Strength Reduction of Integer Division and Modulo Operations

by

Jeffrey W. Sheldon

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ....
Saman Amarasinghe
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Strength Reduction of Integer Division and Modulo Operations

by

Jeffrey W. Sheldon

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Integer modulo and division operations are frequently useful and expressive constructs both in hand written code and in compile generated transformations. Hardware implementations of these operations, however, are slow, and, as time passes, the relative penalty to perform a division increases when compared to other operations. This thesis presents a variety of strength reduction like techniques which often can reduce or eliminate the use of modulo and division operations. These transformations analyze the values taken on by modulo and divisions operations within the scope of for-loops and exploit repeating patterns across iterations to provide performance gains. Such patterns frequently occur as the result of data transformations such as those performed by parallelizing compilers. These optimizations can lead to considerable speedups especially in cases where divisions account for a sizeable fraction of the runtime.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

This work is based upon previous work [7] done by Saman Amarasinghe, Walter Lee, and Ben Greenwald and it would not have been possible to complete this without this preceding foundation.

Contents

List of Figures

Chapter 1

Introduction

Compilers frequently employ a class of optimizations referred to as strength reductions to transform the computation of some repetitive expression to a form which is more efficient to compute. The presence of these transformations frequently allow programmers to expression a computation in a more natural, albeit more expensive, form without incurring any additional overhead in the resulting executable.

Classical strength reductions have focused on such tasks as transforming stand alone operations, such as multiplications by powers of two, into lighter bitwise operations or transforming operations involving loop indexes such as converting repeated multiplications by a loop index to an addition based on the previous iteration's value[1]. Compilers have long supported optimizing modulo and division operations into bitwise operations when working with known power of two denominators. They are also capable of introducing more complicated expressions of bitshifts and subtractions to handle non power of two, but still constant, denominators. None of these existing techniques, however, are useful when dealing with arbitrary denominators. Unlike previous techniques which have considered division operations in isolation, we feel that analyzing modulo and division operations within the context of loops will often make it possible to eliminate both modulo and division operations in much the same way multiplications are typically removed from loops without requiring a known denominator.

With the knowledge that a strength reduction pass will occur during compilation, programmers are free to represent the intended computation at a higher level. This, naturally, can lead to greater readability and maintainability of the code. Leaving the original code expressed at a higher level also permits other compiler optimizations to be more effective. For instance classic strength reductions would allow an expression which is four times a loop index could be strength reduced to be a simple addition each time through the loop near the end of the compilation phase. This delays the introduction of another loop carried dependency which can complicate analysis.

1.1 Motivation

Both modulo and division operations are extremely costly operations to implement. Unfortunately hardware implementations of division units are notoriously slow, and, in smaller processor cores, are sometimes omitted entirely. For instance the Intel P6 core requires 12–36 cycles to perform an integer divide, while only 4 cycles to perform a multiply [6]. To make matters worse the iterative nature of the typical hardware implementation of division prevents the division unit from being pipelined. Thus while the multiplication unit is capable of a throughput of one multiply per cycle the division unit still remains only capable of executing one divide every 12–36 cycles. In the MIPS product line from the R3000 processor to the R10000 the ration of costs of div/mul/add instructions has changed from 35/12/1 to 35/6/1, strengthening the need to convert divisions into cheaper instructions. On the Alpha the performance difference is even more dramatic since integer based divides are performed using software emulation.

Such a large disparity of cost between multiplications and divisions is result of a long running trend in which operations such as multiplications benefit from massive speedups while division times have lagged behind. For instance increases of chip area have allowed for parallel multipliers but have not led to similar advances for division units. While it is possible to create a faster and pipelineable division unit, doing so is still very costly and not space efficient for most applications. The performance difference between multiplication and division has led to some work to precompute the reciprocal of dividends and use floating point multipliers to perform integer divisions [2].

The present performance variation of operations is very similar to the weighty cost multiplications used to represent in comparison to additions. The unevenness of performance between multiplication and division led to widespread use of strength reduction techniques to replace multiplications with additions and bitwise shift operations. Such widespread use of strength reductions not only benefitted program performance, but in time benefitted software readability. The reductions that these passes dealt with could generally be performed by the programmer during development, but doing so is error prone and costly both in terms of legibility of the underlying algorithm and development time. Over fewer and fewer strength reductions were performed by hand pushing such work down to a compiler pass.

Today's performance characteristics of modulo and division operations parallels the former performance of multiplication and would likely benefit in many of the same ways as did multiplication. Much like programs before strength reduction, most of today's programs typically do not contain a number of instances of modulo or division operations which can be eliminated. This would be expected as any such elimination would typically performed by hand, but such instances are not as common as uses of multiplications which can be eliminated. The initial value of tackling the problem of reducing modulo and division operations comes not from removing those originally present in the program, but from reducing away those that get introduced in the process of performing other compiler optimizations. There are many sorts of compiler optimizations which are apt to introduce modulo and division operations typically in the process of performing data transformations.

For example the SUIF parallelizing compiler [10] frequently transforms the layout of arrays in memory. In order to minimize the number of cache misses from processors competing for data, it reshapes the layout of data to provide each processor with a contiguous chunk of memory. These data transformations typically introduce both modulo and division operations into the array indexes of all accesses of the array. Unfortunately the introduction of these operations incurs such a performance penalty so

Before Transform:

```
for t ← 0 to T
    do for i ← 0 to N^2 − 1
            do A[i%N] ← 0
```

After Transform:

```
for t ← 0 to T
    do for ii ← 0 to N − 1
            do modVal ← 0
                for i ← ii ∗ N to min(ii ∗ N + N − 1, N^2 − 1)
                    do A[modVal] ← 0
                        modVal ← modVal + 1
```

Figure 1-1: Example of optimizing a loop with a modulo expression

as to result in poorer overall performance than before the data transformation. Such a high cost forced the SUIF parallelizing compiler to perform these data transformations without using any modulo or divisions, substantially increasing the complexity of the transformation.

By transforming a simple for loop using a modulo operation as shown in Figure 1-1, one can realize speed gains of 5 to 10 times the performance of the original code. When tested on an Alpha which does not contain a hardware division unit, the speed up was 45 times. A similar transformation using a division rather than a modulo is shown in Figure 1-2. A graph showing the speedups that result from removing the modulo or division operations on different architectures is shown in Figure 1-3. These performance gains highlight the cost of performing modulo and division operations. In real world codes the performance increases will be somewhat less impressive as the percentage of time spent performing divisions is relatively lower.

Similar array transformations exist in the Maps compiler-managed memory system [4], the Hot Pages software caching system [8], and the C-CHARM memory system [5]. Creating systems such as these would be simplified if a generalized system of strength reductions for integer modulo and division operations were available and could be used as a post pass to these other optimizations.

Before Transform:

```
for t ← 0 to T
    do for i ← 0 to N^2 − 1
            do A[i/N] = 0
```

After Transform:

```
for t ← 0 to T
    do divVal ← 0
        for ii ← 0 to N − 1
            do for i ← ii ∗ N to min(ii ∗ N + N − 1, N^2 − 1)
                    do A[divVal] ← 0
                divVal ← divVal + 1
```

Figure 1-2: Example of optimizing a loop with a division expression

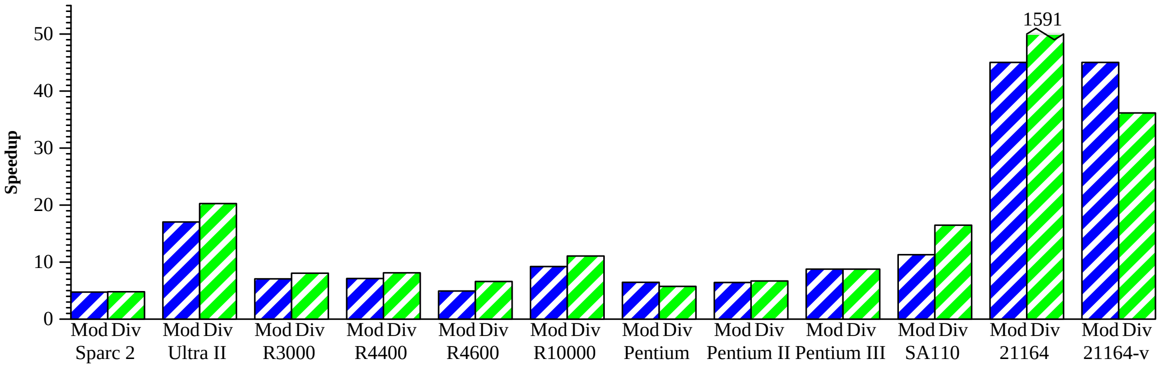

Figure 1-3: Performance improvement on several architectures of example modulo and divsion transformations.

