Increasing and Detecting Memory Address Congruence

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

Abstract

A static memory reference exhibits a unique property when its dynamic memory addresses are congruent with respect to some non-trivial modulus. Extraction of this congruence information at compile-time enables new classes of program optimization. In this paper, we present methods for forcing congruence among the dynamic addresses of a memory reference. We also introduce a compiler algorithm for detecting this property. Our transformations do not require interprocedural analysis and introduce almost no overhead. As a result, they can be incorporated into real compilation systems.

On average, our transformations are able to achieve a five-fold increase in the number of congruent memory operations. We are then able to detect 95% of these references. This success is invaluable in providing performance gains in a variety of areas. When congruence information is incorporated into a vectorizing compiler, we can increase the performance of a G4 AltiVec processor up to 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

Memory address congruence is depicted in Figure 1. Here, consecutive addresses are low-order interleaved across four banks. The shaded cells represent dynamic addresses of a static memory reference. In contrast to the locations in part (a), those in part (b) are congruent with respect to the total bank width. As a result, the reference only accesses a single bank. The importance of this property was first recognized in the early 1980s with the advent of the first VLIW architectures [7, 8]. These designs consisted of clusters of processing units divided among multiple boards. Memory was also clustered, with each cluster gaining fast access to its local bank. In this design, compile-time knowledge of the bank location meant the associated memory operation could be placed on the local cluster, thereby improving performance.

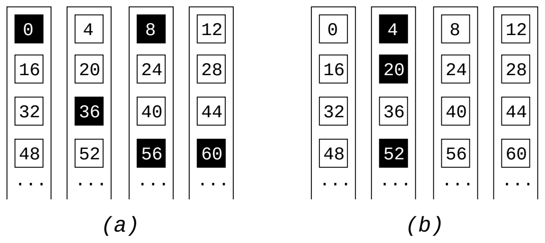


Figure 1. Consecutive words are low-order interleaved across memory banks. (a) A reference whose dynamic instances are not congruent with respect to the bank width. (b) All dynamic references are congruent.

This partitioning of memory operations is only possible when the dynamic addresses generated by a static reference are congruent with respect to the bank width. Unfortunately, this property is rare in programs. Our experiments indicate that only 14% of dynamic accesses exhibit this congruence property. To improve the situation, we have developed a comprehensive set of program transformations that improve memory address congruence. These techniques are able to increase the number of congruent references to 84% in the SPECfp95 benchmarks and 64% in the MediaBench benchmarks. When congruent references are present, we are then able to resolve the bank number, or *offset*, with a success rate of 95%.

In addition to its applicability to banked memory designs, congruence information is central to optimizations that target the cache. When congruence is determined relative to cache line size, the offset and datatype precisely specify which locations are accessed within the line. Since the width of a cache line is conventionally a power of two, congruence detection essentially determines the low-order bits of an address at compile-time.

This paper makes the following contributions:

- Introduces a simple formulation for a congruence detection algorithm.

- Provides an extensive suite of congruence-enhancing optimizations.
- Shows that a profile-based system is capable of extracting the necessary global information without reliance on whole-program analysis.
- Demonstrates the effectiveness of obtaining congruence information for two benchmark suites.
- Establishes the importance of congruence 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 congruence information is already being used. In Section 3 we discuss the details of the congruence detection algorithm we have developed. Section 4 describes our suite of transformations for increasing the number of congruent memory references in a program. In Section 5 we present results of our techniques and the effect they have on improving performance. Finally, we outline related work and conclude.

2. Applications of Congruence Information

Congruence information is central to a variety of memory-related compiler optimizations. These range from techniques for increasing 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 and scientific 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 do not directly support unaligned data accesses. If alignment can not be guaranteed, software must explicitly merge data from two consecutive regions. Under these circumstances, 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]. Recently, this effort has been extended by Shin et al. [16]. One of the main objectives of our approach is to combine multiple sequential memory references into a single wide operation. Since congruence information specifies the cache line locations of memory references, it is used to ensure that the resulting wide operations are naturally aligned.

2.2. Compilation for Banked Memory

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, 18]. 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 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 executed safely in parallel. Congruence information is used to determine which bank is accessed at runtime.

