A BRIEF SURVEY OF TECHNIQUES OF LOOP TRANSFORMATION

Leila A. Plummer, Nelson L. Passos, and Ranette Halverson
Department of Computer Science
Midwestern State University
3410 Taft Boulevard
Wichita Falls, TX 76308
plummer@cs.mwsu.edu
(fpassos|rhalver)@nexus.mwsu.edu
http://www.cs.mwsu.edu/~caesar

ABSTRACT

Techniques of loop transformation—the modification of loops in such a way that the new and previous loops produce the same results—are often used to increase task scheduling efficiency in parallel systems. Because of the knowledge prerequisite for mastering them, however, they often prove difficult to investigate and their methodologies can be misunderstood. This paper explains several of the most important techniques of loop transformation, including unrolling and retiming, and presents the background necessary to understand them.

1. INTRODUCTION

Despite increases in computer technology, performance demands seem to be always one step ahead of current capabilities. Continual improvements in speed are required. Since scientific and engineering applications pass most of their execution time in loops, optimization research has been centered mainly in the area of loop restructuring [20]. Although other methods for loop restructuring exist, such as the affine-by-statement technique [4, 5], the index shift method [11], the MD rotation technique [19], software pipelining [1, 6, 7], and push-up scheduling [2, 12, 13, 18], the most common is the use of more trivial techniques of loop transformation, such as loop interchange, loop fusion, loop coalescing, loop unrolling, loop skewing, and loop retiming [3, 8, 9, 14, 20].

Arguably one of the most important concepts of parallel processing [15], a loop transformation is the rearrangement of statements within a code loop, done in such a fashion that the equivalence of the loop is maintained. It is normally performed at compile time [9], and is an umbrella term which covers several methods, each of which possesses its own rules. Usually, several loop transformations are combined in order to achieve the best results. Choosing appropriate techniques and applying them in an
appropriate order requires an understanding of how they operate.

A simple example is loop interchange. Loop interchange successfully switches, or interchanges, the order of loops in order to increase parallelism. The code in figures 1(a) and 1(b) are identical, except for one distinction: the \( i \) and \( j \) loops have been switched. The results of both loops are identical. Despite this, however, parallelism is increased in figure 1(b). This occurs because the potential now exists for all iterations of the \( i \) loop to be run simultaneously.

\[
\begin{align*}
\text{for } i &= 1 \text{ to } 100 \text{ do} \\
&\quad \text{for } j = 1 \text{ to } 100 \text{ do} \\
&\quad \quad A[i,j] = A[i,j-1] + 2; \\
&\quad \text{endfor;}
\end{align*}
\]

\[
\begin{align*}
\text{for } j &= 1 \text{ to } 100 \text{ do} \\
&\quad \text{for } i = 1 \text{ to } 100 \text{ do} \\
&\quad \quad A[i,j] = A[i,j-1] + 2; \\
&\quad \text{endfor;}
\end{align*}
\]

(a) (b)

Figure 1. Loop body (a) before and (b) after loop interchange.

The origin of this paper may be found in an online tutorial [21] that the authors developed in order to explain loop transformation basics to incoming research assistants. Because some of the methods have few references in literature on the subject, several difficulties were encountered. Sources were hard to find, and some published information presented minor flaws. For instance, one of the main sources used in the preparation of the online tutorial [9] incorrectly presented an example of the loop skewing technique.

In order to present the loop transformations material, the remainder of this paper is organized as follows. Ideas necessary for understanding loop transformations are given in the next section. Section 3 consists of an overview of several of the most important techniques of loop transformation, and is followed by a summary, which concludes the paper.

2. BACKGROUND

A major challenge in parallel processing is task scheduling, the process of deciding which instructions will be run by which processor, and in which order. Loop transformations are often used to make task scheduling easier and more efficient. A common aid used in scheduling tasks is the task graph.

A task graph consists of a set of nodes and directed edges, where nodes represent tasks and an edge from node A to node B signifies that task A must be completed before task B. Each node is divided into two sections by a horizontal line. The top half of a node contains the task number, and the bottom half contains the task’s execution time [9]. Figure 2 shows a task graph and its corresponding pseudocode.
for i = 1 to 100 do
1: A[i] = B[i] + 2;
2: C[i] = A[i] + 4;
3: D[i] = A[i] + 4;
4: E[i] = C[i] + D[i];
endfor:

(a) 
(b)

Figure 2. (a) A task graph. (b) Its corresponding pseudocode.

