Techniques for Increasing and Detecting Memory Alignment

Samuel Larsen, Emmett Witchel and Saman Amarasinghe
MIT Laboratory for Computer Science
Cambridge, MA 02139
{slarsen,witchel,saman}@lcs.mit.edu

Abstract

Memory alignment is an important property in memory system performance. Extraction of alignment information at compile-time enables the possibility for new classes of program optimization. In this paper, we present methods for increasing and detecting the alignment of memory references in a program. Our transformations and analyses do not require interprocedural analysis and introduce almost no overhead. As a result, they can be incorporated into real compilation systems.

On average, our techniques are able to achieve a five-fold increase in the number of dynamically aligned memory references. We are then able to detect 94% of these operations. This success is invaluable in providing performance gains in a range of different areas. When alignment information is incorporated into a vectorizing compiler, we can increase the performance of a G4 AltiVec processor by more than a factor of two. Using the same methods, we are able to reduce energy consumption in a data cache by as much as 35%.

1 Introduction

An important focus of past compiler research has been optimizations that target the memory system. Examples include register allocation, loop tiling, array padding, and data prefetching. In the late 1980's, the Bulldog compiler introduced the importance of alignment issues in memory system performance [7]. However, in the years following, little work has been done in this area. One of the major reasons for this is that very few memory references are actually aligned in practice. In the SPEC95fp and MediaBench benchmark suites only 14% of the dynamic accesses are aligned.

In this paper, we introduce a comprehensive set of program transformations to improve memory alignment. These techniques are able to increase alignment to 80% in the SPEC95fp benchmarks and 62% in the MediaBench benchmarks. We also present an algorithm that automatically detects aligned memory references. The availability of quality alignment information at compile-time enables a new class of program optimizations. We demonstrate the need for alignment information in improving the performance of multimedia and clustered architectures, as well as reducing energy consumption in low-power processors.

MIT/LCS Technical Memo, November, 2001.

Ordinarily, the compiler abstracts data memory as a linear array of cells. However, it is often beneficial to view a memory structure as a two-dimensional array of rows and columns. Cache memory, for example, is composed of a collection of cache lines that are loaded and evicted atomically. Here, the cache lines can be viewed as rows and the addressable units within the cache line can be viewed as columns. Based on this row and column paradigm, detecting memory alignment consists of computing the columns accessed by a particular static memory reference. This is most useful when it can be determined that a load or store operation in fact accesses the same column for each dynamic invocation. In this situation, we consider the memory reference to be *aligned*. For aligned references, we associate an *alignment* that specifies which column is accessed.

In a cache memory, the alignment of an aligned reference and the width of the operation specify which bytes within the cache line are accessed. Since the width of a cache line is conventionally a power of two, alignment detection can be regarded as determining the low-order bits of an address at compile-time. As another example, consider the application of memory alignment to a clustered or banked-memory architecture. In a clustered design, memory is divided among a number of distinct banks. Here, it is natural to regard a row as a slice through the banks. With this view, the alignment of a memory reference specifies the bank in which the data are located.

This paper makes the following contributions:

- Introduces a two-dimensional abstraction for data memory that leads to a simple formulation for an alignment detection algorithm.
- Provides an extensive suite of alignment enhancing optimizations.
- Shows that a profile-based system is capable of extracting the necessary global alignment information without reliance on whole-program analysis.
- Demonstrates the effectiveness of obtaining alignment information for two benchmark suites.
- Establishes the importance of alignment information in improving performance and energy consumption.

The remainder of this paper proceeds as follows: In the next section we overview the areas in which alignment information is already being used. In Section 3, we discuss

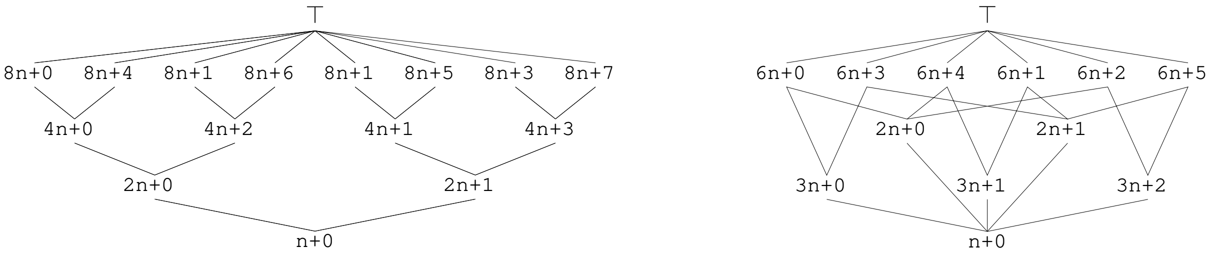

Figure 1: Dataflow lattices for problems with eight and six columns.

the details of the alignment detection algorithm we have developed. Section 4 describes our suite of transformations for increasing the number of aligned memory references in a program. In Section 5, we present results of our techniques and the effect they have on improving performance in real systems. Finally, we outline related work and conclude.

2 Applications of Alignment Detection

Alignment information is central to a wide variety of memory-related compiler optimizations. These range from techniques to increase parallelism, to methods for reducing energy consumption. The following subsections discuss some of the areas in which our compiler system is already being employed.

2.1 Multimedia Compilation

Multimedia instructions are now common among general-purpose microprocessors. These extensions add a set of short SIMD or vector instructions to the ISA in order to exploit the data parallelism available in multimedia applications. One of the key benefits provided by these extensions is the ability to load or store multiple data items using a single wide memory instruction. In order to achieve the best performance, however, these operations must be *naturally aligned*, meaning that a transfer of n bytes must fall on an n-byte boundary.

Architectures such as Motorola's AltiVec are unable to operate on data that are not naturally aligned. If alignment can not be guaranteed, software must explicitly merge data from two consecutive regions. Proper alignment can improve performance by as much as a factor of two, with an average improvement of 20% [1]. Even architectures that are capable of accessing misaligned data can incur a performance penalty. For example, the wide load instructions offered in the Pentium II and Pentium III require six to nine extra cycles if the data cross a cache line boundary [11].

In previous work, we presented a compiler algorithm that automatically extracts SIMD parallelism from sequential programs without using complicated vectorization techniques [13]. We use alignment information to ensure that all wide memory operations fall on a natural boundary. In our approach, alignment information greatly simplifies the parallelization algorithm.

