Strength Reduction of Integer Division and Modulo Operations

Jeffrey Sheldon, Walter Lee, Ben Greenwald, Saman Amarasinghe *
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.
{jeffshel,walt,beng,saman}@lcs.mit.edu

ABSTRACT

Integer division, modulo, and remainder operations are expressive and useful operations. They are logical candidates to express complex data accesses such as the wrap-around behavior in queues using ring buffers. In addition, they appear frequently in address computations as a result of compiler optimizations that improve data locality, perform data distribution, or enable parallelization. Experienced application programmers, however, avoid them because they are slow. Furthermore, while advances in both hardware and software have improved the performance of many parts of a program, few are applicable to division and modulo operations. This trend makes these operations increasingly detrimental to program performance.

This paper describes a suite of optimizations for eliminating division, modulo, and remainder operations from programs. These techniques are analogous to strength reduction techniques used for multiplications. In addition to some algebraic simplifications, we present a set of optimization techniques that eliminates division and modulo operations that are functions of loop induction variables and loop constants. The optimizations rely on algebra, integer programming, and loop transformations.

1. INTRODUCTION

This paper describes a suite of optimizations for eliminating division, modulo, and remainder operations from programs. In addition to some algebraic simplifications, we present a set of optimization techniques that eliminates division and modulo operations that are functions of loop induction variables and loop constants. These techniques are analogous to strength reduction techniques used for multiplications.

Integer division, modulo, and remainder are expressive and useful operations. They are often the most intuitive way to represent many algorithmic concepts. For example, use of a modulo operation is the most concise way of implementing queues with ring buffers. In addition, many modern compiler optimizations heavily employ division and modulo operations when they perform array transformations to improve data locality or enable parallelization. The SUIF parallelizing compiler [2, 5], the Maps compiler-managed memory system [6], the Hot Pages software caching system [15], and the C-CHARM memory system [13] all introduce these operations to express the array indexes after transformations.

However, the cost of using division and modulo operations is often prohibitive. Despite their suitability for representing various concepts, experienced application programmers avoid them when they care about performance. On the MIPS R10000, for example, a divide operation takes 35 cycles, compared to six cycles for a multiply and one cycle for an add. Furthermore, unlike the multiply unit, the division unit has dismal throughput because it is not pipelined. In compiler optimizations that attempt to improve cache behavior or reduce memory traffic, the overhead from the use of modulo and division operations can potentially negate any performance gained.

Advances in both hardware and software make optimizations on modulo and remainder operations more important today than ever. While modern processors have taken advantage of increasing silicon area by replacing iterative multipliers with faster, non-iterative structures such as Wallace multipliers, similar non-iterative division/modulo functional units have not materialized technologically [16]. Thus, while the performance gap between an add and a multiply has narrowed, the gap between a divide and the other arithmetic operations has either widened or remained the same. In the MIPS family, for example, the ratio of costs of div/mul/add has gone from 35/12/1 on the R3000 to 35/6/1 on the R10000. Similarly, hardware advances such as caching and branch prediction help reduce the cost of memory accesses and branches relative to divisions. From the software side, better code generation, register allocation, and strength reduction of multiplies increase the relative execution time of portions of code that uses division and modulo operations. Thus, in accordance with Amdahl's law, the benefit of optimizing away these operations is ever increasing.

This paper presents optimizations that focus on eliminating division and modulo operations from loop nests where the numerators and the denominators are linear functions of loop induction variables and loop constants. The concept is similar to strength reduction of multiplications. However, a strength reducible multiplication in a loop creates a simple linear data pattern, while modulo and division instructions create values with complex saw-tooth and step patterns. We use algebra, loop iteration space analysis, and integer programming techniques to identify and simplify these patterns. The elimination of division and modulo operations requires complex loop transformations to break the patterns at their

*This research is funded in part by Darpa contract # DABT63-96-C-0036 and in part by an IBM Research Fellowship.

```
for(t = 0; t < T; t++)
  for(i = 0; i < NN; i++)
    A[i%N] = 0;
```

(a) Loop with an integer modulo operation

```
_invt = (NN-1)/N;
for(t = 0; t <= T-1; t++) {
  for(_Mdi = 0; _Mdi <= _invt; _Mdi++) {
    _peeli = 0;
    for(i = N*_Mdi; i <= min(N*_Mdi+N-1,NN-1); i++) {
      A[_peeli] = 0;
      _peeli = _peeli + 1;
    }
  }
}
```

(b) Modulo loop after strength reduction optimization

```
for(t = 0; t < T; t++)
  for(i = 0; i < NN; i++)
    A[i/N] = 0;
```

(c) Loop with an integer division operation

```
_invt = (NN-1)/N;
for(t = 0; t <= T-1; t++) {
  for(_mDi = 0; _mDi <= _invt; _mDi++) {
    for(i = N*_mDi; i <= min(N*_mDi+N-1,NN-1); i++) {
      A[_mDi] = 0;
    }
  }
}
```

(d) Division loop after strength reduction optimization

Figure 1: Two sample loops before and after strength reduction optimizations. The run-time inputs are T=500, N=500, and NN=N*N.

discrete points.

We believe that if the compiler is able to eliminate the overhead of division and modulo operations, their use will become prevalent. Both user code and compiler generated code will benefit. Similar to how strength reduction of multiplications helped the acceptance of early Fortran compilers into the scientific programming community, strength reduction of modulo and division can increase the attractiveness and impact of automatic parallelization and locality optimization.

The algorithms shown in this paper have been effective in eliminating most of the division and modulo instructions introduced by the SUIF parallelizing compiler, Maps, Hot Pages, and C-CHARM. In some cases, they improve the performance of applications by more than a factor of ten.

Related Work An early article by Cocke and Markstein describes one of the optimizations presented in this paper and shows how it complements locality improving array transformations [7]. Other previous work on eliminating division and modulo operations have focused on the case when the denominator is a compile-time constant [1, 12, 14]. In [12], a division with constant denominator is turned into a load-constant, a multiplication, one or two shifts, and two add/subtracts; a modulo with constant denominator is converted into those instructions plus a multiply and a subtract. In contrast, our approach focuses on loop nests, but it only requires that the denominator is loop invariant. In cases where both approaches are applicable, our approach usually leads to more efficient code, with mod cost as low as one add, and div cost as low as an add amortized over the iteration count.

The rest of the paper is organized as follows. Section 2 motivates our work. Section 3 describes the framework for our optimizations. Section 4 presents the optimizations. Section 5 presents results. Section 6 concludes.