One of the primary challenges of task scheduling is balancing communication delays with parallelism [9]. A communication delay is the amount of time needed to transfer information from one task being run on one processor to another task being run on another processor. Increasing parallelism usually increases communication delays. Task scheduling determines the number of communication delays present during execution time, and so effective scheduling must pay close attention to this problem. Two possible schedulings for the task graph of figure 2, represented by two Gantt charts, are given in figures 3(a) and 3(b). Figure 3(a) shows a minimum of communication delays since all tasks are run consecutively. Figure 3(b) shows increases in both parallelism and communication delays. To illustrate the importance of minimizing communication delays, a third potential scheduling is given in figure 3(c). This schedule, which has only a slight improvement over 3(a), however takes longer to run that does 3(b).

A second important concept in the area of task scheduling is that of data dependence. Data dependence refers to the precedence relations of data manipulation, and several types exist. Task B is said to be flow dependent on task A if A writes to a location and then B reads from it. B is anti-dependent on A if A reads from a location and then B writes to that same location. B is output dependent on A if A writes to a location and then B writes to that location [9]. It can be shown that instances of anti-dependence and output dependence can be easily eliminated by the renaming of variables. Figure 4, for example, shows how anti dependence can be removed. In the code of 4(a), B is anti-dependent on A because A reads from location C, and then B writes to it. This dependence can be removed by having B write to a different location, as is done in the code of 4(b). Since flow dependence is the only critical type for our purposes, we will consider all data dependences to be flow data dependences. Therefore, task dependencies, within the rest of this paper, will have to do with how many loop iterations a task must wait before being executed.
Figure 3. Gantt charts showing 3 possible schedulings of the task graph of figure 2.

(a) \hspace{4cm} (b) \hspace{4cm} (c)

\begin{verbatim}
for i = 1 to 100 do 
1: A[i] = C[i] * 3;
2: C[i] = B[i] + 2;
endfor;
x = C[100];
\end{verbatim}

(a) \hspace{4cm} (b)

Figure 4. An example of code (a) before and (b) after the removal of anti-dependence.

We use Multi-Dimensional Data Flow Graphs (MDFG’s) as an aid to understanding task dependencies, as we used task graphs to explain communication delays. MDFG’s are cyclic data flow graphs used to represent nested loops. Each node implies a task, or instruction, in the nested loop, and each edge implies a dependency between two operations. An edge label indicates multi-dimensional iteration delays [17]. An example of an MDFG and its corresponding loop body is shown in figure 5.

\begin{verbatim}
for i = 1 to 100 do 
for j = 1 to 100 do 
 A[i,j] = D[i-1,j+1];
 C[i,j] = A[i-1,j] + 3;
 D[i,j] = B[i,j] + C[i-1,j-1];
endfor;
endfor;
\end{verbatim}

(a) \hspace{4cm} (b)

Figure 5. (a) An MDFG. (b) Its corresponding pseudocode.
3. LOOP TRANSFORMATIONS

Now that a basic background on how to model loops, which will be subject to transformation techniques, has been given, it is time to explain some of these techniques. As mentioned earlier, loop transformations are manipulations of code loops that do not change the loops’ results. Their purpose is to improve speed by reducing overhead or increasing the number of tasks that can be executed in parallel. Several of the most significant methods are loop interchange, loop fusion, loop coalescing, loop unrolling, loop skewing, and loop retiming.

\[
\begin{align*}
\text{(a)} & \quad \text{for } i = 1 \text{ to } 100 \text{ do} \\
& \quad 1: A[i] = B[i] * 3; \\
& \quad 2: C[i] = A[i-2]; \\
& \quad \text{endfor;}
\end{align*}
\]

\[
\begin{align*}
\text{(b)} & \quad \text{for } i = 1 \text{ to } 100 \text{ do} \\
& \quad 1: A[i] = B[i] * 3; \\
& \quad \text{endfor;}
\end{align*}
\]

\[
\begin{align*}
\text{(c)} & \quad \text{for } i = 1 \text{ to } 100 \text{ do} \\
& \quad 1: A[i] = B[i] * 3; \\
& \quad \text{endfor;}
\end{align*}
\]

\[
\begin{align*}
\text{(d)} & \quad \text{for } i = 1 \text{ to } 100 \text{ do} \\
& \quad 1: A[i] = B[i] * 3; \\
& \quad 2: C[i] = A[i+2]; \\
& \quad \text{endfor;}
\end{align*}
\]

Figure 6. Loop body (a) before and (b) after loop fusion, and (c) before and (d) after an unsuccessful “attempt” at loop fusion.

