Loop Quantization: an Analysis and Algorithm

Alexander Aiken*
Alexandru Nicolau†

87-821

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*Supported in part by an IBM fellowship.

†Supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

Abstract

Loop unwinding is a well-known technique for reducing loop overhead, exposing parallelism, and increasing the efficiency of pipelining. Traditional loop unwinding is limited to the innermost loop of a set of nested loops and the amount of unwinding is either fixed or must be specified by the user. In this paper we present a general technique, loop quantization, for unwinding multiple nested loops, explain its advantages over other transformations, and illustrate its practical effectiveness. An abstraction of nested loops is presented which leads to results about the complexity of computing quantizations and an algorithm.

Index Terms-parallel processing, loop unwinding, transformation, Loop Quantization, compilation, compaction

1 Introduction

Loop unwinding is known as an effective technique for improving the utilization of pipelined machines. More recently loop unwinding has emerged as a technique for exploiting fine-grain parallelism within loops—notably for Very Large Instruction Word machines, a class of very tightly coupled multiprocessors [1]. Loop unwinding helps exploit fine-grain parallelism by providing a large number of operations (the unwound loop body) for scheduling by operation-level code transformations such as Trace Scheduling [2] or Percolation Scheduling [3]. The operations in the unwound loop body come from previously separate iterations and are thus freer of the order imposed by the original loop. Inside this unwound loop-body operations may be scheduled for parallel execution subject only to data dependencies. Considerable parallelism too irregular to exploit using traditional methods (e.g., vectorization) may be found in this way.

The basic technique is very simple. The body of the loop and the control code (counter and exit-tests) are replicated a number of times. Figure 1b shows the effect of unwinding the loop in Figure 1a three times. This form of unwinding is usually used for simple **for** loops, but other forms of iteration can be dealt with in similar fashion.

The unwound loop may sometimes be simplified by removing the intermediate tests and jumps introduced by unwinding. This may require compile-time knowledge of the number of iterations executed. For example, if the user knows that the loop in Figure 1a executes a multiple of three times, the extra tests and jumps in Figure 1b may be removed. Alternatively, if the pattern of memory references of the loop can be determined by static analysis (using disambiguation techniques [4,5]), additional memory locations may be allocated to allow extra iterations of the unwound loop to execute safely. Even if no information about the bounds of the loop in Figure 1a is available at compile time, adding two extra elements to array A allows the tests and jumps to be removed. Note that the additional storage required is a function of the number of times the loop is unwound, not of the number of times the original loop is executed. More complex methods for removing tests and jumps from unwound loops exist, including testing the loop bounds at run-time and then executing an appropriately unwound loop. While they introduce some overhead, such methods deal effectively with statically unpredictable loops and references.

The above description surveys current uses of loop unwinding. One of the major disadvantages of current techniques is that multiple nested loops cannot be unwound. This

```
For i = x, y do
    A[i] : = expr(i);
End;
```

(a) Original Loop

```
For i = x, y, 3 Do
    A[i] : = expr(i);
    If i>y Then Goto exit;
    A[i + 1] : = expr(i + 1);
    If i + 1>y Then Goto exit;
    A[i + 2] : = expr(i + 2);

End;
exit:
```

(b) Loop Unwound three times

Figure 1: Simple loop unwinding.

significantly limits the usefulness of loop unwinding, particularly for architectures designed to exploit fine-grain parallelism (i.e., VLIW's [6,7], ROPE [8], Microflow [9] and Alliant [10]). It is often the case that parallelism is found across several nested loops; this situation is illustrated in Section 3. In the rest of this paper we present a technique, called *loop quantization*, that overcomes the problem by allowing correct multiple-loop unwinding for arbitrary nested loops. [1]

Quantization allows the extraction of parállelism that was previously believed to be detectable only at runtime [11]. Furthermore, quantization can extract significant amounts of fine-grain parallelism in cases where higher-level parallelism exploitation methods (i.e., vectorization and loop interchange) do not apply.

Loop Quantization can help achieve significant speedups in scientific code. The main loop of weather code is naturally amenable to quantization, as are the Livermore loops[12] in their nested context. Several of the Livermore loops (e.g., 5,6,11,24) are considered "hazard bound" [13] and do not yield to previous techniques. These loops can be successfully quantized, allowing fine-grain paralellization. Using Percolation Scheduling to exploit the fine-grain parallelism exposed by loop quantization, the speedups achieved in our experiments range from four to six over code produced for the Cray-1 by the CIVIC compiler. The amount of parallelism exposed is often limited only by the hardware resources available. Because quantization rearranges the order of execution of the loop iterations less dramatically than other transformations and because it can expose even irregular fine-grain parallelism, quantization is applicable to a large

[1] Loop unwinding should not be confused with loop jamming (fusion), a transformation that merges the bodies of two loops into a single loop, thus reducing loop overhead.

class of programs. Furthermore, quantized loops naturally map onto parallel architectures with hypercube or cube-connected-cycles topologies (e.g., Cosmic-cube[14], Microflow[9]).

The remainder of this paper is divided into six major sections. Section two describes and presents correctness conditions for loop quantization. Section three develops two examples. Section four presents an abstract representation of loops that captures the information required for quantization. In section five this representation is used to prove a number of properties of loop quantization and to develop an algorithm. The appendix contains proofs of the technical theorems and details of the algorithms.

2 Loop Quantization

Given enough processors, optimal speedups can be achieved if all nested loops are fully unwound. Because no artificial constraints are introduced by the indexing order, limitations are imposed only by data dependencies. Techniques such as Percolation Scheduling can then be applied to exploit the available parallelism. Unfortunately, complete unwinding is not usually feasible because of processor and memory limitations and because loop bounds are not always known at compile time. Still, even if the amount of unwinding is limited, the ability to unwind all loops is necessary. Unwinding the innermost loop exposes parallelism only in that loop. This is not satisfactory, because the parallelism is often available between several of the outer loops.

The basic idea of loop quantization is to unwind a few iterations of all nested loops. The unwinding is constrained by three factors:

1. *Correctness.* Quantization must not alter the execution order of dependent statements.

2. *Available parallelism.* The unwinding should maximize the parallelism exposed.

3. *Space considerations.*

2.1 Formalization of Loop Quantization

Consider n nested loops as shown in Figure 2. Loop n is the innermost; loop 1 is the outermost. A ***loop index*** I_l and an *increment* K_l are associated with each loop l. Values of I_l and K_l are denoted by i_l and k_l respectively. The loops are assumed initially to be normalized to range from 1 to N_l with increments of 1; an algorithm for bringing loops into this normal form is well-known [15].

In the following discussion we assume that the n loops are the only ones affected by the quantization and that indirect references are linear functions of the iteration vectors. (An iteration- or index-vector [16] consists of the setting of the induction variables that uniquely identify an iteration of the nested loops.) We also assume that two references to an array $A[i_1, i_2, \ldots, i_n]$ and $A[j_1, j_2, \ldots, j_n]$ are equal if and only if $i_1 = j_1, i_2 = j_2, \ldots, i_n = j_n$. That is, no ambiguous **equivalence** and **common** statements are used.

To unwind loop i k_i times, the loop-body is duplicated k_i times. In the first duplication of the original body the index I_i is unchanged; in the second, each occurrence of I_i is replaced by $I_i + 1$, and so on, up to $I_i - k_i - 1$. The loop increment K_i is set to k_i. For the next outermost loop, $(i - 1)$, the newly unwound body is itself replicated k_{i-1} times, with indexes I_{i-1} to $I_{i-1} + k_{i-1} - 1$ replacing I_{i-1}. This is essentially equivalent to unwinding all nested loops fully when the upper bounds are $k_1, \ldots . k_n$.

For this unwinding to be useful, it must preserve the semantics of the original loop. Informally, this requires that each statement S executed for some original index-vector value $\bar{i} = (i_1, \ldots, i_n)$ use the same data in the original and unwound loops. In particular, if the statement accesses a value computed in an iteration preceding it (i.e., calculated by a statement with an index vector $\bar{j} \preceq \bar{i}$, where $\preceq$ denotes lexicographic ordering), it should access this value in the unwound loop. The goal is to find a set of loop unwindings such that the quantized loops preserve the semantics of the original loops.

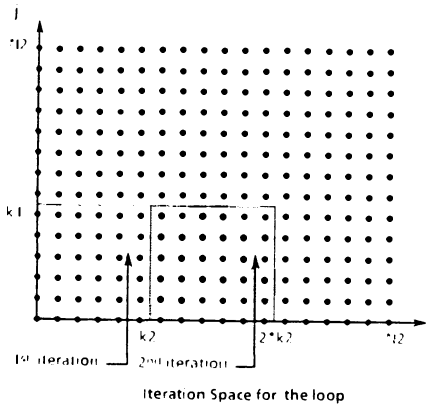
If only the innermost loop is unwound no problems arise; this preserves the execution order of the iterations. However, when other loops are also unwound, an iteration of the quantized loop consists of a *n-dimensional* box B with dimensions k_1 by k_2 by ... by k_n (see Figure 2). All statements in B are executed before the box is shifted (by a "quantum jump") along any of the dimensions. The movement along the n dimensions, while quantized, is still in normal loop order (i.e., first along dimension n, then along dimension $n - 1$, etc.). This obviously alters the order of execution. For example, if loop 1 is unwound a statement S_1 in the original iteration $(i_1 + 1, \ldots, i_n)$ is executed before a statement S_2 in the original iteration $(i_1, \ldots, i_n + k_n)$. The quantization preserves the program's semantics if S_1 does not require the results of S_2 and S_2 does not use information that is altered by S_1. If such dependencies exist the quantization is illegal, although other quantizations may succeed. The execution order of

```
For j = 1, N1 do
   For i = 1, N2 do
     S(i,j)
  End
End;
```

(a) Original loop

```
For j = 1, N1, k1 do
    For i = 1, N2, k2 do
        S(i,j)
        S(i + 1,j)
        ......
        S(i + k2 - 1, j)

        S(i,j + 1)
        S(i + 1,j + 1)

        S(i + k2 - 1, j + 1)

   ..................................

        S(i,,J + k1 - 1)
        ......
        S(i + k2-1,j + k1-1)
    End
End
```

(b) Quantized loop



(c) Quantum box (k1 by k2)

Figure 2: Sample two-dimensional loop quantization.

dependent statements in the same iteration of a quantized loop is preserved by quantization.

If a quantization is illegal because of conflicting references, it may be possible to increase the size of the "box" to include any dependent references. However, this may require unwinding a loop fully. In many applications one loop—of several nested loops—is executed relatively few times; hence, full unwinding on that loop may be feasible. In cases where full unwinding is not practical, limited unrollings must be chosen to ensure correctness.

2.2 Loop Quantization Conditions

2.2.1 Dependencies

Assuming that useless writes to memory are eliminated by dead-code removal, two types of dependencies may be violated as a result of loop quantization: [2]

1. **Write-before-Read:** A memory location is written in iteration $\bar{i}$, and in iteration $\bar{j} > \bar{i}$ the same memory location is read.

2. **Write-after-Read:** A memory location is read in iteration $\bar{i}$, and in iteration $\bar{j} > \bar{i}$ the same memory location is written.

The execution order of such dependent statements must be preserved to ensure correctness. The order of independent statements is irrelevant.

[2] Useless writes arise because of *output dependencies* [17]. Our techniques apply directly to output dependencies that cannot be eliminated; we have made this simplification only for notational brevity.