2.2 Compilation for Banked Memory Architectures

Global wire delay will soon become a significant problem for conventional microprocessor designs [2, 10]. Large, centralized structures will limit cycle time, making it difficult to provide performance improvements. To deal with this, future architectures will likely consist of clusters or tiles [15, 16]. Among other things, these architectures replace a centralized memory with a series of independent banks. Compared to a monolithic memory, decentralized memory banks operate with lower latency and can provide higher aggregate bandwidth.

In a clustered design it is typical that each processing unit has fast access to a subset of the memory banks. Data that are close to a processing unit can be accessed quickly, whereas communication to a remote bank is slower. In this situation, it is desirable that computation be placed near its data. Furthermore, memory operations that access different banks are guaranteed to be distinct and can be safely executed in parallel. As mentioned in the previous section, alignment detection is used to determine the bank in which particular data reside.

The prototype compiler for the Raw machine [16] is currently using our analyses to help parallelize sequential applications across clusters of processing units. Alignment techniques will be useful in any design where memory access time is dependent on the distance to a remote bank.

2.3 Compilation for Low-Power

Low-power microprocessors are garnering more attention recently due to the proliferation of mobile computing devices. One way to improve energy consumption is to eliminate tag checks in the data cache. This can have a significant effect on total performance since low-power caches, such as the one found in the StrongARM microprocessor, expend over 50% of their energy in the tag checks [19].

Tag checks can be eliminated when the location of a data item is known beforehand. As discussed, alignment information reveals the cache line location of a memory operation at compile-time. A simple architectural enhancement can use this information to eliminate data cache tag checks [18]. The techniques described in this paper are essential in the resulting reduction of energy consumption.

3 Alignment Detection

We have developed a simple and robust analysis for detecting alignment in programs. The algorithm operates on arbitrary control flow and low-level address calculations. As such, it is not dependent on language or programming style.

The set of locations accessed by a particular static memory operation is represented using a *stride* and *offset*. If we denote the stride as a and the offset as b, then this set is characterized by the linear equation $an + b$, where n is a non-negative integer. Under this scheme, we say that a memory reference is aligned if its stride is equal to the number of columns in the given application. The alignment of

$\sqcap$	$a = \gcd(a_1, a_2, \|b_1 - b_2\|)$ $b = b_1 \bmod a$
$+$	$a = \gcd(a_1, a_2)$ $b = (b_1 + b_2) \bmod a$
-	$a = \gcd(a_1, a_2)$ $b = (b_1 - b_2) \bmod a$
$\times$	$a = \gcd(a_1 a_2, a_1 b_2, a_2 b_1, C)$ $b = b_1 b_2 \bmod a$

Table 1: Transfer functions for alignment detection. The result of operating on any element e and $\top$ is e. Otherwise, new elements are computed using the number of columns, C, and the inputs $a_1 n + b_1$ and $a_2 m + b_2$.

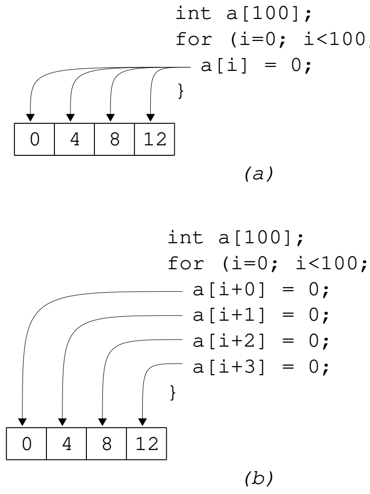

Figure 2: (a) The memory reference in the loop body is unaligned since it accesses consecutive locations. (b) After unrolling, each memory reference accesses only a single location. For simplicity, we will assume the iteration count is always a multiple of the unroll factor.

an aligned reference is given by the offset.

We have implemented alignment detection as an iterative dataflow analysis. For every point in a procedure, we associate each address variable with a linear equation of the form described above. The elements present in the dataflow lattice and the structure of the lattice itself are dictated by the number of columns in the specific problem. Figure 1 shows the lattices for problems with eight and six columns. In order to successfully uncover aligned memory references, the maximum stride represented in the lattice must be equal to the number of columns. The other strides seen in the lattice include all factors of this value. These points are described more carefully in the appendix.

In the lattices for alignment detection, the $\perp$ element usually seen in dataflow analysis is equivalent to the element $n + 0$. This element is used when nothing is known about the value of a variable. In this case, we must assume the variable can take on any value. The $\top$ element has its usual representation. It is associated with variables that have yet to be assigned during iteration over the control-flow graph.

Element values are propagated using the transfer functions listed in Table 1. *Addition*, *subtraction*, and *multiplication* are the operations typically found in address calculations. The *meet* operator is used to merge control flow. Derivations for the transfer functions can be found in the appendix. Any operations not listed in the table result in the element $n+0$. For a constant, D, we assign the element $Cn+d$, where C is the number of columns and $d = D \bmod C$. While this describes values beyond the constant itself, it captures the correct alignment information.

It may be possible to derive transfer functions for other arithmetic or logical operations. However, we have not found any instances in the benchmarks we surveyed where this would be useful. Load instructions appear in address calculations when memory is accessed indirectly, but our analysis does not attempt to track values in memory. As a result, we do not detect alignment for indirect memory references.

4 Alignment-Enhancing Transformations

In order for alignment detection to be useful, aligned references must exist in the program. In Section 5, we present data showing that the vast majority of memory references are in fact unaligned in practice. The primary reason for this is illustrated in the simple loop of Figure 2(a). Here, the array reference within the loop accesses consecutive memory locations on each iteration and is therefore unaligned. This section discusses the transformations we have implemented to greatly increase the number of aligned memory references.

In the next three subsections, we discuss the transformations that form our core approach to increasing alignment. In all of the benchmarks we surveyed, these transformations were universally effective in creating aligned memory references. In addition, these techniques have the advantage of being applied locally and do not require global or whole-program analysis.

4.1 Alignment Conventions

To improve alignment when accessing aggregate data such as arrays and structures, we regulate where these data are placed. This requires that we allocate stack frames in blocks that are a multiple of the number of columns. Alignment of stack-allocated data can depend on the current position of the stack pointer. Aligning stack frames ensures that these data are aligned identically for any invocation of the enclosing procedure.