1.2 Goals

We will describe a set of code transformations which reduce the number of dynamic calls to modulo and division operations focusing on removing such operations from within loops. We will assume that such operations are supported by the target platform and thus can be used outside of the loop in order initialize appropriate variables. Naturally if the target platform did not support such operations at all, they could be implemented in software. As such the transformations will focus not on removing the number of instances of these operations in the resulting code, but reducing the total number of time such operations need to be performed at runtime.

As many of these transformations will be dependent upon being able to show certain properties about the values being transformed, we will also attempt to show what analysis is necessary in order to be able to show that it is safe to apply the strength reductions. We will also present a system which attempts to extract the required information from a program and use it to perform these analyses.

Since reducing modulo and division operations is of primary benefit to post processing code generated by other compiler transformations rather than hand written code, consideration will should also be given to transformations whose preconditions may be difficult to determine from the provided code alone. Data transformations, being the most common sort of transformation to introduce modulo and division operations, may know a great deal about the properties of the transformed space and as such it would not be unreasonable to anticipate that such transformations could provide additional annotations about the values that variables or expressions might take on.

1.3 Outline

In Chapter 2 we present a description of the transformations we propose and conditions under which they can apply. Then in Chapter 3 we discuss our implementation of these transform. Chapter 4 presents results obtained using our implementation and we give our conclusions in Chapter 5. We also work through the transformations performed on one example in detail.

Chapter 2

Technical Approach

Rather than a single transformation to reduce the use if integer modulo and division operations, we will show a number of different techniques which can be used under different circumstances. Since the aim is to reduce dynamic uses of modulo and division operations we look at occurrences of these operations within loops. We also make certain simplifying assumptions about the domain of modulo and division operations we wish to optimize in order to simplify the analysis process. To facilitate the discussion of various transformations we describe certain notations which we will use throughout our descriptions of possible transformations.

2.1 Input Domain Assumptions

To make the analysis more manageable we restrict the space of modulo and division operations which we shall consider when evaluating if some form of strength reduction is possible.

Since the bulk time executing any given type of operation is spent on instances occurring inside of the body of loops, we begin by restricting our input space to modulo and division operations which occur inside the body of loops. For stand alone modulo or division operations it is possible to convert a constant denominator to other operations, and in the case that there is only one instance of the operation and the denominator is unknown it seems unlikely that one can do much better than performing the actual modulo or division. We, furthermore, restrict our attention to for-loops since these typically provide the most information about the iteration space of the loop and hence provide the best information about the value ranges variables inside of the loop will take on. We also require that the bounds and step size of the for-loops are not modified during the execution of the loop itself.

We also assume that loop invariant code and expressions have been moved out of loops. This is done primarily to simplify the analysis such that the modulo or division expression is a function of the loop index of the innermost enclosing for-loop. As such it is possible to model the conditions for transformation entirely upon a single for-loop without having to directly consider loop nests.

For-loops with non-unit step sizes or non-zero lower bounds are also required to be normalized prior to attempting to optimize modulo and division operations. Requiring this simplifies both the analysis as well as the actual transformation process. This transformation can be safely applied to any for-loop and thus does not thereby sacrifice any generality. The primary drawback of this approach is that it can sometimes make it more difficult to determine certain properties of the upper bound of the for loop in the event that step size is not known as the step size is folded into the upper bound. It would be possible to negate this shortcoming if the analysis phase is sufficiently sophisticated to handle expressions in which some of the terms are unknowns.

Furthermore, we assume that the modulo operations which are being dealt with follow C-style semantics rather than the classic mathematical definition of modulo. Thus the result of a modulo with a negative numerator is a negative number. For example $-7 \,\%\, 3 = -1$ rather than 2. The sign of the denominator does not affect the result of the expression. The case of a positive numerator is the same as the mathematical definition of modulo. It should be possible to extend the transformations presented to handle modulo operations defined in the mathematical sense in which the result is always positive. Since the C-style modulo in the case of a negative numerator is just the negative of the difference of the denominator and the mathematical modulo, it should be possible to convert most of the transformations in this thesis conform with the mathematical semantics by changing signs and possible adding an addition.

2.2 Analysis Model

The conditions which must be satisfied at compile time in order to permit a transformation to occur will create a system of inequalities. Solving such a system in general is a very difficult problem, but in the case that the constraints can be expressed as a linear system of inequalities, efficient means exist [9] to determine if a solution exists. Since we will be solving our constraints using linear polynomials we will constrain our input to have both numerators and denominators which are representable as polynomials. We will require that our conditions simplify to linear polynomials. The denominator is required to be a polynomial of loop constants which remain constant across iterations of the innermost for-loop which contains the modulo or division. The numerator can be expressed as a polynomial of loop constants and of the index variable for the innermost for-loop containing the modulo or division.

If the for-loop is nested inside of other for loops, we treat the index variables of the outer for-loops just as loop constants themselves. Since we have already required that the modulo or division operation to have been moved out of any for-loop in which it is invariant, it will have a dependency on the immediate enclosing for-loop. This formulation does prevent the possibility of developing transformations involving nested for-loops simultaneously. However, since most of the benefit is gained from removing the operations from the innermost loop, and since any modulo or division operations which are moved outside of the loop to initialize state could be optimized in a second pass, analyzing a single for-loop at a time does not significantly limit the effectiveness of attempts to strength reduce modulo and division operations.

2.3 Notation

To simplify the description of the transformations we introduce some simple notation. We use this notation both for describing the constraints of the system required for a transformation and within the pseudocode used to describe the actual transformation.

- The index variable of the innermost enclosing for-loop of the modulo or division operation shall be referred to as i.
- We will let N and D represent the numerator and denominator, respectively, of the modulo or division operations in question. Both should be polynomials of loop constant variables, with the caveat that the numerator can also contain a terms of the loop index variable i.
- The upper and lower bounds of the innermost enclosing for-loop can be referred to as L and U respectively. Since we have assumed that loops will be normalized, however, L can generally be omitted as it will be equal to zero.
- We let n represent the coefficient of the index variable in the numerator polynomial. Since the for-loops are normalized such that the index is incremented by one, the value of the entire numerator expression will be incremented by n for each iteration.
- It is frequently useful to consider the portion of numerator which remains unchanged through the iterations of the loop which we shall define as $N^- = N - n * i$. This is simply the numerator polynomial with the terms dependent upon i removed. In the case of normalized loops it is also the starting value of the numerator.

When describing modulo operations we will use the form $a \% b$ to indicate a modulo operation with C-style semantics. When a traditional, always positive, modulo operation is needed we will instead use the classical $a \bmod b$ notation.

```
for i ← 0 to U
    do x ← N % D
       y ← N/D
```

Figure 2-1: Structure of untransformed code

2.4 Transformations

For convenience we will describe our transformations in terms of transforming both modulo and division operations at the same time assuming that the original code contains an instance of each that share the same numerator and denominator. In the event that this is not the case we can simply remove extraneous dead code from the optimization.

All of the following transformations assume that the input code is of the form shown in Figure 2-1 in which the variables represent the values described above. A brief summary of the requirements and resulting code for each of the transforms is presented in Figure 2-2.

2.4.1 Reduction to Conditional

The most straightforward transform is to simply introduce a conditional into the body of the loop. This transform uses code within the loop to detect boundaries of either the modulo or division expressions and thus cannot handle cases in which the numerator pass through more than one boundary in a single iteration. Thus we require that $n < D$ for this transform.

The transformed code, seen in Figure 2-3, is quite straightforward and simply checks on each iteration to see if the modulo value has surpassed the denominator, and, if so, increments the modulo and division values appropriately. The code in Figure 2-3 as shown works for cases where the numerator is increasing. If the numerator is decreasing across iteration of the loop the signs of the incrementing operations need to be modified as does the test in the conditional.

While the constraint on when replacing the modulo or division operator with a

Reduction to Conditional

Precondition: $|n| < |D|$

Effects: Introduces a conditional in each loop iteration providing only moderate speed gains.