To determine whether a dependency exists between two statements and to establish the dependency type, reads in one statement are compared with writes in the other. Because scalar variable accesses are invariant with respect to loops, they cannot affect the semantic correctness of the quantization. Furthermore, dependencies between references in the same iteration are not affected by quantization. Thus, it is sufficient to consider only dependencies caused by indirect references occurring in different iterations; these are the loop-carried dependencies of Kuhn [16].

Techniques for determining (to the extent possible at compile time) whether two indirect references might access the same memory location may be found in [4,5]. For notational simplicity we assume that index references are reduced to the primitive form $(a_1 i_1 + b_1, \ldots, a_n i_n + b_n)$, where $\bar{i}$ is the iteration vector and the a_l's and b_l's are constants. The following discussion applies to arbitrary array indexes.

2.2.2 Quantization Conditions

Given two conflicting indirect references $A[a'_1 i_1 + b'_1, \ldots, a'_n i_n + b'_n]$ and $A[a''_1 j_1 + b''_1, \ldots, a''_n j_n + b''_n]$, we can express the second iteration vector $\bar{j}$ as a function of the first iteration vector $\bar{i}$ by

$$\bar{j} = (\frac{a'_1 i_1 + (b'_1 - b''_1)}{a''_1}, \ldots, \frac{a'_n i_n + (b'_n - b''_n)}{a''_n}) = (a_1 i_1 + b_1, \ldots, a_n i_n + b_n) = (j_1, \ldots, j_n)$$

for values of the i_l's such that the j_l's are all integers. This must be possible; otherwise, there cannot be a conflict.

Equating $\bar{i}$ and $\bar{j}$ element-wise from left to right, we may determine whether $\bar{i} < \bar{j}$, or $\bar{i} > \bar{j}$. For a quantization to be legal, we must ensure that the order of the conflicting references is preserved by the quantization.

Definition 2.1 Let the *lower bound* of a box B be the index vector of the quantized loop for which the original loop iteration with index vector $\bar{i}$ is executed. Assume that original iteration $\bar{i}$ occurs in a box with lower bound $\overline{L} = (l_1, \ldots, l_n)$ and original iteration $\bar{j}$ occurs in a box with lower bound $\overline{L'} = (l'_1, \ldots, l'_n)$. We say the quantization is legal if one of the following conditions holds:

$$\textbf{1. if } \bar{i} > \bar{j} \textbf{ then } \overline{L} \geq \overline{L'} \text{ or } \textbf{2. if } \bar{i} < \bar{j} \textbf{ then } \overline{L} \leq \overline{L'}$$

Because the i_e's runs from 1 to N_e and the new indexes run from 1 to N_e by increments of k_e, we obtain:

$$\overline{L} = (\lfloor \frac{i_1 - 1}{k_1} \rfloor k_1 + 1, \ldots, \lfloor \frac{i_n - 1}{k_n} \rfloor k_n + 1), \text{ and } \overline{L'} = (\lfloor \frac{j_1 - 1}{k_1} \rfloor k_1 + 1, \ldots, \lfloor \frac{j_n - 1}{k_n} \rfloor k_n + 1).$$

As with the original iterations, the lower bounds of the quantized boxes are ordered lexicographically. That is, the box associated with $\overline{L}$ is executed before the box associated with $\overline{L'}$ if and only if $\overline{L} < \overline{L'}$. Using *symbolic* element-wise comparison we can determine if either of the two conditions above is satisfied.[3]

2.3 The Use of Loop Quantization Conditions

The conditions presented above may be used in several ways. The simplest is to let the system perform a prespecified unwinding using some fixed k_e's, modified by range analysis of the bounds of arrays and loops. A deficiency of this approach is that it does not provide any information about what k_e's may be appropriate or how to unwind so as to best utilize the space we trade for speed.

A more reasonable approach, which we are incorporating into our percolation scheduling compiler, improves efficiency and applies even when loop bounds are unknown. We notice that $a > b$ implies $\lfloor \frac{a}{k} \rfloor \geq \lfloor \frac{b}{k} \rfloor$. Furthermore, $a - b \geq k$ implies $\lfloor \frac{a}{k} \rfloor > \lfloor \frac{b}{k} \rfloor$. These rules can be used to compare $\overline{L}$ and $\overline{L'}$ symbolically, without computing specific values for each iteration. This can also help in picking the right unwinding (i.e., pick $k_e \leq k$). Doing such symbolic comparison, we may determine that:

- $\overline{L} \geq \overline{L'}$. If this is the condition needed to ensure the legality of the transformation, the quantization may be performed, using arbitrary k_e's.
- $\overline{L} \leq \overline{L'}$. If this is the condition needed to ensure the legality of the transformation the quantization may be performed using arbitrary k_e's.
- *Undetermined situation.* This occurs if there is no strict ordering; none of $<$, $>$, or $=$ hold between $\overline{L}$ and $\overline{L'}$, and there are some $\leq$'s and $\geq$'s between various elements. For example, for $\overline{L} = (\lfloor \frac{a+1}{k_1} \rfloor, \lfloor \frac{b}{k_2} \rfloor)$ and $\overline{L'} = (\lfloor \frac{a}{k_1} \rfloor, \lfloor \frac{b+1}{k_2} \rfloor)$ we have $\lfloor \frac{a+1}{k_1} \rfloor \geq \lfloor \frac{a}{k_1} \rfloor$ and $\lfloor \frac{b}{k_2} \rfloor \leq \lfloor \frac{b+1}{k_2} \rfloor$. This is undetermined, because

[3] In this context symbolic comparison is simply proving statements about the relationship between two (linear) functions. For instance, if $f_1(i,j) = 2j + 3i + 4$ and $f_2 = i + 3$, our system can prove $\forall i, j > 0 : f_1(i,j) > f_2(i)$.

$$\overline{L} = \overline{L'} \text{ when } a = 1,\ k_1 = k_2 = 3,\ b = 1;$$
$$\overline{L} > \overline{L'} \text{ when } a = 2,\ k_1 = k_2 = 3,\ b = 1;$$
$$\overline{L} < \overline{L'} \text{ when } a = 1,\ k_1 = k_2 = 3,\ b = 2.$$

In this example correctness may be ensured by not unwinding on the first dimension ($k_1 = 1$). This preserves the order of references because $\lfloor \frac{a+1}{1} \rfloor > \lfloor \frac{a}{1} \rfloor$. If k_1 is greater than one, than the second dimension (on b) must be fully unwound.

When a conflict occurs, the system compares $\overline{L}$ and $\overline{L'}$ and decides which dimension(s) (if any) may be fully unwound to satisfy the conditions. For example, if there is a value e such that for all i where $i < e$, $l_i \geq l'_i$ and $l_e \leq l'_e$, then the safety of quantization is undetermined. If full unwinding on dimension e is feasible, doing so forces $l_e = l'_e$ because for any original iterations i_e and j_e, $i_e \leq N_e$ and $j_e \leq N_e$. That is, for all i_e and j_e, $l_e = l'_e$. The conflict on the e^{th} dimension is eliminated because all potentially conflicting references are in the same box.

There is no reason to unwind on a dimension if dependencies exist on that dimension that prevent software pipelining or parallelization. This indicates that there is no possible performance gain for the statements under consideration. To decide if the unwinding will achieve any speedup, all statements in the loop body must be considered. Even when no major overlap can occur, exit-tests might still be eliminated, and initialization and tests from several original iterations might be done in parallel.

Quantization may be hindered by the presence of arbitrary conditional jumps in the loop body that limit the ability to disambiguate dependencies at compile time and to accurately evaluate the run-time speedup achieved by a given unwinding. The accuracy can be significantly improved by the use of conditional-jump probability information. Such information can be obtained by the system using test runs or analysis, or it can be supplied by the user. We have found such information to be easily available in typical scientific code.

3 Examples

3.1 Example 1

Consider how loop quantization techniques deal with recurrences. A simple program is shown in Figure 3. Transformations such as loop interchange or vectorization do not apply here.

```
   Do i=1,n
   Do j=1,n
(1)  T1=X[i+16,j];
(2)  T2=X[i+1,j];    /* Notice that statements (2 and 4) form a recurrence */
(3)  T1=T1+T2;
(4)  X[i+1,j+1]=T1;
   Od; Od;
```

Figure 3: Original recurrence code.

For example, in the original loops, $X[17,2]$ is read in iteration $(i = 1, j = 2)$, and written in iteration $(i = 16, j = 1)$. Because the loop on j is innermost, iteration $(i = 1, j = 2)$ occurs before $(i = 16, j = 1)$; a read before a write. Reversing the loops will write $X[17,2]$ in iteration $(j = 1, i = 16)$ and read it in iteration $(j = 2, i = 1)$; the write occurs before the read, which is incorrect. Vectorization is infeasible, even if expansion and loop distribution are used. The first statement can be vectorized, but this only reduces the execution from $4 * n^2$ to $3 * n^2 + 1$ steps, even for a processor array of size n. This is wasteful if n is large. A similar speedup ($3 * n^2$ execution steps) can be obtained on a multiprocessor by running the first two statements in parallel; attempting to distribute the loop on a traditional multiprocessor with more than two processors will incur very high communication overhead due to the tight dependencies between the iterations.

The loops may be quantized by unwinding on both i and j, which exposes enough fine-grain parallelism to keep the available processors busy (e.g., in a VLIW machine, or any other type of tightly coupled multiprocessor). Quantization succeeds because it preserves the relative order of execution of the critical statements inside the "box". By comparing

$$(1)\ \ T1 := X[i + 16, j]$$

and

$$(4)\ \ X[i' + 1, j' + 1] := T1;$$

the disambiguator can determine that conflicts between iteration $\overline{h} = (i, j)$ and $\overline{h'} = (i', j')$

```
Do i=1,n,15
Do j=1,n,15
  T1=X[i+16,j];       /* j+0, i+0  */
  T2=X[i+1,j];
  T1=T1+T2;
  X[i+1,j+1]=T1;
  .................
  T29=X[i+2,j+15];   /* j+15,i+0  */
  T30=X[i+1,j+15];
  T29=T29+T30;
  X[i+1,j+16]=T29;

........................

  T1=X[i+31,j];       /* j+0, i+15 */
  T2=X[i+1,j];
  T1=T1+T2;
  X[i+16,j+1]=T1;
  ..................
  T29=X[i+31,j+15];   /* j+15,i+15 */
  T30=X[i+16,j+15];
  T29=T29+T30;
  X[i+16,j+16]=T29;
Od; Od;
```

Figure 4: Same recurrence with quantized unwinding.

can occur when $i' = i - 15$ and $j' = j + 1$. Since $i > i'$, it follows that $\overline{h} > \overline{h'}$ and thus

$$\overline{L} = (\lfloor\frac{i-1}{K_i}\rfloor K_i + 1,\ \lfloor\frac{j-1}{K_j}\rfloor K_j + 1)$$

must be greater or equal to

$$\overline{L'} = (\lfloor\frac{i-16}{K_i}\rfloor K_i + 1,\ \lfloor\frac{j}{K_j}\rfloor K_j + 1)$$

if quantization is to be allowed. In general this is not the case, as

$$\lfloor\frac{i-1}{K_i}\rfloor \geq \lfloor\frac{i-16}{K_i}\rfloor$$

but

$$\lfloor\frac{j-1}{K_j}\rfloor \leq \lfloor\frac{j}{K_j}\rfloor$$

and thus there is no definite ordering for arbitrary K_i and K_j. However, because $i - i' = 15$, choosing $K_i = 15$ ensures that

$$\lfloor\frac{i-1}{15}\rfloor > \lfloor\frac{i-16}{15}\rfloor$$

and therefore $\overline{L} > \overline{L'}$. K_j has no influence in this case; any value chosen for it preserves correctness. For this example we assume that $K_j = 15$ as well. The other dependency that must be examined is between

$$(2)\ \ T2 := X[i+1, j]$$

and

$$(4)\ \ X[i'+1, j'+1] := T1;$$

where $\overline{h} = (i, j)$, $\overline{h'} = (i' = i, j' = j - 1)$, and thus $\overline{h} > \overline{h'}$. Since $i = i'$ and $j > j'$, it follows that $\overline{L} \geq \overline{L'}$ allowing semantically correct quantization. The resulting loop is shown in Figure 4.