Since the compiler is responsible for placement of data within the stack, we can force the alignment of local data to an arbitrary value. This is important for alignment detection since the analysis can not determine the alignment of an array reference if the alignment of the base is unknown. The same is true for an access to a structure field. For this reason, we force all aggregate data structures to start on a zero-aligned boundary. When the analysis encounters an immediate value representing the base of an aggregate, it can assume an element value of $Cn + 0$.

This same technique is used for global and heap-allocated data as well. It is simple for the compiler to allocate global data on whatever boundary it chooses. For data allocated from the heap, it was necessary to modify the `malloc` library routine to ensure that it always returns properly aligned pointers.

```
void init(int *x)
{
  int i;
  for (i=0; i<100; i++) {
    if (&x[i] % 16 == 0)
      break;
    x[i] = 0;
  }
  for (; i<100; i += 4) {
    x[i+0] = 0;
    x[i+1] = 0;
    x[i+2] = 0;
    x[i+3] = 0;
  }
}

int main()
{
  int x[200];
  init(&x[0]);
}
```

Figure 3: A pre-loop is used to iterate until a known alignment is reached. This guarantees alignment within the unrolled loop.

4.2 Loop Unrolling

Loop unrolling is used to increase the number of aligned memory references within loop bodies. An example of this is shown in Figure 2(b). When the loop is unrolled by a factor consistent with the number of columns, each new memory reference will access only a single column. In our current approach, we unroll each loop by a factor of C/w where C is the number of columns and w is the width of the largest data type loaded or stored in the loop body.

Since the majority of dynamically executed memory operations are located within loops, loop unrolling is fundamental to ensuring that a large number of memory references are aligned.

4.3 Enforcing Alignment with a Pre-loop

After loop unrolling, we are able to detect the alignment of most accesses to a local or global array. This relies on the fact that the bases of these arrays are guaranteed to be zero-aligned. However, programming languages such as C allow arbitrary pointers into the middle of an array. If an array base is passed as a pointer to a procedure, the alignment of an access derived from this pointer is unknown unless we know the alignment of the pointer upon entry to the procedure.

We can overcome this difficulty using a pre-loop. An example of this is shown in Figure 3. The pre-loop is used to execute a few iterations of the original loop until the memory reference within the loop reaches a known alignment. At this point, we can exit the pre-loop and begin execution of the unrolled version. This has two consequences. First, we guarantee that the memory references within the unrolled loop are aligned. Second, the alignment can be communicated to the alignment detection analysis since the compiler is responsible for choosing the exit condition.

The pre-loop is a simple optimization that allows us to determine the alignments of references derived from pointer arguments. The difficulty is in choosing the exit condition. Our solution is to use a profile-based scheme to observe the run-time alignment behavior. Before we give the details of this solution, we first describe the complexities of choosing the exit condition.

```
void init(int *x)
{
  int i;
  for (i=0; i<100; i+=2) {
    if (&x[i] % 16 == 0)
      break;
    x[i] = 0;
  }
  ...
}

int main()
{
  int x[200];
  init(&x[1]);
  init(&x[0]);
}
```

first call	
i	$\&x[i]\%16$
0	4
1	12
2	4
...	...

second call	
i	$\&x[i]\%16$
0	0
1	8
2	0
...	...

Figure 4: A memory access with non-unit stride complicates the choice for the exit condition. In this example, the exit condition is never satisfied for the first call.

4.3.1 Non-unit Strides

At first glance, it may seem as though the choice for an exit condition is arbitrary. In the example of Figure 3, the pre-loop exits when the alignment of the memory access reaches zero. Regardless of how the array is passed, the pre-loop will execute a small number of iterations until the desired alignment is reached. As long as the total iteration count is sufficiently high, the majority of dynamic memory references will take place in the unrolled loop.

However, an arbitrary choice for the exit condition can lead to a circumstance where the exit condition is never satisfied. Consider a situation in which a memory reference does not have unit stride. An example of this is shown in Figure 4. For the first call to the procedure, an alignment of zero is never observed. This means that all iterations will take place in the pre-loop and we will gain no alignment information. Assuming the iteration count is sufficiently high, the exit condition will be satisfied under the following condition:

Theorem 4.1 *Given a memory reference with stride s, initial access to location x_0, and C columns, there will be an access to column c iff*

$$x_0 \equiv c \pmod{\gcd(s, C)}.$$

The idea is that the memory reference will access exactly the columns $x_0 + sn \pmod{C}$, for integer n. Thus, we are trying to determine if the equation

$$sn \equiv (c - x_0) \pmod{C}$$

has any solutions for n given s, c, x_0 and C. This occurs exactly under the conditions given in the theorem. The proof can be found in any text covering elementary number theory, for example [5].

4.3.2 Calls with Conflicting Alignments

The choice of exit condition becomes more complicated when a pointer parameter is assigned different alignments

```
void copy(int *x, int *y)
{
  int i;
  for (i=0; i<100; i++) {
    if (&x[i] % 16 == 0 &&
        &y[i] % 16 == 0)
      break;
    x[i] = y[i];
  }
  ...
}

int main()
{
  int x[200];
  int y[200];
  copy(&x[0], &y[0]);
  copy(&x[0], &y[1]);
}
```

first call		
i	$x\%16$	$y\%16$
0	0	0
1	4	4
2	8	8
3	12	12
4	0	0
...	...	...

second call		
i	$x\%16$	$y\%16$
0	0	4
1	4	8
2	8	12
3	12	0
4	0	4
...	...	...

Figure 5: A pre-loop with multiple variables. The exit condition can be satisfied for only one of the function calls.

for different invocations of the enclosing procedure. An example of this is also shown in Figure 4. In this situation, it is not possible to find an exit condition that is satisfied for both invocations of the procedure. If the exit condition is set to four, the pre-loop will exit for the first call, but not for the second. If it is left at zero, the reverse is true.

4.3.3 Multiple Variables

The final consideration in constructing the pre-loop exit condition is the inclusion of multiple variables. In real applications, most loops will contain accesses to more than one pointer. A check has to be made for each variable whose alignment we wish to guarantee in the unrolled loop. A simple example of two variables is shown in Figure 5. Here, the discrepancies in alignments among different invocations of the procedure lead to two possible courses of action. First, we could exit from the pre-loop based on the alignment of both variables. This would guarantee their alignment in only one of the function calls. For the other call, the exit condition would never be met. Alternatively, we could exit from the pre-loop contingent on the alignment of $x[i]$ or $y[i]$ alone. With this scheme, we would be able to guarantee the alignment of one of the references for both procedure calls. However, the alignment of the other variable would be unknown.

