Loop Quantization: Unwinding for Fine-Grain Parallelism Exploitation*

Alexandru Nicolau
TR 85-709
October 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work is supported in part by NSF grant DCR-8502884, and the Cornell NSF Supercomputing Center.

Loop Quantization: Unwinding for Fine-Grain Parallelism Exploitation

Alexandru Nicolau[1]

Computer Science Department

Cornell University

Ithaca, NY 14853

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 in a group of nested loops and the amount of unwinding is either fixed or has to be specified by the user, on a case by case basis. In this paper we present a general technique for automatically unwinding multiply nested loops, explain its advantages over other transformation techniques and illustrate its practical effectiveness.

[1]This work is supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

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

```
   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:
```

Figure 1: Simple Loop Unwinding: (a) Original Loop. (b) Loop Unwound 3 times.

1 Introduction

Loop unwinding has been long known as an effective way to increase the efficient utilization of pipelined machines. More recently loop unwinding has emerged as a primary technique for exploiting fine-grained parallelism within loops, notably for Very Large Instruction Word machines, a form of very tightly coupled multiprocessors [7]. Loop unwinding helps exploit fine-grain parallelism by providing a large number of operations (the unwound loop body) which can then be scheduled by operation-level code transformations such as Trace Scheduling [8] or Percolation Scheduling [17]. The operations in the unwound loop body come from previously separate iterations and thus are freer of the order imposed by the original loop. Inside this (larger) loop-body, operations can be scheduled for parallel execution (by the compiler) subject only to data dependencies. Large amounts of parallelelism which is too irregular to exploit by traditional methods (e.g., vectorization) can be found in this way.

The basic unwinding technique is very simple. The body of the loop together with the control code (i.e., counter and exit-test) is replicated a number of times. For example, 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 can sometimes be simplified by removing the intermediate tests and jumps introduced by the unwinding process. This may require static (i.e., compile-time) knowledge about the number of times the loop will be executed. For example, if the user knows that the loop will be executed a multiple of three times, the extra tests and jumps in figure 1b could be removed. Alternatively, if the pattern of memory references of the loop can be determined by static analysis (e.g., using disambiguation techniques [3], [16]), some extra memory locations can be allocated to allow extra iterations of the unwound loop to execute safely. For example, even if no information about the number of times the loop in figure 1a is executed is available at compile time, we may still safely remove the tests and jumps as before, if we add 2 extra elements to array A. Note that this increase is a function of the number of times the loop is unwound, not of the number of times the original loop body gets executed. More complex methods for the removal of tests and jumps from unwound loops exist, including testing at run-time the number of times a loop is executed and then executiing an appropriatedly unwound loop based on this information. While such methods can effectively deal with statically unpredictable loops and references, they introduce some overhead.

The above description surveys current uses of loop unwinding. But current techniques do not allow unwinding multiply nested loops. This is a major disadvantage, as it significantly limits the usefulness of loop unwinding, particularly for fine-grained parallelism architectures such as VLIW's [6], [9], ROPE [18], Microflow [20] and Alliant [2]. In this context, the parallelism may be distributed across several nested loops, so unwinding only one of the loops will not achieve the best results. This situation occurs frequently and is illustrated in section 3. In the rest of this paper we present a technique, called Loop Quantization, that overcomes this problem by allowing correct multiple-loop unwinding for arbitrary nested loops.[2] We will also describe how the decision on the bounds of Quantization can be automated. That is, we will show how to automatically determine the amount of unwinding needed to efficiently execute a loop on a particular machine. While the user may be able to make such decisions for single loop unwinding, dealing with multiple loop unwinding by hand requires significant effort and is extremely error prone. Loop quantization is being integrated in a parallelizing compiler under development at Cornell.

Quantization allows the extraction of parallelism that was previously believed to be detectable only at runtime [11]. Furthermore, quantization applies in cases where higher-level parallelism exploitation methods (like vectorization and loop interchange) do not apply, but where a significant amount of fine-grained parallelism is still present.