Elimination from a Continuous Range

Precondition: $N \geq 0 \implies D \geq 0 \wedge \exists\, k \text{ s.t. } kD \leq N < (k+1)D$

$N < 0 \implies D \geq 0 \wedge \exists\, k \text{ s.t. } (k+1)D < N \leq kD$

Effects: Completely removes modulo and division operations, but requires determining k.

Elimination from Integral Stride

Precondition: $n \,\%\, D = 0 \ \wedge\ \exists\, k \text{ s.t. } kD \leq N^- < (k+1)D$

Effects: Replaces modulo and divisions with additions.

Elimination from Integral Stride with Unknown Alignment

Precondition: $n \,\%\, D = 0$

Effects: Modulo and division operations are removed from the loop, but are still used to initialize variables before the loop executes. A extra addition is used within the body of the loop.

Absence of Discontinuity

Precondition: $n > 0 \implies n * \mathit{niter} \leq n - (N^- \,\%\, D) + D - 1$

$n < 0 \implies n * \mathit{niter} \geq n - (N^- \,\%\, D)$

Effects: Removes modulo and division operations from the loop replacing them with an addition to maintain state across loops. Modulo and division operations are used to before the body of the loop.

Loop Partitioning

Precondition: $D \,\%\, |n| = 0 \ \wedge\ \mathit{niter} \leq D/|n|$

Effects: Removes modulo and divisions from the loops, but still uses both operations in the setup code before the loop. Also this results in the final code containing two copies of the body of the loop.

Aligned Loop Tiling

Precondition: $D \,\%\, n = 0 \ \wedge\ N^- = 0$

Effects: Replaces modulo and divisions within the body of the loop with an addition, but still uses such operations during the setup code. Introduces one extra comparison every D iterations of the loop.

Loop Tiling

Precondition: $D \,\%\, n = 0$

Effects: Replaces modulo and divisions within the body of the loop with an addition, but still uses such operations during the setup code. Introduces one extra comparison every D iterations of the loop. Results the final code containing two copies of the body of the loop.

Figure 2-2: Summary of available transformations

```
modVal ← N⁻ % D
divVal ← N⁻/D
for i ← 0 to U
     do x ← modVal
        y ← divVal
        modVal ← modVal + n
        if modVal ≥ D
          then modVal ← modVal − D
               divVal ← divVal + 1
```

Figure 2-3: Transformed code for reduction to conditional

```
for i ← 0 to U
     do x ← N − kD
        y ← k
```

Figure 2-4: Elimination from a positive continuous range

conditional is easy to satisfy, this optimization is generally not very attractive in most cases as it does introduce a branch into each iteration of the loop. This is likely to be an especially bad problem for cases in which the denominator is small as having to take the extra branch frequently is likely to thwart attempts by a branch predictor of eliminating the cost of the extra branch. This approach, however, is still useful in cases where the cost of a modulo or division operation is still substantially higher than a branch, such as systems which emulate this operations in software.

2.4.2 Elimination for a Continuous Range

Since data transformations used in parallelizing compilers are frequently designed to lay out the data accessed by a given processor contiguously in memory, it is not uncommon for a modulo or division expression to only span a single continuous range. When this can be detected the modulo or division operator can be removed entirely.

For positive numerators, if it can be shown that $N \geq 0$, $D \geq 0$ and an integer k can be found such that $kD \leq N < (k+1)D$ then it is safe to eliminate the modulo and division operations entirely as shown in Figure 2-4.

An analogous transformation can be performed if the numerator expression is

for $i \leftarrow 0$ **to** U
    do $x \leftarrow N + kD$
        $y \leftarrow k$

Figure 2-5: Elimination from a negative continuous range

for $i \leftarrow 0$ **to** U
    do $x \leftarrow N^{-} - kD$
        $y \leftarrow (n/D)i + k$

Figure 2-6: Transformed code for integral stride

negative. This transformation requires $N < 0$, $D \geq 0$, and that an integer k can be found such that $(k+1)D < N \leq kD$. The resulting code is shown in Figure 2-5.

Both of these transforms are very appealing as they eliminate both modulo and division operations while adding very little other overhead. They, unfortunately, require the ability for the optimizing pass to compute k which often requires extracting a lot of information about the values in the original program if one is not fortunate enough to get loops with constant bounds.

2.4.3 Elimination from Integral Stride

Another frequently occurring pattern resulting from data transformations are instances where the numerator strides across discontinuities but the increment size of the numerator is a multiple of the denominator. This transformation also requires that the N^{-} value does not cross discontinuities itself over multiple executions of the loop. Thus when $n \,\%\, D = 0$ and when an integer k can be found such that $kD \leq N^{-} < (k+1)D$ one can eliminate the modulo and division operations as shown in Figure 2-6. In the event that k turns out to be less than, then the values of k in the transformed code need to be incremented by 1.

This transformation is very similar to the previous one in that it adds very little overhead to the resulting code after removing the modulo and division operations, but once again it requires that a sufficient amount of information can be extracted from the program to determine the value of k.

$\mathit{modVal} \leftarrow N^{-} \,\%\, D$
$\mathit{divVal} \leftarrow N^{-}/D$
for $i \leftarrow 0$ **to** U
    do $x \leftarrow \mathit{modVal}$
        $y \leftarrow \mathit{divVal}$
        $\mathit{divVal} \leftarrow \mathit{divVal} + (n/D)$

Figure 2-7: Transformed code for integral stride with unknown alignment

$\mathit{modVal} = k$
$\mathit{divVal} = N^{-}/D$
for $i \leftarrow 0$ **to** U
    do $x \leftarrow \mathit{modVal}$
        $y \leftarrow \mathit{divVal}$
        $\mathit{modVal} \leftarrow \mathit{modVal} + n$

Figure 2-8: Transformation from absence of discontinuity

If one cannot statically determine the value of k one can use a slightly more general form of the transformation which only requires that $n \,\%\, D = 0$. The code in this case can be transformed into the code shown in Figure 2-7.

2.4.4 Transformation from Absence of Discontinuity

As described above it is common for a modulo or division expression to not encounter any discontinuities in the resulting value. In cases where one cannot statically determine the appropriate range to use as required by the transformation in Section 2.4.2, we prevent this alternate transformation with less stringent requirements. If we let $k = N^{-} \,\%\, D$ and can show that $(n > 0 \ \wedge \ n * \mathit{niter} \leq D - k + n - 1) \vee (n < 0 \ \wedge \ n * \mathit{niter} \geq n - k$ where niter is the number of iterations of the for-loop (which as long as the loops are normalized this will be the same as U). The resulting code in this case is shown in Figure 2-8.

While the preconditions for this transformation are often easier to verify than those from Section 2.4.2, it also contains a loop carried dependency which might make other optimizations more difficult, although this particular dependency could be removed if necessary.

```
modVal ← N⁻ % D
divVal ← N⁻ / D
breakPoint ← min(⌈(D − modVal)/n⌉ − 1, U)
for i ← 0 to breakPoint
    do x ← modVal
       y ← divVal
       modVal ← modVal + n
modVal ← modVal ∓ D
divVal ← divVal ± 1
for i ← breakPoint + 1 to U
    do x ← modVal
       y ← divVal
       modVal ← modVal + n
```

Figure 2-9: Transformed code from loop partitioning

2.4.5 Loop Partitioning

While most of the transformation thus far have exploited cases in which the loop iterations were conveniently aligned with the discontinuities in the modulo and division expressions, this is not always the case. If a loop can be found which can be shown to cross at most one discontinuity in modulo and division expressions, then one can partition the loop at this boundary. Therefore if $D \,\%\, |n| = 0$ and $niter \leq D/|n|$, where *niter* is the number of iterations which equals U, then one can duplicate the loop body, while removing the modulo and division operations as shown in Figure 2-9. The sign of the increment for *divVal* needs to be set to match the sign of n.

This transformation adds very little overhead within the body of the loops, but does introduce some extra arithmetic to setup certain values. Possibly more problematic is that it replicates the body of the loop which becomes increasingly undesirable as the size of the body grows. Figure 2-10 shows an example of a simple loop being partitioned using this transformation.