The prototype compiler for the Raw machine [18] is currently using our analyses to help parallelize sequential applications across clusters of processing units. In addition to the obvious performance improvement, maximizing the number of local memory operations seems to provide the instruction scheduler with a good initial seed for partitioning non-memory operations as well [17].

2.3. Compilation for Low-Power

Low-power microprocessors are garnering more attention due to the proliferation of mobile computing devices. One way to decrease 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 [21].

Tag checks can be eliminated when the location of a data item is known beforehand. As discussed, congruence information reveals the cache line location of a memory operation at compile-time. A simple architectural enhance-

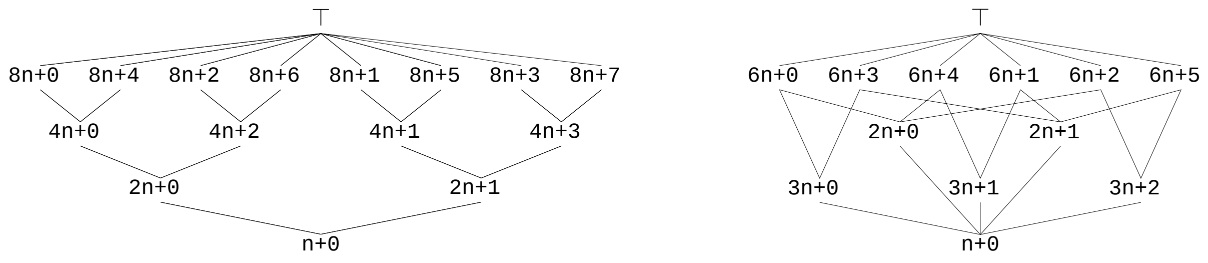

⊤
8n+0 8n+4 8n+2 8n+6 8n+1 8n+5 8n+3 8n+7
4n+0 4n+2 4n+1 4n+3
2n+0 2n+1
n+0
⊤
6n+0 6n+3 6n+4 6n+1 6n+2 6n+5
2n+0 2n+1
3n+0 3n+1 3n+2
n+0

Figure 2. Dataflow lattices for problems with moduli of eight and six.

ment can use this information to eliminate data cache tag checks [20]. The techniques described in this paper are essential in the resulting reduction of energy consumption.

3. Congruence Detection

We have developed a simple and robust analysis for detecting congruence 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 an *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 *resolved* if its stride is equal to the modulus of the given application. For example, an address with a value of $32n + 4$ always accesses the fourth byte in a 32-byte cache line.

We have implemented congruence 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 modulus associated with the specific problem. Figure 2 shows lattices for problems with moduli of eight and six. In order to successfully resolve memory references, the maximum stride represented in the lattice must be equal to the modulus. The other strides seen in the lattice include all factors of this value.

In the lattices for congruence detection, the $\perp$ element typically seen in dataflow analysis is equivalent to $n + 0$. This element is used when nothing is known about the value of a variable. In these cases, 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 generally found in address calculations. The *meet* operator is used to merge control flow. Any operations not listed in the table result in the element $n + 0$. This includes load operations which are present in indirect memory references. For any constant, D, we assign the element $Mn + d$, where M is the modulus and $d = D \bmod C$. While this describes values beyond the constant itself, it captures the appropriate information.

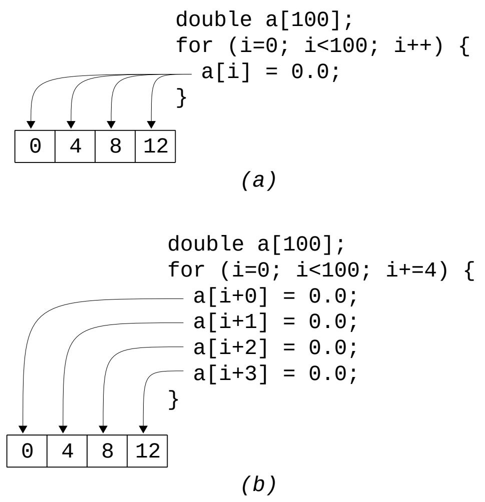
It may be possible to derive transfer functions for other arithmetic or logical operations. However, we have not found any instances in the benchmarks we surveyed where this would be profitable.