4.3.4 Choosing the Exit Condition

Now that we have outlined the choices available in constructing the pre-loop exit condition, we now discuss our specific implementation. The goal is to maximize the number of aligned references we are able to detect at compile-time. The best choice depends on the number of memory references in the loop, their initial alignment and stride, and the total iteration count of the loop.

Our solution is to use a profile-based approach. The technique is as follows. Before each inner loop, we insert code to record the initial alignment of every memory reference in the loop. Each set of alignments is then augmented with the total iteration count across all invocations of the loop. After executing the application, this information can be analyzed offline. In the worst case, profiling data could grow unmanageably large. Specifically, in a loop with n memory references, there are C^n possible sets of alignments for a target of C columns. If profiling data become too large, we could choose to store only the most frequently encountered sets of alignments. However, we have not observed this problem in the applications we surveyed. None of our benchmarks require more than a 28 kilobyte text file to store all profiling information.

In order to determine the upper limit on performance we can achieve with profiling, we have implemented an algorithm that chooses the exit condition using an exhaustive search. For each memory reference in the original loop body, we need to decide if it will be included in the pre-loop exit condition. If not, then we can not guarantee its alignment in the unrolled loop. If it is included, we need to choose the alignment against which to compare. The exhaustive search iterates over all possible exit conditions. For a given set of alignments, we can compute the other sets that can be satisfied using Theorem 4.1. Based on the profile data, we can calculate the number of iterations that would be spent in the unrolled loop versus the pre-loop. This number is then multiplied by the number of references that will have guaranteed alignment in the unrolled body. The exit condition with the highest resulting value maximizes alignment information.

The exhaustive search finishes quickly for most of our benchmarks. However, it requires an unreasonable amount of time for two loops in `applu`. Since an exhaustive search requires exponential running time, a heuristic algorithm is needed for a general solution. The simplest approach merely chooses the set of alignments with the highest associated iteration count. Here, all memory references within the loop are included in the exit condition. With this approach, we are able to detect over 97% of the aligned references detected with the exhaustive search. Since this result is near-optimal, we do not explore other heuristics.

A potential shortcoming in any profile-based scheme is that program transformations are based on the results of a single data set. If alignment characteristics vary widely with input data, profiling will not produce good results. In Section 5, we present data showing that our results are highly immune to the particular choice of profile data set. In fact, we achieve nearly identical results regardless of the input data. As such, we believe that profiling is particularly well-suited to the alignment problem.

An alternative to profiling is a completely static approach that employs interprocedural analyses. In fact, we have implemented a version of our analysis that propagates alignment information across call boundaries. However, we strongly believe that whole-program analysis is not practical for real applications. Traditionally, whole-program analyses do not scale to large program sizes. Furthermore, they usually require that the entire source be available. This makes it difficult to use separate compilation or dynamically-linked libraries. Even when whole-program analysis is practical, the presence of pointer aliasing makes it difficult to maintain precise alignment information.

4.4 Secondary Transformations

The transformations described above form the basis of our approach for increasing the amount of aligned memory references. These techniques are generally applicable and provide large improvements for all of our benchmarks. This section covers other techniques that can be used to further increase alignment. The transformations listed here are more specialized, increasing alignment in specific situations.

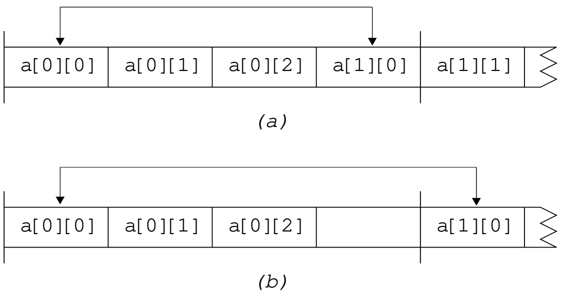

Figure 6: Memory layout of a two-dimensional array with three elements in the low-order dimension. (a) The same index into the low-order dimension is aligned differently for different indexes into the high-order dimension. (b) After padding, they have the same alignment.

4.4.1 Padding Multidimensional Arrays

When accessing multidimensional arrays, it may be useful to pad the array in the lowest dimension. A particular reference into the low-order dimension of the array will be consistently aligned only if the size of the lowest dimension is a multiple of the number of columns. For example, consider the memory layout depicted in Figure 6. Part (a) shows the layout without padding. Here, an index into the low-order dimension is aligned differently for different indexes into the high-order dimension. Part (b) shows the layout after padding. In this case, the alignment is dependent only on the low-order index.

Padding of multidimensional arrays must be handled carefully. In Fortran, common block reshapes can be used to view an array with an arbitrary number of dimensions, regardless of how it is used elsewhere. In C, type casting achieves the same effect. We have implemented an analysis that determines when padding can be applied safely. Since the algorithm must look at all uses of an array, it necessarily requires whole-program analysis. Fortunately, we have discovered that we can obtain the same benefits of padding using the pre-loop transformation. When multiple loop nests are used to iterate over the elements of a multidimensional array, a pre-loop is placed before the innermost loop. This has the effect of realigning the memory references within the inner loop on every iteration of the outer loop. Since we expect the majority of memory references to take place in the unrolled inner loop, most references are aligned. In our experiments, we were able to obtain the same results without padding.

4.4.2 Duplicating Constant Tables

Many multimedia codes contain references to arrays of constants. These tables are often accessed in non-uniform patterns, making their dynamic alignment erratic. Since these arrays are usually small, we can duplicate them for each possible alignment. For any reference to the table, we can arbitrarily choose which copy to access.

This approach is effective, but has some limitations. The first is an increased usage of memory. If the array elements are b bytes, then we will require C/b copies of the array. Also, we should make sure the table is never written. Otherwise, modifications would have to be duplicated for every alignment.

We have implemented a simple transformation that duplicates constant arrays that are below a size of our choosing. The analysis only duplicates arrays that are local to the source file and for which no modifications are performed. In practice, these tables were small enough that the increased memory usage was unnoticeable.

4.4.3 Other Loop Transformations

In situations when an inner loop does not iterate over the low-order dimension of an array, unrolling will not create aligned memory references. Under certain circumstances, it may be possible to rearrange the order of loops in a loop nest such that the innermost loop ranges over the low-order dimension. This transformation is known as loop interchange [3]. While powerful, it is limited in that loops must be perfectly nested. Also, loop-carried dependences may render the transformation unsafe.