Loop Quantization can help achieve significant speedups in scientific code by allowing

[2]Loop unwinding should not be confused with loop jamming (fusion), a transformation that merges the bodies of two loops (that have the same iteration bounds) into a single loop, which eliminates the loop overhead for one of the loops. While both loop quantization and jamming require testing that program correctness is preserved (as do all useful transformations), the test for jamming is obvious: jamming is feasible iff given an operation of iteration j in the second loop that depends on the result of an operation done during iteration i of the first loop, we have $i \leq j$. For loop quantization, on the other hand, the test is much more complex and was previously unknown; thus automatic multiple loop unwinding was not feasible.

better pipelining and fine-grain parallelism exploitation. The main loop of weather code, for example, is naturally amenable to Quantization, as are the Livermore loops[14] in their nested context. Several of the Livermore loops (e.g., 5,6,11,24) are considered "hazard bound" [10] and do not yield to previous techniques. These loops can be successfully quantized, allowing fine-grain paralellization. The speedups achieved in our experiments range from 4 to six over the code produced for Cray-1 by the CIVIC compiler. These experiments use Percolation Scheduling to exploit the fine-grain parallelism exposed by Loop Quantization. The amount of parallelism exposed is often limited only by the hardware resources available (e.g., the number of processors). Since Quantization rearranges the order of execution of the loop iterations less than other transformations, and since it can expose even irregular fine-grain parallelism, it can be applied to a large variety of code. Furthermore, quantized loops naturally map onto parallel architectures with hypercube or cube-connected-cycles topologies (e.g., Cosmic-cube[19], Microflow[20]).

2 Loop Quantization

Given enough processors, optimal speedups can be achieved if all nested loops were fully unwound. Since no artificial constraints would be introduced by the indexing order, the only limitation would be imposed by data dependencies.[3] Techniques such as Percolation Scheduling could then be applied to exploit the available parallelism. Unfortunately, such complete unwinding is not usually feasible because of processor and memory limitations[4] and because the loop bounds are not always known at compile time. Still, even if the amount of unwinding is limited, we need the ability to unwind all loops. Unwinding just the innermost loop is not satisfactory, since it exposes parallelism only in that loop, whereas 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 way we do the unwinding is constrained by three factors:

1. Correctness. Quantization should not altering the order in which data-dependent statements are executed.

[3] A data dependency is a relationship between two instructions that use a common register or memory location. The second operation cannot begin until the first operation is finished using the register or memory.

[4] Mainly processor limitations—if we were to have enough processing power to execute the unwound loops in parallel, we could probably afford the memory.

2. Available parallelism. Unwind so as to maximize the parallelism exposed – i.e., minimize dependency chains.

3. Space considerations.

2.1 Formalization of Loop Quantization

We will now formalize the problem of Loop Quantization. This will enable us to derive formal conditions for when and how quantization can be applied. These conditions are introduced in section 2.2. The actual use of these conditions in the automatic quantizations of loops is described in section 2.3.

We will consider n nested loops, where loop n is the innermost and loop 1 is the outermost, as shown in figure 2. A *loop index* I_l is associated with each loop l. We will denote by i_l particular values of I_l. An *increment*, K_l, may also be specified for each loop l. We will denote particular values of K_l by k_l. Originally, the loops are assumed to be normalized to range from 1 to N_l with increments of 1 (our system can bring them into this normal form).

In the following discussion we assume that the n loops are the only ones affected by the quantization and that indirect references (i.e. references involving computed addresses) are linear functions of the iteration vectors.[5] We will also assume that two references to an array A, namely $A[i_1, i_2, ..., i_n]$ and $A[j_1, j_2, ..., j_n]$ are equal if and only if $i_1 = j_1, i_2 = j_2, ..., i_n = j_n$. 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 with $I_i + 1$, and so on, all the way up to $I_i - k_i - 1$. 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 the nested loops fully, when their upper bounds are $k_1, .., k_n$ respectively. For this unwinding to be useful, it must preserve the semantics of the original loop. Informally, this means that each statement S executed for some original index-vector value $\bar{i} = (i_1, ..., i_n)$, will have available to it exactly the same data when the unwound loop is executed as it would have in the original loop. In particular, if the statement was accessing a value computed in an iteration preceding it (i.e., calculated by a