Loop fusion, also known as loop jamming, is a way of combining two similar, adjacent loops. Its purpose is to reduce overhead, and it may be performed whenever data dependence relations will not be infringed. The loops in figure 6(a) can be fused, since no data dependence relation would be changed. The resulting loops are shown in figure 6(b). The loops of figure 6(c), however, cannot be fused. Figure 6(d) shows the result of an unsuccessful “attempt” at loop fusion. Notice that in 6(c), statement 2 uses elements produced by statement 1, while in 6(d), statement 2 does not. Instead, it uses whatever is in A before statement 1 has been executed. The data dependence relation is not preserved—statement 2 in fact changes from being flow dependent to anti dependent on statement 1—so loop fusion is not possible.

Loop coalescing is another loop transformation technique, which is similar to loop fusion in that it combines loops. It is different because it begins with nested rather than adjacent ones. It converts a multi-dimensional iteration space into a one-dimensional iteration space, simplifying the scheduling problem and reducing overhead. Figure 7 shows the results of an instance of loop coalescing.
for $i = 1$ to 100 do
  for $j = 1$ to 100 do
  endfor;
endfor;

for $i = 1$ to 10,000 do
  $A[\text{ceiling}(i/100), (i - 1 \mod 100) + 1] = B[\text{ceiling}(i/100), ((i-1) \mod 100) + 1]$;
endfor;

(a) 

Figure 7. Loop body (a) before and (b) after loop coalescing.

While loop fusion and loop coalescing try to reduce overhead caused by loop control, other techniques try to improve the parallel execution. Loop unrolling peels off iterations at the beginning, middle, or end of a loop in order to discover loop-carried dependencies which may allow several iterations to be executed at the same time. In figure 8, loop unrolling makes several parallel possibilities apparent. Iterations 3 and 4, followed by 5 and 6, can be run simultaneously, for instance.

A slightly more complex example of loop unrolling is shown in figures 9 and 10. Figures 9(a) and 10(a) show a pair of loops before loop unrolling and its corresponding MDFG, respectively, and figures 9(b)-(d) and 10(b) present the results of loop unrolling. As becomes evident in figure 9(d), $a^{00}$ and $a^{01}$ are not dependent on each other, and neither are $a^{10}$ or $a^{11}$. So $a^{00}$ and $a^{01}$ can be run together in parallel, as can $a^{10}$ and $a^{11}$. This information can be given to a scheduler, and thus the number of tasks done in parallel can be raised. Without loop unrolling, the parallelism would have stayed invisible.

for $i = 3$ to 9 do
endfor;


(a) 

Figure 8. Loop body (a) before and (b) after loop unrolling.
for $i = 1$ to 100 do
  for $j = 1$ to 100 do
  endfor;
endfor;

for $i = 1$ to 100 do
  for $j = 1$ to 100 step 2 do
    $a^{00}$: $A[i,j] = A[i-1,j]$;
  endfor;
endfor;

(a)

for $i = 1$ to 100 step 2 do
  for $j = 1$ to 100 do
  endfor;
endfor;

(b)

Figure 9. Loop body (a) before unrolling, (b) after unrolling innermost loop once, (c) after unrolling outermost loop once, and (d) after unrolling both loops at once.

![Diagram](attachment:image.png)

Figure 10. MDFG for loop bodies of (a) figure 9a and (b) figure 9d.

Another tool used for parallel execution improvement is loop skewing. Loop skewing takes a given nested loop and changes the indices in such a way that increased parallelization becomes possible, while changing the iteration space and maintaining dependencies. The iteration space changes from square to rhomboid. Figure 11(a) shows a pair of loops, and figure 12(a) represents their iteration space. As is evident, $A[3,3]$ is computed during iteration $(i,j) = (3,3)$. $A[3,3]$ depends on $A[2,3]$ and $A[3,2]$, which are determined in $(i,j) = (2,3)$ and $(i,j) = (3,2)$. As the loops are
currently indexed, no potentials for parallelism are obvious. Since each task depends on the one computed directly before it, execution must be sequential. If we could skew the indices, however, it might be possible to uncover some possibilities. Figures 11(b) and 12(b) show the results of this skewing. Now all \( j \) iterations can be run in parallel, without affecting task dependencies. The values computed in iteration \((i, j) = (3,3)\) is now computed in iteration \((4,3)\). This task depends on what was computed in \((3,2)\) and \((2,3)\), which are now computed in \((3,2)\) and \((3,3)\). They are computed in the \( i=3 \) iteration, and the task in question is computed in the \( i=4 \) iteration.

```
for i = 2 to 5 do
    for j = 2 to 5 do
    endfor;
endfor;
```

```
for i = 2 to 8 do
    for j = max(2,i-3) to min(5,i) do
    endfor;
endfor;
```

Figure 11. Loop body (a) before and (b) after loop skewing.