2. MOTIVATION

We illustrate by way of example the potential benefits from strength reducing integer division and modulo operations. Figure 1(a) shows a simple loop with an integer modulo operation. Figure 1(b) shows the result of applying our strength reduction techniques to the loop. Similarly, Figure 1(c) and Figure 1(d) show a loop with an integer divide operation before and after optimizations. Figure 2 shows the performance of these loops on a wide range of processors. The results show that the performance gain is universally significant, generally ranging from 4.5x to 45x.[1] The thousand-fold speedup for the division loop on the Alpha 21164 arises because, after the division has been strength reduced, the compiler is able to recognize that the inner loop is performing redundant stores. When the array is declared to be volatile, the redundant stores are not optimized away, and the speedup comes completely from the elimination of divisions. This example illustrates that, like any other optimizations, the benefit of mod/div strength reduction can be multiplicative when combined with other optimizations.

3. FRAMEWORK

Definition 1. *Let $x \in R$, $n, d \in Z$. We define integer operations div, rem, and mod as follows:*

$$\begin{aligned} TRUNC(x) &= \begin{cases} \lfloor x \rfloor & x \geq 0 \\ \lceil x \rceil & x < 0 \end{cases} \\ n \ div \ d &= TRUNC(n/d) \\ n \ rem \ d &= n - d * TRUNC(n/d) \\ n \ mod \ d &= n - d * \lfloor n/d \rfloor \end{aligned}$$

For the rest of this paper, we use the traditional symbols / and % to represent integer divide and integer modulo operations, respectively.

To facilitate presentation, we make the following simplifications. First, we assume that both the numerator and denominator expressions are positive unless explicitly stated otherwise. The full compiler system has to check for all the cases and handle them correctly, but sometimes the compiler can deduce the sign of an expression from its context or its use, *e.g.*, an array index expression. Second, we describe our optimizations for modulo operations, which are equivalent to remainder operations when both the numerators and the divisors are positive.

Most of the algorithms introduced in this paper strength reduce integer division and modulo operations by identifying their value patterns. For that, we need to obtain the value

[1] The speedup on the Alpha is more than twice that of the other architectures because its integer division is emulated in software.

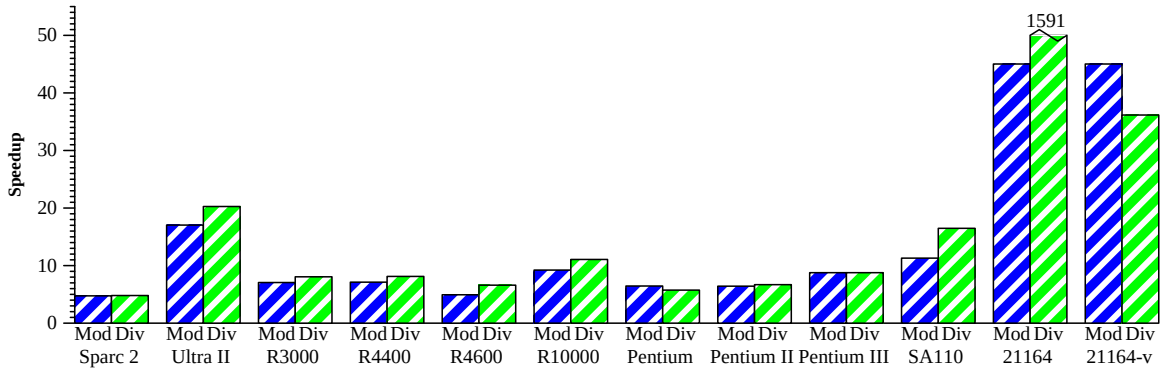

Figure 2: Performance improvement obtained with the strength reduction of modulo and division operations on several machines. See http://cag.lcs.mit.edu/~walt/mdopt/graph.pdf for full tabular results.

ranges of numerator and denominator expressions of the division and modulo operations. We concentrate our effort on loop nests by obtaining the value ranges of the induction variables, since many of the strength-reducible operations are found within loops, and optimizing modulo and division operations in loops has a much higher impact on performance. Finding the value ranges of induction variables is equivalent to finding the iteration space of the loop nests.

First, we need a representation for iteration spaces of the loop nests and the numerator and denominator expressions of the division and modulo operations. Representing arbitrary iteration spaces and expressions accurately and analyzing them is not practical in a compiler. Thus, we restrict our analysis to loop bounds and expressions that are affine functions of induction variables and loop constants. We choose to view the iteration spaces as multi-dimensional convex regions in an integer space [2, 3, 4]. We use systems of inequalities to represent these multi-dimensional convex regions and program expressions. The analysis and strength reduction optimizations are then performed by manipulating the systems of inequalities.

Definition 2. *Assume a p-deep (not necessarily perfectly nested) loop nest of the form:*

FOR $i_1 = \max(l_{1,1}..l_{1,m_1})$ TO $\min(h_{1,1}..h_{1,n_1})$ STEP s_1 DO
  FOR $i_2 = \max(l_{2,1}..l_{2,m_2})$ TO $\min(h_{2,1}..h_{2,n_2})$ STEP s_2 DO
    ....
      FOR $i_p = \max(l_{p,1}..l_{p,m_p})$ TO $\min(h_{p,1}..h_{p,n_p})$ STEP s_p DO
        /* the loop body */

where $v_1, ..., v_q$ are loop invariant, and $l_{x,y}$ and $h_{x,y}$ are affine functions of the variables $v_1, ..., v_q, i_1, ..., i_{x-1}$. We define the context of the k^{th} loop body recursively:

$$\mathcal{F}_k = \mathcal{F}_{k-1} \wedge \left\{ i_k \;\middle|\; \begin{array}{l} \bigwedge_{j=1,...,m_k} i_k \geq l_{k,j} \;\wedge \\ \bigwedge_{j=1,...,n_k} i_k \leq h_{k,j} \end{array} \right\}$$

The loop bounds in this definition contain max and min functions because many compiler-generated loops, including those generated in Optimizations 9 and 10 in Section 4.3.2, produce such bounds.

Note that the symbolic constants $v_1, ..., v_q$ need not be defined within the context. If we are able to obtain information on their value ranges, we include them into the context. Even without a value range, the way the variable is used in an expression (*e.g.,* its coefficient) can provide valuable information on the value range of the expression.

We perform loop normalization and induction variable detection analysis prior to strength reduction so that all the FOR loops are in the above form. Whenever possible, any variable defined within the loop nest is written as affine expressions of the induction variables.