2.4.6 Loop Tiling

There remains a great many iteration patterns which involve crossing multiple discontinuities in the modulo and division expressions. We can handle many these with

Before Transform:

```
for i ← 0 to 8
    do A[(i + 26)/10] ← 0
       B[(i + 26) % 10] ← 0
```

After Transform:

```
modVal ← 6
divVal ← 2
for i ← 0 to 3
    do A[modVal] ← 0
       B[divVal] ← 0
       modVal ← modVal + 1
modVal ← modVal − 10
divVal ← divVal + 1
for i ← 4 to 8
    do A[modVal] ← 0
       B[divVal] ← 0
       modVal ← modVal + 1
```

Figure 2-10: Loop partitioning example

loop tiling. By doing so we can arrange for the discontinuities of the modulo and division expressions to fall on the boundary's of the inner most loop. To use loop tiling we must be able to show that $D \,\%\, n = 0$. If we can furthermore show that $N^{-} = 0$ then we can use the aligned loop tiling transformation as shown in Figure 2-11. Once again the sign of the increment of *divVal* needs to match the sign of n.

If, however, the loop does not begin executing properly aligned then it is necessary to also add a prelude loop to execute enough iterations of the loop to align the

```
divVal ← N⁻/D
for ii ← 0 to ⌊U/(D/n)⌋
    do modVal ← 0
       for i ← ii * (D/n) to min((ii + 1) * (D/n) − 1, U)
           do x ← modVal
              y ← divVal
              modVal ← modVal + n
       divVal ← divVal ± 1
```

Figure 2-11: Transformed code for aligned loop tiling

```
if N⁻ < 0
  then breakPoint ← (−N⁻ % D)/|n| + 1    ▷ N approaching zero
  else breakPoint ← (N⁻/D + 1)(D/|n|) − N⁻/|n|    ▷ N moving away from zero
modVal ← N⁻ % D
divVal ← N⁻/D
for i ← 0 to breakPoint − 1
    do x ← modVal
       y ← divVal
       modVal ← modVal + n
if breakPoint ≥ 0
  then divVal ← divVal ± 1
modInit ← (n ∗ breakPoint + N⁻) % D
for ii ← 0 to ⌊U/(D/|n|)⌋
    do modVal ← modInit
       lowBound ← (ii) ∗ D/|n| + breakPoint
       highBound ← min((ii + 1) ∗ (D/|n|) + breakPoint − 1, U)
       for i ← lowBound to highBound
           do x ← modVal
              y ← divVal
              modVal ← modVal + n
       divVal ← divVal ± 1
```

Figure 2-12: Transformed code for loop tiling for positive n

numerator before passing off control to tiled body. The structure resulting from this transformation is shown in Figure 2-12 for positive n.

Once again the sign of the *divVal* increments should match the sign of the n. In order to align the iterations with discontinuities in the values of modulo and division operations requires the loop tiling transformation to compute the first break point at which this will occur. Unfortunately the formula required to determine this break point required to align the loop iterations with discontinuities in the values of modulo and division expressions, depends both on the sign of n and the sign of N^-. Thus this transformation either requires the compiler to determine the sign of n at compile time or to introduce another level of branching. The code to compute *breakPoint* shown in Figure 2-12 is valid for positive n. For negative n the conditional shown in Figure 2-13 should be used instead.

The unaligned version of the transform introduces far more setup overhead than

if $N^- > 0$
  then $breakPoint \leftarrow (N^- \,\%\, D)/|n| + 1$
  else $breakPoint \leftarrow ((-N^-/D + 1) * (D/|n|) - (-N^-)/|n|)$

Figure 2-13: Loop tiling break point computation for negative n

any of the other transformations, but can still outperform the actual modulo or division operations when the loop is executed. The transformation does add quite a bit of code both in terms of setup code and from duplicating the body of the loop. The example transforms shown in Figures 1-1 and 1-2 are examples of aligned loop tiling. An example which has unknown alignment is shown in Figure 2-14.

2.4.7 Offset Handling

As data transformations which introduce modulo and division expressions into array index calculations are a primary target of these optimizations, and since programs using array frequently access array elements offset by fixed amounts, our reductions should be able to transform all situations which contain multiple modulo or division expressions each with a numerator offset from the others by a fixed amount.

Some of the transformations can naturally handle these offsets by simplifying being applied multiple times on each of the different expressions. Other transformations, such as loop tiling, alter the structure of the loop dramatically complicating the ability to get efficient code from multiple applications of the transformation even when all of the divisions share the same denominator. To allow for such offsets we have extended the transformation to transform all of these offset expressions together. The basic approach we took to handling this was to use the original loop tiling transform as previously described, but to replicate the inner most loop. With each of the new bodies all offset expressions are computed by just adding an offset from the base expression (which is computed the normal way). We choose which expression to use as the base arbitrarily. In the steady state it doesn't really matter, but ideally it would be best to choose one that would allow us to omit the prelude loop as the aligned loop tiling transformation does. Each of these inner loops has it's bounds set

Before Transform:

```
for i ← 0 to 14
    do A[(2i + m)/6] ← 0
       B[(2i + m) % 6] ← 0
```

After Transform:

```
breakPoint ← (m/6 + 1) * (6/2) − m/2
modVal ← m % 6
divVal ← m/6
for i ← 0 to breakPoint − 1
    do A[divVal] ← 0
       B[modVal] ← 0
       modVal ← modVal + 2
if breakPoint ≥ 0
  then divVal ← divVal + 1    ▷ m not aligned; previous loop was executed
modInit ← (2 * breakPoint + m) % 6
for ii ← 0 to 4
    do modVal ← modInit
       lowerBound ← (ii) * 3 + breakPoint
       upperBound ← min((ii + 1) * 3 + breakPoint − 1, 14)
       for i ← lowerBound to upperBound
           do A[divVal] ← 0
              B[modVal] ← 0
              modVal ← modVal + 2
       divVal ← divVal + 1
```

Figure 2-14: Loop tiling example. m is unknown at compile time, but the expression must be positive since it is being used as an array access.

```
for i ← 0 to U
    do a ← N % D
       b ← (N + 1) % D
       c ← (N − 1) % D
```

Figure 2-15: Offset example before transformation

```
off_b ← offInit_b
off_c ← offInit_c
upperBound ← max(((ii) + 1) ∗ D/|n|, U)
lowerBound ← ii ∗ D/|n|
for i ← lowerBound to min(lowerBound + break_c − 1, upperBound)
    do a ← modVal
       b ← modVal + off_b
       c ← modVal + off_c
       modVal ← modVal + n
off_c ← off_c − D
for i ← lowerBound + break_c to min(lowerBound + break_b − 1, upperBound)
    do a ← modVal
       b ← modVal + off_b
       c ← modVal + off_c
       modVal ← modVal + n
off_b ← off_b − D
for i ← lowerBound + break_b to min(lowerBound + break_a − 1, upperBound)
    do a ← modVal
       b ← modVal + off_b
       c ← modVal + off_c
       modVal ← modVal + n
```

Figure 2-16: Inner loops of offset example after transformation

such so as to end when one of the offset expressions reaches a discontinuity. Between loops the offset variables are appropriately updated.

A simple example of this is shown in Figures 2-15 and 2-16. Figure 2-15 shows the original code and Figure 2-16 shows the innermost set of for loops after going through the loop tiling optimization. Variables such as *offInit* and *break* are precomputed before executing any portion of the loop. This transform creates three copies of the loop body. These loops are split based on discontinuities in the offset expressions such that the differences between the expressions within a single one of the loops

	$N > 0$	$N < 0$
$n > 0$	$break_a = \left\lceil \frac{D - (a \bmod D) - modInit}{n} \right\rceil$ $modOffInit_a = a \bmod D$ $divOffInit_a = \frac{N^- + a}{D} - divInit$ $modOffInc_a = -D$ $divOffInc_a = 1$	$break_a = \left\lceil \frac{1 - (a \bmod D) - modInit}{n} \right\rceil$ $modOffInit_a = a \bmod D$ $divOffInit_a = \frac{N^- + a}{D} - divInit$ $modOffInc_a = -D$ $divOffInc_a = 1$
$n < 0$	$break_a = \left\lceil \frac{(a \bmod D) + modInit - D}{n} \right\rceil$ $modOffInit_a = a \bmod D - D$ $divOffInit_a = \frac{N^- + a}{D} - divInit$ $modOffInc_a = D$ $divOffInc_a = 1$	$break_a = \left\lceil \frac{(a \bmod D) + modInit}{n} \right\rceil$ $modOffInit_a = a \bmod D - D$ $divOffInit_a = \frac{N^- + a}{D} - divInit$ $modOffInc_a = D$ $divOffInc_a = 1$

Figure 2-17: Formulas to initialize variables for an offset of a

is constant. For this example $break_c = 1$, $break_b = D - 1$, $offInit_c = D - 1$, and $offInit_b = 1$. Divisions are handled in the same way, only need to be incremented between each of the loops rather than within each of the loops.