![Figure 12](image)

Figure 12. Iteration space for loop bodies of (a) figure 11a and (b) figure 11b.

Like loop skewing, loop retiming[16] also shifts the order of execution in such a way that increased parallelization becomes possible, while maintaining dependencies. A difference is that retiming changes the internal scheduling of the iterations, while loop skewing does not.

Retiming changes the sequence of execution of statements within a loop body by applying a retiming function. In the case of a two-dimensional set of nested loops, a retiming function is defined by two numbers and is applied to a task. Within the task, index variables controlled by the outer and inner loops are increased respectively by the first and second numbers. An example is shown in figures 13 and 14.
for i = 1 to 100 do
  for j = 1 to 100 do
    D[i,j] = B[i-1,j] + C[i-1,j-1];
    A[i,j] = D[i,j] + 1;
    C[i,j] = A[i,j] * 2;
  endfor;
endfor;

for i = 1 to 100 do
  for j = 1 to 100 do
    D[i,j+1] = B[i-1,j+1] + C[i-1,j];
    A[i,j] = D[i,j] + 1;
    C[i,j] = A[i,j] * 2;
  endfor;
endfor;

Figure 13. Loop body (a) before and (b) after loop retiming by r(D) = (0,1).

(a)                  (b)

Figure 14. MDFG for loop bodies of (a) figure 13(a) and (b) figure 13(b).

In the loops of figures 13 and 14, loop retiming makes visible several parallel possibilities. Before retiming, D must be run before A, and A must be run before B and C. After retiming, D may be run simultaneously with A, although B and C must still wait until after A has been executed. While some improvement has been made, the loop still cannot be run entirely in parallel. Chained retiming, the successive application of retiming functions to the loop [16], may however yield complete parallelism. Consider retiming the loops of 13(a) by r(D) = (0,2) and r(A) = (0,1), the results of which are shown in figure 15. Now all four statements A, B, C and D may be executed in parallel.

It may be useful to again compare loop skewing and loop retiming. Loop skewing shifts the order of the execution of iterations, while loop retiming shifts the order of the execution of the within the loop body. In our examples, loop skewing was done in order to cause all j iterations to be executable in parallel, while loop retiming was performed in order to cause A, B, C, and D to be executable in parallel.

Up to this point, we have ignored the fact that we have occasionally “lost” or “gained” some statements during our discussions of transformations. For instance, in the loop retiming shown in figures 13(a) and 15, the first element of the D array to be filled changes from [100,100] to [100,102]. Clearly, this is unacceptable. The
prologue and epilogue, short pieces of code which respectively precede and succeed the nested loops, are designed to correct this oversight. Although for convenience the final result of loop retiming was shown to be a single pair for nested loops, it is in fact a pair of nested loops, plus the prologue and epilogue. Figure 16 shows the complete result of retiming the code of figure 13(a).

for i = 1 to 100 do
  for j = 1 to 100 do
    D[i, j+2] = B[i-1, j+2] + C[i-1, j+1];
    A[i, j+1] = D[i, j+1] + 1;
    C[i, j] = A[i, j] * 2;
  endfor;
endfor;

Figure 15. (a) Result of loop retiming the loops of figure 13(a) by r(D) = (0, 2) and r(A) = (0, 1). (b) The corresponding MDFG.

for i := 1 to 100 do
  //prologue:
  D[i, 1] = B[i-1, 3] + C[i-1, 2];
  D[i, 2] = B[i-1, 4] + C[i-1, 3];
  A[i, 1] = D[i, 2] + 1;
  //optimized loop
  for j = 1 to 98 do
    D[i, j+2] = B[i-1, j+2] + C[i-1, j+1];
    A[i, j+1] = D[i, j+1] + 1;
    C[i, j] = A[i, j] * 2;
  endfor;

  //epilogue:
  A[i, 100] = D[i, 101] + 1;
  B[i, 100] = A[i, 100] + 3;
  C[i, 100] = A[i, 100] * 2;
endfor;

Figure 16. Result of loop retiming the loops of figure 11a by r(D) = (0, 2) and r(A) = (0, 1), including prologue and epilogue.

4. SUMMARY

Loop transformations, which are manipulations of code loops that leave results unchanged, can lead to computation speed-up and are regarded as one of the most important areas of parallel processing. Techniques of loop transformation vary in their rules and uses, and are often used in combination. Several of the most significant methods are loop interchange, loop fusion, loop coalescing, loop unrolling, loop skewing, and loop retiming. This paper has presented a brief discussion on these
techniques and presented examples of their application.

REFERENCES