Definition 3. *Given context $\mathcal{F}$ with symbolic constants $v_1, ..., v_q$ and loop index variables $i_1, ..., i_p$, an affine integer division (or modulo) expression within it is represented by a 3-tuple $\langle N, D, \mathcal{F} \rangle$ where N and D are defined by the affine functions: $N = n_0 + \sum_{1 \leq j \leq q} n_j v_j + \sum_{1 \leq j \leq p} n_{j+q} i_j$, $D = d_0 + \sum_{1 \leq j \leq q} d_j v_j + \sum_{1 \leq j \leq p-1} n_{j+q} i_j$. The division expression is represented by N/D. The modulo expression is represented by $N\%D$.*

We restrict the denominator to be invariant within the context (i.e., *it cannot depend on i_p). We rely on this invariance property to perform several loop level optimizations.*

3.1 Expression relation

Definition 4. *Given affine expressions A and B and a context $\mathcal{F}$ describing the value ranges of the variables in the expressions, we define the following relations:*

- *Relation$(A < B, \mathcal{F})$ is true iff the system of inequalities $\mathcal{F} \wedge \{A \geq B\}$ is empty.*
- *Relation$(A \leq B, \mathcal{F})$ is true iff the system of inequalities $\mathcal{F} \wedge \{A > B\}$ is empty.*
- *Relation$(A > B, \mathcal{F})$ is true iff the system of inequalities $\mathcal{F} \wedge \{A \leq B\}$ is empty.*
- *Relation$(A \geq B, \mathcal{F})$ is true iff the system of inequalities $\mathcal{F} \wedge \{A < B\}$ is empty.*

Using the integer programming technique of Fourier-Motzkin Elimination [8, 9, 10, 18, 20], we manipulate the systems of inequalities for both analysis and loop transformation purposes. In many analyses, we use this technique to identify if a system of inequalities is empty, *i.e.,* no set of values for the variables will satisfy all the inequalities. Fourier-Motzkin elimination is also used to simplify a system of inequalities by eliminating redundant inequalities. For example, a system of inequalities $\{I \geq 5, I \geq a, I \geq b, a \geq 10, b \leq 4\}$ can be simplified to $\{I \geq 10, I \geq a, a \geq 10, b \leq 4\}$. In many optimizations discussed in this paper, we create a

new context to represent a transformed iteration space that will result in elimination of modulo and division operations. We use Fourier-Motzkin projection to convert this system of inequalities into the corresponding loop nest. This process guarantees that the loop nest created has no empty iterations and loop bounds are the simplest and tightest [2, 3, 4].

3.2 Iteration count

Definition 5. *Given a loop* FOR $i = L$ TO U DO *with context* $\mathcal{F}$*, where* $L = max(l_1, ..., l_n)$*,* $U = min(u_1, ..., u_m)$*, the number of iterations niter can be expressed as follows:*

$$niter(L, U, \mathcal{F}) = \min\{k | k = u_y - l_x + 1; x \in [1, n]; y \in [1, m]\}$$

The context is included in the expression to allow us to apply the max/min optimizations described in Section 4.4.

4. OPTIMIZATION SUITE

This section describes our suite of optimizations to eliminate integer modulo and division instructions.

4.1 Algebraic simplifications

First, we describe simple optimizations that do not require any knowledge about the value ranges of the source expressions.

4.1.1 Algebraic axioms

Many algebraic axioms can be used to simplify division and modulo operations [11]. Even if the simplification does not immediately eliminate operations, it is important because it can lead to further optimizations.

Optimization 1. *Simplify the modulo and division expressions using the following algebraic simplification rules.* f_1 *and* f_2 *are expressions,* x *is a variable or a constant, and* c*,* c_1*,* c_2 *and* d *are constants.*

$$\begin{array}{rcl}
(f_1x + f_2)\%x & \Longrightarrow & f_2\%x \\
(f_1x + f_2)/x & \Longrightarrow & f_1 + f_2/x \\
(c_1f_1 + c_2f_2)\%d & \Longrightarrow & ((c_1\%d)f_1 + (c_2\%d)f_2)\%d \\
(c_1f_1 + c_2f_2)/d & \Longrightarrow & ((c_1\%d)f_1 + (c_2\%d)f_2)/d \\
 & & +(c_1/d)f_1 + (c_2/d)f_2 \\
(cf_1x + f_2)\%(dx) & \Longrightarrow & ((c\%d)f_1x + f_2)\%(dx) \\
(cf_1x + f_2)/(dx) & \Longrightarrow & ((c\%d)f_1x + f_2)/(dx) + (c/d)f_1
\end{array}$$

4.1.2 Reduction to conditionals

A broad range of modulo and division expressions can be strength reduced into a conditional statement. Since we prefer not to segment basic blocks because it inhibits other optimizations, we attempt this optimization as a last resort.

Optimization 2. *Let* $\langle N, D, \mathcal{F} \rangle$ *be a modulo or division expression in a loop of the following form:*

```
FOR i = 0 TO U DO
  x = N%D
  y = N/D
ENDFOR
```

Let n be the coefficient of i in N, and let $N^- = N - n * i$. Then if $n < D$, the loop can be transformed to the following:

```
_Mdx = N^-%D
_mDy = N^-/D
FOR i = 0 TO U DO
  x = _Mdx
  y = _mDy
  _Mdx = _Mdx + n
  IF _Mdx ≥ D THEN
    _Mdx = _Mdx - D
    _mDy = _mDy + 1
  ENDIF
ENDFOR
```

The code shown is for $N > 0$ and $n > 0$. Other cases can be handled by changing signs appropriately.

4.2 Optimizations using value ranges

The following optimizations not only use algebraic axioms, they also take advantage of compiler knowledge about the value ranges of the variables associated with the modulo and division operations.

4.2.1 Elimination via simple continuous range

Suppose the context allows us to prove that the range of the numerator expression does not cross a multiple of the denominator expression. Then for a modulo expression, we know that there is no wrap-around. For a division expression, the result has to be a constant. In either case, the operation can be eliminated.

Optimization 3. *Given a modulo or division expression* $\langle N, D, \mathcal{F} \rangle$*, if* $Relation(N \geq 0 \wedge D \geq 0, \mathcal{F})$ *and* $Relation(kD \leq N < (k+1)D, \mathcal{F})$ *for some* $k \in Z$*, then the expressions reduce to* k *and* $N - kD$ *respectively.*

Optimization 4. *Given a modulo or division expression* $\langle N, D, \mathcal{F} \rangle$*, if* $Relation(N < 0 \wedge D \geq 0, \mathcal{F})$ *and* $Relation((k+1)D < N \leq kD, \mathcal{F})$ *for some* $k \in Z$*, then the expressions reduce to* k *and* $N + kD$*, respectively.*

4.2.2 Elimination via integral stride and continuous range

This optimization is predicated on identifying two conditions. First, the numerator must contain an index variable whose coefficient is a multiple of the denominator. Second, the numerator less this index variable term does not cross a multiple of the denominator expression. These conditions are common in the modulo and division expressions that are part of the address computations of compiler-transformed linearized multidimensional arrays.