Our system will decide that the 15 unwindings along i (i to $i + 14$) are independent of one another and can be done in parallel. Applying folding and balancing (tree height reduction [17,18,19]) techniques to each such group (for j to $j+14$), each group can, resources permitting, be executed in 6 steps: 1 for loading, 4 for computing sums and storing, and 1 for storing the last results (see Figure 5). To fully exploit this parallelism, 16 processors are needed per

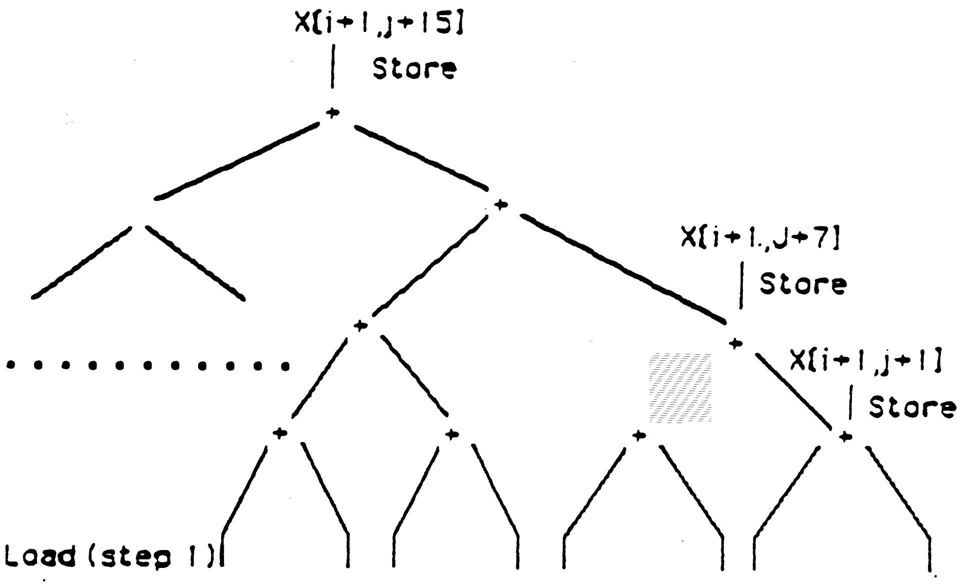


Figure 5: Balanced tree for i, j..j+14

unwinding on i, for a total of 15*16 processors[4]. The speedup achieved is from $4 * n^2$ steps to $6 * (n/k_1) * (n/k_2)$. (In this example, $k_1 = k_2 = 15$.) If more resources are available, the quantization may be greater, resulting in even more dramatic speedups.

3.2 Example 2

This example illustrates how Loop Quantization can expose parallelism previously observable only at runtime [11].

Consider the loop in Heuft and Little's example, shown in Figure 6. To simplify the discussion we assume that intermediate exit-tests have been removed by any of the methods described above. The upper bound for each of the nested loops is nine in the original example; because quantization is trivial for such a loop, we have made the bounds of the outer two loops arbitrary. In the section on *mitred quantization* we show how the example may be quantized even if the inner loop has unknown bounds.

The tests described in section 2.3 show that the loop can be quantized safely. The details of the quantization of this loop are similar to those of the previous example and are omitted. Based on symbolic analysis, the system can determine a correct and effective quantization

[4]By reducing the speedup slightly, from 6 to 8 time steps per unwinding of i, we could do with only 15*8 processors by adding 2 extra steps for initial and intermediate storage of results.

```
      DO 100 I1 = 0, N
      DO 100 I2 = 0, N
      DO 100 I3 = 0, 9
S1:      A(I1,I2+1,I3] = B(I1,I2,I3+2) * C(I1,I2) + U(I1,I2) * V(I1,I2)
S2:      B(I1+1,I2,I3) = A(I1,I2,I3+6)* D(I1,I3)
100: CONTINUE
```

Figure 6: Sample loop from [HeuLitt82]

subject to the data-dependencies present and the resources available. Assuming that enough processors are available[5] the quantization shown in Figure 7 yields optimal speedups.

4 An Abstraction of Nested Loops

The remainder of this paper investigates properties of loop quantization. In this section an abstract representation of loops is developed; the abstraction captures information about loop-carried dependencies [16]. Our representation is quite general and is particularly powerful when applied to loop quantization. We prove a number of properties of loop quantization using the abstraction.

The final section uses the abstraction to develop an algorithm for computing a *strict* quantization. A quantization is said to be strict if all the unwound copies of the original loop body are data-independent for all possible executions of the quantized loop body; the unwound iterations can always be executed in parallel. Thus, the compacted version of a strictly quantized loop may be represented as a list of unwindings and the compacted body of the original loop—no explicit unwinding is necessary. Note that a strict quantization is not necessarily vectorizable; there may be dependencies between iterations of the original loop that occur in different iterations of the quantized loop.

In what follows, L is a set of n nested loops, A is an n-dimensional array, and $A[f_1, \ldots, f_n]$ is a reference to A where f_j is an expression for the jth subscript of A using iteration variable i_j. We assume for simplicity that all arrays have the same number of

[5]Three hundred are required for this example, under the same assumptions as in [11]

```
      DO 100 I1 = 0, N, 9
      DO 100 I2 = 0, N, 9
      DO 100 I3 = 0, 9, 9
S1:      A(I1,I2+1,I3] = B(I1,I2,I3+2) * C(I1,I2) + U(I1,I2) * V(I1,I2)
S2:      B(I1+1,I2,I3) = A(I1,I2,I3+6)* D(I1,I3)
      ..............................................
S1:      A(I1+9,I2+10,I3+9] = B(I1+9,I2+9,I3+11) * C(I1+9,I2+9)
                            + U(I1+9,I2+9) * V(I1+9,I2+9)
S2:      B(I1+10,I2+9,I3+9) = A(I1+9,I2+9,I3+15)* D(I1+9,I3+9)
100: CONTINUE

S1 in iteration (i1,i2,i3) depends on S2 from iteration (i1-1,i2,i3+2)
S2 in iteration (i1,i2,i3) depends on S1 from iteration (i1,i2-1,i3+6)
```

Figure 7: Quantized Loop and Dependency Pattern

dimensions.

The abstraction consists of a *comparison* table and a *distance* table for L. The comparison table captures the ordering of read and write references across iterations. The distance table describes the "distance" (in numbers of iterations) between conflicting references.

Comparison tables are based on the "direction vectors" of Wolfe [15]. Comparison tables have also been defined independently by Kennedy [20]; he uses "direction matrices" to explore vectorization. Distance vectors (also called "difference vectors") were originally defined by Lamport [21]. Cytron makes use of what are essentially distance tables to develop an algorithm for computing delays in DO-ACROSS loops [22].

4.1 Comparison Tables

Definition 4.2 The *time* at which an element $[j_1, \ldots, j_n]$ of array A is addressed by the reference $A[f_1, \ldots, f_n]$ is the tuple $\langle f_1^{-1}(j_1), \ldots f_n^{-1}(j_n)\rangle$, if all of the inverses exist. Times are ordered lexicographically.

Loop quantization requires that the read and write references of an array in L be compared; we present a concise notation for comparing references. We assume that the upper and lower bounds of index variables are unknown, and that array references contain no constant subscripts. Thus, the inverses in Definition 4.2 are always well-defined. These restrictions are included to make the development concise; our techniques can be easily extended to accommodate constant references and knowledge of upper and lower loop bounds.

A quantization is correct if and only if it preserves the order of reads and writes for each memory location. We begin, therefore, by defining a notation for discussing this order for a particular location.

Definition 4.3 Let $A[r]$ and $A[w]$ be a read and write reference (respectively) of a one-dimensional array A in L. Let p be the pair $\langle r, w\rangle$. Then x^p, the order of reads and writes at location x with respect to p, is:

$$x^p = \begin{cases} =_o & \Leftrightarrow \quad r^{-1}(x) = w^{-1}(x) \wedge I \\ \neq_o & \Leftrightarrow \quad r^{-1}(x) \notin \mathcal{N} \vee w^{-1}(x) \notin \mathcal{N} \\ >_o & \Leftrightarrow \quad r^{-1}(x) > w^{-1}(x) \wedge I \\ <_o & \Leftrightarrow \quad r^{-1}(x) < w^{-1}(x) \wedge I \end{cases}$$

where I is the predicate $r^{-1}(x) \in \mathcal{N} \wedge w^{-1}(x) \in \mathcal{N}$

If x^p is $=_o$ then x is both read and written in the same iteration. The relation $<_o$ specifies x is read before it is written, $>_o$ that it is written before it is read. $\neq_o$ indicates that there is no conflict at x.

The generalization to multi-dimensional arrays is straightforward.

Definition 4.4 Let $A[r_1, \ldots, r_n]$ and $A[w_1, \ldots, w_n]$ be a read and write reference of A in L. Let p be the tuple $\langle p_1, \ldots, p_n\rangle$ where $p_k = \langle r_k, w_k\rangle$. If $x = \langle x_1, \ldots, x_n\rangle$ is a location in array A, then $x^p = \langle x_1^{p_1}, \ldots, x_n^{p_n}\rangle$.

Having categorized the possible read/write orderings for a pair of references with respect to a single location, we can express the possible read/write orderings for a pair of references over all possible locations.

Definition 4.5 Let $B[r]$ and $B[w]$ be a read and a write reference (respectively) of a one-dimensional array B in L. $B[r]$ and $B[w]$ are compared as follows:

$$\delta(r,w) = \begin{cases} 1. & \neq_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w\rangle} = \neq_o \\ 2. & =_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w\rangle} = =_o \vee Q \\ 3. & >_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w\rangle} = >_o \vee Q \\ 4. & <_\delta & \text{if } \forall x \in \mathcal{N}\ x^{\langle r,w\rangle} = <_o \vee Q \\ 5. & ?\!=_\delta & \text{if } \exists x,y,z \in \mathcal{N} \text{ s.t. } x^{\langle r,w\rangle} = >_o \wedge y^{\langle r,w\rangle} = <_o \wedge z^{\langle r,w\rangle} = =_o \\ 6. & ?_\delta & \text{if } \exists x,y \in \mathcal{N} \text{ s.t. } x^{\langle r,w\rangle} = >_o \wedge y^{\langle r,w\rangle} = <_o \end{cases}$$

where Q is the predicate $x^{\langle r,w\rangle} = \neq_o$. The value of $\delta(r,w)$ is the lowest numbered case which applies.