In the general case, when N and n may not be positive and when one might need a prelude loop, unlike in the given example, the transformation is slightly more complex. The setup code is required to find values for five variables for each one of the offset expressions: *break* is the offset of the discontinuity relative to the base expression, *modOffInit* is the value that the modulo expression's offset should be reinitialized to every time the base expression crosses a discontinuity, *divOffInit* is the value that the division expression's offset should be reinitialized to every time the base expression crosses a discontinuity, *modOffInc* is the amount that the modulo expression's offset needs be incremented by at its discontinuity, and *divOffInc* is the amount that the division expression's offset needs to be incremented by at its discontinuity. These expressions used to compute these values depend upon the sign of both N and n requiring the addition of runtime checks if these signs cannot be determined statically. The expressions to compute each of these variables are shown in Figure 2-17.

The inner for loops in both the prelude loop and the main body of the loop tiling case can then be split across the boundaries for each of the offset expressions as was shown above. If the prelude loop executes any iterations *divVal* must also be incremented before entering the main tiled loops.

Chapter 3

Implementation

While the primary intent of providing strength reductions for modulo and divisions operations is to act as a post pass removing such operations that get introduced during data transformations performed in other optimization passes, all current passes of this type necessarily contain some form of modulo and division operations folded into them. As such we were unable to focus on a particular data transformation's output as a target for the strength reductions. We have therefore implemented a general purpose modulo and division optimizing pass in anticipation of future data transformations passes which will not need to contain additional complexity to deal with modulo and division optimizations.

3.1 Environment

To simplify and speed the development of the optimization pass and to encourage future use we choose to implement the system using the SUIF2 compiler infrastructure as well as the Omega Library as to solve equality and inequality constraints.

3.1.1 SUIF2

We choose to implement the modulo and division transformations as a pass in the using SUIF2 infrastructure. SUIF2 provides an extensible compiler infrastructure for experimenting with, and implementing new types of compiler structures and optimizations. The SUIF2 infrastructure provides a general purpose and extensible IR as well as the basic support infrastructure around this IR to both convert source code into the IR and to output the IR to machine code or even back into source code.

The SUIF2 infrastructure is targeted toward research work in the realm of optimizing compilers and in particular parallelizing compilers. As transforms involved paralleling are the prime candidate for using a post pass to eliminate modulo and division operations, targeting a platform on which such future parallelizing passes are likely to be written seems ideal. The original SUIF infrastructure has already been widely used for many such projects. SUIF2 is an attempt to build a new system to overcome certain difficulties that arose from the original version of SUIF system. For instance it provides a full extensible IR structure allowing additions of new structures without necessitating a rewrite of all previous passes. Since it seems likely that in the near future the bulk of development will move to the newer platform, and since these modulo and division optimizations will likely have less benefit for already existing transformations which have been implemented in the original SUIF, it seems reasonable to target the newer version.

3.1.2 The Omega Library

All of the transformations we present to remove modulo and division operations require satisfying certain conditions during compilation in order to be safely performed. Verifying these conditions often requires solving a system of inequalities composed of the condition itself as well as conditions extracted from analyzing the code. The Omega library is capable of manipulating sets and relations of integer tuples and can be used to solve systems of linear inequalities in an efficient manner.

The representation used by the Omega library to store systems of equations is somewhat awkward to manipulate (though understandably so as it was designed to allow for a highly efficient implementation). As such we have chosen to use the Omega library only for the purposes of for solving systems and not for any sort of data representation with in our pass. Thus should the analysis that the Omega library performs ever prove insufficient, substituting another package would result in minimal impact on the code base for our optimization.

3.2 Implementation Structure

Many of the requirements preconditions on the structure of the input are often simple to satisfy and as such as part of our implementation we have created simple prepasses for such things as loop normalization. In order to facilitate gathering and manipulating information about the values and ranges variables can take on we created an expression representation as well as the needed infrastructure to evaluate expressions of this form using the omega library. Separated from the analysis portion of our pass we have naturally implemented the logic required to perform the actual code transformations on the SUIF2 IR.

3.2.1 Preliminary Passes

As described above our analysis framework presupposes a number of conditions about the for-loops it considers for optimization so as to simplify the framework. While these conditions are common in compiler's SUIF2, being a very young framework, did not contain infrastructure to provide them. We have, therefore, implemented two passes intended to preprocess the code prior to the attempt to eliminate modulo and division operations.

As our analysis focuses on attempting to transform a modulo or division operations with the context of its enclosing for-loop, it is necessary for the expression to actually be dependent on upon the iterations of that for-loop. As such we have implemented a pass using SUIF2 to detect loop invariant statements and move them outside of loops within which they do not vary. Unfortunately our implementation is somewhat limited as it currently only supports moving entire loop invariant statements and does not attempt to extract portions of expressions which may be loop invariant. Our current code would likely be somewhat more effective when combined with a common subexpression eliminator, which currently does not yet exist in SUIF2, but even as such may fail to move some loop invariant modulo and division operations. Expanding the prepass to check for loop invariant expressions, however, is a straight forward problem.

Our analysis also requires that for-loops be normalized such that their index variable begins at 0 and increments each iteration by 1. Doing so results in references to the original loop index becoming linear functions of the new, normalized, index variable. Since our analysis operates on modulo and division expressions with polynomial numerators and denominators this transformation will not result in causing an expression which was previously in the supported form to be transformed into a form which can no longer be represented. Unfortunately this transformation does often result in a more complex upper bound.

3.2.2 Data Representation

The majority of the time spent attempting to optimize away modulo and division operations, as well as the majority of the code involved in the process, deals with attempting to verify the conditions upon which the various transformations depend. This effort largely involves examining a given constraint, substituting in variables and values which need to be extracted from the code. In order to represent these expressions and constraints we developed a data representation for integer and boolean expressions.

While it was somewhat unfortunate that we needed to develop another representation when our system already contained two other representations capable of representing such expressions (the SUIF2 IR and the Omega libraries relations), neither proved to be well suited to our tasks. The Omega library's interface is severely restrained in order to allow an internal representation that allows for highly efficient analysis. Unfortunately the interface was too limiting to use unaugmented and we wished not to become permanently dependent upon the Omega library in case a more general, if slower, analysis package became necessary in the future. SUIF2, on the other hand, has a very flexible representation of expressions as part of its IR. As part of the IR, however, the SUIF2 tree is dependent upon a strict ownership hierarchy of all of its nodes in order to properly manage memory. This was somewhat inconvenient for our purposes as we largely wished to deal with expressions which were not part of any such ownership tree. We also wished to not have to deal with many of the intricacies that arose from SUIF2's extensible nature as we were just working with simple algebra. As such we implemented our own, albeit simple, tree-structured expression data structure.

While technically not a pass that is run prior to the modulo and division optimization pass, the first step of the modulo and division pass is to combine all instances of modulo *or* division operations which share the same numerator and denominator and occur within the same for-loop so that they can be transformed together. We also combine together instances with the same denominator but numerators offset by a fixed amount. In the case that there are no such offsets this is essentially a specialized for of common subexpression elimination, but it needs to be performed in order to allow us to detect and handle offset expressions efficiently.

3.2.3 Analyzing the Source Program

Each possible transformation starts out with its own condition which needs to be verified before the transformation can occur. The general conditions first need to be instantiated with the actual expressions from the program. Then the variables in the resulting expression need to each be examined and possibly expanded to reflect their values, or constraints upon the values they can take on. Once the test expression is instantiated with expressions from the input code, we need to find values, expressions, or ranges for the values in the test expression in order to find a solution to it.