Optimization 5. *Given a modulo or division expression* $\langle N, D, \mathcal{F} \rangle$*, let* i *be an index variable in* $\mathcal{F}$*,* n *be the coefficient of* i *in* N*, and* $N^- = N - n * i$*. If* $n\%D = 0$ *and there exists an integer* k *such that* $kD \leq N^- < (k+1)D$*, then the modulo and division expressions can be simplified to* $N^- - kD$ *and* $(n/D)i + k$*, respectively for* $k > 0$ *and to* $N^- - (k+1)D$ *and* $(n/D)i + k + 1$ *for* $k < 0$*.*

If alignment of the loop, k, is not constant, or cannot be determined one can instead perform a slightly more general transformation.

Optimization 6. *Let* $\langle N, D, \mathcal{F} \rangle$ *be a modulo or division expression in a loop of the following form:*

```
FOR i = 0 TO U DO
  x = N%D
  y = N/D
ENDFOR
```

Let n be the coefficient of i in N and $N^- = N - n * i$. Then if $n\%D = 0$, the loop can be transformed to the following:

$\_Mdx = N^-\%D$
$\_mDy = N^-/D$
FOR $i = 0$ TO U DO
  $x = \_Mdx$
  $y = \_mDy$
  $\_mDy = \_mDy + (n/D)$
ENDFOR

4.2.3 Elimination through absence of discontinuity

Many modulo and division expressions do not create discontinuities within the iteration space. If this can be guaranteed, then the expressions can be simplified. Figure 3(a) shows an example of such an expression with no discontinuity in the iteration space.

Optimization 7. *Let $\langle N, D, \mathcal{F}\rangle$ be a modulo or division expression in a loop of the following form:*

FOR $i = 0$ TO U DO
  $x = N\%D$
  $y = N/D$
ENDFOR

Let n be the coefficient of i in N, $N^- = N - n * i$, and $k = N^-\%D$. For $n > 0$ the loop can be transformed into the following if $Relation(n * niter(0, U, \mathcal{F}) \leq D - k + n - 1, \mathcal{F})$ while for $n < 0$ the loop can be transformed into the following if $Relation(n * niter(0, U, \mathcal{F}) \geq n - k$, the loop can be transformed to the following:

$\_mDy = N^-/D$
$\_Mdx = k$
FOR $i = 0$ TO U DO
  $x = \_Mdx$
  $y = \_mDy$
  $\_Mdx = \_Mdx + n$
ENDFOR

4.3 Optimizations using Loop Transformations

The next set of optimizations perform loop transformations to create new iteration spaces that have no discontinuity. For each loop, we first analyze all its expressions to collect a list of necessary transformations. We then eliminate any redundant transformations.

4.3.1 Loop partitioning to remove one discontinuity

For some modulo and division expressions, the number of iterations in the loop will be less than the distance between discontinuities. But a discontinuity may still occur in the iteration space if it is not aligned to the iteration boundaries. When this occurs, we can either split the loop or peel the iterations. We prefer peeling iterations if the discontinuity is close to the iteration boundaries. This optimization is also paramount when a loop contains multiple modulo and division expressions, each with the same denominator and whose numerators are in the same uniformly generated set [22]. In this case, one of the expressions can have an aligned discontinuity while others may not. Thus, it is necessary to split the loop to optimize all the modulo and division expressions. Figure 4(a) shows an example where loop partitioning eliminates a single discontinuity.

Optimization 8. *Let $\langle N, D, \mathcal{F}\rangle$ be a modulo or division expression in a loop of the following form:*

FOR $i = 0$ TO U DO
  $x = N\%D$
  $y = N/D$
ENDFOR

Let n be the coefficient of i in N and $N^- = N - n * i$. Then if $D\%n = 0$ and $Relation(niter(0, U, \mathcal{F}) \leq D/n, \mathcal{F})$, the loop can be transformed to the following:

$k = N^-\%D$
$\_Mdx = k$
$\_mDy = N^-/D$
$\_cut = min((D - \_kx + n - 1)/n + L - 1, U)$
FOR $i = 0$ TO $\_cut$ DO
  $x = \_Mdx$
  $y = \_mDy$
  $\_Mdx = \_Mdx + n$
ENDFOR
IF $\_Mdx \geq D$ THEN
  $\_Mdx = \_Mdx - D$
  $\_mDy = \_mDy + 1$
ENDIF
FOR $i = \_cut + 1$ TO U DO
  $x = \_Mdx$
  $y = \_mDy$
  $\_Mdx = \_Mdx + n$
ENDFOR

4.3.2 Loop tiling to eliminate discontinuities

In many cases, the value range identified contains discontinuities in the division and modulo expressions. This section explains how to perform loop transformations to move discontinuities to the boundaries of the iteration space. The result is that modulo and division operations can be eliminated or propagated out of the inner loops. Figure 4(b) shows an example requiring this optimization.

When the iteration space has a pattern with a large number of discontinuities repeating themselves, breaking a loop into two loops such that the discontinuities occur at the boundaries of the second loop will let us optimize the modulo and division operations. Optimization 9 adds an additional restriction to the lower bound so that no preamble is needed. Optimization 10 eliminates that restriction.