The order notation has been used in the definition for clarity. In practice, standard diophantine equation analysis may be used to compute $\delta(r,w)$. (It is easily verified that exactly one case holds for all r and w.) We will frequently write $r{>_\delta}w$ for $\delta(r,w) = >_\delta$.

Elements are read and written in the same iteration if $r{=_\delta}w$. The relation $\neq_\delta$ is included for completeness; clearly there is no dependency between two references if $r{\neq_\delta}w$. We assume that all such read/write pairs are eliminated from consideration. The relation $r{<_\delta}w$ indicates elements are read before being written. The interpretation of $r{>_\delta}w$ is symmetric. $r?_\delta w$ holds if and only if for some elements y and z, y is read before it is written $(r^{-1}(y) < w^{-1}(y))$ and z is written before it is read $(r^{-1}(z) > w^{-1}(z))$. In this case no *a priori* order of references can be established. The relation $r?{=_\delta}w$ is included to distinguish the case where references can occur in either order and there is an element u where $r^{-1}(u) = w^{-1}(u)$.

This case analysis is similar to, but more refined than, the method of Wolfe [15]. We have added the relations $?_\delta$ and $?{=_\delta}$. We use the more complex analysis because our methods apply even if $r?_\delta w$ or $r?{=_\delta}w$. The inclusion of $?_\delta$ and $?{=_\delta}$, however, greatly complicates the technical presentation. We have chosen to omit these from most of what follows; the extensions necessary to accommodate $?_\delta$ and $?{=_\delta}$ are included in the appendix.

The following definition extends the comparison of references to multi-dimensional arrays.

Definition 4.6 Let $A[r_1,\ldots,r_n]$ and $A[w_1,\ldots,w_n]$ be a read and write reference of A in L. Then

$$\delta(A[r_1,\ldots,r_n], A[w_1,\ldots,w_n]) = \langle \delta(r_1,w_1),\ldots,\delta(r_n,w_n)\rangle$$

This is a comparison tuple.

For example, if the statement $A[x, y-4, z] := A[x+3, 2y+1, z]$ appears in L, the comparison tuple for this read/write reference pair is $\langle <_\delta \; ?_\delta \; =_\delta \rangle$.

Definition 4.7 Let $c = \langle c_1, \ldots, c_n \rangle$ be a comparison tuple, where each c_i is $>_\delta$, $<_\delta$, or $=_\delta$. Let j be the least index such that c_j is not $=_\delta$.

If there is such a j then c_k is a *defining position* for c if $c_k = c_j \wedge \forall z < k, c_z = =_\delta \vee c_z = c_j$.

If the is no such j, then $n+1$ is taken to be c's only defining position.

A defining position of a comparison tuple is an entry that determines the lexicographic ordering of read and write references. For instance, let $c = \langle =_\delta \; <_\delta \; =_\delta \; <_\delta \; >_\delta \rangle$. The defining positions are c_2 and c_4; this comparison tuple indicates that locations are read before being written. Intuitively, any legal quantization must preserve this order by not unwinding "too much" on one of c_2 or c_4. This will be formalized in the next section.

Definition 4.8 A comparison table of C for loop L contains a comparison tuple

$$\delta(A_k[r_1, \ldots, r_n], A_k[w_1, \ldots, w_n])$$

for every distinct pair of read and write references of each array A_k. C_{ij} is the jth component of tuple i.

As an example, consider Figure 3. There are two pairs of read and write references—$(X[i+1, j], X[i+1, j+1])$ and $(X[i+16, j], X[i+1, j+1])$. It is easily verified that the comparison table for this loop is:

$$\begin{bmatrix} =_\delta & >_\delta \\ <_\delta & >_\delta \end{bmatrix}$$

4.2 Distance Tables

In this section we describe the *distance table*, which is used in conjunction with the comparison table to compute quantizations.

For the computation of the unrolling of each loop, a notion of "distance" (in time) between references to the same array element is needed. A one-dimensional array B is used to introduce the idea.

Definition 4.9 Let $B[r]$ and $B[w]$ be read and write references of B in L using iteration variable i. We say a k-unwinding of loop l_i is *strict* if and only if

$$\forall x \in \mathcal{N}\ r^{-1}(x) \Box w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{k} \right\rfloor \Box \left\lfloor \frac{w^{-1}(x)}{k} \right\rfloor$$

where $\Box$ is either $<$ or $>$ and Q is the predicate $r^{-1}(x) \in \mathcal{N} \wedge w^{-1}(x) \in \mathcal{N}$.

A quantization is said to be strict if the execution of the quantized loop does not contain dependent array references in an iteration of the quantized loop that were not in the same iteration of the original loop.

Definition 4.10 Let $B[r]$ and $B[w]$ be read and write references of B in L.

$$\rho(r,w) = \max_{z \in \mathcal{Z}^+ \cup \{\infty\}} \forall x \in \mathcal{N}\ r^{-1}(x) \Box w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{z} \right\rfloor \Box \left\lfloor \frac{w^{-1}(x)}{z} \right\rfloor$$

where $\Box$ is either $<$ or $>$ and Q is the predicate $r^{-1}(z) \in \mathcal{N} \wedge w^{-1}(z) \in \mathcal{N}$.

Thus $\rho(r,w)$ is the greatest strict unwinding of a loop with respect to r and w. Note that ∞ is a possible value. This is introduced to handle the case where there is no greatest strict unwinding; ∞ is taken to be greater than any value of $\mathcal{Z}^+$.

The value of $\rho(r,w)$ depends on $\delta(r,w)$. For example, if $\delta(r,w) = =_\delta$, then $\rho(r,w) = \infty$. This follows trivially because all dependent references occur in the same iteration if $r =_\delta w$. $\rho(r,w)$ can be computed for the other cases of $\delta(r,w)$. Interestingly, it is possible to have strict unwindings greater than one even if $r ?_\delta w$ or $r ?=_\delta w$. The details are included in the appendix.

We now define distance tuples and tables analogous to the comparison tuples and tables introduced earlier.

Definition 4.11 Let $A[r_1,\ldots,r_n]$ and $A[w_1,\ldots,w_n]$ be a read and write reference (respectively) of A in L. Then

$$\rho(A[r_1,\ldots,r_n], A[w_1,\ldots,w_n]) = \langle \rho(r_1,w_1),\ldots,\rho(r_n,w_n) \rangle$$

This is a distance tuple.

Definition 4.12 A distance table D for loop L contains a distance tuple

$$\delta(A_k[r_1,\ldots,r_n], A_k[w_1,\ldots,w_n])$$

for every distinct pair of read and write references of each array A_k. The order of the tuples in the table is assumed to be compatible with the comparison table for L. D_{ij} is the jth component of tuple i.

As an example, consider once more Figure 3. The distance table is:

$$\begin{bmatrix} \infty & 1 \\ 15 & 1 \end{bmatrix}$$

4.3 Correspondence Between Tables and Loops

From the previous discussion it should be clear that C and D can be computed for every L. In this section we show that every *compatible* C and D pair corresponds to some L (compatibility is a simple well-formedness condition). This permits us to dispense with loops entirely and work only with the abstract representation.

Definition 4.13 Let C be a comparison table and D a distance table, with the elements of C restricted to $>_\delta$, $<_\delta$, $=_\delta$. C and D are compatible if and only if

1. C and D have the same dimensions
2. $C_{ij} \in \{<_\delta, >_\delta\} \Rightarrow D_{ij} \in \mathcal{Z}^+$
3. $C_{ij} \in \{=_\delta\} \Rightarrow D_{ij} = \infty$

The following shows how to compute a compatible comparison and distance table from L.

Definition 4.14 Let L be a set of n nested loops. Then L^C, a comparison table, and L^D, a distance table, are defined for L:

$$L^C_{ij} = \delta(r^i_j, w^i_j)$$

$$L^D_{ij} = \rho(r^i_j, w^i_j)$$

where r^i_j is the jth component of the read reference of the ith read-write reference pair (w^i_j is similarly defined).

Theorem 4.15 For every compatible pair of comparison and distance tables C and D, there exists an L such that $L^C = C$ and $L^D = D$.

Proof: For each pair of tuples c_i of C and d_i of D construct a statement $R_i := W_i$, where R_i is a read reference and W_i a write reference of an array A_i such that $\delta(R_i, W_i) = c_i$ and $\rho(R_i, W_i) = d_i$. This can be done as each component of the tuple is independent of the others and C and D are compatible. □

5 Computing Quantizations

In this section we prove several results about the complexity of computing quantizations, and develop an algorithm for computing strict quantizations.

The order notation must be extended to work with comparison tuples rather than read/write reference pairs. x^c, where x is a location and c is a comparison tuple for a read/write reference pair p, is defined to be x^p. x_i^c is the ith component of x^c. The reader may question this change; after all, a particular c represents many possible reference pairs. The following key lemma justifies the change in notation.

Lemma 5.16 Let c and d be compatible comparison and distance tuples. For simplicity, we assume that c has no $?_\delta$ or $?{=}_\delta$ entries. If the quantization unrolls loop l_i more than d_i times then there is a location x such that $x_i^c = =_o$ in the quantized loop. If the quantization unrolls loop l_i less than d_i times then x_i^c in the quantized loop and x_i^c in the original loop are always equal.

Proof: Follows immediately from the definition of d_i. □

We will also make use of the fact that regardless of the unwinding of a loop l_i and comparison tuple c, there is always some element x such that x_i^c is the same in both the quantized and original loops. This is true because we assume upper and lower loop bounds are arbitrary, and therefore loops may not be fully unwound. As stated before, our techniques can be extended to cover the case where loop bounds are known.

5.1 Properties of Loop Quantization

We now prove some general properties of loop quantization. The development is geared toward pinpointing exactly which quantizations can be feasibly computed.

Definition 5.17 A quantization q of L is an n-tuple $\langle q_1, \ldots, q_n \rangle$, $q_i \in \mathcal{Z}^+ \cup \{\infty\}$, where $q_i = \infty$ indicates that loop l_i may be unwound an arbitrary number of times.

Definition 5.18 Let p and q be quantizations of L. Then $p \preceq q \Leftrightarrow \forall j \; p_j \leq q_j$ (i.e., the product ordering). We take ∞ to be greater than any element of $\mathcal{Z}^+$.

A quantization q is said to be *maximal* if it is a legal quantization and there is no quantization q' that is legal and $q \preceq q'$. A quantization q is *maximum* if it is legal and for any other legal quantization q', $q' \preceq q$.

Fact 5.18.1 For every L, there exists a maximal quantization.

Lemma 5.19 There exists an L such that L has maximal quantizations q_1 and q_2 and $q_1 \npreceq q_2$ and $q_2 \npreceq q_1$.

Proof: Let $C = [>_\delta >_\delta <_\delta]$ and $U = [1\ 1\ 1]$. Then $\langle \infty\ 1\ \infty \rangle$ and $\langle 1\ \infty\ \infty \rangle$ are both maximal quantizations.

To see this, assume that $q_1 > 1$ and $q_2 > 1$. Applying Lemma 5.16, there is a location x such that $x^c = \langle =_o =_o <_o \rangle$ in the quantized loop. However, $x^c = \langle >_o >_o <_o \rangle$ in the original loop, showing that the quantization does not preserve the order of conflicting references. Another check shows that if $q_1 = 1$ or $q_2 = 1$ the order of references is preserved. Thus both quantizations are maximal. □

Barring the existence of a unique maximum, what should be the criteria for an optimal quantization? Ideally, we would like to maximize two things: the number of ∞ entries in the quantization and the product of the other components. Maximizing the number of components that are ∞ permits the greatest leeway in code generation for an arbitrary machine and generally provides the greatest speedup (subject to the length of dependency chains induced by loop-carried dependencies).