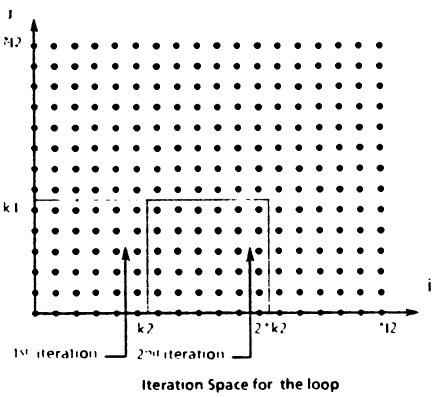
[5] An iteration (or index) vector [13] consists of the seting of the induction variables that uniquely identify an iteration of the nested loops.

```
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 2 dimensional loop quantization

statement with an index vector $\bar{j} \leq \bar{i}$, where "$\leq$" denotes lexicographic ordering[6]), it should be able to access this value in the unwound loop. Similarly, the statement S should not access values computed in later iterations of the original loop(s). Our goal is to find a set $\{k_1, ...k_n\}$ of loop increments (steps) such that the quantized loops preserve the semantics of the original ones (i.e., given the same input they both produce the same output).

If only the innermost loop is unwound no problems arise, as this preserves the execution order of the iterations. However, when other loops are also unwound, the looping consists of a *n-dimensional* box B with dimensions k_1 by k_2 by...k_n (see figure 2). All the 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 the normal loop order (i.e., first along dimension n, then along dimension $n-1$, etc). This obviously alters the order of execution. A statement S_1 in the original iteration $(i_1+1, i_2, ..., i_n)$ may be included in this box (in fact it will be, if loop 1 is unwound at all) and thus may be executed before a statement S_2 in the original iteration $(i_1, ..., i_n + k_n)$ (which already falls outside the box), even though in the original execution the latter would precede the former. This will not present any problem if S_1 does not require the results of S_2, or if S_2 does not use information that is altered by S_1. If such dependencies exist, however, the particular quantization is illegal, although other quantizations may succeed. We will discuss the chioce of a particular quantization in section 2.3. We need not worry about incorrect execution order when both

[6]That is, the relation is determined by the pairwise comparison from left to right of the elements of $\bar{i}$ and $\bar{j}$. The first pair of elements that are not equal determines the result of the comparison. This comparison can often be made (symbolically) even when the elements are expressions involving variables. For example the expression $2j + 3i + 4$ is greater than or equal to $i + 3$ for all positive values of i and j. Our system can automatically determine this. Note that this is *not* syntactic comparison, but symbolic evaluation akin to MACSYMA [4].

dependent statements are inside the same box since the execution order of such statements is preserved by the quantization.[7]

To fix the dependency problem, we could increase the size of the box to include any dependent references. This would ensure correctness, since all dependent pairs of operations whose order would be reversed as a result of quantization would now be contained in the box and thus executed subject to their dependencies. Unfortunately, this process can be self replicating, and the unwinding might have to repeat until the bound of the array encompasing the problematic references is reached. Since in many applications one of the dimensions of the array is quite small, full unwinding on that dimension may be reasonable.

In cases where dimensions are large or full unwinding along some dimension cannot resolve the problem, the alternative is to only allow unwindings of loops which do not cause dependencies to be violated. For example, a quantization yielding a box B bounded by $(i_1, ..., i_n)$ and $(i_1 + k_1 - 1, ...i_n + k_n - 1)$ must be disallowed if B contains references to a variable written in the original iteration $\bar{j} = (i_1, ..., i_{n-1} + k_{n-1} - 2, i_n + k_n)$, which occurs in a box executed after B. Limiting the amount of unwinding to avoid violating dependencies should not be much of a constraint as long as we are utilizing the (limited) resources fully. If utilization is too low we can unwind one of the loops further.

2.2 Loop Quantization Conditions

2.2.1 Dependencies

We will now formally define the dependencies and the conditions under which they may be violated by loop quantization. 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:

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.

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 all statements connected by such dependencies must be preserved to ensure semantic correctness. The order of independent statements is irrelevant.

[7]Since Loop Quantization changes the order of the iterations less drastically than more global transformations (e.g., interchange), quantization can succeed even when global transformations would not apply.

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. Since scalar variable accesses are invariant with respect to the 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 only consider dependencies caused by indirect references occurring in different iterations.

Memory disambiguation techniques for determining (to the extent possible at compile time) whether two indirect references might access the same memory location can be found in [3], [16]. They involve the derivation of primitive expressions for the array indexes. These expressions contain compile-time constants, and variables whose values cannot be derived at compile-time. To determine whether two references conflict, the primitive index expressions are symbolically equated and the resulting diophantine equation is solved. If there are (integer) solutions to the equation, then the two references might access the same memory location, and a potential conflict (data-dependency) must be assumed. For simplicity of notation we will assume that index references are reduced to the primitive form $(a_1 i_1 + b_1, ..., a_n i_n + b_n)$, where $\vec{i}$ is the iteration vector, and the a_l's and b_l's are constants. The following discussion applies to arbitrary array indexes. Of course, the more complex the index expressions, the less likely they are to be disambiguated by the compiler increasing the number of dependencies that have to be assumed.

2.2.2 Quantization Conditions

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

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

for values of i_l's such that the j_l's are all integers (this must be possible, otherwise there would be no conflict).

Equating $\vec{i}$ and $\vec{j}$ elementwise from left to right, we may determine whether $\vec{i} < \vec{j}$, or $\vec{i} > \vec{j}$. For quantization to be legal, we need to ensure that the order of the conflicting references is preserved under quantization. Assuming that original iteration $\vec{i}$ occurs in a box with *lower bound*[6] $\overline{L} = (l_1, ..., l_n)$ and original iteration $\vec{j}$ occurs in a box with lower bound $\overline{L} = (l'_1, ..., l'_n)$,