4. Improving Memory Address Congruence

In order for detection to be useful, congruent memory references must exist in programs. In Section 5, we present data that show this is not the case. The primary reason for this is illustrated in the simple loop of Figure 3(a). Here, the array reference within the loop accesses consecutive cache line locations on each iteration. This section discusses the

$\sqcap$	$a = \gcd(a_1, a_2, \lvert b_1 - b_2 \rvert)$ $b = b_1 \bmod a$
+	$a = \gcd(a_1, a_2)$ $b = (b_1 + b_2) \bmod a$
-	$a = \gcd(a_1, a_2)$ $b = (b_1 - b_2) \bmod a$
$\times$	$a = \gcd(a_1 a_2, a_1 b_2, a_2 b_1, M)$ $b = b_1 b_2 \bmod a$

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



```
double a[100];
for (i=0; i<100; i++) {
  a[i] = 0.0;
}
```

```
double a[100];
for (i=0; i<100; i+=4) {
  a[i+0] = 0.0;
  a[i+1] = 0.0;
  a[i+2] = 0.0;
  a[i+3] = 0.0;
}
```

Figure 3. (a) The memory reference addresses consecutive cache line locations. (b) After unrolling, each reference accesses only a single offset.

transformations we have implemented to increase the number of congruent memory references.

The next three subsections discuss the transformations that form our core approach to increasing congruence. In all of the benchmarks we surveyed, these transformations were universally effective in creating congruent memory references. In addition, these techniques do not require global or whole-program analysis.

4.1. Congruence Conventions

To improve congruence 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 modulus. In the absence of this constraint, the offset of stack-allocated data can vary depending on the depth of the call stack. Intelligent stack allocation forces local data to have the same offset for every call to the enclosing procedure. RISC compilers already perform this type of padding in order to ensure that all basic data types are aligned on a natural boundary.

The compiler is responsible for placement of data within the stack. As such, we can force the offset of local data to an arbitrary value. This is important for congruence detection since the analysis can not determine the offset of an array reference if the offset 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 offset. When the detection analysis encounters an immediate value representing the base of an aggregate, it can assume an element value of $Mn + 0$.

The same discipline is used for global and heap-allocated data. It is simple for the compiler to allocate global data on whatever boundary it chooses. For data allocated from the heap, it is necessary to modify the `malloc` library routine to ensure that it always returns pointers with zero offset.

4.2. Loop Unrolling

Since the majority of dynamically executed instructions are located within loops, it is crucial that most memory operations within inner loops exhibit the congruence property. The Bulldog compiler [7] was the first to use loop unrolling as a means towards this end. An example of this is shown in Figure 3(b). When the loop is unrolled by a factor consistent with the modulus, each memory reference in the unrolled body accesses a single offset. We unroll each loop by a factor of M/w where M is the modulus and w is the width of the smallest datatype loaded or stored in the loop body. When the iteration count is unknown or is not a multiple of the unroll factor, a post-loop is required to execute the final iterations. For simplicity of illustration, the post-loop is omitted from our examples.

4.3. Enforcing Congruence with a Pre-loop

After loop unrolling, we are able to resolve the offset of most accesses to a local or global array. This relies on the fact that the bases of these arrays are placed at zero offsets. 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 offset of an access derived from the pointer is unknown.

We can overcome this difficulty using a pre-loop. An example of this is shown in Figure 4. The pre-loop is used to execute a few iterations of the original loop until the memory reference within the loop reaches a known offset. At this point, we 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 exhibit the congruence property. Second, the offset can be communicated to the congruence detection analysis since the compiler is responsible for choosing the exit condition.

The pre-loop was first proposed in [7]. It was discussed in simple cases such as the one shown in Figure 4. However, as we will demonstrate below, effective pre-loop construction is complicated for realistic applications. Our solution employs a profile-based scheme to observe memory addresses at runtime. We can then analyze these data offline to construct an efficient pre-loop.

```
void init(double *x)
{
  int i;
  for (i=0; i<100; i++) {
    if ((int)&x[i] % 32 == 0)
      break;
    x[i] = 0.0;
  }
  for (; i<100; i += 4) {
    x[i+0] = 0.0;
    x[i+1] = 0.0;
    x[i+2] = 0.0;
    x[i+3] = 0.0;
  }
}

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

Figure 4. A pre-loop is used to iterate until a known offset is reached. This guarantees congruence within the unrolled loop.