Let the *infinity number* $|q|_\infty$ of quantization q be the number of components that are ∞.

Definition 5.20 A quantization q of L is *best* if q is legal and

$$\forall q' \text{ s.t. } q' \text{ is legal, } |q|_\infty > |q'|_\infty \vee (|q|_\infty = |q'|_\infty \wedge \prod_{q_i \in \mathcal{Z}^+} q_i \geq \prod_{q'_i \in \mathcal{Z}^+} q'_i)$$

Theorem 5.21 Computing a best quantization of a set of nested loops L is NP-hard.

Proof: In appendix. □

It is unfortunate that computing best quantizations is NP-hard. However, careful examination of the proof reveals that the exponential cost is proportional only to the number of dimensions of the largest array involved. If a loop contains references to n-dimensional arrays, a brute-force algorithm can compute a best quantization by performing 2^n simple tests. (The approach is to check every possible subset of the columns of D and C to find the one that optimizes $|q|_\infty$—see the appendix.) Because array dimensions rarely exceed five or six in practice this is a reasonable approach. We will present a polynomial time algorithm to compute a maximal strict quantization (a simple variation computes a maximal quantization); nevertheless, computing a best quantization is probably feasible in practice.

5.2 Strict Quantizations

Following the line of development of the previous section, we say a quantization q is *maximal strict* if q is strict and there is no other strict quantization q' such that $q \preceq q'$. A quantization q is *maximum strict* if q is strict and for all other strict quantizations q', $q' \preceq q$. Finally, a quantization is *best strict* if it is strict and

$$\forall q' \text{ s.t. } q' \text{ is strict,} |q|_\infty > |q'|_\infty \vee (|q|_\infty = |q'|_\infty \wedge \prod_{q_i \in \mathcal{Z}^+} q_i \geq \prod_{q'_i \in \mathcal{Z}^+} q'_i)$$

These definitions parallel those from the previous section.

Theorem 5.22 The following are true:

1. For every L there is a maximal strict quantization.
2. There exists an L with no maximum strict quantization.
3. Computing the best strict quantization of L is NP-hard.

Proof: We develop an algorithm to compute a maximal strict quantization in the next section. Proofs of 2 and 3 mirror proofs from the previous section. □

The problems in this and the previous section stem from the difficulty of defining an "optimal" quantization. Three definitions were proposed for both the general and strict cases: maximal, maximum, and best. The structure of the two problems is identical. Maximal is computable in polynomial time, maximum does not always exist, and best is NP-hard. The questions that need investigation are: How often does a maximum exist in real programs? How often is a maximal quantization substantially worse than the maximum? If a loop has a maximum, then the following corollary shows that we can efficiently compute an optimal quantization.

Corollary 5.23 If a maximum quantization q exists for L then:

1. q is best
2. q is computable in polynomial time

Proof: One is trivial by the definitions of maximum and best. Two follows from the fact that a maximum must be the unique maximal quantization. □

5.3 Computing a Maximal Strict Quantization

We now develop an algorithm for computing a maximal strict quantization. As stated previously, a strict quantization is useful because the loops need not be unrolled to perform compaction; the unrolled iterations are always independent. The algorithm also illustrates one application of the comparison and unrolling tables to quantization. A slightly modified algorithm computes a maximal quantization.

The algorithm consists of two passes over the tables. In the first pass a simple greedy heuristic computes an approximate quantization. The second pass refines this to a maximal strict quantization. We assume for technical simplicity that there are no $?_\delta$ or $?{=}_\delta$ entries in C.

The algorithm makes use of the fact that any strict quantization must preserve strictness in a defining position of every comparison tuple. This is formalized in the following theorem.

Theorem 5.24 Let C and D be compatible comparison and distance tables of size $m \times n$. Let C_i be a tuple in C. If C_i has a defining position less than $n+1$, then any strict quantization q must satisfy:

$$\exists k \text{ s.t. } C_{ik} \text{ is a defining position of } C_i \wedge q_k \preceq D_{ik}$$

Proof: Straightforward application of Lemma 5.16 and the definition of a defining position. □

The strategy of pass one is as follows. Consider each column of C and D in order for $j = 1..n$. Let

$$A_j \quad = \quad \{D_{ij} | C_{ij} \text{ is the leftmost defining position of tuple } i\} \qquad (1)$$

Then

> **if** $A_j \neq \emptyset$
>
> **then** $q_j \leftarrow \text{MIN}(A_j)$
>
> **else** $q_j \leftarrow \infty$;

The idea behind this computation is simple. Let c be a tuple in C. If c is defined at position k, then the relation c_k specifies the lexicographic ordering of references for the read/write reference pair which c represents. One way to preserve strictness is to unwind strictly on dimension k with respect to c. Note that strictness also guarantees independence of the iterations. For example, let c be the tuple $\langle =_\delta \; =_\delta \; <_\delta \; =_\delta \; >_\delta \rangle$. c is defined at c_3. By unwinding strictly with respect to c on the third dimension, we guarantee that conflicting references occur in the correct order (reads before writes) and in separate iterations. The first phase enforces strictness at the first defining position of each tuple.

Minimizing over the set A_j gives an unrolling which satisfies the above requirements for all tuples. If $A_j = \emptyset$, then the column under consideration has no impact on the order of references (as determined be $q_1 \ldots q_{j-1}$) and may be unwound arbitrarily. Note that this may occur before the algorithm has passed a defining point of all tuples. For example, consider the comparison and distance tables:

$$C = \begin{bmatrix} <_\delta & =_\delta & =_\delta & >_\delta \\ =_\delta & =_\delta & <_\delta & =_\delta \\ >_\delta & >_\delta & =_\delta & =_\delta \end{bmatrix} D = \begin{bmatrix} 1 & \infty & \infty & 1 \\ \infty & \infty & 1 & \infty \\ 1 & 1 & \infty & \infty \end{bmatrix}$$

In this case, pass one generates $q = \langle 1\ \infty\ 1\ \infty \rangle$, although the second tuple is not defined until column three. The reason is that because pass one unrolls strictly on column one, the correct order of reads and writes for tuple three is already guaranteed. Column one is called tuple three's *selected defining position.* Tuples one and two do not require any particular unwinding for column two; thus, $q_2 = \infty$.

Pass one does not, in general, compute a maximal strict quantization. A counterexample:

$$C = \begin{bmatrix} <_\delta & <_\delta & <_\delta \\ =_\delta & >_\delta & <_\delta \end{bmatrix} D = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In this example, pass one computes $q = \langle 1\ 1\ \infty \rangle$; however, $q' = \langle \infty\ 1\ \infty \rangle$ is also strict and $q' \succ q$. The problem is that ordering and strictness for the first tuple can be satisfied by unrolling strictly on any dimension. The second pass fixes this problem.

The second pass considers each column from right to left ($n..1$). Fix a column j. Let J be the set of comparison tuples with a selected defining position less than j. For each $C_i \in J$, check if q_j, C_{ij}, and D_{ij} together guarantee the strictness of C_i—i.e., whether j is a defining position of C_i, and $D_{ij} \geq q_j$. If so, then let x be the current selected defining position of C_i. Set C_i's selected defining position to j, and recompute q_x without considering C_{ix}. Note that q_j need not be recomputed—the unwinding of dimension j is already sufficient to guarantee the strictness of C_i.

The idea is that the first pass establishes strictness as "early" as possible—at the leftmost defining position. The second pass then moves the selected defining position as far right as possible while maintaining consistency with the quantization. In the example above, it is easily confirmed that this yields $q = \langle \infty\ 1\ \infty \rangle$, which is maximal (and maximum).

An algorithm to compute a maximal strict quantization appears in Figure 8. We assume without loss of generality that all tuples containing $\neq_\delta$ have been removed from C.

We outline a proof of correctness for this algorithm.

Proof: [Correctness] Let q be the quantization computed by the algorithm. By construction this is a legal strict quantization. To show that it is maximal we show that any $q' \succ q$ is not a legal strict quantization.

Input: C $m \times n$ comparison table

D $m \times n$ distance table

```
declare array defined[1..m], unroll[1..n], q[1..n]
∀i defined[i] ← nil
(* Pass 1 *)
for j = 1 to n do
    if ∀i(defined[i] ∨ C[i,j] = =_δ)
        then q[j] ← ∞
        else q[j] ← min_{i|defined[i]=nil} D[i,j];
    for i = 1 to n do if C[i,j] ∈ {<_δ, >_δ} ∧ defined[i] = nil
            then defined[i] ← j;
(* Pass 2 *)
for j = n to 1 do
    for each tuple C_i
        if j is a defining position of C_i and D[i,j] ≥ q[j] and defined[i] < j
            then
                x ← defined[i];
                defined[i] ← j;
                q[x] ← min_{i|defined[i]=x} D[i,x];
```

Figure 8: Algorithm for computing a maximal strict quantization

Let $q' \succ q$. The there is an j such that $q'_j > q_j$. Clearly $q_j < \infty$. Consider q_j after columns $n..j+1$ have been processed by pass two. There is at least one tuple C_i with the following property:

(*) C_i's defining position is j and j is the least such position.

If C_i's selected defining position cannot be moved by the pass two examination of column j, then it follows that there is no defining position k of C_i greater than j that also has a strict unrolling for C_i (i.e., $q_k < D_{ik}$). Thus, by property (*) and Theorem 5.24, j is the *only* defining position of C_i that preserves strictness (and possibly legality, too). Choose C_i from all tuples satisfying these criteria so that U_{ij} is minimized. Then $q_j = D_{ij}$, and by Lemma 5.16 no larger unrolling is permissible. □

Extensions for the relations $?_\delta$ and $?=_\delta$ are included in the appendix.

Turning once more to Figure 3, the maximal strict quantization computed by the algorithm for this example is $\langle 15\ 1 \rangle$. A simple variation on this algorithm computes a maximal quantization, the difference being that the constraint on the unwinding is to preserve legality rather than strictness. A maximal (and maximum) quantization for the example is $\langle 15\ \infty \rangle$.

6 Mitred Quantization

In practice, the important loop carried dependencies are generated by array references of the form $A[I+c]$ for an iteration variable I and a constant c. References of this form are said to be *affine*. If some references are affine, we can make use of a technique, *mitred quantization*, that greatly increases the power of loop quantization.

Thus far it has been impossible (without unwinding fully on at least one dimension) to quantize a loop containing a statement of the following form:

$$\begin{array}{ll} (*) & A[i,j] \leftarrow f(A[i-k, j+k']); \\ \text{or} & A[i-k, j+k'] \leftarrow f(A[i,j]); \end{array}$$

The quantization tables for these statements are $\left[\ <_\delta \quad >_\delta\ \right]\left[\ k \quad k'\ \right]$ and $\left[\ >_\delta \quad <_\delta\ \right]\left[\ k \quad k'\ \right]$.

Any loop containing (*) may be unwound at most k times on loop L_i. The problem is that the execution order of two dependent statements is reversed if L_i is unwound "too much". However, if subscript expressions of the dependencies that prevent quantization are affine, then the corresponding dependent statements must lie on lines in the iteration space. In this case the quantization box can be skewed to include any dependent statements. An illustration is given in Figure 9.