[6]The lower bound of a box B is the index vector of the quantized loop for which the original loop iteration with index vector $\vec{i}$ gets executed.

then one of the following conditions must hold:

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

Since 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 get:

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

As with the original iterations, the lower bounds of the quantized boxes occur in lexicographic order. That is, the box associated with $\bar{L}$ will precede during execution the box associated with $\bar{L'}$ if and only if $\bar{L} < \bar{L'}$. Thus by element-wise (symbolic) comparison we can determine if either one of the two conditions above is satisfied.

2.3 The Use of Loop Quantization Conditions

The conditions presented above can be used in several ways. The simplest is to let the system perform a prespecified unwinding (i.e., use some fixed k_e's, possibly determined by range analysis of the bounds of arrays and loops). Dependent statements are checked against the conditions derived in the previous section, for all values of each I_e. If the conditions are satisfied then the unwinding is performed, otherwise k_e's are decreased and the process is repeated. While this could be extremely expensive, it provides a straightforward solution for cases where only a few nested loops with few dependent statements exist and the N_i's are known at compile time. A deficiency of this approach is that it does not provide any information about what k_e's may be appropriate or on how to unwind so as to best utilize the space we are willing to trade for speed.

A more reasonable approach, wich we are incorporating in our percolation scheduling compiler improves efficiency and applies even when the bounds of the loops are not known. 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 $\bar{L}$ and $\bar{L'}$ 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:

- $\bar{L} \geq \bar{L'}$. If this is the condition needed to ensure the legality of the transformation, the quantization can be performed, using arbitrary k_e's. Otherwise an attempt is made to change "$\geq$" to "$=$" to ensure correctness. This may require full unwinding on one (or more) dimensions.

- $\overline{L} \leq \overline{L'}$. If this is the condition needed to ensure the legality of the transformation the quantization can be performed using arbitrary k_e's. Otherwise an attempt is made to change $\leq$'s to $=$ to ensure correctness. This may require full unwinding on one (or more) dimensions.

- *Undetermined situation.* This will occur whenever we have no strict ordering (i.e., none of "<" or ">" or "=" hold) between $\overline{L}$ *and* $\overline{L'}$, and there are some "$\leq$'s" and "$\geq$'s" between various elements. For example, for $\overline{L} = (\lfloor \frac{a+1}{k} \rfloor, \lfloor \frac{b}{k} \rfloor)$ and $\overline{L'} = (\lfloor \frac{a}{k} \rfloor, \lfloor \frac{b+1}{k} \rfloor)$ we have $\lfloor \frac{a+1}{k} \rfloor \geq \lfloor \frac{a}{k} \rfloor$ and $\lfloor \frac{b}{k} \rfloor \leq \lfloor \frac{b+1}{k} \rfloor$. This is undetermined, since

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

  In such cases we may obtain the desired relation by changing $\leq$'s to $=$'s (or $\geq$'s to $=$'s) by using different k_e's, or if that fails, possibly by full unwinding on some dimensions.

In general, $i_e - j_e \geq k + (c - c') \Rightarrow \lfloor \frac{i_e}{k} \rfloor > \lfloor \frac{j_e}{k} \rfloor$, where $i_e = dk + c$ and $j_e = d'b + c'$. This together with the other constraints on the various variables can be used with integer programming techniques to find a set of maximal k_e's satisfying the above conditions. The simpler alternative used in our compiler is to establish an upper limit on the unwinding, say K_x for loop x. Such a bound occurs naturally in practice, as a function of the size of resources (the number of processors) available. In this case, we can start with K_x and apply the legality tests for quantization. if the tests succeed, we're done, and if they fail we repeat the process, performing a binary search on the range of possible values $k_x \leq K_x$. In the worst case, we will have performed $log(K_x)$ attempts before finding a legal k_x.

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

As for space efficiency, intuitively, there is no point in unwinding on a dimension if dependencies exist on that dimension that prevent software pipelining or parallelization.[9] 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 may be done in parallel. Whenever additional unwinding is not deemed to be worth the cost (a user or system settable parameter) unwinding stops.

This method 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 real (run-time) speedup achieved by a given unwinding. This accuracy can be significantly improved by the use of conditional-jump probability information. Such information can be obtained by the system using test runs and/or analysis, or can be supplied by the user. We have found such information to be easily available in typical scientific code.

2.4 Folding and Balancing

To best exploit the parallelism exposed by quantization, we can apply another transformation similar to tree height reduction [12]. This expresses each left-hand-side (LHS) of an assignment in terms of initial values or irreducible variables, by using repeated back-substitution (and constant/variable folding [16]) on the quantized body of the loops. The highest tree is then balanced by using efficient algorithms that take advantage of commutativity, distribution, and associativity [15], [5]. Subtrees corresponding to other LHS—if any—are identified and reused (common subexpression elimination). The process is repeated for all independent statements in the loop and each operation is scheduled for execution subject to its dependencies (e.g., by Percolation Scheduling).

If the machine is "reasonably well utilized" by the final schedule (a system or user specified bound), or the code size is getting to be "large" (another user or system specified bound) quantization stops. Otherwise, we increase the unwinding (incrementally) and build a new overall schedule. If the new part of this schedule is isomorphic to the part added in the previous schedule and there is no overlap between them (i.e., a *no-gain situation*—the trees have exactly the same structure and the two halves have to be scheduled completely sequentially to ensure semantics preservation) then the last unwinding was not worthwhile, so we undo it and stop.

[9] See [13] for a rigorous definition of loop carried dependencies.