Optimization 9. *Let $\langle N, D, \mathcal{F}\rangle$ be a modulo or division expression in a loop of the following form:*

FOR $i = 0$ TO U DO
  $x = N\%D$
  $y = N/D$
ENDFOR

Let n be the coefficient of i in N and $N^- = N - n * i$. Then if $D\%n = 0$ and $N^- = 0$, the loop can be transformed to the following:

$\_mDy = N^-/D$
FOR $ii = 0$ TO $U/(D/n)$ DO
  $\_Mdx = 0$
  FOR $i = ii * (D/n)$ TO
      $min((ii + 1) * (D/n) - 1, U)$ DO
    $x = \_Mdx$
    $y = \_mDy$
    $\_Mdx = \_Mdx + n$
  ENDFOR
  $\_mDy = \_mDy + 1$
ENDFOR

Optimization 10. *For the loop nest and the modulo and division statements described in optimization 9, if $D\%n = 0$ then the above loop nest is transformed to the following:*

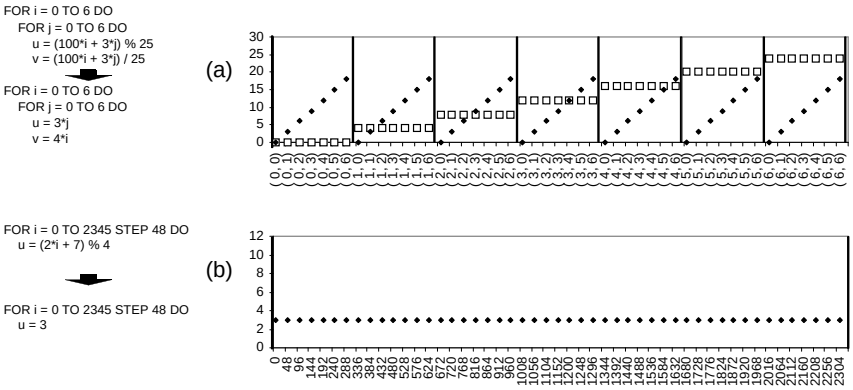

Figure 3: Original and optimized code segments for several modulo and division expressions. The x-axes are the iteration spaces. The y-axes are numeric values. The solid diamonds are values of the modulo expression. The open squares are the values of the division expression. The solid lines represent the original iteration space boundaries. The dash lines represent the boundaries of the transformed loops.

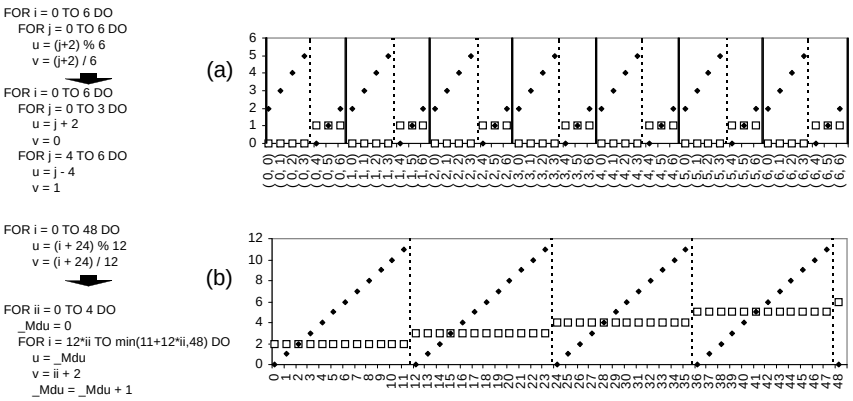

Figure 4: Original and optimized code segments for several modulo and division expressions. See caption for Figure 3.

```
_Mdx = N⁻%D
_mDy = N⁻/D
FOR i = 0 TO _brklb − 1 DO
   x = _Mdx
   y = _mDy
   _Mdx = _Mdx + n
ENDFOR
IF _brklb ≥ 0 THEN
   _mDy = _mDy + 1
ENDIF
_Mdi = (n * _brklb + N⁻)%D
FOR ii = 0 TO U/(D/n) DO
   _Mdx = _Mdi
   FOR i = ii * (D/n) + _brklb
           TO min((ii + 1) * D/n − 1 + _brklb, U) DO
      x = _Mdx
      y = _mDy
      _Mdx = _Mdx + n
   ENDFOR
   _mDy = _mDy + 1
ENDFOR
```

The expression to determine _brklb depends on the sign of both N and n. If $N > 0$ and $n > 0$, $\_brklb = (N^-/D + 1)(D/n) - N^-/n$. If $N < 0$ and $n > 0$, $\_brklb = (-N^-\%D)/n + 1$. If $N > 0$ and $n < 0$, $\_brklb = (-N^-/D + 1) * (D/|n|) * (D/|n|) - (-N^-)/|n|$. If $N < 0$ and $n < 0$, $\_brklb = (N^-\%D)/|n| + 1$.

4.3.3 General loop transformation: single access class

It is possible to transform a loop to eliminate discontinuities with very little knowledge about the iteration space and value ranges. The following transformation can be applied to any loop containing a single arbitrary instance of affine modulo/division expressions.

Optimization 11. *Let $\langle N, D, \mathcal{F} \rangle$ be a modulo or division expression in a loop of the following form:*

```
FOR i = L TO U DO
   x = N%D
   y = N/D
ENDFOR
```

Let n be the coefficient of i in N and $N^- = N - n * i$. Then the loop can be transformed to the following:

```
SUB FindNiceL(L, D, n, N⁻)
   IF n = 0 THEN
      RETURN L
   ELSE
      VLden = ((L * n + N⁻ − 1)/D) * D
      VLbase = L * n + N⁻ − VLden
      NiceL = L + (D − VLbase + n − 1)/n
      RETURN NiceL
   ENDIF
ENDSUB

k = n/D
r = n − k * D

IF r ≠ 0 THEN
   perIter = D/r
   niceL = FindNiceL(L, D, r, N⁻)
   niceNden = (U − niceL + 1)/D
   niceU = niceL + niceNden * D
ELSE
   perIter = U − L
   niceL = L
   niceU = U + 1
ENDIF

modval = (n * L + N⁻)%D
divval = (n * L + N⁻)/D
i = L
FOR i2 = L TO niceL − 1 DO
   x = modval
   y = divval
   modval = modval + r
   divval = divval + k
   i = i + 1
   IF modval ≥ D THEN
      modval = modval − D
      divval = divval + 1
   ENDIF
ENDFOR

WHILE i < niceU DO
   FOR i2 = 1 TO perIter DO
      x = modval
      y = divval
      modval = modval + r
      divval = divval + k
      i = i + 1
   ENDFOR
   IF modval < D THEN
      x = modval
      y = divval
      modval = modval + r
      divval = divval + k
      i = i + 1
   ENDIF
   IF modval ≠ 0 THEN
      modval = modval − D
      divval = divval + 1
   ENDIF
ENDWHILE

FOR i2 = niceU TO U DO
   x = modval
   y = divval
   modval = modval + r
   divval = divval + k
   i = i + 1
   IF modval ≥ D THEN
      modval = modval − D
      divval = divval + 1
   ENDIF
ENDFOR
```

The loop works as follows. First, note that within the loop, N is a function of i only and D is a constant.

For simplicity, consider the case when $n < D$. We observe that if $N(i)\ \%D \in [0, n)$, then there is no discontinuity in the functions $N(i)/D$, $N(i)\%D$ in the range $[i, i + \lfloor D/n \rfloor])$. Furthermore, the discontinuity must occur either after $i + \lfloor D/n \rfloor$ or $i + \lfloor D/n \rfloor + 1$.