```
void init(double *x)
{
  int i;
  for (i=0; i<100; i+=2) {
    if ((int)&x[i] % 32 == 0)
      break;
    x[i] = 0.0;
  }
  ...
}

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

first call	
i	$\&x[i]\%32$
0	8
2	24
4	8
...	...

second call	
i	$\&x[i]\%32$
0	0
2	16
4	0
...	...

Figure 5. A memory access with non-unit stride complicates the choice of 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 of exit condition is arbitrary. In the example of Figure 4, the pre-loop exits when the offset 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 value is reached. As long as the total iteration count is sufficiently high, the majority of dynamic memory references take place in the unrolled loop.

However, an arbitrary choice for the exit condition can lead to a situation in which the exit condition is never satisfied. Consider the example shown in Figure 5 where the index variable has non-unit stride. For the first call to the procedure, an offset of zero is never reached. This means that all iterations take place in the pre-loop and we gain no useful congruence information.

Assuming the iteration count is sufficiently high, the exit condition is satisfied under the following condition:

Theorem 4.1 *Given a memory reference with stride s, initial access to location x_0, and modulus M, there will be an access to offset b iff*

$$x_0 \equiv b \pmod{\gcd(s, M)}.$$

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

$$sn \equiv (b - x_0) \pmod{M}$$

has any solutions for n given s, b, x_0 and M. 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 Offsets