We structured these transformations as an iterative process of analyzing each variable in turn and determining a context for each variable. We then merged this context with the test expression and proceeded to the next variable. Each context was modeled an optional expression and a list, perhaps empty, of constraints which are known to be true about the expression. If the context has an expression then it is considered equivalent to substitute the variable for that expression. If possible, the substitution is performed and contexts are looked up for any new variables introduced by the expression. We utilize this structure in order to simplify the future process of adding new methods of extracting constraints about variables. Since we anticipate that in many instances the data transformation pass that introduced the modulo and division operations will be capable of easily supplementing our analysis with information that it already has computed we can use this interface to query any annotations it may have added to the code. Naturally it will also facilitate adding new analyses ourselves.

In the present implementation of our system we use comparatively simple techniques in order to find contexts for variables of interest. Presently only use two very simplistic techniques for extracting information for to provide a context for a variable. If the variable has a single reaching definition we provide that expression as a substitution for the variable provided the substitution consists of operations representable in our framework. If the variable in question is the index variable in an enclosing for script then constraints are created which bound it to be between the lower and upper bounds of the loop. One need not also constrain that the value strides by the step size if, indeed, all for-loops are normalized in the normalization pass.

This sort of recursive walking of the definitions of variables is adequate for many cases, but often proves insufficient as it is unable to deal with conditional data flow. It would often be more effective to attempt to compute working ranges using a full dataflow based algorithm. While adding such a feature would offer a clear improvement over the current limited means of analysis, SUIF2 does not yet contain a flexible dataflow framework and we lacked sufficient resources to implement the complete dataflow algorithm.

After gathering available information from the input code, it is then necessary to evaluate the resulting condition. The condition as well as all constraints are then converted into the representation used in the omega library and then a solution is searched for.

3.2.4 Performing Transformations

As there are a number of different transformations which need to be considered we have modularized the implementation of each transformation from the remaining infrastructure required in order to simplify any future attempts to add new transformations. This structure also made it simple to attempt to analyze the code in the context of each possible transformation and then, if multiple transformations can be applied, use global heuristics to select the preferred one. While certain transformations are clearly preferable to others, such as eliminating the entire instance of the modulo or division when compared to introducing a conditional, in some cases the best solution may vary based on factors such as the expected magnitude of the denominator and the relative costs of divisions, branches, and comparison for the target architecture.

Once the transform is selected, applying the transformation is straightforward. The transformations as written in our implementation, do assume that output code will at least be run through both a constant propagater and a copy propagater as they liberally introduce additional temporary variables for convenience.

Chapter 4

Results

As the primary intended use of the modulo and division optimizations presented in this thesis is to be used as a post pass of other compiler optimizations that introduce such operations into the code, this pass requires such transformed code in order to be used. Current optimizing passes, however, are already required to have sufficient logic incorporated into their transformations to prevent adding the modulo and divisions operations into the generated code as doing so would introduce a unreasonable amount of overhead. This, unfortunately, makes it difficult to use our transformations in the context in which they are intended to be used.

In order to examine the performance of our system against algorithm that have undergone data transformations we have made use of a small sample set of benchmarks which were parallelized and transformed by hand. Such samples should provide an reasonably accurate impression of the performance of the transformations we present, but it is less clear if the provide an accurate view of the effectiveness of the analysis code as the hand generated code is likely to be structured somewhat differently than that produced by an earlier optimization pass.

The two sample algorithms that we have used are implementations of a 5-point stencil running over a two dimensional array and an matrix LU factorization algorithm. Each of the two algorithms was initially written as a simple single processor implementations. Each was then converted by hand into a parallelized version using a simple straightforward partitioning of the data space for each processor. Since this

```
for t ← 1 to numSteps
    do for i ← 1 to n − 1
        do for j ← 1 to n − 1
            do A[j][i] ← f(B[j][i], B[j][i + 1], B[j][i − 1],
                           B[j + 1][i], B[j − 1][i])
      swap(A, B)
```

Figure 4-1: Original five point stencil code

has poor performance due to cache conflicts the data was rearranged in order to improve locality. The structure of this rearrangement different for each algorithm, but both introduced both modulo and division operations into each array access severely hampering performance. We then used this code with the data remapped as sample input for our optimization pass.

Since the SUIF2 infrastructure still lacks many traditional optimizations like copy propagation we used the SUIF2 system as a C source to source translator such that after attempting to remove modulo and division operations we converted our IR back into C source. This allowed us to run gcc on the resulting code in order to provide such normal optimizations and to provide a robust code generation system.

4.1 Five Point Stencil

The first sample code we considered was a five point stencil which iteratively computes a new value for each element of a two dimensional array by computing a function of the value at that location and the values from each of the four surrounding locations. We used a simple implementation which used two copies of the array alternating source and destination for each iteration. The pseudocode for the original algorithm is shown in Figure 4-1.

While there are a number of mappings that could be used to assign the elements of the array to different processors in order to parallelize, one would ideally wish to minimize the required amount of interprocess communication, which can be achieved by dividing the data space into roughly square regions for each processor. After doing so one wishes to remap the data elements such that each processor's elements are arranged contiguously in memory. If we assume that the data space was divided amongst by dividing the n by n array into *procHeight* chunks vertically and *procWidth* chunks horizontally, we can map the two dimensional array to a single one dimensional array with each processor being assigned a contagious chunk of memory by using the following formula (based on [3]):

$$offset = (j/denom * n) * denom + i * denom + j \,\%\, denom$$

where

$$denom = \left\lceil \frac{n}{procHeight} \right\rceil$$

In order to simplify the measurement process we have modified the multiprocessing part of the code to simply schedule each processors iterations in turn.

Running our algorithm directly on the transformed modified code unfortunately failed to perform any optimizations. Since the borders of the array are not iterated over, so as to prevent the stencil from accessing data out of bounds, conditionals needed by the algorithm to be used to setup the proper bounds. Although this conditional simply amount to checking if the lower bound is zero and if so setting it to one, it does unfortunately hamper our analysis as we are currently lacking a dataflow based value range solver. This impeded all of our transformations as it prevented the pass from establishing that the range of the numerator was either strictly positive or strictly negative. We can work around this by allowing the compiler to assume that the array accesses will always be done with positive offsets. Since array indexes should be positive this is not an unreasonable assumption to make. After doing so the pass is then able to apply a loop tiling optimizations two remove the modulo and division operations from the inner loop. The parallelization, data transformation, and modulo and division optimizations are detailed in Appendix A.

Removing the modulo and division operations from the inner loop of the algorithm resulted in a sizeable performance increase. The running time for a matrix of 1000 elements square dropped for 50 iterations from an average of 8.59 sec to 4.41 sec. These numbers were computed using 4 processors (tiled 2 by 2). Varying the number of processors did not noticeably effect the results. These results are especially impressive as the time spent in the inner loop for this example is largely spent computing array indexes.

4.2 LU Factoring

The second algorithm we had available which introduces modulo and division operations as part of data transformations to improve performance was a matrix LU factoring algorithm. This algorithm allocated the original matrix to processors by rows with each of *numProcs* processors getting every *numProcs* row of the original matrix. Using this transform, also based on [3], the one dimensional offset in memory of the original value located at $A[j,i]$ is:

$$\left(j/numProcs + (j \,\%\, numProcs) * \left\lceil \frac{n}{numProcs} \right\rceil \right) * n + i$$

Since the innermost loop of the factoring algorithm iterates over the elements in a row (incrementing i), the innermost loop does not contain any modulo or division operations. The modulo and division operations are therefore much less frequently occurring in this example than in the previous one. Even when not present in the inner loop, however, they can still lead to a noticeable impact on performance.

Once again our implementation had difficulty deducing that the modulo and division expressions were always positive. If we once again allow it the liberty to assume so, it was able to transform the loop eliminating the innermost modulo and division. Even though the modulo and divisions being removed are not in the inner loop, a noticeable improvement was still noticed. Performing 60 iterations of LU factoring on a 1000 by 1000 matrix split across 10 processors the original data-transformed code completed in 12.02 seconds while after optimizations it ran in 11.66 seconds. It should be noted optimizations also had to be manually turned off for the initialization loop, to prevent it's speed up from distorting the results.

Chapter 5

Conclusion

The introduction of modulo and division operations as part of the process of performing data transformations in compilers, whether for parallelizing applications, improving cache behavior, or other reasons, need not result in a performance loss of the compiled code. Such passes also need not make attempts to remove the modulo and division expression which are naturally inserted into array bound calculations as transformation exist which are capable of removing them as part of a separate stage.