Another way to create aligned memory references when loop nests are not conveniently ordered is to unroll outer loops. In [4], Barua et al. developed precise equations for computing the unroll factors of loops in a loop nest. This technique has the advantage that it can create aligned references in the presence of unpadded multidimensional arrays. The only drawback to this approach is a potentially large increase in code size due to excessive unrolling.

In the benchmarks we surveyed, we found limited opportunities for improving alignment using these loop transformations. If future investigations reveal a need, we will add them to our toolchain.

5 Results

In this section, we present the results of our alignment transformations and analyses. All compiler passes were implemented in the SUIF [17] infrastructure. Where possible, results are shown for two benchmark suites: MediaBench [14] and SPEC95fp. For MediaBench, we have excluded `ghostscript` since the complex compilation process for this benchmark is currently stressing our toolchain.

5.1 Effectiveness

We first show the ability of our transformations to increase the number of aligned references and the success of our analysis in detecting them. For both suites, alignment was determined relative to a column width of 32 bytes, which is a typical size for a cache line. In Figure 7, the left bar for each benchmark shows the number of aligned references as a percentage of the total dynamic memory operations. A single dynamic memory reference is considered aligned if its alignment is the same for all other dynamic instances of the same static operation. Results were obtained by instrumenting each benchmark to record the alignment of every memory reference at run-time. The transformed code was converted to C, compiled natively, and then executed. For the SPEC95fp benchmarks, the profile data set was used for profiling and the reference data set was used to gather the numbers shown in the graphs. The MediaBench benchmarks do not have a standard profile data set, so the same input was used for both runs.

The graphs in Figure 7 show alignment before modification (original), and after successive application of each core transformation. These include alignment conventions (conven), inner loop unrolling (unrolling), pre-loop using the simple heuristic (simple), and pre-loop using the exhaustive search (exhaust). For the MediaBench benchmarks, we also include duplication of constant tables (duplicate). This

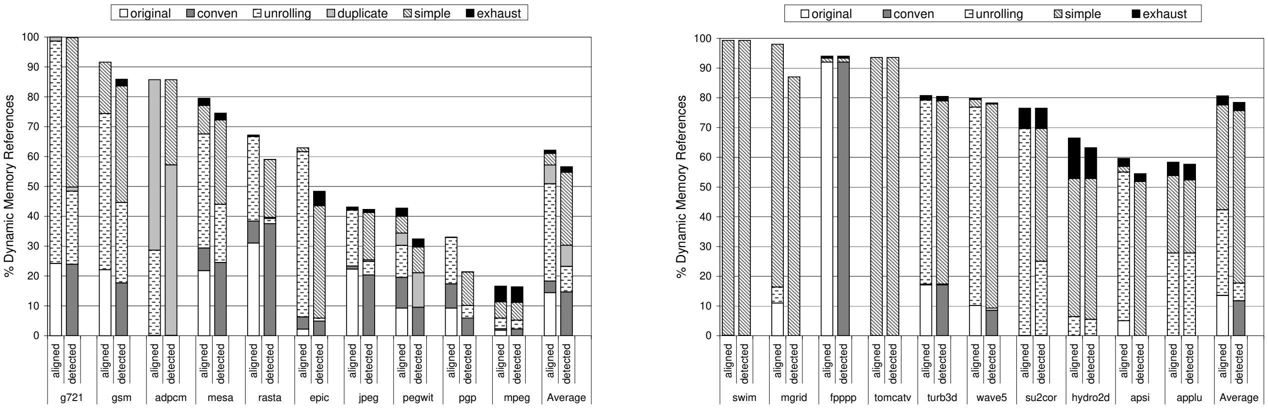

Figure 7: Alignment results for the MediaBench and SPEC95fp benchmark suites. The left bar for each benchmark shows the percentage of aligned memory references. The right bar shows the percentage detected by our analysis. Results are shown after successive application of each alignment-increasing transformation.

transformation is particularly important for `adpcm`. Padding of multidimensional arrays did prove useful for some of the SPEC95fp benchmarks. However, we have not included the effect of this transformation since we were able to obtain the same results using the pre-loop.

As can be seen from the figures, we rely heavily on the alignment-enhancing techniques. Before the transformations are applied, only `fpppp` has a significant percentage of aligned memory references. After the transformations, 62% and 80% of the dynamic memory references are aligned in the MediaBench and SPEC95fp benchmarks, respectively. It is interesting to note the high contribution of the pre-loop transformation in many of the benchmarks. For `applu`, `hydro2d`, and `mgrid`, arguments to key procedures are passed with different alignments for different invocations of the procedure. For `swim` and `tomcatv`, important multidimensional arrays have a size that is inconsistent with the number of columns and would otherwise require padding. Without the pre-loop optimization, we observe very few aligned references.

Figure 7 also presents the percentage of dynamically aligned memory references that were detected by our dataflow analysis. This is shown on the right bar for each benchmark. Without our transformations, we are unable to detect any alignment. At a minimum, alignment conventions are needed to guarantee the position of array and structure bases. After application of every transformation, the analysis is able to detect nearly all of the aligned references. The average percentage detected across each benchmark suite was 57% for MediaBench and 78% for SPEC95fp. Overall, the alignment detection algorithm was able to uncover 94% of the aligned references available in the transformed benchmarks.

Our compiler infrastructure does not include a back-end. Therefore, the results presented in this section do not account for memory operations generated from register spills and parameter passing. Since we already require the alignment of stack frames, any scalar stack accesses will be guaranteed to be aligned. Furthermore, the compiler is responsible for placement of these data within the stack, meaning their alignment is known at compile-time. As a result, the numbers presented here are a conservative estimate of what can be achieved using our techniques.

5.2 Overheads

We next analyze the overheads associated with our transformations. The most worrisome is a possible increase in execution time. This could negatively impact any performance gains we achieve using alignment information. There are two potential sources for an increase in execution time. The first is unrolling, which can negatively impact the performance of the instruction cache by increasing code size. The second is the pre-loop, which introduces runtime alignment checks.

To test the impact of our transformations, we timed the execution of the benchmarks in the SPEC95fp suite after applying each transformation. Benchmarks were converted to C from SUIF, compiled natively with gcc using full optimization, and timed using the Unix *time* command. The MediaBench benchmarks execute too quickly in comparison to the precision offered by the *time* command. Therefore, we were unable to achieve meaningful results for these benchmarks.