Thus, the transformation uses a startup loop that executes iterations of i until $N(i)$ falls in the range $[0, n)$. It then enters a nested loop whose inner loop executes $\lfloor D/n \rfloor$ iterations continuously, then conditionally executes another iteration if the execution has not reached the next discontinuity. This main loop continues for as long as possible, and a cleanup loop finishes up whatever iterations the main loop is unable to execute.

The loop handles the case when $n \geq D$ by using $n\%D$ as the basis for calculating discontinuities.

Note that the FindNiceL subroutine can be shared across optimized loops.

4.3.4 General loop transformation: arbitrary accesses

Finally, the following transformation can be used for loops with arbitrarily many affine accesses.

Optimization 12. *Given a loop with affine modulo or division expressions:*

```
FOR i = L TO U DO
   x_1 = (a_1 * i + b_1) op_1 d_1
   x_2 = (a_2 * i + b_2) op_2 d_2
   ...
   x_n = (a_n * i + b_n) op_n d_n
ENDFOR
```

where op_j is either mod or div, the loop can be transformed into:

```
SUB FindBreak(L, U, den, n, k)
   IF n = 0 THEN
      RETURN U + 1
   ELSE
      VLden = ((L * n + k)/den) * den
      VLbase = L * n + k - VLden
      Break = L + (den - VLbase + n - 1)/n

      RETURN Break
   ENDIF
ENDSUB

FOR j = 1 TO n DO
   k_j = a_j/d_j
   r_j = a_j - k_j - d_j

   val_j[mod] = (a_j * L + b_j)%d_j
   val_j[div] = (a_j * L + b_j)/d_j

   break_j = FindBreak(L, U, d_j, r_j, b_j)
ENDFOR

i = L
WHILE i ≤ U DO
   Break = min(U + 1, {break_j | j ∈ [1, n]})
   FOR i = i TO Break DO
      x_1 = val_1[op_j]
      val_1[mod] = val_1[mod] + r_1
      val_1[div] = val_1[div] + k_1
      x_2 = val_2[op_j]
      val_2[mod] = val_2[mod] + r_2
      val_2[div] = val_2[div] + k_2
      ...
   ENDFOR

   FOR j = 1 TO n DO
      IF Break = break_j THEN
         val_j[mod] = val_j[mod] - d_j
         val_j[div] = val_j[div] + 1
         break_j = FindBreak(i + 1, U, d_j, r_j, b_j)
      ENDIF
   ENDFOR
ENDWHILE
```

Note that the *val*[] associative arrays are used only for the purpose of simplifying the presentation. In the actual implementation, all the op_j's are known at compile time, so that for each expression only one of the array values needs to be computed. Also, note that the FindBreak subroutine can be shared across optimized loops.

The loop operates by keeping track of the next discontinuity of each expression. Within an iteration of the WHILE loop, the code finds the closest discontinuity and executes all iterations until that point in the inner FOR loop. Note, however, that because one needs to perform at least one division within the outer loop to update the set of discontinuities, the more complex control flow in the transformed loop may lead to slowdown if the iteration count of the inner loop is small (possibly due to a small D or a large n).

4.4 Min/Max Optimizations

Some loop transformations, such as those in Section 4.3, produce minimum and maximum operations. This section describes methods for eliminating them. For brevity, we only present the Min optimizations when the Max optimizations can be defined analagously (Optimizations 13, 14, and 17).

4.4.1 Min/Max elimination by evaluation

If we have sufficient information in the context to prove that one of the operand expressions is always greater (smaller) than the rest of the operands, we can use that fact to get rid of the max (min) operation from the expression.

Optimization 13. *Given a min expression $min(N_1, ..., N_m)$ with a context $\mathcal{F}$, if there exists k such that for all $0 \leq i \leq m$, $Relation(N_k \leq N_i, \mathcal{F})$, then $min(N_1, ..., N_m)$ can be reduced to N_k.*

4.4.2 Min/Max simplification by evaluation

Even if we are able to prove few relationships between pairs of operands, it can result in a min/max operation with fewer number of operands.

Optimization 14. *Given a min expression $min(N_1, ..., N_m)$ with a context $\mathcal{F}$, if there exists i, k such that $0 \leq i, k \leq m$, $i \neq k$, $Relation(N_i \leq N_k, \mathcal{F})$ is valid, then $min(N_1, ..., N_m)$ can be reduced to $min(N_1, ..., N_{k-1}, N_{k+1}, ..., N_m)$.*

4.4.3 Division folding

The arithmetic properties of division allow us to fold a division instruction into a min/max operation. This folding can create simpler division expressions that can be further optimized. If further optimizations do not eliminate these division operations, however, the division folding should be un-done to remove potential negative impact on performance.

Optimization 15. *Given an integer division expression with a min/max operation $\langle min(N_1, ..., N_m), D, \mathcal{F}\rangle$ or $\langle max(N_1, ..., N_m), D, \mathcal{F}\rangle$, if $Relation(D > 0, \mathcal{F})$ holds, rewrite min and max as $min(\langle N_1, D, \mathcal{F}\rangle, ..., \langle N_m, D, \mathcal{F}\rangle)$ and $max(\langle N_1, D, \mathcal{F}\rangle, ..., \langle N_m, D, \mathcal{F}\rangle)$ respectively.*

For brevity, we omit the dual optimization when $D < 0$.

4.4.4 Min/Max elimination in modulo equivalence

Since $a \leq b$ does not lead to $a\%c \leq b\%c$, there is no general method for folding modulo operations. If we can prove that the results of taking the modulo of each of the min/max operands are the same, however, we can eliminate the min/max operation.

Optimization 16. *Given an integer modulo expression with a min /max operation $\langle min(N_1, ..., N_m), D, \mathcal{F}\rangle$ or $\langle max(N_1, ..., N_m), D, \mathcal{F}\rangle$ if $\langle N_1, D, \mathcal{F}\rangle \equiv ... \equiv \langle N_m, D, \mathcal{F}\rangle$, then we can rewrite the modulo expression as $\langle N_1, D, \mathcal{F}\rangle$.*

Note that all $\langle N_k, D, \mathcal{F}\rangle$ $(1 \leq k \leq m)$ are equivalent, thus we can choose any one of them as the resulting expression.