The choice of exit condition becomes more complicated when a pointer parameter is assigned different offsets for different invocations of the enclosing procedure. An example of this is also shown in Figure 5. 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 eight, 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 contain accesses to multiple pointer variables. A check has to be made for each variable whose congruence we wish to guarantee in the unrolled loop. A simple example of two variables is shown in Figure 6. Here, the discrepancies in offsets between different invocations of the procedure lead to two possible courses of action. First, we could exit from the pre-loop based on the offset of both variables. This guarantees congruence in only one of the procedure calls. For the other call, the exit condition is never be met. Alternatively, we could exit from the pre-loop contingent on the offset of $x[i]$ or $y[i]$ alone. With this scheme, we guarantee congruence for only one of the references.

```
void copy(double *x, double *y)
{
  int i;
  for (i=0; i<100; i++) {
    if ((int)&x[i] % 32 == 0 &&
        (int)&y[i] % 32 == 0)
      break;
    x[i] = y[i];
  }
  ...
}

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

first call		
i	$x\%32$	$y\%32$
0	0	0
1	8	8
2	16	16
3	24	24
4	0	0
...	...	...

second call		
i	$x\%32$	$y\%32$
0	0	8
1	8	16
2	16	24
3	24	0
4	0	8
...	...	...

Figure 6. A pre-loop with multiple variables. The exit condition can be satisfied for only one of the procedure calls.

4.3.4. Choosing the Exit Condition

Having outlined some of the intracies of pre-loop construction, we now discuss our specific implementation. The goal is to maximize the number of congruent references executed at run-time. The best choice depends on the number of memory references in the loop, their initial offset and stride, and the total number of loop iterations. The major difficulty in solving this optimization problem is the presence of multiple variables, each having a different offset for different procedure invocations.

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 offset of every memory reference in the loop. Each set of offsets 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 M^n possible sets of offsets for a modulus of M. If profiling data become too large, we could choose to store only the most frequently encountered sets of offsets. However, we have not observed this problem in the applications we surveyed. All of our benchmarks require less than 28 kilobytes of text 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 congruence in the unrolled loop. If it is included, we need to choose the offset against which to compare. The exhaustive search iterates over all possible exit conditions. For a given set of offsets, we compute the other sets that are satisfied using Theorem 4.1. Based on the profile data, we 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 having guaranteed congruence in the unrolled body. The exit condition with the highest resulting value maximizes congruence 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 offsets 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 congruent references detected with the exhaustive search. Since this 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 congruence 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 congruence problem.

An alternative to profiling is a completely static approach that employs interprocedural analysis. We have implemented a version of our algorithm that propagates congruence 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 the entire source to 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 congruence information.

4.4. Secondary Transformations

The transformations described above form the basis of our approach for increasing the number of congruent 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 our effectiveness. The transformations listed here are more specialized, increasing congruence in specific situations.

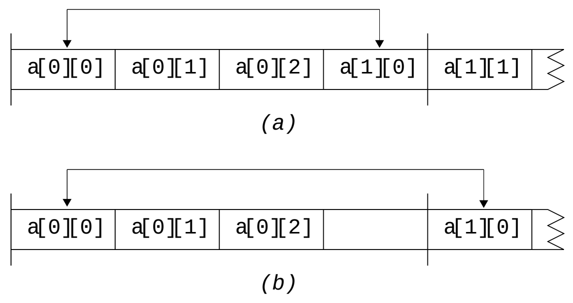


Figure 7. Memory layout of a two-dimensional array with three elements in the low-order dimension. (a) The same index into the low-order dimension has a different offset for different indexes into the high-order dimension. (b) After padding, they have the same offset.

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 have the same offset only if the size of the lowest dimension is a multiple of the modulus. For example, consider the memory layout depicted in Figure 7. Part (a) shows the layout without padding. Here, an index into the low-order dimension has different offsets for different indexes into the high-order dimension. Part (b) shows the layout after padding. In this case, the offset 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 resetting the offset of memory references within the inner loop on every iteration of the outer loop. Since the majority of memory references take place in the inner loop, most references are resolved to a single offset. 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 runtime addresses unpredictable. Since these arrays are usually small, we can duplicate them for each possible offset. 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 use of memory. If the array elements are b bytes, then we require M/b copies of the array. Also, we must ensure that the table is never written. Otherwise, modifications would have to be duplicated for every offset.