```
   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

Otherwise we repeat the process until either one of the bounds is reached or a no-gain situation is encountered.

3 Examples

3.1 Example 1

As an illustration, consider the way Loop Quantization techniques deal with recurrences. In figure 3 we have a simple piece of code. Transformations such as loop interchange or vectorization do not apply here. For example, in the original loops, $X[17,2]$ will be read in iteration ($i = 1, j = 2$), and written in iteration ($i = 16, j = 1$). Since the loop on j is innermost, $(1,2)$ occurs before $(16,1)$ and thus we have a read before a write. Reversing the loops will write into $X[17,2]$ in iteration ($j = 1, i = 16$) and read from it in iteration (j=2,i=1); but since the loops are now reversed, the write will occur before the read, which is obviously not semantically equivalent to the original code. The interchange is therefore illegal. Vectorization is unfeasible, even if we use expansion and loop distribution (breaking); the first statement could be vectorized, but that would only reduce the execution from $4 * n^2$ to $3 * n^2 + 1$ steps, even assuming an array of processors of size n. This would be wasteful if n is large. A similar speedup ($3 * n^2$ execution steps) could be achieved on a multiprocessor by running the first two statements in parallel. Even if more than two processors were available on a traditional multiprocessor, attempting to distribute the loop across the processors will incur very heavy communication costs due to the tight dependencies between the iterations.