Increases in code size and execution time are shown in Table 2. As can be seen, the execution time overheads for both transformations are extremely low. For many of the benchmarks, unrolling actually has a positive effect on the execution time. The only benchmark that shows a noticeable increase in execution time is `wave5`. Unrolling increases execution time by 3.75%. Combined with the pre-loop, execution time is increased by 4.58%.

	Code size		Execution time	
	Unrolling	+ Pre-loop	Unrolling	+ Pre-loop
applu	2.26	2.79	-6.27%	-5.28%
apsi	1.46	1.49	0.93%	1.13%
fpppp	1.67	1.85	0.00%	0.00%
hydro2d	1.42	1.58	0.99%	0.39%
mgrid	1.23	1.31	0.72%	0.72%
su2cor	1.78	1.98	-0.32%	0.11%
swim	1.39	1.49	-0.96%	-0.17%
tomcatv	1.09	1.14	-0.18%	0.65%
turb3d	1.28	1.31	-0.80%	1.72%
wave5	2.05	2.05	3.75%	4.58%

Table 2: Factor increase in code size and percentage increase in execution time due to unrolling and the pre-loop.

Run data	test		train		ref	
Profile data	train	ref	test	ref	test	train
applu	0.02%	0.02%	0.39%	0.11%	0.00%	0.02%
apsi	0.02%	0.00%	8.13%	8.13%	0.00%	0.02%
fpppp	0.07%	0.00%	0.11%	0.11%	0.00%	0.02%
hydro2d	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
mgrid	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
su2cor	0.00%	0.04%	0.00%	0.04%	0.00%	0.00%
swim	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
tomcatv	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
turb3d	0.00%	0.00%	0.44%	0.00%	0.00%	0.00%
wave5	0.00%	0.00%	0.00%	0.00%	0.13%	0.13%

Table 3: Percentage of dynamic aligned references undetected when different data sets are used for profiling and execution.

Another possible overhead is an inflated use of data memory. Specifically, alignment of stack frames could result in an increase in total stack space. To see if this was the case, we monitored the memory usage of each benchmark after successive application of each transformation. On our host platform, memory is allocated by the operating system in pages of 4 kilobytes. Inspection of the maximum number of allocated pages revealed that none of our transformations caused the use of extra pages.

5.3 Profiling Accuracy

Next, we examine the effect of the profile data set on the percentage of aligned memory references detected by the analysis. The SPEC95fp benchmarks are distributed with three data sets. This means there are nine possible pairings of profile and execution data sets. Assuming that alignment results are best when the same data set is used for both runs, we computed the percentage of memory references that were undetected when a different data set was used. The results are shown in Table 3.

As can be seen from the table, there is only one case where a significant number of memory references go undetected. This occurs for `apsi` when we use the *test* or *ref* data sets for profiling, and the *train* data set to gather alignment results. For all other cases, the differences are negligible.

5.4 Application of Alignment

Finally, we evaluate the impact of alignment information in two real systems. The following two subsections describe the importance of alignment techniques in providing energy savings for a low-power architecture, and in providing speed increases for a multimedia processor.

5.4.1 Energy Savings

Alignment information is a key component in a design that has been shown to reduce energy consumption in a low-power architecture [18]. In this approach, architectural extensions are able to use compile-time information to eliminate tag checks in the data cache.

The compiler algorithm attempts to identify pairs of memory references that access the same cache line. If the first dominates the second, it is guaranteed that the second will hit in the cache. In this case, a pair of special memory operations is issued. The first performs a normal load or store, but records the *way* in which the cache line is located. The second can then use this information to access the cache line directly, skipping the expensive tag checks.

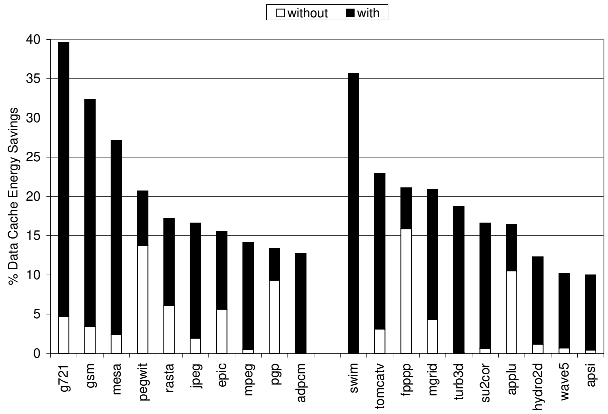

Figure 8: Data cache energy savings for the MediaBench and SPEC95fp benchmarks. Results are shown with and without the use of alignment information.

Tag checks can be eliminated using this mechanism when it can be proven that a pair of memory references access the same location. These will be guaranteed to access the same cache line. In the majority of situations, however, more information is necessary. When two references are separated by a non-zero amount that is less than the width of a cache line, alignment information is required to determine whether or not the references access the same line.

Reduction in energy consumption was computed by instrumenting the benchmarks with calls to a cache simulator. The cache line size was set to 32 bytes. The simulator detected when tag checks were successfully eliminated. Energy consumption for tag-checked and tag-unchecked accesses were computed using a detailed hardware model based on physical layout information [12].

Figure 8 shows the energy savings with and without our alignment techniques. When alignment is unknown, tag checks can only be eliminated when a pair of references is guaranteed to access the same location. It can be seen from the graph that most of the energy savings are a result of alignment information. Data cache energy savings ranged from about 12% to 40% for the MediaBench benchmarks and 10% to 35% for the SPEC95fp benchmarks. In the best case, our analyses are responsible for a 35% reduction in energy consumption.

5.4.2 SIMD Compilation

As discussed in Section 2, the SIMD memory operations available in popular multimedia extensions are more efficient when the data are naturally aligned. To measure the impact of alignment techniques on performance, we targeted a G4 PowerMac workstation running Linux. The G4 microprocessor incorporates the AltiVec multimedia extension which supports 128-bit SIMD operations. AltiVec instructions operate on various datatypes packed into a 128-bit superword. Therefore, the effective vector length depends on the size of the elements.

