Strength Reduction of Integer Division and Modulo Operations

Saman Amarasinghe, Walter Lee, Ben Greenwald *
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.
{saman,walt,beng}@lcs.mit.edu
http://www.cag.lcs.mit.edu/raw

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, array address calculations in data distribution, and cache locality compiler-optimizations. 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 which eliminates division and modulo operations that are functions of loop induction variables and loop constants. The optimizations rely on number theory, 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 which 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, compiler optimizations have many opportunities to simplify code generation by using modulo and division instructions. Today, a few compiler optimizations use these operations for address calculation of transformed arrays. The SUIF parallelizing compiler [2, 5], the Maps compiler-managed memory system [7], the Hot Pages software caching system [18], 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

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

MIT-LCS-TM-600, November 1999

not pipelined. In compiler optimizations which 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 [19]. 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 which uses division and modulo operations. Thus, in accordance with Amdahl's law, the benefit of optimizing away these operations is ever increasing.

We believe that if the compiler is able to eliminate the overhead of division and modulo operations, their use will become prevalent. A good example of such a change in programmer behavior is the shift in the use of multiplication instructions in FORTRAN codes over time. Initially, compilers did not strength reduce multiplies [15, 16]. Therefore, many legacy FORTRAN codes were hand strength reduced by the programmer. Most modern FORTRAN programs, however, use multiplications extensively in address calculations, relying on the compiler to eliminate them. Today, programmers practice a similar laborious routine of hand strength reduction to eliminate division and modulo operations.

This paper introduces optimizations that can eliminate this laborious process. Most of these optimizations concentrate on eliminating division and modulo operations from loop nests where the numerators and the denominators are 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 number theory, loop iteration space analysis, and integer programming techniques to identify and simplify these patterns. The elimination of division and modulo operations require complex loop transformations to break the patterns at their discrete points.

Previous work on eliminating division and modulo operations have focused on the case when the denominator is a compile-time constant [1, 12, 17]. We are not aware of any work on the strength reduction of these operations when the denominator is not a compile-time constant.

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 that by more than a factor of ten.

The result 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. Table 1 and 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

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

```
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.

comes completely from the elimination of divisions. This example illustrates that, like any other optimizations, the benefit of div/mod strength reduction can be multiplicative when combined with other optimizations.

Processor	Clock Speed (MHz)	Integer modulo loop			Integer division loop		
		No opt. Figure 1(a)	Opt. Figure 1(b)	Speedup	No opt. Figure 1(c)	Opt. Figure 1(d)	Speedup
SUN Sparc 2	70	198.58	41.87	4.74	194.37	40.50	4.80
SUN Ultra II	270	34.76	2.04	17.03	31.21	1.54	20.27
MIPS R3000	100	194.42	27.54	7.06	188.84	23.45	8.05
MIPS R4600	133	42.06	8.53	4.93	43.90	6.65	6.60
MIPS R4400	200	58.26	8.18	7.12	56.27	6.93	8.12
MIPS R10000	250	10.79	1.17	9.22	11.51	1.04	11.07
Intel Pentium	200	32.72	5.07	6.45	32.72	5.70	5.74
Intel Pentium II	300	24.61	3.83	6.43	25.28	3.78	6.69
Intel StrongARM SA110	233	48.24	4.27	11.30	43.99	2.67	16.48
Compaq Alpha 21164	300	19.36	0.43	45.02	15.91	0.01	1591.0
Compaq Alpha 21164 (volatile array)	300	19.36	0.43	45.02	15.91	0.44	36.16

Table 1: Performance improvement obtained with the strength reduction of modulo and division operations on several machines. Results are measured in seconds.

3 Framework

Definition 3.1 *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

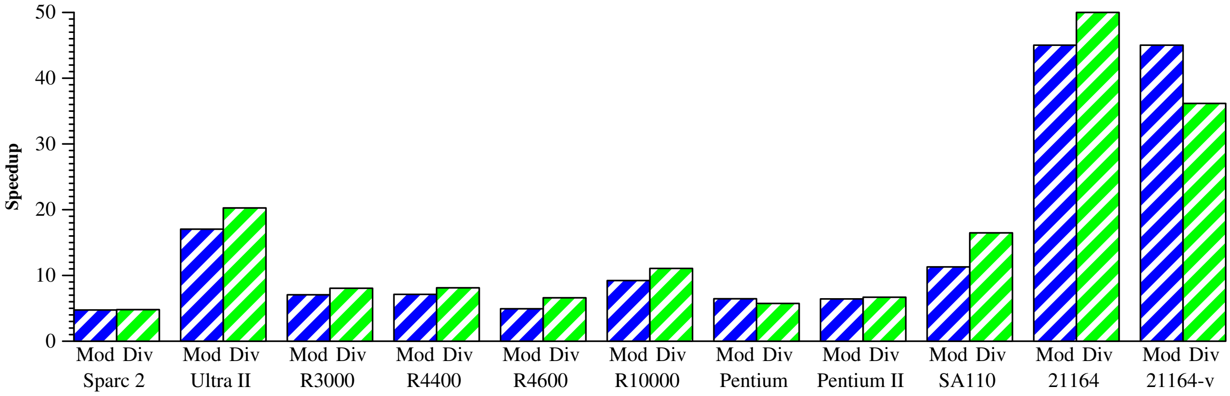


Figure 2: Performance improvement obtained with the strength reduction of modulo and division operations on several machines.

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 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 have 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. In this domain, many representations are possible [6, 14, 20, 21, 24, 25]. 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 3.2 *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 $\min(h_{1,1}..h_{1,n_1})$ DO
  FOR $i_2 = \max(l_{2,1}..l_{2,m_2})$ TO $\min(h_{2,1}..h_{2,n_2})$ DO
    ....
      FOR $i_p = \max(l_{p,1}..l_{p,m_p})$ TO $\min(h_{p,1}..h_{p,n_p})$ 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:

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

The loop bounds in this definition contain max and min functions because many compiler-generated loops, including those generated in Optimizations 10 and 11 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 3.3 *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 in 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 3.4 *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, 22, 26], 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 3.5 *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 division and modulo 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 Number theory axioms

Many number theory 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.*

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

4.1.2 Special case for power-of-two denominator

When the numerator expression is positive and the denominator expression is a power of two, the division or modulo expression can be strength reduced to a less expensive operation.

Optimization 2 *Given a division or modulo expression $\langle N, D, \mathcal{F}\rangle$, if $D = 2^d$ for some constant positive integer d, then the division and modulo expression can be simplified to a right shift $N >> d$ and bitwise and $N\&(D-1)$, respectively.*

4.1.3 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 3 *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 let $N^- = N - n * i$. Then if $n < D$, the loop can be transformed to the following:

```
_Mdx = (n * L + N⁻)%D
_mDy = (n * L + N⁻)/D
FOR i = L TO U DO
  x = _Mdx
  y = _mDy
  _Mdx = _Mdx + n
  IF _Mdx ≥ D THEN
    _Mdx = _Mdx − D
    _mDy = _mDy + 1
  ENDIF
ENDFOR
```

Note that the statement $x = x\%D$ can be simplified to $x = 0$ when $n = 1$.

4.2 Optimizations using value ranges

The following optimizations not only use number theory 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 4 *Given a division or modulo 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 5 *Given a division or modulo expression* $\langle N, D, \mathcal{F}\rangle$*, if* $Relation(N < 0 \wedge D \geq 0, \mathcal{F})$ *and* $Relation(kD < N \leq (k-1)D, \mathcal{F})$ *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 divisor 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 which are part of the address computations of compiler-transformed linearized multidimensional arrays.

Optimization 6 *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.*

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:*

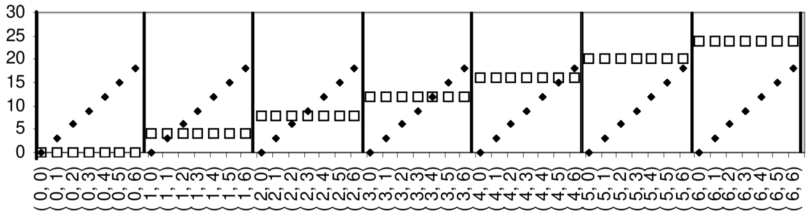
```
FOR i = L TO U DO
   x = N%D
   y = N/D
ENDFOR
```

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