The following lemma limits the cases we must consider.

Lemma 6.25 Let C be a comparison table for loop L with affine references. Assume further that for any tuple c_i of C, there is no k and l such that $c_{ik} = >_\delta$ and $c_{il} = <_\delta$. Then any quantization is legal for L.

Proof: Trivial. □

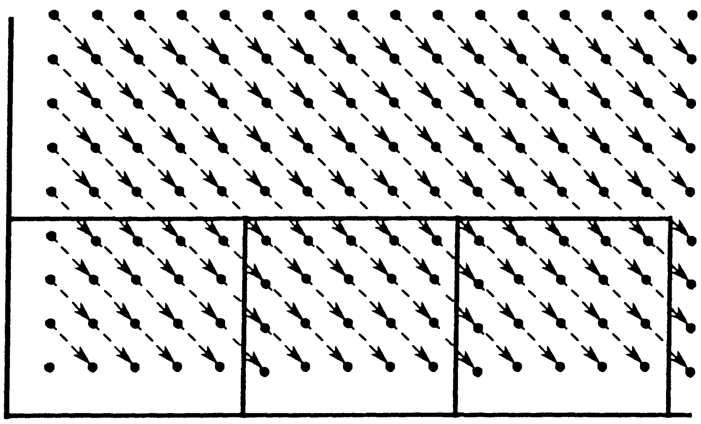
The lemma implies that the only cases of concern are of the type presented above (nothing more can be done if a tuple contains $?_\delta$ or $?{=}_\delta$). We limit our discussion to the two-dimensional case; the technique extends to arbitrary dimensions, but the essential ideas are illustrated best with examples that can be drawn in a plane. The resulting algorithm can be used to generate arbitrarily large quantizations of any loop where the dependencies preventing arbitrary quantization consist of affine references. In particular, this allows arbitrarily large quantizations of any loop with only affine references.

Consider a loop L with a read reference $A[i+i_r, j+j_r]$ and a write reference $A[i+i_w, j+j_w]$. We assume that L consists of two nested loops, with j being the induction variable of the outer loop. The *slope* of the dependency of a read/write reference pair is defined to be

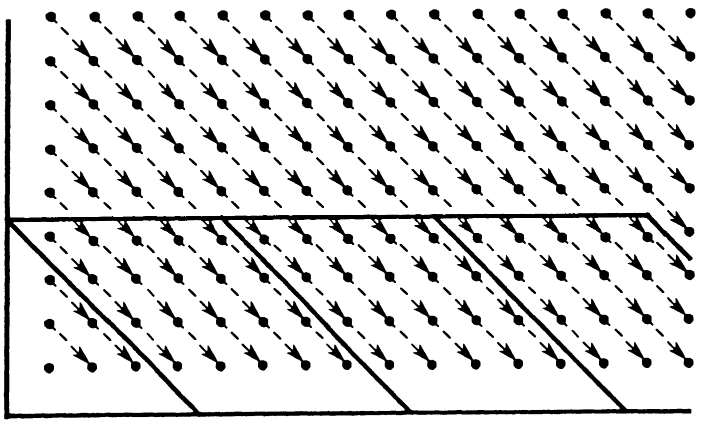
$$\frac{j_r - j_w}{i_r - i_w}$$

If the slope of a dependency is negative, then the corresponding comparison tuple must be $[<_\delta\ >_\delta]$ or $[>_\delta\ <_\delta]$. Let p be the read/write reference pair with the smallest negative slope s. It can be shown that our method ensures correctness for all dependencies if correctness is ensured for p.

The presentation of the technique involves arguments about what statements are dependent on a point (i, j) in the iteration space. The iteration space can be depicted as a plane; however, the convention that loops are normalized to iterate from 1 to N puts the first interesting point of this plane at the point $(1, 1)$ instead of the origin. We have respected this convention here; some of the formulas, however, are simpler if expressed assuming loops iterate from 0 to N.



(a) Normal quantzation "cuts" a dependency illegally.



(b) Skewed boxes capture dependent statements.

Figure 9: Quantization with affine references. Arrows represent dependencies.

Let Δj and Δi be unwindings of L_i and L_j respectively. We require that Δj satisfy the constraint that $\frac{\Delta j-1}{-s}$ is an integer. The dependencies of p lie on lines of the form $j = si + c$ for some constant c. With an unwinding Δj the point $(\Delta i, \Delta j)$ must be executed after all points on the line $j - \Delta j = s(i - \Delta i)$ such that $1 \leq i < \Delta i$. These are exactly the dependent statements that would be executed before $(\Delta i, \Delta j)$ in normal loop execution order.

The i-intercept of the line $j = 1$ is

$$\frac{\Delta j - 1}{-s} + \Delta i$$

This is to the right of the point $(\Delta i, 1)$ as s is negative. If we wish to execute the point $(\Delta j, \Delta i)$, we must either include in the quantization box, or have executed in an earlier box, all dependent statements on the line between the points $(\Delta j, \Delta i)$ and $(\frac{\Delta j-1}{-s} + \Delta i, 1)$. We choose the the limits of the quantization box to be the parallelogram bounded by the points $(1, \Delta j)$, $(\Delta i + 1, \Delta j)$, $(\frac{\Delta j-1}{-s} + 1, 1)$, $(\frac{\Delta j-1}{-s} + \Delta i + 1, 1)$. An illustration is given in Figure 10. Note that all dependent references are included in the box or are satisfied by executing boxes in normal loop order (i.e., first along the i dimension, then the j dimension).

One detail that must be covered is what happens to the points in the iteration space that are not in the skewed boxes at the limits of the loop (area M in Figure 10). These are executed, in normal loop order, as a prologue and epilogue to executions of the quantized loop L_i. In general, it will be necessary have a prologue for loop L_j as well, as Δj will not evenly divide the upper bound of loop L_j. A general scheme for a quantization $(\Delta j, \Delta i)$ is given in Figure 11. Note that there is an upper bound, computable at compile time, on the number iterations executed in the prologues and epilogues; thus these "extra" loops may themselves be quantized, up to and including full unwinding.

As stated earlier, this method can be extended to higher dimensions. We do not discuss the general case here, except to mention that there are several added difficulties in the analysis, but the resulting algorithm is similar to the one presented. Using this method we can, for example, obtain arbitrary quantizations of the example in Figure 6 when the upper bound of the inner loop is unknown.

6.1 Dimension Reversal

Another simple trick further increases the power of quantization. Consider a two-dimensional iteration space as above. If there are no dependencies of the form $[=_\delta <_\delta]$ or $[=_\delta >_\delta]$, then

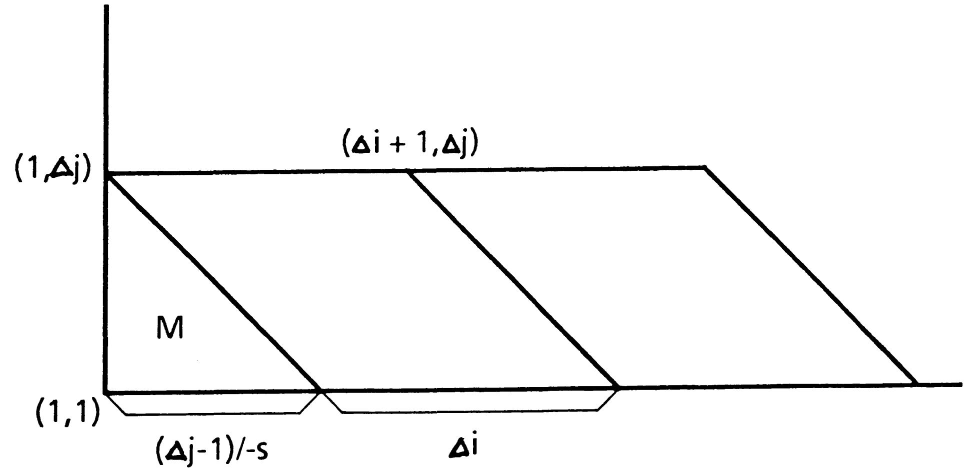


Figure 10: Computation of the skewed box for Δj and Δi

the innermost loop L_i may iterate either from 1 to N_i or N_i to 1. Reversing the direction of the loop also changes the dependencies with negative slope; thus if mitred quantization fails, reversing the loop L_i may allow mitred quantization to succeed. Even if mitred quantization is not applicable in either case, it is possible that reversing the direction of L_i may result in a better (normal) quantization. The idea extends to outer loops and higher dimensions.

7 Conclusions

Loop Quantization is being integrated into an environment for scientific programming under development at Cornell [23]. The early results are very encouraging, suggesting that loop quantization combined with compaction techniques form a viable complement or even alternative to higher-level, more monolithic transformations.

We are currently exploring other aspects of loop quantization. A number of important questions are still open; most important perhaps is the question of how much, in general, of a quantized and compacted loop can be computed symbolically (i.e., without actually having to unwind and compact). We are also exploring applications of the comparison and distance tables to other transformations.

(*

Original loop is

```
    for J = 1, N_J              loop L_J
        for I = 1, N_I          loop L_I
            B;
```

*)

(* L_J prologue *)

for $J = 1, N_J \bmod \Delta j$

  for $I = 1, N_I$

    B;

for $J = (N_J \bmod \Delta j) + 1, N_J, \Delta j$

  (* L_I prologue—computes area M in Figure 10 *)

  for $j = J, J + \Delta j - 1$

    for $i = 1, \frac{\Delta j - 1}{-s} - \lfloor \frac{(J-j)}{-s} \rfloor + (N_I \bmod \Delta i)$

      B; where j replaces J, i replaces I

  (*The skewed quantization box. *)

  for $I = \frac{\Delta j - 1}{-s} + (N_I \bmod \Delta i) + 1, N_I, \Delta i$

    B_{00};

    B_{01};

    ...

    $B_{1,\Delta i-1}$;   where $I + i - \lfloor \frac{j}{-s} \rfloor$ replaces I in B_{ji}

    B_{11}; $J + j$ replaces J in B_{ji}

    ...

    $B_{\Delta j-1,\Delta i-1}$;

  (* L_I epilogue *)

  for $j = J + 1, J + \Delta j - 1$

    for $i = N_I - \lfloor \frac{(j-J)}{-s} \rfloor, N_I$

      B; where i replaces I, ij replaces J

Figure 11: Scheme for arbitrary two-dimensional quantization $\Delta j, \Delta i$.

A Appendix

A.1 Computation of $\rho(r, w)$

Lemma A.26 Let $r(i) = a_r i + b_r, a_r \neq 0$ and $w(i) = a_w i + b_r, a_w \neq 0$.

Then the following statements hold:

1. $r =_\delta w \Rightarrow \rho(r, w) = \infty$

2. $r \neq_\delta w \Rightarrow \rho(r, w) = \infty$

3. $r <_\delta w \Rightarrow \rho(r, w) = b_r - b_w$

4. $r >_\delta w \Rightarrow \rho(r, w) = b_w - b_r$

Proof: The first two cases are trivial; we prove only the third.