We have shown a variety of transformations which can be used to remove modulo and divisions under different circumstances. Numerous different transformations need to be available as each is applicable under different situations as each exploits certain properties about the modulo and division expressions that hold across iterations of loops. The transformations also vary in the amount of overhead they require in order to eliminate the modulo or division expression from cases in which the operator can just be removed entirely to cases where an conditional branch must be taken during each iteration of the loop.

Simply having a set of transforms is not, however, sufficient to have a useful optimizing postpass; one must also be able to safely and effectively identify instances where the transformations are applicable. It is unfortunately in this area that our current system falls somewhat short. While our system is perfectly capable of identifying possible applications of each of our transforms in simple test programs, when used against real algorithms which have gone through data transformations of the form we expect our system to be used with it currently still needs to be coaxed through the analysis slightly before it is willing to perform the optimization.

We can attribute these failures to primarily our inability to frequently determine the value ranges that a variable might take on. Often simply being able to determine that a variable's value will be positive is all the rest of the analysis depends on. These problems arise from the lack of a dataflow based scheme to compute these ranges. This problem was largely anticipated and put off while SUIF2's dataflow framework matures as we lacked the resources to build up the required infrastructure ourselves.

In addition to the obvious work that should be done to employee a dataflow analysis to compute value ranges, the next logical extension to this project is to attempt to use it in conjunction with an implementation of a data transformation which would benefit from not needing to implement modulo and division optimizations itself. This would provide a better space within which to test and refine the analysis required for the transformations. In the event that some of the analysis required for a large class of programs proves infeasible to perform, it would also allow for the chance to experiment with integration of information known by passing information known about the program prior to the transformations, such as the sign of terms, to the optimizing pass to remove the modulo and divisions.

The introduction of the of a optimization of the form of a strength reductions is often difficult as the optimizations that are being attacked are typically not present in existing code. It is not until the optimization is available that future code is free to make use of those previously inefficient structures that are being reduced without concern for their former slowdown. It is in this vein that we hope that providing this set of transformations that future work done with data transformations that introduce modulo and division operations will be able to be simplified and future effort involved in removing modulo and division operations can be shared across projects.

Appendix A

Five Point Stencil

Figure A-1 once again shows the original 5 point stencil algorithm which iterates over the contents of a two dimensional replacing the contents of one array with the result of a function applied to the corresponding location and surrounding values in the other array. For the purposes of simplicity we have removed the outer loop shown in Figure 4-1 which swaps the arrays and repeats the procedure. The outer loop does not effect the analysis of the loop as it simply repeats the loops execution for some number of iterations.

The original algorithm does not contain any modulo or division operations, as they are introduced in the process of parallelizing the code. Let us first consider a simple parallelization of the algorithm in which we assign each processor a strip of the array. Even though we are only assigning processor along a single dimension we shall refer to both *procWidth* and *procHeight* as the dimensions of our processor grid as later assignments will split the array across both dimensions. Figure A-2 shows the loop split into strips with each strip assigned to a different processor.

The downside to assigning each processor a strip of the two dimensional array is

for $i \leftarrow 1$ **to** $n-1$
    do for $j \leftarrow 1$ **to** $n-1$
        do $A[j][i] \leftarrow f(B[j][i],\ B[j][i+1],\ B[j][i-1],$
            $B[j+1][i],\ B[j-1][i])$

Figure A-1: Original five point stencil code

```
w ← ⌈n/(procWidth * procHeight)⌉
for id ← 0 to procWidth * procHeight,  in parallel
    do for i ← max(id * w, 1) to min((id + 1) * w − 1, n − 1)
        do for j ← 1 to n − 1
            do A[j][i] ← f(B[j][i], B[j][i + 1], B[j][i − 1],
                           B[j + 1][i], B[j − 1][i])
```

Figure A-2: Simple parallelization of stencil code

```
w ← ⌈n/procWidth⌉
h ← ⌈n/procHeight⌉
for id ← 0 to procWidth * procHeight, in parallel
    do iStart ← (id % procWidth) * w
       for i ← max(iStart, 1) to min(iStart + w − 1, n − 1)
           do jStart ← (id/procWidth) * h
              for j ← max(jStart, 1) to min(jStart + h − 1, n − 1)
                  do A[j][i] ← f(B[j][i], B[j][i + 1], B[j][i − 1],
                                 B[j + 1][i], B[j − 1][i])
```

Figure A-3: Stencil code parallelized across square regions

that the cost of interprocessor communication is higher than if each processor were assigned a roughly square portion of the original array. Having a square portion essentially decreases the perimeter to area ratio of the processor's region of the array. Having a relatively smaller perimeter means that there are fewer array elements which need to be shared with other processors and thus reduces parallelization overhead. Figure A-3 shows the 5 point stencil split among processors this way.

Unfortunately splitting data between processors this way does not immediately groove as useful as one might expect. Each processor's assignment from the original two-dimensional array is no longer laid out contagiously in memory. Unless one is lucky enough to have all of the boundaries between processor's regions perfectly aligned with cachelines, this causes thrashing in the cache reducing performance. This can be solved by relaying out the contents of the array in such a way as to make the memory for each processor contiguous again. The result of this transform, taken from [3], is shown in Figure A-4. After the transform w is the width of the region of the original array that each processor operates on and h is the height. The body of the innermost for has been split intro two lines to improve readability.

```
w ← ⌈n/procWidth⌉
h ← ⌈n/procHeight⌉
for id ← 0 to procWidth * procHeight, in parallel
    do iStart ← (id % procWidth) * w
       for i ← max(iStart, 1) to min(iStart + w − 1, n − 1)
           do jStart ← (id/procWidth) * h
              for j ← max(jStart, 1) to min(jStart + h − 1, n − 1)
                  do tmp ← f(B[(j/h * n) + i][j % h],
                             B[(j/h * n) + (i + 1)][j % h],
                             B[(j/h * n) + (i − 1)][j % h],
                             B[((j + 1)/h * n) + i][(j + 1) % h],
                             B[((j − 1)/h * n) + i][(j − 1) % h])
                     A[(j/h * n) + i][j % h] ← tmp
```

Figure A-4: Stencil code after undergoing data transformation

```
w ← ⌈n/procWidth⌉
h ← ⌈n/procHeight⌉
for id ← 0 to procWidth * procHeight, in parallel
    do iStart ← (id % procWidth) * w
       for i ← max(iStart, 1) to min(iStart + w − 1, n − 1)
           do jStart ← (id/procWidth) * h
              j_off ← max(jStart, 1)
              j_max ← min(jStart + h − 1, n − 1)
              for k ← 0 to j_max − j_off
                  do j ← k + j_off
                     tmp ← f(B[(j/h * n) + i][j % h],
                             B[(j/h * n) + (i + 1)][j % h],
                             B[(j/h * n) + (i − 1)][j % h],
                             B[((j + 1)/h * n) + i][(j + 1) % h],
                             B[((j − 1)/h * n) + i][(j − 1) % h])
                     A[(j/h * n) + i][j % h] ← tmp
```

Figure A-5: Stencil code after normalization

The data transformation introduced into the innermost loop 3 distinct pairs of modulos and divisions: $\langle\ j/h,\ j\,\%\,h\ \rangle$, $\langle\ (j+1)/h,\ (j+1)\,\%\,h\ \rangle$, and $\langle\ (j-1)/h,\ (j-1)\,\%\,h\ \rangle$. Before attempting the modulo and division optimizations, we will normalize the innermost loop as is required by our analysis. In reality we would normalize all of the loops, but in an effort to maximize readability we will leave the outer loops as shown. The normalized code is shown in Figure A-5. The normalized code changes the index of the innermost loop from j to k and computes the value of j inside the loop as $k + j_{off}$.

Applying our analysis to the now normalized code will result in determining that it is safe to apply the loop tiling transform.

Figure A-6 shows the 5 point stencil after being loop tiled. References to j/h and $j\,\%\,h$ were replaced with *divVal* and *modVal* respectively. The offset versions modulos and divisions have not been removed yet. As mentioned above, the loop tiling transform introduced to copies of the innermost loop which can start on lines 13 and 25.

