 0, then \texttt{ival_next} stores the compressed values of \texttt{ival} and \texttt{next}, as previously discussed. If bit 31 is 1, however, the remaining 31 bits of \texttt{ival_next} instead stores a pointer to an instance of \texttt{example_overflow} which contains the uncompressed values of \texttt{ival} and \texttt{next}. In the studied architecture, bit 31 of a pointer is known to be 0, so this bit is available for use as a flag.

In this way, occasional exceptions to the compressibility of fields may be tolerated without changing the size of individual instances of the compressed structure. Obviously, if the data is allocated uncompressed elsewhere, the memory usage is greater than the original structure, so it is important to ensure this happens in only relatively few of the allocated instances.

As an aid to understanding, Figure 13(a) shows a simple loop which sums all instances’ \texttt{ivals}. Figure 13(b) shows the functionally equivalent code for the compressed structure. Note that this code is for example only, and uses a minimum of tricks to enable clarity; the transformations in practice would probably look nothing like this.

10.2.3 Instruction set support

When this kind of compression is performed, there are obvious costs associated with checking if compression is possible, compressing and decompressing the data, and allocating more memory if compression cannot be applied. To defray these costs, the authors propose a modest ISA (instruction set architecture) extension to support these kinds of operations.

They propose two new conditional branch instructions. The first checks the high-order 17 bits of two 32-bit values, and branches to a target if they are not the same. The target is intended to be the special case allocation of additional storage, obviously, while the fall-through is for the case where compression may be employed. The second conditional branch similarly jumps to a given location if the most significant 18 bits of a single 32-bit value are not all the same (i.e., all 0 or all 1, as required for data range compression).

The authors then propose four extract and expand instructions, to minimize the decompression costs. The authors do not explicitly mention why they do not also propose similar extensions for performing compression, but obviously the proposal should be modest in its ISA extensions, in
struct example *sp;
struct example_overflow *op;
int i, j, k;
...
while (sp)
{
/*
 * If data was compressed
 */
    if (!(sp->ival_next & 0x80000000))
    {
        j = (sp->ival_next) >> 16;
        if (j & 0x00004000) /* negative # */
            j |= 0xFFFF8000;
        i += j;
        j = ((int) sp) & 0xFFFF8000;
        k = sp->ival_next & 0x00007FFF;
        sp = (struct example *) (j | k);
    }
    /*
    * If data was not compressible
    */
    else
    {
        op = (struct example_overflow *)
            (sp->ival_next & 0x7FFFFFFF);
        i += op->ival;
        sp = op->next;
    }
}

(a) Uncompressed access

(b) Unoptimized compressed access

Figure 13: Looping over structure before (a) and after (b) compression
order to minimize the hardware costs required for efficient support, and reads (extractions) typically occur much more frequently than writes (compressions).

This proposed ISA extension is referred to as *DCX* (data compression extensions).

### 10.2.4 Results

In order to study the benefits of these techniques, the authors utilized the *simplescalar* [?] superscalar processor simulator, and studied a single-processor version of the Olden test suite [?]. The Olden test suite consists of six benchmarks that are highly dependent on dynamically allocated memory. Averaging across all six benchmarks executed with both a large and small problem sizes, dynamic memory usage was reduced by almost 25\% (ranging from 10\% − 33\% across the different benchmarks).

When the DCX instructions were utilized, the better use of the memory hierarchy enabled by compression decreased average execution time by roughly 30\%. Without DCX, the authors still report a 10\% decrease in this simulated execution time.