Fact A.26.1 $r <_\delta w \Leftrightarrow a_r = a_w \wedge b_r > b_w$

If no lower bounds are known, then $r(i) = w(i) + (b_r - b_w)$. Dividing by $(b_r - b_w)$ and taking floors we get:

$$\forall x \in \mathcal{N} \left(\left\lfloor \frac{r(x)}{b_r - b_w} \right\rfloor = \left\lfloor \frac{w(x) + (b_r - b_w)}{b_r - b_w} \right\rfloor \right)$$

From this it immediately follows that

$$\forall x \in \mathcal{N} \left(\left\lfloor \frac{r(x)}{b_r - b_w} \right\rfloor = \left\lfloor \frac{w(x)}{b_r - b_w} \right\rfloor + 1 \right)$$

So $b_r - b_w$ is strict for r and w. To see that it is also the largest value, note that using $b_r - b_w + 1$ results in

$$\left\lfloor \frac{r(x)}{b_r - b_w + 1} \right\rfloor$$

and

$$\left\lfloor \frac{w(x) + b_r - b_w}{b_r - b_w + 1} \right\rfloor$$

If we choose $x = 0$ then

$$\left\lfloor \frac{b_r}{b_r - b_w + 1} \right\rfloor = \left\lfloor \frac{b_w + b_r - b_w}{b_r - b_w + 1} \right\rfloor$$

so $b_r - b_w + 1$ is *not* strictness preserving. □

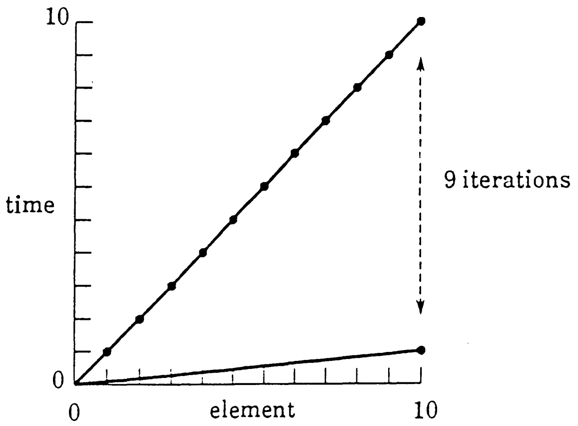


Figure 12: Comparing references $A[10x]$ and $A[x]$.

The most complicated cases have not been treated—what if $r?_\delta w$ or $r?{=}_\delta w$? Although reads and writes may occur in either order in these cases, it is still possible to have strict unwindings greater than one.

As an example, assume the statement $A[10x] := A[x]$ appears in L. A graph of the time functions for the read and write references is shown in Figure 12. Two conflicting references are shown. One is at element zero, but it occurs in the same iteration (time zero) for both references and is therefore preserved by any quantization. The other conflict is at element ten. $A[10x]$ writes it at time one; $A[x]$ reads it at time ten. Thus the loop l_x could be unwound nine times and still preserve strictness. This follows because conflicting references in the positive direction must be more than nine iterations apart. A symmetric argument shows the same to be true in the negative direction.

Note that if the lower bound of x were something other than one the loop could be unwound more and still preserve strictness. If, for instance, the loop began with $x = 0$ the greatest strictness-preserving unwinding would be ten. If the loop began at $x = 2$ the conflict at element ten would not exist and the unwinding could be as great as eighteen (the next conflict is at element twenty). The method we present can be easily extended to take advantage of knowledge of the upper and lower bounds of loop indexes; for the moment we continue to assume nothing is known about these bounds.

The generalization of the example requires a few simple facts about integer-valued linear functions. We state these facts without proof.

Definition A.27 Let $r(x) = ax + b$, where a and b are rationals. We define the set of integer solutions of r to be

$$\mathcal{I}(r) = \{y | y \in \mathcal{N} \wedge ay + b \in \mathcal{N}\}$$

There are linear expressions without integer solutions—e.g., $x + \frac{1}{2}$. We exclude these from consideration.

Fact A.27.1 Let $r(x) = ax + b$, where a and b are rationals and $a \neq 0$. Let x_1 be any integer solution of r. Let x_2 be the smallest integer solution of r greater than x_1. Then

$$\mathcal{I}(r) = \{a(x_1 + \Delta(x_2 - x_1)) + b | \Delta \in \mathcal{N}\}$$

$(x_2 - x_1)$ is called the *period* of r and is denoted by $\hat{r}$.

Lemma A.28 Let $r(x) = a_r x + b_r$ and $w(x) = a_w x + b_w$ be two linear expressions.

$$\begin{array}{ll} \text{If} & \mathcal{I}(r) \cap \mathcal{I}(w) \neq \emptyset \\ \text{then} & \mathcal{I}(r) \cap \mathcal{I}(w) = \{\Delta \cdot lcm(\hat{r}, \hat{w}) + z | \Delta \in \mathcal{N}\} \\ \text{where} & z \in \mathcal{I}(r) \cap \mathcal{I}(w) \end{array}$$

Proof: Omitted. □

We can now describe how to compute the greatest strict unrolling when $r?_\delta w$ (or $?{=}_\delta$). From the previous discussion, it is clear that the objective is to compute the minimum vertical distance between r^{-1} and w^{-1} at a common integer solution; this corresponds to the "closest" conflicting reference. More precisely, if $r?_\delta w$ or $r?{=}_\delta w$ then $\min_{j \in \mathcal{I}(r) \cap \mathcal{I}(w)} |r^{-1}(j) - w^{-1}(j)|$ may be computed as follows:

1. Compute any integer solution k in $\mathcal{I}(r) \cap \mathcal{I}(w)$. This can be done using standard diophantine equation analysis.
2. Compute $\alpha = lcm(\hat{r}, \hat{w})$.
3. Compute $p =$ the intersection point of r^{-1} and w^{-1}. p is a real number.
4. Using k and α, compute the smallest common integer solution greater than p. Call this g. The formula is:

$$\begin{array}{ll} g = k + \alpha \left\lceil \frac{p-k}{\alpha} \right\rceil & \text{if } p \text{ is not a common integer solution} \\ g = p + \alpha & \text{if } p \text{ is a common integer solution} \end{array}$$

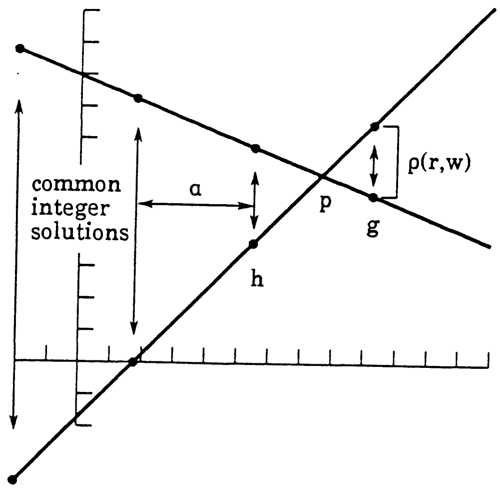


Figure 13: Computation of $\rho(r, w)$.

5. Compute the largest common integer solution less than p. This is

$$h = g - \alpha \quad \text{if } p \text{ is not a common integer solution}$$
$$h = p - \alpha \quad \text{if } p \text{ is a common integer solution}$$

6. Set $\rho(r, w) = \min(|r^{-1}(g) - w^{-1}(g)|, |r^{-1}(h) - w^{-1}(h)|)$.

Figure 13 illustrates the values computed by the algorithm.

We will not present a formal proof of correctness for this algorithm. The idea of the proof is to note that the smallest "distance" in iterations must occur near the intersection point of the two lines. In fact, it must be with $lcm(\hat{r}, \hat{w})$ on either side of the intersection point. Thus we need only consider the two common integer solutions lying on either side of the intersection.

A.2 Extensions to the Maximal Strict Quantization Algorithm

Unfortunately, $\rho(r, w)$ does not provide enough information to extend the algorithm for computing a maximal strict quantization to handle $?_\delta$.

Definition A.29 Let

$$\rho^{<}(r, w) = \max_{z \in \mathcal{Z}^{+} \cup \{\infty\}} \forall x \in \mathcal{N}\ r^{-1}(x) < w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{z} \right\rfloor < \left\lfloor \frac{w^{-1}(x)}{z} \right\rfloor$$

$$\rho^{>}(r,w) = \max_{z \in \mathcal{Z}^{+} \cup \{\infty\}} \forall x \in \mathcal{N}\ r^{-1}(x) > w^{-1}(x) \wedge Q \Rightarrow \left\lfloor \frac{r^{-1}(x)}{z} \right\rfloor > \left\lfloor \frac{w^{-1}(x)}{z} \right\rfloor$$

where Q is the predicate $r^{-1}(z) \in \mathcal{N} \wedge w^{-1}(z) \in \mathcal{N}$.

The values $\rho^{<}(r,w)$ and $\rho^{>}(r,w)$ are the arguments of the minimum in step six of the algorithm to compute $\rho(r,w)$ when $r ?_{\delta} w$. We extend the definition of distance tables to include pairs $\langle \rho^{<}(r,w), \rho^{>}(r,w) \rangle$ for $?_{\delta}$ entries in a compatible comparison table.

The following lemma extends Lemma 5.16 for $?_{\delta}$ and motivates the need for $\langle \rho^{<}, \rho^{>} \rangle$ pairs in the distance table.

Lemma A.30 Let c and d be compatible comparison and distance tuples with $c_i = ?_{\delta}$ or $?{=}_{\delta}$. Assume c_i is computed from r_i and w_i. Let q be a quantization of L. If $q_i > \rho^{<}(r_i, w_i)$ (resp.$\rho^{>}(r_i, w_i)$), then there is an x s.t. $x_i^c = <_o$ $(>_o)$ in the original loop and $x_i^c = =_o$ in the quantized loop.

Proof: The result follows from the definition of $\rho^{<}(r_i, w_i)$ and $\rho^{>}(r_i, w_i)$. □

Consider the comparison and distance tables:

$$C = [?_{\delta} >_{\delta}]\ D = [\langle 2, 1 \rangle\ 2]$$

What is a maximal strict quantization for these tables? The obvious thing to do is to unwind strictly on the first dimension and arbitrarily on the second. Thus one possibility is $q_{max} = \langle 1\ \infty \rangle$. Another possibility is $q_{max} = \langle 2\ 2 \rangle$; the first component guarantees the strictness for locations x where $x_1^c = <_o$, the second component guarantees the strictness for all other locations. This is why we need more information to handle $?_{\delta}$; the fact that $\langle 2\ 2 \rangle$ is a maximal strict quantization could not be inferred if the distance entry for $?_{\delta}$ was $\rho(r,w)$.

Now consider the tables:

$$C = [?_{\delta} >_{\delta} >_{\delta} >_{\delta}]\ D = [\langle 2, 1 \rangle\ 2\ 2\ 2]$$

A best quantization is $\langle 1\ \infty\ \infty\ \infty \rangle$. The quantization $\langle 2\ 2\ \infty\ \infty \rangle$ is maximal strict for this example, as are $\langle 2\ \infty\ 2\ \infty \rangle$ and $\langle 2\ \infty\ \infty\ 2 \rangle$. In this example, every position can act like a defining position. The $?_{\delta}$ component is the most important— it demands a strict unwinding that at least protects the read/write order for locations x where $x_1^c = <_o$. However, because all the other components in the table are $>_\delta$, the first dimension need not be unrolled to ensure strictness for locations x where $x_1^c = >_o$. This can be done by unwinding strictly on any other dimension.

The relation $?{=}_\delta$ is not as difficult to handle, as the following lemma shows.

Lemma A.31 Let $r ?{=}_\delta w$. Then $\rho^<(r, w) = \rho^>(r, w)$.

Proof: A simple geometric argument based on the fact that the intersection point of r^{-1} and w^{-1} is a common integer solution. □

Thus $\rho(r, w)$ provides enough information when $r ?{=}_\delta w$.