```
_mDy = (n * L + N^-)/D
_Mdx = k
FOR i = L TO U DO
   x = _Mdx
   y = _mDy
   _Mdx = _Mdx + n
ENDFOR
```

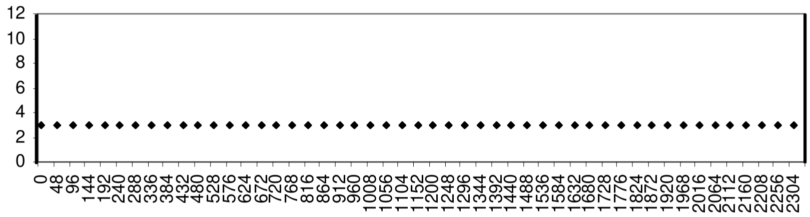
```
FOR i = 0 TO 6 DO
   FOR j = 0 TO 6 DO
      u = (100*i + 3*j) % 25
      v = (100*i + 3*j) / 25
FOR i = 0 TO 6 DO
   FOR j = 0 TO 6 DO
      u = 3*j
      v = 4*i
```

(a)



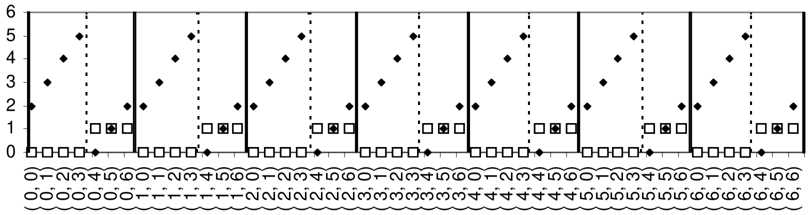
```
FOR i = 0 TO 2345 STEP 48 DO
   u = (2*i + 7) % 4

FOR i = 0 TO 2345 STEP 48 DO
   u = 3
```

(b)



```
FOR i = 0 TO 6 DO
   FOR j = 0 TO 6 DO
      u = (j+2) % 6
      v = (j+2) / 6
FOR i = 0 TO 6 DO
   FOR j = 0 TO 3 DO
      u = j + 2
      v = 0
   FOR j = 4 TO 6 DO
      u = j - 4
      v = 1
```

(c)



```
FOR i = 0 TO 48 DO
      u = (i + 24) % 12
      v = (i + 24) / 12

FOR ii = 0 TO 4 DO
   _Mdu = 0
   FOR i = 12*ii TO min(11+12*ii,48) DO
      u = _Mdu
      v = ii + 2
      _Mdu = _Mdu + 1
```

(d)

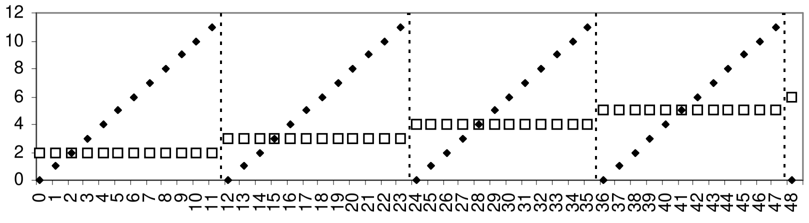


Figure 3: Original and optimized code segments for several modulo and division expressions. The x-axes are the iteration spaces. The y-axes are numeric values. The solid diamonds are values of the modulo expression. The open squares are the values of the division expression. The solid lines represent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

4.2.4 Optimization for non-unit loop steps

If a loop has a step size which is a multiple of the coefficient of the loop index in the numerator expression, the modulo expression is constant within the loop and the division expression is linear. Figure 3(b) provides an example of such an expression.

Optimization 8 *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 STEP S DO
  x = N%D
  y = N/D
ENDFOR
```

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

```
_Mdx = (n * L + N^-)%D
_mDy = (n * L + N^-)/D
FOR i = L TO U STEP S DO
  x = _Mdx
  y = _mDy
  _mDy = _mDy + (S/n)
ENDFOR
```

4.3 Optimizations using Loop Transformations

The next set of optimizations perform loop transformations to create new iteration spaces which 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 a single 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 [28]. 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 3(c) shows an example where loop partitioning eliminates a single discontinuity.

Optimization 9 *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 if $D\%n = 0$ and $Relation(niter(L, U, \mathcal{F}) < n*D, \mathcal{F})$, the loop can be transformed to the following:

```
_kx = (n * L + N^-)%D
_Mdx = _kx
_mDy = (n * L + N^-)/D
_cut = min((D - k + n - 1)/n + L, U)
FOR i = L TO _cut - 1 DO
  x = _Mdx
  y = _mDy
  _Mdx = _Mdx + 1
ENDFOR
_Mdx = _kx
_mDy = _mDy + 1
FOR i = _cut TO U DO
  x = _Mdx
  y = _mDy
  _Mdx = _Mdx + 1
ENDFOR
```

4.3.2 Loop tiling to eliminate discontinuities

In many cases, the value range identified still leads to discontinuities in the division and modulo expressions. This section explains how to strength reduce these expressions by performing loops transformations such that the resulting loop nest will move the discontinuities to the boundaries of the iteration space. Thus, modulo and division optimizations can be completely eliminated or propagated out of the inner loops. Figure 3(d) 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 10 adds an additional restriction to the lower bound so that no preamble is needed. Optimization 11 eliminates that restriction.