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

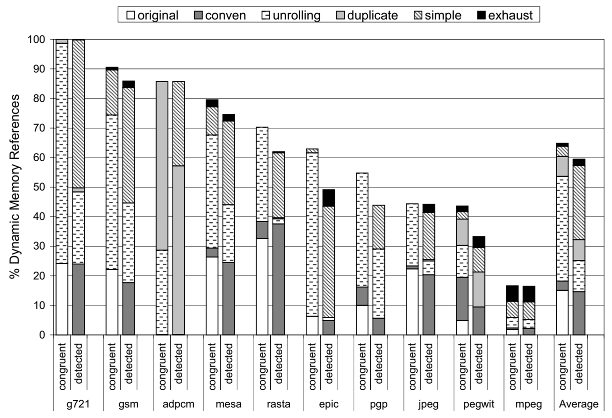
For situations in which an inner loop does not iterate over the low-order dimension of an array, unrolling will not create congruent memory references. Under certain circumstances, it may be possible to rearrange the order of loops in a loop nest such that the innermost loop ranges over the low-order dimension. This transformation is known as loop interchange [3]. While powerful, it has implications in areas such as cache behavior that can seriously impact performance. Also, loop-carried dependences may render the transformation unsafe.

Another way to create congruent memory references when loop nests are not conveniently ordered is to unroll outer loops. Barua et al. [4] developed precise equations for computing the unroll factors of loops in a loop nest. This technique has the advantage that it can create congruent 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 congruence 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 congruence transformations and analyses. All compiler passes were implemented in the SUIF infrastructure [19]. Where possible, results are shown for two benchmark suites: MediaBench [14] and SPECfp95. For MediaBench, we have excluded `ghostscript` since our toolchain does not handle its complex compilation process.



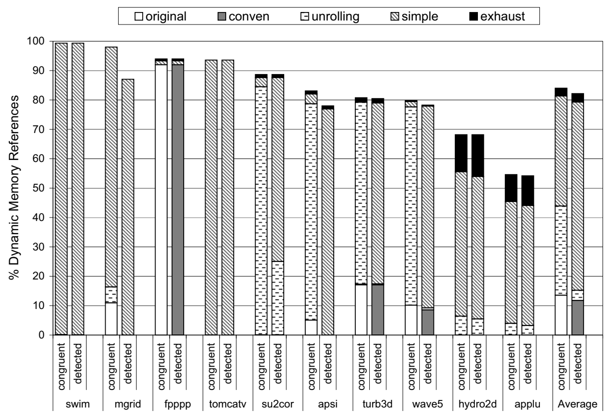


Figure 8. Congruence results for the MediaBench and SPECfp95 benchmark suites with a modulus of 32 bytes. The left bar for each benchmark shows the percentage of congruent memory references. The right bar shows the percentage detected by our analysis. Results are shown after successive application of each congruence-increasing transformation.

5.1. Effectiveness

We first show the ability of our transformations to increase the number of congruent references and the success of our analysis in detecting them. For both suites, congruence was determined relative to 32 bytes. This is a typical cache line size, making it the likely candidate for the applications discussed in Section 2. Since our analysis and transformations depend on the particular modulus, all benchmarks are compiled specifically for a 32-byte cache line size.

In Figure 8, the left bar for each benchmark shows the number of congruent references as a percentage of the total dynamic memory operations. A single dynamic memory reference is considered congruent if its offset is the same as all other dynamic instances of the same static instruction. For example, a static load that accesses the same offset twice is counted as two congruent accesses. A static load that accesses the same offset nine times and a different offset once is counted as ten non-congruent accesses. We report results in this fashion since we consider an operation congruent only when every dynamic instance accesses the same offset.

Results were obtained by instrumenting each benchmark to record the offset of every memory reference at run-time. The transformed code was converted to C, compiled natively, and then executed. For the SPECfp95 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 8 show congruence before modification (original), and after successive application of each core transformation. These include congruence 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 transformation is particularly important for `adpcm`. Padding of multidimensional arrays did prove useful for some of the SPECfp95 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 seen from the figures, we rely heavily on the congruence-enhancing techniques. Before the transformations are applied, only `fpppp` has a significant percentage of congruent memory references. After the transformations, 64% and 84% of the dynamic memory references exhibit congruence in the MediaBench and SPECfp95 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 offsets for different invocations of the procedure. For `swim` and `tomcatv`, important multidimensional arrays have a size that is inconsistent with the modulus and would otherwise require padding. Even in cases where the pre-loop is not needed to generate congruent accesses, it is usually required for detection. This is because most array accesses are based on parameters whose offset we can not guarantee without whole-program analysis.

Figure 8 also presents the percentage of dynamically congruent memory references that are detected by our dataflow analysis. This is shown on the right bar for each benchmark. Without our transformations, we are unable to detect any congruence. At a minimum, congruence conventions are needed to guarantee the position of array and

	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 pre-loop.

Run data	test		train		ref	
Profile data	train	ref	test	ref	test	train
applu	0.02%	0.02%	0.35%	0.04%	0.00%	0.00%
apsi	11.48%	11.48%	6.53%	6.53%	0.00%	0.00%
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%	0.02%	0.00%	0.02%	0.01%	0.01%
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%	0.00%	0.00%	0.00%	0.00%
wave5	0.00%	1.21%	0.00%	1.19%	0.99%	0.99%

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