4.4.5 Min/Max expansion

Normally min/max operations are converted into conditionals late in the compiler during code generation. However, if any of the previous optimizations are unable to eliminate the mod/div instructions, lowering the min/max will simplify the modulo and division expressions, possibly leading to further optimizations. To simplify the explanation, we describe Optimizations 17 with only two operands in the respective min and max expressions.

Optimization 17. *A mod/div statement with a min operation, $res = \langle min(N_1, N_2), D, \mathcal{F}\rangle$, gets lowered to*

IF $N_1 < N_2$ THEN
  $res = \langle N_1, D, \mathcal{F} \wedge \{N_1 < N_2\}\rangle$
ELSE
  $res = \langle N_2, D, \mathcal{F} \wedge \{N_1 \geq N_2\}\rangle$
ENDIF

5. RESULTS

We have implemented the optimizations described in this paper as a compiler pass in SUIF [21] called Mdopt. We are also in the process of implementing the optimizations in SUIF2 using the Omega integer programming solver package [17]. Mdopt has been used as part of several compiler systems: the SUIF parallelizing compiler [2], the Maps compiler-managed memory system in the Raw parallelizing compiler (Rawcc) [6], the Hot Pages software caching system [15], and the C-CHARM memory system [13]. All those systems introduce modulo and division operations when they manipulate array address computations during array transformations. This section presents some of the performance gain when applying Mdopt to code generated by those systems.

5.1 SUIF Parallelizing Computer

The SUIF parallelizing compiler uses techniques based on linear inequalities to parallelize dense matrix programs [2, 5]. This section uses a hand coded example to illustrate how such a system benefits from Mdopt.

We begin with a five-point stencil code. The stencil code involves iterating over the contents of a two dimensional matrix with a 5-point stencil writing resulting values into another matrix, swapping the matrixes, and repeating. In a traditional parallelizing compiler, the matrix is divided into as many stripes as there are processors, with each stripe assigned to a single processor. Each processor portion of the matrix is a contiguous block of memory, thus ensuring good cache locality. In Figure 5, the *stripe* lines show the performance of this parallelization technique for two matrix sizes on an SGI Origin with MIPS R10000 processors, each with a 4MB L2 cache.

This parallelization technique suffers the drawback that it has a large edge-to-area ratio. Edges are undesirable because the processors need to communicate edge values to their neighbors in each iteration. This interprocessor communication can be reduced by dividing the original matrix into roughly square sections rather than stripes. The *square* lines in Figure 5 show the performance of these square partitions. In this approach, however, each processor's data is no longer a contiguous block of memory. This property increases the likelihood of conflict misses. As a result, *square* actually performs uniformly worse than *stripe*.

Square partitions can be made contiguous through array transformations. This transformation restores cache locality, but it introduces division and modulo operations into address computations. The *datatrans* and *mdopt* lines in Figure 5 show the performance of this approach without and with Mdopt optimizations, respectively. Using a 4096x4096 matrix, little speedup is gained by performing the data transformation and modulo/division optimizations on small processor configurations. This is because each processor's working set is sufficiently large that the computation is memory bound. As the number of processors increase past 32, the working set of each processor begins to fit in the L2 caches. The application becomes CPU bound, so that the benefits of div/mod optimizations becomes visible. For a smaller 2048x2048 matrix, the application is CPU bound for correspondingly smaller configurations, and we see an overall performance gain for up to 48 processors. For larger configurations, however, synchronization costs at the end of each iteration overshadows any performance gains from Mdopt.

5.2 C-CHARM Memory Localization System

The C-CHARM memory localization compiler system [13] attempts to do much of the work traditionally done by hardware caches. The goal of the system is to generate code for an exposed memory hierarchy. Data is moved explicitly from global or off-chip memory to local memory before it is needed and vice versa when the compiler determines it can no longer hold the value locally.

Benchmarks	Speedup
convolution	15.6
jacobi	17.0
median-filter	2.8
sor	8.0

Table 1: Speedup from applying Mdopt to C-CHARM generated code run on an Ultra 5 Workstation.

C-CHARM analyses the reuse behavior of programs to determine how long a value should be held in local memory. Once a value is evicted, its local memory location can be reused. This local storage equivalence for global memory values is implemented with a circular buffer. References are mapped into the same circular buffer, and their address calculations are rewritten with modulo operations. It is these modulo operations that map two different global addresses to the same local address. It is these operations that Mdopt removes.

Table 1 shows the speedup from applying modulo/division optimizations on C-CHARM generated code running on a single processor machine.

5.3 Maps Compiler Managed Memory

Maps is the memory management front end of the Rawcc parallelizing compiler [6], which targets the MIT Raw architecture [19]. It distributes the data in a sequential input program across the individual memories of the Raw tiles. The system low-order-interleaves arrays whose accesses are affine functions of enclosing loop induction variables. That is, for an N-tile Raw machine, the k^{th} element of an "affine" array A becomes the $(k/N)^{th}$ element of partial array A on tile $k\%N$. Mdopt is used to simplify the tile number into a constant, as well as to eliminate the division operations in the resultant address computations.

Table 2 shows the impact of the transformations. It contains results for code targeting a varying number of tiles, from 1 to 32. The effects of the transformations depend on the number of affine-accessed arrays and the computation to data ratio. Because Mdopt plays an essential correctness role in the Rawcc compiler (Rawcc relies on Mdopt to reduce the tile number expressions to constants), it is not possible to directly compare performance on the Raw machine with and without the optimization. Instead, we compile the C sources before and after the optimization on an Ultrasparc workstation, and we use that as the basis for comparison.

The left column of each configuration shows the performance measured in slowdown after the initial low-order interleaving data transformation. This transformation intro-

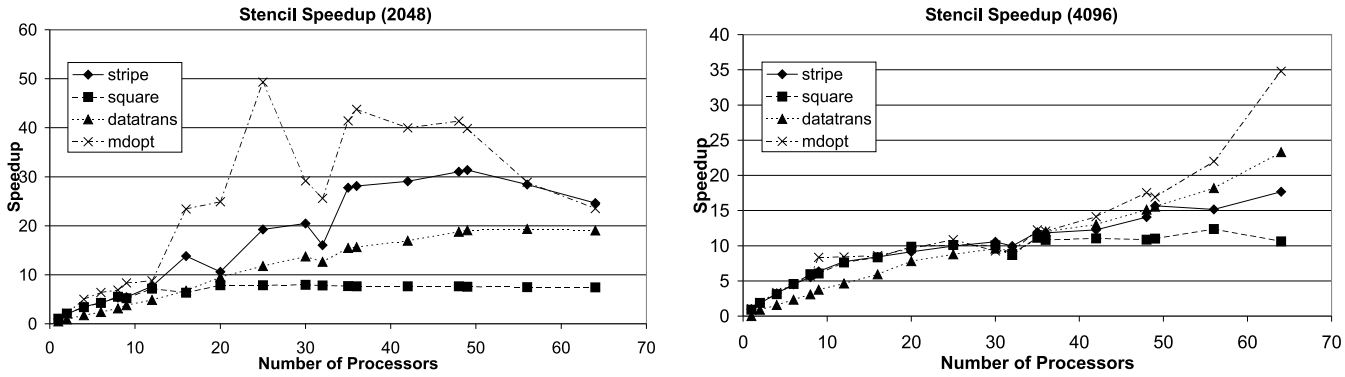

Figure 5: Performance of stencil code using varying parallelization techniques.