Consider the tables:

$$C = [?{=}_\delta \ ?{=}_\delta \ >_\delta \ >_\delta] \ D = [1\ 1\ 1\ 1]$$

There are only two maximal quantizations for this example: $q_{max} = \langle 1\ 1\ 1\ \infty \rangle$ or $q_{max} = \langle 1\ 1\ \infty\ 1 \rangle$.

This deserves some explanation. If q_1 were greater than one, then, by Lemma A.30, there would be a location x where $x_1^c = >_o$ in the original loop and $x_1^c = =_o$ in the quantized loop. Therefore we may choose values of the iteration variables of the quantized loop such that $x^c = \langle =_o <_o \ \ldots \rangle$; an unrolling greater than one for q_1 interchanges the read/write order for at least one point. A similar argument shows that q_2 must be one.

One of q_3 and q_4 must be one. This follows because the first two components of the comparison tuple are $?{=}_\delta$. By definition, there is a location x such that $x^c = \langle =_o =_o \ \ldots \rangle$. Guaranteeing strictness here requires a strict unwinding in either the third or fourth dimension.

Most of the lemmas and definitions presented thus far require only trivial modifications to allow for $?_\delta$ and $?{=}_\delta$. One that requires considerable change is the definition of a defining position.

Definition A.32 Let c be a comparison tuple of length n. Let j be the least index such that c_j is not $=_\delta$ or $?{=}_\delta$. If there is no such j, then c's only defining position is $n + 1$.

If $c_j \neq ?_\delta$, then c_k is a defining position for c if $c_k = c_j \wedge \forall z, j < z < k, c_z \in \{=_\delta, c_j\}$

If $c_j = ?_\delta$, then j is the *primary* defining position of c. Let c_i be the least index greater than j that is not $=_\delta$ or $?{=}_\delta$. If c_i is either $<_\delta$ or $>_\delta$, then c_k is a *secondary* defining position of c if $c_k = c_i \wedge \forall z, i < z < k, c_z \in \{=_\delta, c_i\}$

The following lemmas may be proved by straightforward application of Lemmas 5.16 and A.30.

Lemma A.33 Let c and d be compatible comparison and distance tuples. Let j be the least defining position of c. If there is a position c_k s.t. $k < j \wedge c_k = ?{=}_\delta$, then any strict quantization q must satisfy $q_k \leq d_k$.

Lemma A.34 Let c and d be compatible comparison and distance tuples. Let j be the least defining position of c and let $c_j = ?_\delta$. Let $d_j = \langle \rho^<, \rho^> \rangle$. Then any strict quantization q must satisfy one of the following

1. $q_j \leq \min(\rho^<, \rho^>)$
2. $q_j \leq \rho^<$ and there is a secondary defining position c_k s.t. $c_k = >_\delta$ and $q_k \leq d_k$.
3. $q_j \leq \rho^>$ and there is a secondary defining position c_k s.t. $c_k = <_\delta$ and $q_k \leq d_k$.

The revised algorithm is shown in Figure A.2. The proof of correctness is similar to the proof for the previous algorithm. Lemmas A.33 and A.34 are used to cover the new cases.

A.3 Computing a Best Quantization is NP-hard

Proof: We actually prove a stronger statement: computing the infinity number of a best quantization is NP-hard. To help motivate the proof, consider the following tables:

$$L_C = \begin{bmatrix} l_1 & l_2 & l_3 \\ >_\delta & >_\delta & <_\delta \end{bmatrix} L_D = \begin{bmatrix} l_1 & l_2 & l_3 \\ 1 & 1 & 1 \end{bmatrix}$$

In these tables, the l_i are column labels. Let $A[r_1, r_2, r_3]$ and $A[w_1, w_2, w_3]$ be the read and write references which generate these tables. Clearly, one of l_1 or l_2 cannot be unrolled more than once if the quantization is to be legal. To see this, assume that both l_1 and l_2 are unwound k times, where $k > 1$. Then, by Lemma 5.16, there is a location x such that $x^{\langle r,w \rangle} = \langle =_o =_o <_o \rangle$ in the quantized loop and $x^{\langle r,w \rangle} = \langle >_o >_o <_o \rangle$ in the original loop. By unwinding more than once on l_1 and l_2 the quantization interchanges the order of references for at least one point.

Input: C $m \times n$ comparison table

D $m \times n$ distance table

declare array *defined*$[1..m]$, *unroll*$[1..n]$, $q[1..n]$

$\forall i, j$ **if** $D[i,j] = \langle \rho^<, \rho^> \rangle$

    then $select(D[i,j]) \leftarrow \min(\rho^<, \rho^>)$

    else $select(D[i,j]) \leftarrow D[i,j]$;

$\forall i$ $defined[i] \leftarrow nil$

(* Pass 1 *)

for $j = 1$ **to** n **do**

    if $\forall i(defined[i] \vee C[i,j] = =_\delta)$

        then $q[j] \leftarrow \infty$

        else $q[j] \leftarrow \min_{\{i \mid defined[i]=nil\}} select(D[i,j])$;

    for $i = 1$ **to** n **do if** $C[i,j] \in \{<_\delta, >_\delta, ?_\delta\} \wedge defined[i] = nil$

            then $defined[i] \leftarrow j$;

(* Pass 2 *)

for $j = n$ **to** 1 **do**

    for each tuple C_i

        if j is a defining position of C_i, $D[i,j] \geq q[j]$, $defined[i] < j$, and $C[i,j] \neq ?_\delta$ **then**

            $x \leftarrow defined[i]$;

            $defined[i] \leftarrow j$;

            $q[x] \leftarrow \min_{\{i \mid defined[i]=x\}} D[i,x]$;

        if j is a defining position of C_i and $C[i,j] = ?_\delta$, and k is a secondary defining position s.t. $select(D[i,k]) \leq q[k]$ **then**

            let $\langle \rho^<, \rho^> \rangle = D[i,j]$;

            if $C[i,k] = >_\delta$ **then** $select(D[i,j]) = \rho^<$;

            if $C[i,k] = <_\delta$ **then** $select(D[i,j]) = \rho^>$;

            $q[j] \leftarrow min_{\{i \mid defined[i]=j\}} select(D[i,j])$;

Figure 14: Extended algorithm for computing a maximal strict quantization

The problem of computing the maximum infinity number is equivalent to selecting a minimum number of components of the quantization which must be set to some finite number. We can transform Hitting Set [24] to this problem.

Hitting Set: Given a finite set S and $T_1, \ldots, T_N$ subsets of S, what is the smallest cardinality of a set which contains at least one element of each of the T_i's.

Reduction: We construct a comparison table C of size $N \times |S| + 1$. Each row corresponds to a T_i and each column (except the last) to an element of S.

$$C_{ij} = \begin{cases} >_\delta & \text{if element } s_j \text{ is in set } T_i \\ <_\delta & \text{if } j = |S| + 1 \text{ (the last column)} \\ =_\delta & \text{otherwise} \end{cases}$$

We also need a distance table of the same size:

$$D_{ij} = \begin{cases} 1 & \text{if } C_{ij} \neq =_\delta \\ \infty & \text{otherwise} \end{cases}$$

Let

$$\begin{aligned} H &= \text{minimum hitting set cardinality of the } T_i\text{'s} && (2) \\ I &= \text{maximum infinity number of quantization for } C \text{ and } U && (3) \end{aligned}$$

Lemma A.35 H = N - I

Proof: C looks like:

$$\begin{bmatrix} >_\delta & =_\delta & >_\delta & >_\delta & \cdots & <_\delta \\ =_\delta & =_\delta & >_\delta & =_\delta & \cdots & <_\delta \\ >_\delta & >_\delta & =_\delta & =_\delta & \cdots & <_\delta \\ \cdots & & & & & \end{bmatrix}$$

From the previous discussion, if a tuple in C has $>_\delta$ entries at (say) positions i, j, and k, then a legal quantization must be strict in component i, j, or k. Thus a legal quantization corresponds in the obvious way to selecting a hitting set for the T_i's. Minimizing the size of the hitting set corresponds to maximizing the infinity number—in fact, the size of the minimum hitting set is the same as the number of components which must be set to a finite number in a best quantization. □

To complete the proof, note that by Theorem 4.15 a quantization table can be mapped to a loop in polynomial time, thus completing the reduction. □

References

[1] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: a smart compiler and a dumb machine," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pp. 37–47, June 1984.

[2] J. A. Fisher, *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources.* PhD thesis, New York University, 1979.

[3] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," in *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 614–618, Aug. 1985.

[4] U. Banerjee, "Speedup of ordinary programs," Tech. Rep. UIUCDS-R-79-989, University of Illinois at Urbana-Champaign, Oct. 1979.

[5] A. Nicolau, *Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers.* PhD thesis, Yale University, 1984.

[6] A. E. Charlesworth, "An approach to scientific array processing: the architectural design of the AP-120b/FPS-164 family," *IEEE Computer*, vol. 14, no. 3, pp. 18–27, 1981.

[7] J. A. Fisher, "Very long instruction word architectures and the ELI-512," Tech. Rep. 253, Yale University, 1982.

[8] A. Nicolau and K. Karplus, "ROPE: a statically scheduled supercomputer architecture," in *Proceedings of the First International Conference on Supercomputing Systems*, (St. Petersburg, FL), Dec. 1985.

[9] J. Solworth and A. Nicolau, "Microflow: a fine-grain parallel processing approach," Tech. Rep. 85-710, Cornell University, 1985.

[10] *Product Summary.* Alliant Computer Systems Corporation, Acton Mass., Jan. 1985.

[11] R. W. Heuft and W. D. Little, "Improved time and parallel processor bounds for Fortran-like loops," *IEEE Trans. Computers*, vol. C-31, Jan. 1982.

[12] F. H. McMahon, *Lawrence Livermore National Laboratory FORTRAN Kernels: MFLOPS.* Livermore, CA., 1983.

[13] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: a VLSI decoupled architecture," in *Proceedings of the 12th Annual International Symopsium on Computer Architecture*, pp. 20–27, June 1985.

[14] C. L. Seitz, "The Cosmic Cube," *Communications ACM*, vol. 28, Jan. 1985.

[15] M. J. Wolfe, *Techniques for Improving the Inherent Parallelism in Programs.* Master's thesis, University of Illinois at Urbana-Champaign, July 1978.

[16] R. H. Kuhn, *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[17] D. Kuck, "Parallel processing of ordinary programs," in *Advances in Computers*, pp. 119–179, Academic Press, 1976.

[18] Y. Muraoka, "Parallelism exposure and exploitation in programs," Tech. Rep. 71-424, University of Illinois at Urbana-Champaign, 1971.

[19] R. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM*, vol. 21, pp. 201–206, 1974.

[20] K. Kennedy, "Compiling scientific programs for execution on parallel machines," Jan. 1987. Notes from a lecture presented at the 1987 SIGACT-SIGPLAN Symposium on Principles of Programming Languages.

[21] L. Lamport, "The parallel execution of DO loops," *Communications ACM*, vol. 17, pp. 83–93, Feb. 1974.

[22] R. Cytron, "Doacross: beyond vectorization for multiprocessors," in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 836–844, Aug. 1986.

[23] A. Nicolau, "A development environment for scientific parallel programs," *Applied Mathematics and Computation*, vol. 20, pp. 175–183, Sep. 1986.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.