structure bases. After application of every transformation, the analysis is able to detect nearly all of the congruent references. The average percentage detected across each benchmark suite was 60% for MediaBench and 82% for SPECfp95. Overall, the congruence detection algorithm was able to uncover 95% of the congruent 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 padded stack frames, any scalar stack accesses are guaranteed to exhibit congruence. Furthermore, the compiler is responsible for placement of these data within the stack, meaning their offset is known at compile-time. As a result, the numbers presented here are a conservative estimate of what can be achieved.

5.2. Overheads

Next, we analyze the overheads associated with our transformations. The most worrisome is a possible increase in execution time. This could nullify any performance gains achieved using congruence information. There are two potential sources for such an increase. The first is unrolling, which can negatively impact instruction cache performance through an increase in code size. The second is the pre-loop, which introduces runtime checks.

To test the impact of our transformations, we timed the execution of the benchmarks in the SPECfp95 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, prohibiting us from gathering meaningful results for these benchmarks.

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

Another possible overhead is an inflated use of data memory due to stack frame padding. To see if this occurs, we monitored the memory usage of each benchmark with and without congruence conventions. On our host platform, memory is allocated by the operating system in pages of 4 kilobytes. In no cases did the use of conventions require extra pages.

5.3. Profiling Accuracy

Next, we examine the effect of the profile data set on the percentage of congruent memory references detected by the analysis. The SPECfp95 benchmarks are distributed with three data sets, providing nine possible pairings of profile and execution data sets. Since results are best when the same data set is used for both runs, we computed the difference in detected congruent references when a different data set is used. The percentages are shown in Table 3.

As seen from the table, `apsi` is the only benchmark in which a noticeable number of memory references go undetected when a different data set is used. This occurs when the *test* or *ref* data sets are used to gather final results. In all other cases, the differences are negligible. This suggests that profiling works exceptionally well for the congruence problem. In contrast, when profiling is used in the traditional application of branch prediction, results can vary greatly for different profile data sets [9].

5.4. Scalability

Figure 9 shows the percentage of dynamic congruent references observed and detected over a range of moduli. For both benchmark suites, congruence information degrades

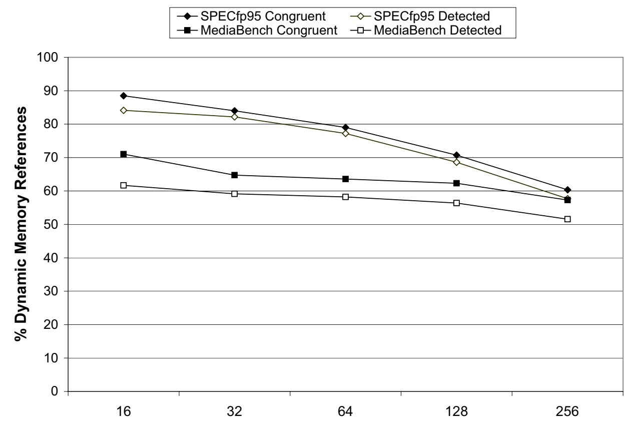


Figure 9. Percentage of dynamic congruent references present and detected as the modulus is increased.

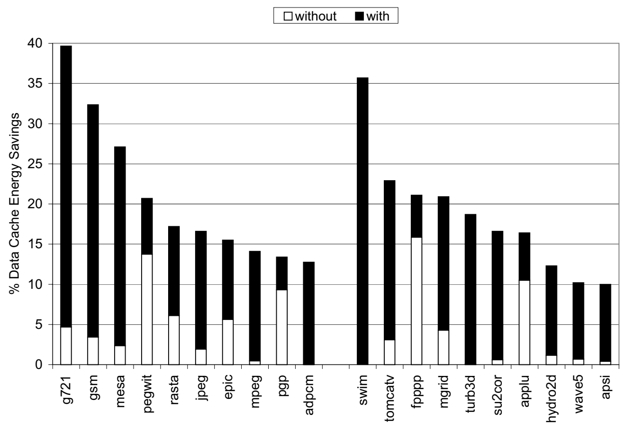


Figure 10. Data cache energy savings for the MediaBench and SPECfp95 benchmarks. Results are shown with and without the use of congruence information.

gracefully as the modulus is increased. The downward trend can be attributed to two phenomena. First, larger unroll factors are required to guarantee congruence with respect to a larger modulus. At some point, the unroll factor becomes larger than the iteration count, causing all iterations to execute in the post-loop. More importantly, a larger modulus allows for a greater number of possible offsets. This makes discovery of a good pre-loop more difficult. In both cases, fewer total iterations take place in the unrolled body.

5.5. Application of Congruence Information

Finally, we evaluate the impact of congruence information on two real systems. The following two subsections describe the importance of congruence techniques in providing energy savings on a low-power architecture, and in providing performance improvements on a multimedia processor.

5.5.1. Energy Savings

Congruence information is a key component in a design that has been shown to reduce energy consumption in a low-power architecture [20]. In this approach, architectural extensions use compile-time information to eliminate tag checks in the data cache.

The compiler 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, two special memory operations are issued. The first performs a normal load or store, but records the *way* in which the cache line is located. The second uses this information to access the cache line directly, skipping the expensive tag check.

Tag checks can be eliminated using this mechanism when it can be proven that a pair of memory references access the same location. These are 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, congruence 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 10 shows the energy savings with and without our techniques. When the offset is unknown, tag checks can be eliminated only when a pair of references is guaranteed to access the same location. It is seen from the graph that most of the energy savings are a result of congruence information. Data cache energy savings ranged from about 12% to 40% for the MediaBench benchmarks and 10% to 35% for the SPECfp95 benchmarks. In the best case, our analyses are responsible for a 35% reduction in energy consumption.

5.5.2. SIMD Compilation