Transforming the offset modulo and division operations requires making a number of modifications to the code shown in A-6. First the definition of *modInit* is moved up in the code to occur in front of the prelude loop as it's value is used for setting up some of the variables used for handling the offset loops. Prior to the prelude loop we also define *break*, *modOffInit*, and *divOffInit* as defined in Figure 2-17. The variables with a subscript n are for the offsets of -1 and p are for the offsets of $+1$. To actually remove the offset modulo and division operations the basic technique (since they are offset by just one in either direction) is to peel an iteration off the front and the back of the loop. Since the transformation is structured to handle arbitrary offsets, the peeled loops are written as for loops themselves. Within each of these split for the modulo and division operations can be replaced with a constant offset from *modVal* and *divVal* respectively. Between each of the for for loops the offset for exactly one of the pairs of modulo and division expressions changes.

```
w ← ⌈n/procWidth⌉
h ← ⌈n/procHeight⌉
for id ← 0 to procWidth * procHeight, in parallel
    do iStart ← (id % procWidth) * w
       for i ← max(iStart, 1) to min(iStart + w − 1, n − 1)
           do jStart ← (id/procWidth) * h
              j_off ← max(jStart, 1)
              j_max ← min(jStart + h − 1, n − 1)
              breakPoint ← (j_off/h + 1)(h) − j_off
              modVal ← j_off % h
              divVal ← j_off/h
              for k ← 0 to breakPoint
                  do j ← k + j_off
                     tmp ← f(B[(divVal * n) + i][modVal],
                             B[(divVal * n) + (i + 1)][modVal],
                             B[(divVal * n) + (i − 1)][modVal],
                             B[((j + 1)/h * n) + i][(j + 1) % h],
                             B[((j − 1)/h * n) + i][(j − 1) % h])
                     A[(divVal * n) + i][modVal] ← tmp
                     modVal ← modVal + 1
              if breakPoint ≥ 0
                then divVal ← divVal + 1
              modInit ← (breakPoint + j_off) % h
              for kk ← 0 to (j_max − j_off)/h
                  do modVal ← modInit
                     lowBound ← max((kk)h + breakPoint, L)
                     highBound ← min((kk + 1)h + breakPoint − 1,
                                     j_max − j_off)
                     for k ← lowBound to highBound
                         do j ← k + j_off
                            tmp ← f(B[(divVal * n) + i][modVal],
                                    B[(divVal * n) + (i + 1)][modVal],
                                    B[(divVal * n) + (i − 1)][modVal],
                                    B[((j + 1)/h * n) + i][(j + 1) % h],
                                    B[((j − 1)/h * n) + i][(j − 1) % h])
                            A[(divVal * n) + i][modVal] ← tmp
                            modVal ← modVal + 1
                     divVal ← divVal + 1
```

Figure A-6: Stencil code after loop tiling

The following is the fully transformed code including handling of modulo and division operations with offset numerators.

```
w ← ⌈n/procWidth⌉
h ← ⌈n/procHeight⌉
for id ← 0 to procWidth * procHeight, in parallel
    do iStart ← (id % procWidth) * w
       for i ← max(iStart, 1) to min(iStart + w − 1, n − 1)
           do jStart ← (id/procWidth) * h
              j_off ← max(jStart, 1)
              j_max ← min(jStart + h − 1, n − 1)
              breakPoint ← (j_off/h + 1)(h) − j_off
              modVal ← j_off % h
              divVal ← j_off/h
              modInit ← (breakPoint + j_off) % h
              break_p ← h − 1 − modInit
              modOffInit_p ← 1
              divOffInit_p ← (j_off + 1)/h − divVal
              break_n ← 1 − modInit
              modOffInit_n ← h − 1
              divOffInit_n ← (j_off − 1)/h − divVal
              for k ← max(0, k_off) to min(break_n − 1, breakPoint + k_off)
                  do tmp ← f(B[divVal * n + i][modVal],
                             B[divVal * n + (i + 1)][modVal],
                             B[divVal * n + (i − 1)][modVal],
                             B[(divVal + 0) * n + i][modVal + 1],
                             B[(divVal − 1) * n + i][modVal + h − 1])
                     A[(divVal * n) + i][modVal] ← tmp
                     modVal ← modVal + 1
              for k ← max(break_n, k_off) to min(break_p − 1, breakPoint + k_off)
                  do tmp ← f(B[divVal * n + i][modVal],
                             B[divVal * n + (i + 1)][modVal],
                             B[divVal * n + (i − 1)][modVal],
                             B[(divVal + 0) * n + i][modVal + 1],
                             B[(divVal + 0) * n + i][modVal − 1])
                     A[(divVal * n) + i][modVal] ← tmp
                     modVal ← modVal + 1
              for k ← max(break_p, k_off) to breakPoint
                  do tmp ← f(B[divVal * n + i][modVal],
                             B[divVal * n + (i + 1)][modVal],
                             B[divVal * n + (i − 1)][modVal],
                             B[(divVal + 1) * n + i][modVal − h + 1],
                             B[(divVal + 0) * n + i][modVal − 1])
                     A[(divVal * n) + i][modVal] ← tmp
                     modVal ← modVal + 1
              if breakPoint ≥ 0
                then divVal ← divVal + 1
```

$\textbf{for } kk \leftarrow 0 \textbf{ to } (j_{max} - j_{off})/h$

$\quad \textbf{do } modVal \leftarrow modInit$

$\quad\quad lowBound \leftarrow \max((kk)h + breakPoint, L)$

$\quad\quad highBound \leftarrow \min((kk+1)h + breakPoint - 1, j_{max} - j_{off})$

$\quad\quad \textbf{for } k \leftarrow lowBound \textbf{ to } \min(lowBound + break_n - 1, highBound)$

$\quad\quad\quad \textbf{do } tmp \leftarrow f(B[(divVal * n) + i][modVal], B[(divVal * n) + (i+1)][modVal], B[(divVal * n) + (i-1)][modVal], B[(divVal + 0) * n + i][modVal + 1], B[(divVal - 1) * n + i][modVal + h - 1])$

$\quad\quad\quad\quad A[(divVal * n) + i][modVal] \leftarrow tmp$

$\quad\quad\quad\quad modVal \leftarrow modVal + 1$

$\quad\quad \textbf{for } k \leftarrow lowBound + break_n \textbf{ to } \min(lowBound + break_p - 1, highBound)$

$\quad\quad\quad \textbf{do } tmp \leftarrow f(B[(divVal * n) + i][modVal], B[(divVal * n) + (i+1)][modVal], B[(divVal * n) + (i-1)][modVal], B[(divVal + 0) * n + i][modVal + 1], B[(divVal + 0) * n + i][modVal - 1])$

$\quad\quad\quad\quad A[(divVal * n) + i][modVal] \leftarrow tmp$

$\quad\quad\quad\quad modVal \leftarrow modVal + 1$

$\quad\quad \textbf{for } k \leftarrow lowBound + break_p \textbf{ to } highBound$

$\quad\quad\quad \textbf{do } tmp \leftarrow f(B[(divVal * n) + i][modVal], B[(divVal * n) + (i+1)][modVal], B[(divVal * n) + (i-1)][modVal], B[(divVal + 1) * n + i][modVal - h + 1], B[(divVal + 0) * n + i][modVal - 1])$

$\quad\quad\quad\quad A[(divVal * n) + i][modVal] \leftarrow tmp$

$\quad\quad\quad\quad modVal \leftarrow modVal + 1$

$\quad\quad divVal \leftarrow divVal + 1$

Bibliography

[1] Frances E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[2] R. Alverson. Integer division using reciprocals. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 186–190, Grenoble, France, 1991. IEEE Computer Society Press, Los Alamitos, CA.

[3] Saman Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities.* PhD dissertation, Stanford University, Department of Electrical Engineering, January 1997.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: a Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symbosium on Computer Architecture*, Alanta, GA, May 1999.

[5] B. Greenwald. A technique for Compilation to Exposed Memory Hierarchy. Master's thesis, M.I.T., Department of Electrical Engineering and Computer Science, September 1999.

[6] Linley Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2), February 1995.

[7] Walter Lee, Benjamin Greenwald, and Saman Amarasinghe. Strength reduction of integer division and modulo operations. Technical Report LCS-TM-600, M.I.T., November 1999.

[8] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for raw microprocessors. Technical Report LCS-TM-599, M.I.T., September 1999.

[9] William Pugh. The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs, 1994.

[10] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers, 1994.