Our techniques, on the other hand, can quantize the loop by unwinding on both i and 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

which will expose enough fine-grain parallelism to keep the available processors busy (e.g., in a VLIW machine, or any other type of tightly coulpled multiprocessor). This quantizing is legal even though loop interchange is impossible. This is because quantization, unlike interchange, preserves the relative order of execution of the critical statements inside the "box". By comparing

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

and

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

our disambiguator system can determine that conflicts between iteration $\bar{i} = (i, j)$ and $\bar{j} = (i', j')$ can occur when $i' = i - 15$ and $j' = j + 1$. Since $i > i'$, it follows that $\bar{i} > \bar{j}$ 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, K_j. However, since $i - i' = 15$, chosing $K_i = 15$ will ensure that

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

and therefore $\overline{L} > \overline{L'}$. K_j has no influence over this, and thus any value choosen for it will still preserve correctness. For this example we will assume that $K_j = 15$ as well. The other dependency that needs to be examined for the legality of loop quantization is between

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

and

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

where $\bar{i} = (i, j)$, $\bar{j} = (i' = i, j' = j-1)$, and thus $\bar{i} > \bar{j}$. 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 each other, and can be done in parallel. Applying folding and balancing (tree height reduction) techniques to each such group (for j to j+14), we can, resources permitting, execute each 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, we need 16 processors per unwinding on i, for a total of 15*16 processors[10]. The speedup achieved is from $4*n^2$ steps to $6*(n/k_1)*(n/k_2)$, (in our example, $k_1 = k_2 = 15$). If more resources were available, we could unwind more, getting even more dramatic speedups. In particular, if the loops were fully unwound the speedups would be optimal. Even when this is impossible, we will still get good performance given the available resources.

3.2 Example 2

This example illustrates how Loop Quantization can expose parallelism previously observable only at runtime (e.g., [11]). The following is an illustration of this point, based on Heft's and Little's original example in [11].

To simplify the discussion we will assume that intermediate exit-tests have been removed by any of the methods described above. Consider the original loop in Heuft's and Little's example, shown in figure 6. As a result of the tests described in section 2.3, we can determine automatically that the loop can be quantized safely. The details of the quantization proper for this loop are similar to those for the previous example and are omitted. Based on symbolic analysis, the system can determine a correct and effective quantization subject to the data-dependencies present and the resources available. Assuming that we had enough processors[11] the quantization shown in figure 7 would yield optimal speedups.

If more processors were available, we would like to automatically infer by analyzing the

[10] 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.

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

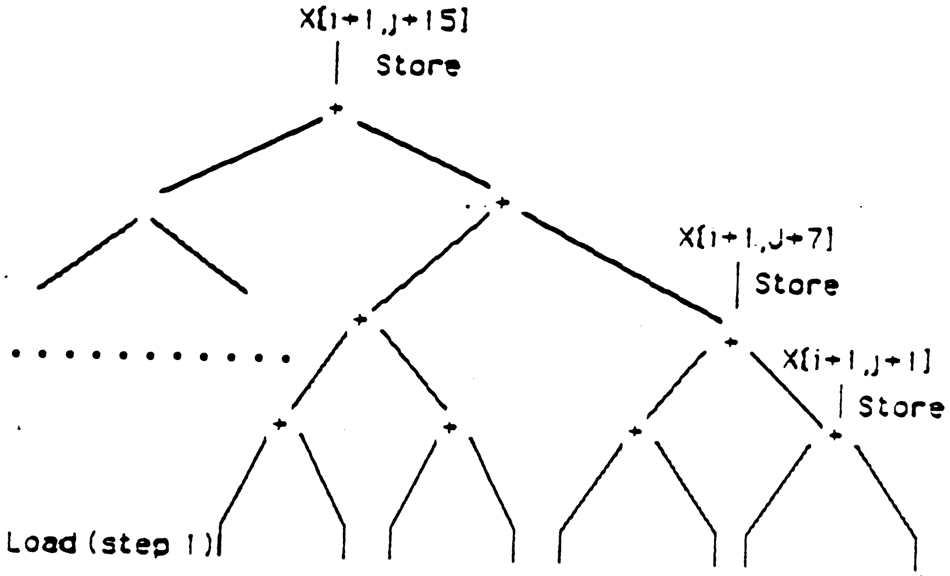


Figure 5: Balanced tree for i, j..j+14

```
     DO 100 I1 = 0, N
     DO 100 I2 = 0, N
     DO 100 I3 = 0, N
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]