Ideally, we would like to compare parallelization when alignment is known to parallelization when it is unknown. The SIMD compiler we presented in [13] is completely dependent on alignment information to achieve parallelization. As a result, it has not been optimized to generate efficient code for unaligned loads and stores. In an attempt to isolate the effects of alignment, we have used a commercial vector-

	-O0	-O1	-O2	-O3
float (32-bit)	1.76	1.64	1.45	1.45
int (32-bit)	1.79	1.25	1.35	1.36
short (16-bit)	1.61	2.47	1.97	1.97
char (8-bit)	1.69	2.47	2.21	2.21

Table 4: Speedup of an aligned vector addition operation over the unaligned version for various data sizes and optimization levels.

izer for this study. The VAST compiler [1] can still provide performance gains in the absence of alignment information by producing efficient code for merging two consecutive unaligned regions.

Unfortunately, the mechanism for communicating alignment information to the VAST compiler is limited. Procedures can be designated as *aligned* using command line options or pragmas. This asserts that every memory reference on the first iteration of each loop is aligned on a 128-bit boundary. This narrow channel allows us to communicate only a small fraction of the alignment information we are able to extract. As a result, we limit our study to a vector addition operation. With this simple benchmark, we are able to effectively communicate alignment information since all arrays are identically aligned.

Table 4 shows the speedup obtained when alignment is enforced using a pre-loop. Without the pre-loop, all SIMD memory accesses are unaligned. The VAST compiler generates C code with AltiVec macros inserted where vectorization is successful. This output is then compiled natively using gcc. Since the impact of the pre-loop seems to differ depending on the degree of gcc optimization, we show speedups for several different levels. We have also shown results for different basic data types, since this varies the degree of parallelization.

The vector addition operation is easily vectorized by the VAST compiler and makes full use of the AltiVec execution unit. As a result, the numbers shown in Table 4 represent an upper bound on the performance improvement we can expect from alignment information. Nonetheless, the speedups are considerable. It is clear that alignment techniques are necessary to achieve the best performance from multimedia architectures.

6 Related Work

To the best of our knowledge, Fisher [8] and Ellis [7] were the first to discuss the importance of alignment information. They used loop unrolling as a method for increasing the number of aligned memory references. Their work was done in the context of the Bulldog compiler that targeted a clustered VLIW. In their architecture, main memory was distributed across a set of banks. Alignment was important because local memory accesses had lower latency than remote accesses. In addition, each bank could be accessed in parallel, provided that every cluster accessed a local bank. The use of a pre-loop was also proposed to ensure aligned references in cases where an array base is unknown. However, this transformation was done by hand and apparently only used in simple cases. This research did not propose a mechanism for choosing the exit condition when a loop body contained several references, each with different alignments.

In order to determine which bank was accessed by a particular memory reference, a constraint-based system called *Memory Bank Disambiguation* was used. In order to be successful, this system required the programmer to provide hints about the alignment of certain variables. Comparatively, our alignment detection algorithm requires no programmer intervention in order to detect aligned references.

Barua et al. proposed a more complicated form of loop unrolling [4] to aid in compilation for the Raw machine [16]. The Raw architecture is composed of a mesh of identical tiles, each with a local memory bank. In this design, data access time is a function of the distance to the bank containing the data. Unrolling was used to create memory references that were guaranteed to access a single bank. Precise equations were presented to determine the unroll factors of arbitrary loop nests.

Davidson et al. [6] discussed alignment issues in their work on *Memory Access Coalescing*. This research focused on combining narrow width load and store instructions into wide memory operations. The goal was to provide better memory bus utilization. Since RISC architectures typically require memory operations to be naturally aligned, dynamic checks were inserted to ensure that wide memory operations were aligned properly. In our approach, all alignment information is determined at compile-time.

Equations very similar to our transfer functions appear in a different context called *Bounded Regular Section Analysis* [9]. Researchers at Rice University discussed an efficient method for characterizing the memory locations accessed by a procedure. This information could then be used for dependence checking when parallelizing loops that contain function calls. In this scheme, a range of memory locations was described with a lower bound, upper bound, and stride. This is similar to the stride and offset representation used in our analysis. As a result, the equations for computing a new stride when combining two ranges are the same as the equations used for combining our lattice elements. In Bounded Regular Section Analysis, it is necessary to represent all possible range and stride combinations. Alignment detection requires a small number of stride and offset pairs. Consequently, the lattice describing our analysis is much more concise.

7 Conclusion

Compiler optimizations that target the memory system are important for improving computer performance. The extraction of quality alignment information is one area that can unlock a new class of program optimizations. However, due to a lack of aligned memory references in real programs, little progress has been made since the Bulldog compiler first considered alignment issues over a decade ago.

In this paper we present simple and robust methods for increasing and detecting alignment. Our methods have proven very effective, and we are able to increase the percentage of aligned memory references by a factor of five. We are then able to detect the alignment for over 94% of these operations. Using a novel profiling technique, we can extract the needed global information without using whole-program analysis. Furthermore, our transformations introduce almost no overhead in terms of execution time or memory usage.

The methods discussed in this paper are being used to improve performance in radically different systems. These include the prototype Raw compiler, SIMD compilation, and compilation for energy reduction. When alignment information is used in conjunction with a commercial vectorizer for a G4 AltiVec processor, we can observe a greater than two-fold improvement in execution time. When combined with a design for a low-power data cache, we see a reduction in energy consumption by as much as 35%.

Due to the problems with scaling current architectures, we believe alignment techniques will become even more important in the future. Tomorrow's designs will most likely consist of clusters or tiles, in which memory is divided among independent banks. In this approach, memory access time typically depends on the distance to the referenced bank. Alignment information will be crucial in providing the best performance.

Acknowledgments

We would like to thank the members of the Commit group who helped solidify and improve the ideas presented in this paper. We are especially grateful to Matt Frank, Michael Gordon, and Mark Stephenson.

This research was partially supported by NSF Grant CCR-0073510 and DARPA PAC/C award F3060200-2-0562.

References

[1] VAST-C/AltiVec Product Website. http://www.psrv.com.

[2] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] J. R. Allen and K. Kennedy. Automatic Loop Interchange. In *Procedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246, Montreal, Quebec, June 1984.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the Fifth International Conference on High Performance Computing*, Chennai, India, Dec 1998.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[6] J. W. Davidson and S. Jinturkar. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 186–195, Orlando, FL, June 1994.

[7] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, Massachussetts, 1985.

[8] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.

[9] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[10] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. In *Proceedings of the IEEE*, pages 490–504, Apr 2001.

[11] Intel Corporation. *Intel Architecture Optimization Reference Manual*, 1999.