As discussed in Section 2, the SIMD memory operations available in popular multimedia extensions are more efficient when data are naturally aligned. To measure the impact of our techniques on performance, we targeted a G4 PowerMac workstation running Linux. The G4 microprocessor incorporates the AltiVec multimedia extension which

datatype	vector length	speedup (unaligned)	speedup (aligned)	improvement
float	4	3.25	4.75	46%
int	4	2.15	2.93	36%
short	8	2.98	5.87	97%
char	16	5.21	11.53	121%

Table 4. Speedup of a vector addition operation after vectorization. Results are compared for data that are unaligned and aligned.

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 in the presence of congruence information to parallelization when it is absent. The SIMD compiler we presented in [13] is completely dependent on congruence information to achieve parallelization. As a result, it is impossible to isolate its contribution to the speedups we observe. Instead, we have chosen to use a commercial vectorizer for this study. The VAST compiler [1] can still provide performance gains in the absence of congruence information by producing efficient code for merging two consecutive unaligned regions.

Unfortunately, the mechanism for communicating congruence information to the VAST compiler is limited. Procedures can be designated as *aligned* using command line arguments or pragmas. This asserts that every memory reference on the first iteration of a loop will have a zero offset. This narrow channel allows us to communicate only a small fraction of the information we are able to extract. As a result, we are forced to limit our study to a vector addition operation. This kernel has simple congruence characteristics which allow us to effectively communicate the information via pragmas.

Table 4 shows the speedup obtained with different vector lengths. Results are shown when data are unaligned and when natural alignment is enforced using a pre-loop. The VAST compiler generates C code with AltiVec macros inserted where vectorization is successful. This output is then compiled natively using gcc. We compiled with full optimization in both stages.

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 congruence information. Nonetheless, the speedups are considerable. It is clear that congruence 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 congruence information. They used loop unrolling as a method for increasing the number of congruent 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. Bank location 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 congruent references in cases where an array base was 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 automatically, particularly when a loop body contains several references, each with different initial offsets.

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 offset of certain variables. Comparatively, our congruence detection algorithm requires no programmer intervention in order to resolve congruent references.

Barua et al. proposed a more complicated form of loop unrolling [4] to aid in compilation for the Raw machine [18]. 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 are 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 information is determined by the compiler.

7. Conclusion

A static memory reference is considered *congruent* if its dynamic memory addresses are all congruent with respect to some non-trivial modulus, typically the size of a cache line. In this paper, we presented practical methods for both improving and detecting address congruence. With almost no overhead, these techniques increase the percentage of congruent memory references by a factor of five. Our congruence detection algorithm is then able to resolve the offset for 95% of these operations.

At the heart of our approach is a profiling technique that allows us to construct highly optimized pre-loops for the complex loops found in real applications. Imperical results suggest that profiling is particularly effective since congruence information varies little among data sets.

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 congruence information is used in conjunction with a commercial vectorizer for a G4 AltiVec processor, it is possible to improve performance by as much as a factor of two. When combined with a design for a low-power data cache, our techniques are responsible for reducing energy consumption by up to 35%.

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 grants CCR-0073510 and EIA-0071841, DARPA PCA award F29601-01-2-0166, 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 *Proceedings 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, MA, 1985.

[8] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proc. 10th ISCA*, pages 140–150. Computer Society Press, 1983.

[9] J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Oct 1992.

[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, Vancouver, BC, 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] J. Shin, J. Chame, and M. W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architecture. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, Sep 2002.

[17] S. M. Swenson. Spatial Instruction Scheduling for Raw Machines. Master's thesis, Massachusetts Institute of Technology, Feb 2002.

[18] 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.

[19] 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.

[20] 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*, pages 124–133, Austin, TX, Dec 2001.

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