```
      DO 100 I1 = 0, N, 9
      DO 100 I2 = 0, N, 9
      DO 100 I3 = 0, N, 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

dependency pattern (figure 7) that unwinding more on the innermost loop would not lengthen the execution time, since this would not introduce any longer dependency-chains. While we can achieve this by explicitly unwinding as described in the previous section, such symbolic inference would be more efficient, and is the goal of current research. The system could then decide to unwind on the inner loop until all the processors are utilized. Finally the actual code generation has to be done, to map the 3-d structure of the quantized loop body, onto the actual hardware. This can be achieved naturally (subject to the dependency constraints discovered by the system) for a machine with a cube-connected-cycles topology, such as Microflow [20]. Ideally, the system could express this information implicitly in a compact symbolic representation such as that shown in figure 8, rather than explicitly (i.e., explicit unwinding) as currently done. The code derived from such a representation for execution on the Microflow architecture is shown in figure 9.

This is equivalent, given the number of processors used, to the bounds claimed by Heuft and Little.

```
For X=0,1,...
Set A = {All instances of S1 for index values:
                    (I3=X,X+9; (I2=X,X+9; (I1=i)))
            All instances of S2 for index values:
                    (I3=X,X+9; (I2=X; (I1=X,X+9))) }

Set B = {All instances of S1 for index values:
                    (I3=X,X+9; (I2=X; (I1=X+1,X+9)))
            All instances of S2 for index values:
                    (I3=X,X+9; (I2=X+1,X+9; (I1=X;))) }
```

Figure 8: Compact representation of Quantized loop

```
0. X:=0;
1. Broadcast X;
2. Execute all members of Set A in parallel;
3. Execute all members of Set B in parallel;
4. Increment X and Repeat steps 1..4, unless Set B is empty.
```

Figure 9: Compact representation of Quantized loop

References

[1] J.R.Allen and K.Kennedy. *Automatic Loop Interchange.* In the Proceedings of the Symposium on Compiler Construction, SIGPLAN Notices, Vol.19 No.6, 1984.

[2] Alliant. Product Summary. *Alliant Computer Systems Corporation.* Acton Mass. January 1985.

[3] U.Banerjee. *Speedup of Ordinary Programs.* University of Illinois Computer Science Technical Report UIUCDS-R-79-989, Oct. 1979.

[4] R.Bogen. *MACSYMA Reference Manual.* Symbolics Inc., Cambridge, Mass. December 1983.

[5] R.Brent. *The Parallel Evaluation of General Arithmetic Expressions.* Journal of the ACM 21, pp. 201-206, 1974.

[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, J.R.Ellis, J.C.Ruttenberg and A.Nicolau *Parallel Processing: A Smart Compiler and a Dumb Machine.* Proc. of the ACM Symposium on Compiler Construction, 1984.

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

[9] J.A.Fisher *Very long instruction word architectures and the ELI-512.* Yale University Department of Computer Science, Technical report # 253, 1982.

[10] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, H. C. Young. *PIPE: A VLSI Decoupled Architecture.* The 12^{th} Annual International Symopsium on Computer Architecture, June 17–19, 1985, Boston, MA, 20–27.

[11] R.W.Heuft and W.D.Little. *Improved Time and Parallel Processor Bounds for Fortran-like Loops.* IEEE Transactions on Computers Vol.31, No.1, 1982.

[12] D.J.Kuck. *Parallel Processing of Ordinary Programs.* In Advances in Computers, Vol. 15, pp. 119-179, 1976.

[13] R.H.Khun. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks and Decision Trees.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1980.

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

[15] Y.Muraoka. *Parallelism Exposure and Exploitation in Programs.* University of Illinois, Urbana, Dept. of Computer Science, Tech. Rep. 71-424, 1971.

[16] A.Nicolau. *Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers.* Yale University Ph.D. Thesis, June 1984.

[17] A.Nicolau. *Percolation Scheduling: A Parallel Compilation Technique.* Cornell University, Dept. of Computer Science Technical Report TR-85-678, May 1985.

[18] A. Nicolau and K. Karplus. ROPE: a Statically Scheduled Supercomputer Architecture. *First International Conference on Supercomputing Systems*, St. Petersburg, FL, December 1985.

[19] C.L.Seitz. *The Cosmic Cube.* Communications of the ACM, Vol.28, No.1 January 1985.

[20] J.Solworth and A.Nicolau. *Microflow: A fine-grain Parallel Processing Approach.* Cornell University, Dept. of Computer Science Technical Report TR-85-710