Optimization 10 *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 if $D\%n = 0$ and $(n * L + N^-) = 0$, the loop can be transformed to the following:

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

Optimization 11 *For the loop nest and the modulo and division statements described in optimization 10, if $D\%n = 0$ then the above loop nest is transformed to the following form:*

```
_brklb = min((D/n) * ((n * L + N^-)/D + 1) - N/n, U)
_Mdu = (n * L + N^-)%D
_mDv = (n * L + N^-)/D
FOR i = L TO _brklb - 1 DO
  u = _Mdu
  v = _mDv
  _Mdu = _Mdu + 1
ENDFOR
_stu = (n * _brklb + N^-)%D
FOR ii = _brklb/(D/n) TO (U + D/n - 1)/(D/n) DO
  _Mdu = _stu
  _mDv = _mDv + 1
  FOR i = _brklb TO min(ii * D/n + D/n - 1, U) DO
    u = _Mdu
    v = _mDv
    _Mdu = _Mdu + 1
  ENDFOR
ENDFOR
```

4.3.3 General loop transformation for loop access of single 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 12 *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 which 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 for arbitrary loop accesses

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

Optimization 13 *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.

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 14 *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.*

Optimization 15 *Given a max expression $max(N_1, ..., N_m)$ with a context $\mathcal{F}$, if there exists k such that for all $0 \leq i \leq m$, $Relation(N_k \geq N_i, \mathcal{F})$, then $max(N_1, ..., N_m)$ 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 16 *Given a min expression $min(N_1, ..., N_m)$ with a contxt $\mathcal{F}$, 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 $min(N_1, ..., N_{k-1}, N_{k+1}, ..., N_m)$.*

Optimization 17 *Given a max expression $max(N_1, ..., N_m)$ with a context $\mathcal{F}$, if there exists i, k such that $0 \leq i, k \leq m$, $i \neq k$, $Relation(N_k \geq N_i, \mathcal{F})$, then $min(N_1, ..., N_m)$ can be reduced to $max(N_1, ..., N_{k-1}, N_{k+1}, ..., N_m)$.*

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. However, if further optimizations do not eliminate these division operations, the division folding should be un-done to remove potential negative impact on performance.

Optimization 18 *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.*

Optimization 19 *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 $max(\langle N_1, D, \mathcal{F}\rangle, ..., \langle N_m, D, \mathcal{F}\rangle)$ and $min(\langle N_1, D, \mathcal{F}\rangle, ..., \langle N_m, D, \mathcal{F}\rangle)$ respectively.*

4.4.4 Min/Max elimination in modulo equivalence

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

Optimization 20 *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 div/mod instructions, lowering the min/max will simplify the modulo and division expressions, possibly leading to further optimizations. To simplify the explanation, we describe Optimizations 21 and 22 with only two operands in the respective min and max expressions.

Optimization 21 *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

Optimization 22 *A mod/div statement with a max operation, $res = \langle max(N_1, N_2), D, \mathcal{F}\rangle$, gets lowered to*

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

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 times. 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.

5 Results

We have implemented the optimizations described in this paper as a compiler pass in SUIF [27] called Mdopt. This pass has been used as part of several compiler systems: the SUIF parallelizing compiler [2], the Maps compiler-managed memory system in Rawcc, the Raw parallelizing compiler [7], the Hot Pages software caching system [18], 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 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 3: 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. The references which share memory values are mapped into the same circular buffer, and their address calculations are rewritten with modulo operations. It is these modulo operations which map two different global addresses to the same local address. It is these operations we have sought to remove with Mdopt.

Table 3 shows the speedup from applying modulo/division optimizations on C-CHARM generated code running on a single processor machine.

5.2 Maps Compiler Managed Memory System

Maps is the memory management front end of the Rawcc parallelizing compiler [7], which targets the MIT Raw architecture [23]. It distributes the data in an input sequential 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 one 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 introduces 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 [7].

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 work which provide modulo and division optimizations for expressions whose denominators are non-constants.

We have implemented our suite of optimizations in 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.

Acknowledgments

The idea of a general modulo/division 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 the Raw compiler, while 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.

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] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[7] 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.

[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, M.I.T., Department of Electrical Engineering and Computer Science, September 1999.

[14] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[15] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization (part I). *Internat. J. Computer Math*, 11:21–41, 1982.

[16] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization (part II). *Internat. J. Computer Math*, 11:111–126, 1982.

[17] 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.

[18] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for raw microprocessors. (LCS-TM-599), Sept 1999.

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

[20] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[21] C. Rosene. *Incremental Dependence Analysis*. PhD thesis, Dept. of Computer Science, Rice University, Mar. 1990.

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

[23] 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.

[24] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[25] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[26] H. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory*, 21:118–123, 1976.

[27] 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.

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