Number of Tiles	1		2		4		8		16		32	
Benchmarks	Slow down	Speed up	Slow down	Speed up	Slow down	Speed up	Slow down	Speed up	Slow down	Speed up	Slow down	Speed up
life	1.02	1.00	3.23	2.20	2.86	2.17	3.85	6.03	2.86	19.42	3.57	17.64
jacobi	1.00	1.00	4.76	4.22	8.33	6.51	10.00	3.33	10.00	2.52	33.33	6.44
cholesky	1.00	1.00	3.57	3.62	4.35	4.12	5.00	3.41	5.55	2.54	11.11	1.85
vpenta	1.00	1.00	1.39	1.18	1.92	1.48	2.50	1.98	*	*	*	*
btrix	1.00	1.00	2.94	3.19	3.45	2.26	1.35	1.00	1.25	1.00	1.28	0.96
tomcatv	0.88	1.00	3.13	2.81	4.17	3.19	5.89	7.49	7.14	6.86	*	*
ocean	1.00	1.00	1.37	1.60	1.69	1.70	1.41	2.00	2.44	2.33	2.94	3.82
swim	1.00	1.00	1.00	1.00	1.06	1.00	1.00	1.00	1.00	1.00	1.00	0.95
adpcm	1.10	1.00	1.10	1.00	1.23	1.00	1.10	1.00	1.10	1.00	1.10	1.00
moldyn	1.00	1.03	0.99	1.00	0.99	1.03	1.00	1.00	1.06	1.00	1.14	0.97

Table 2: Performance of Maps code during transformation targeting a varying number of Raw tiles. For each configuration, the left column shows the slowdown from low-order interleaving array transformation. The right column shows the performance recovered when Mdopt optimization is applied. * indicates missing entries because gcc runs out of memory.

duces division and modulo operations and leads to dramatically slower code, as much as 33 times slowdown for 32-way interleaved jacobi. The right column of each configuration shows the speedup attained when we apply Mdopt on the low-order interleaved code. These speedups are as dramatic as the previous slowdown, as much as an 18x speedup for 32-way interleaved life. In many cases the Mdopt is able to recover most of the performance lost due to the interleaving transformation. This recovery, in turn, helps make it possible for the compiler to attain overall speedup by parallelizing the application [6].

6. CONCLUSION

This paper introduces a suite of techniques for eliminating division, modulo, and remainder operations. The techniques are based on number theory, integer programming, and strength-reduction loop transformation techniques. To our knowledge this is the first extensive work that provides modulo and division optimizations for expressions whose denominators are non-constants.

We have implemented our suite of optimizations as a SUIF compiler pass. The compiler pass has proven to be useful across a wide variety compiler optimizations which does data transformations and manipulate address computations. For some benchmarks with high data to computation ratio, an order of magnitude speedup can be achieved.

We believe that the availability of these techniques will make divisions and modulo operations more useful to programmers. Programmers will no longer need to make the painful tradeoff between expressiveness and performance when deciding whether to use these operators. The optimizations will also increase the impact of compiler optimizations that improve data locality or enable parallelization at the expense of introducing modulo and division operations.

Acknowledgments The idea of a general mod/div optimizer was inspired from joint work with Monica Lam and Jennifer Anderson on data transformations for caches. Chris Wilson suggested the reduction of conditionals optimization. Rajeev Barua integrated Mdopt into Rawcc; Andras Moritz integrated the pass into Hot Pages. We thank Jennifer Anderson, Matthew Frank, and Andras Moritz for providing valuable comments on earlier versions of this paper.

7. REFERENCES

[1] R. Alverson. Integer Division Using Reciprocals. In *Proceedings of the Tenth Symposium on Computer Arithmetic*, Grenoble, France, June 1991.

[2] S. Amarasinghe. Parallelizing Compiler Techniques Based on Linear Inequalities. In *Ph.D Thesis, Stanford University. Also appears as Techical Report CSL-TR-97-714*, Jan 1997.

[3] C. Ancourt and F. Irigoin. Scanning Polyhedra with Do Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Williamsburg, VA, Apr. 1991.

[4] M. Ancourt. *Génération Automatique de Codes de Transfert pour Multiprocesseurs à Mémoires Locales*. PhD thesis, Université Paris VI, Mar. 1991.

[5] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages

166–178, Santa Barbara, CA, July 1995.

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

[7] J. Cocke and P. Markstein. Strength Reduction for Division and Modulo with Application to Accessing a Multilevel Store. *IBM Journal of Research and Development*, 24(6):692–694, November 1980.

[8] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[9] G. Dantzig and B. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[10] R. Duffin. On Fourier's Analysis of Linear Inequality Systems. In *Mathematical Programming Study 1*, pages 71–95. North-Holland, 1974.

[11] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, 1989.

[12] T. Granlund and P. Montgomery. Division by Invariant Integers using Multiplication. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[13] B. Greenwald. A Technique for Compilation to Exposed Memory Hierarchy. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1999.

[14] D. Magenheimer, L. Peters, K. Peters, and D. Zuras. Integer Multiplication and Division On the HP Precision Architecture. *IEEE Transactions on Computers*, 37:980–990, Aug. 1988.

[15] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. (LCS-TM-599), Sept 1999.

[16] S. Oberman. *Design Issues in High Performance Floating Point Arithmetic Units.* PhD thesis, Stanford University, December 1996.

[17] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[18] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, Chichester, Great Britain, 1986.

[19] M. B. Taylor. Design Decisions in the Implementation of a Raw Architecture Workstation. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1999.

[20] H. Williams. Fourier-Motzkin Elimination Extension to Integer Programming Problems. *Journal of Combinatorial Theory*, 21:118–123, 1976.

[21] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), Dec. 1996.

[22] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Dept. of Computer Science, Stanford University, Aug. 1992.