[12] R. Krashinsky. Microprocessor Energy Characterization and Optimization through Fast, Accurate, and Flexible Simulation. Master's thesis, Massachusetts Institute of Technology, May 2001.

[13] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, USA, Dec 1997.

[15] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 161–171, Vancouver, Canada, 2000.

[16] M. Taylor, J. Kim, J. Miller, F. Ghodrat, B. Greenwald, P. Johnson, W. Lee, A. Ma, N. Shnidman, V. Strumpen, D. Wentzlaff, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Processor - A Scalable 32-bit Fabric for Embedded and General Purpose Computing. In *Proceedings of Hot Chips XIII*, Aug 2001.

[17] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994.

[18] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct Address Caches for Reduced Power Consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec 2001.

[19] M. Zhang and K. Asanović. Highly-Associative Caches for Low-Power Processors. *Kool Chips Workshop, MICRO-33*, 2000.

Appendix

A Transfer Functions

The dataflow elements in alignment detection are linear equations of the form $an + b$, where n is a non-negative integer and a and b are integer constants. These equations describe sets of integer values that can be assigned to a variable at a particular program point. In discussing the transfer functions, we refer to the following two linear equations as input:

$$a_1 n + b_1 \tag{1}$$

$$a_2 m + b_2 \tag{2}$$

Our dataflow lattice also contains the special element, $\top$, which represents a value that has yet to be assigned. The result of operating on $\top$ and any element e is simply e. The transfer functions described below assume neither input is $\top$.

meet

Computing the *meet* of two elements corresponds to finding the union of the sets described by their linear equations. In some cases we will not be able to precisely characterize the resulting set using the form $an + b$. Therefore, the new linear equation may contain values that are not present in the input sets.

To derive the equation for the stride a, we first assume that $b_1 \leq b_2$. If this is true, then b_1 is the smallest value that we need to include in the new set. We therefore start with b_1 as the offset. To this we add multiples of a to produce the other values described in the input sets. In other words, some multiple of a must be equal to all of the following:

$$a_1, 2a_1, 3a_1, ...$$

$$b_2 - b_1, 2a_2 + b_2 - b_1, 3a_2 + b_2 - b_1, ...$$

The *largest* stride that matches this criteria is calculated by finding the *greatest common divisor* of each of these values:

$$a = \gcd(a_1, 2a_1, ..., b_2 - b_1, a_2 + b_2 - b_1, 2a_2 + b_2 - b_1, ...)$$

Using two theorems from number theory,

$$\gcd(x, xy) = x$$

$$\gcd(x, x + y) = \gcd(x, y)$$

we simplify the calculation for the stride to:

$$a = \gcd(a_1, a_2, b_2 - b_1)$$

This computes the stride accurately when $b_1 \leq b_2$. We can combine this result with the situation where $b_2 \leq b_1$ by using the difference between b_1 and b_2:

$$a = \gcd(a_1, a_2, |b_1 - b_2|)$$

Once we have determined the stride, we can now refine the offset. Above, we used b_1 as the offset for the new linear equation. In our representation, b is always be less than a. This can be insured with:

$$b = b_1 \bmod a$$

Using b_2 in place of b_1 gives the same result since b_1 and b_2 are separated by a multiple of a.

addition

Adding linear equations (1) and (2) yields:

$$a_1 n + a_2 m + b_1 + b_2$$

We need to approximate the set described by this equation using an equation of the form $an + b$. The lowest value in the set is $b_1 + b_2$, so we start with this as the offset. The remaining values are produced by adding multiples of a_1 and a_2. The largest number whose multiples equal these values is again given by the gcd:

$$a = \gcd(a_1, a_2)$$

As with the *meet* transfer function, we now guarantee that the offset is less than the stride:

$$b = (b_1 + b_2) \bmod a$$

subtraction

Subtracting (2) from (1) results in:

$$a_1 n - a_2 m + b_1 - b_2$$

Here, we can use the same method we used for addition:

$$a = \gcd(a_1, -a_2)$$

From number theory, this is equivalent to:

$$a = \gcd(a_1, a_2)$$

The offset is given by:

$$b = (b_1 - b_2) \bmod a$$

At first glance, it may appear as though this could result in a negative offset. However, the mod operation is defined as follows:

$$x \bmod y = x - y\lfloor x/y \rfloor$$

This will always result in a positive number when the modulus (a in this case) is positive.

multiplication

Multiplying (1) and (2) gives:

$$a_1 a_2 nm + a_1 b_2 n + a_2 b_1 m + b_1 b_2$$

At this point, the pattern is clear. The strides $a_1 a_2$, $a_1 b_2$, and $a_2 b_1$ are all represented in the set. A single stride that best approximates all three is given by:

$$a = \gcd(a_1 a_2, a_1 b_2, a_2 b_1)$$

Since this could result in a value greater than the number of columns, we saturate the stride by including the number of columns in the calculation:

$$a = \gcd(a_1 a_2, a_1 b_2, a_2 b_1, C)$$

The offset is given by:

$$b = b_1 b_2 \bmod a$$

B Lattice Height

The maximum stride represented in the lattice must be equal to the number of columns. It is clear the lattice must be at least this high in order to distinguish aligned references. However, it is not clear that this is the largest stride needed. At first glance, it may seem that higher lattice elements could provide more precision, allowing us to find more aligned references.

Given an alignment problem with C columns and one of our transfer functions,

$$an + b \leftarrow f(a_1 n + b_1, a_2 n + b_2)$$

we want to know if it is possible to obtain $a \geq C$ when $a2 < C$ and $a_1 > C$, but not when $a_1 = C$. Careful inspection of the transfer functions reveals this can not be the case. Since the stride is always computed using the gcd, the new stride is restrained by the lower of a_1 and a_2. Therefore, we will not capture any more aligned references if we include strides greater than C.

C Lattice Elements

The ultimate goal of alignment detection is to find memory references whose address is characterized by a linear equation with a stride equal to the number of columns, C. Given two dataflow elements where at least one of them has a stride less than C, the only transfer function that can produce an element with a stride equal to C is multiplication. Without multiplication, we would only need to distinguish between two kinds of elements: those with strides of C, and those without strides of C. With multiplication, however, we need to include all strides that are a multiple of C. This can be seen from the stride calculation for the multiplication transfer function. Since the product $a_1 a_2$ is used in the gcd calculation, we can achieve a stride of C when both a_1 and a_2 are factors of C. For this reason, the lattice for a particular application of alignment detection includes all strides that are factors of C.