ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

Survey on Need of Loop Transformations for Automatic Parallelization

Nisha[1], Rafiya Shahana[2], Mustafa B[3]

Bearys Institute of Technology, Mangalore, India[1,2,3]

ABSTRACT: With the increasing proliferation of multicore processors, parallelization of applications has become a priority task. One considers parallelism while writing new applications, to exploit the available computing power. Similarly, parallelization of legacy applications for performance benefits is also important. In modern computersystems loops present a great deal of opportunities for increasing Instruction Level andThread Level Parallelism. Techniques are needed to avoid unnecessary checks to assure that only the correct number of iterations are executed. In this paper we present a survey of basic loop transformation techniquesthat can improve the performance by eliminating some unnecessary conditional instructions checking for iteration bounds. The objective of this work is to come up with a nice survey of loop transformations. We present information on the number of instructionseliminated as well as on thegeneral transformations, mostly at the source level. Depending on the target architecture, the goal of loops transformations are: improve data reuse and data locality, efficient use of memory hierarchy, reducing overheads associated with executing loops, instructions pipeline, maximize parallelism.Loop transformations can be performed at different levels by the programmer, the compileror specialized tools. Loop optimization is the process of the increasing execution speed and reducing the overheads associated of loops. Thus, loops optimization is critical in high performance computing. Our techniques are applicable to mostmodern architecture including superscalar, multithreaded, VLIW or EPIC systems.

KEYWORDS: Loop transformation technique, Loop optimization,parallelization.

I. INTRODUCTION

Most of the time, the most time consuming part of a program is on loops. Loop-level parallelism is well known techniques in parallel programming. Domain decomposition is used for solving computer vision applications, while loop-level parallelism is a common approach used by standards like Open MP. Thus, loops optimization is critical in high performance computing. Depending on the target architecture, Loop transformations have the following goals: Improve data reuse and data locality, efficient use of memory hierarchy, reducing overheads associated with executing loops instructions pipeline and to maximize parallelism.

Loop transformations can be performed at different levels by the programmer, the compiler, or specialized tools. At high level, some well known transformations commonly considered are: Loop interchange, Loop reversal, Loop skewing, Loop blocking, Loop (node) splitting, Loop fusion, Loop fission, Loop unrolling, Loop un-switching, Loop inversion, Loop vectorization and Loop parallelization.

II. MOTIVATION

Main motivation is to enabling portable programming without sacrificing performance. Loop transformation can change the order in which the iteration space is traversed. It can also expose parallelism, increase available ILP, or improve memory behavior. Dependence testing is required to check validity of transformation.

Optimizing frame-work includes improving the order of memory accesses to exploit all levels of the memory hierarchy, such as in cache line size. It also improves cache reuse by dividing the iteration space into tiles and iterating over these tiles. In order to provide greater ILP, we can unroll the loop such that each new iteration actually corresponds to

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

several iterations of the original loop. Thus unrolling is useful in a variety of processors, including simple pipelines, statically scheduled Superscalar and VLIW systems.

There exists a dependence if there two statement instances that refer to the same memory location and (at least) one of them is a write.There should not be a write between these two statement instances. Also dependency occurs if there is a flow dependence between two statements S1 and S2 in a loop, then S1 writes to a variable in an earlier iteration than S2 reads that variable. The various dependency tests available are:
Separability test, GCD test, Range test and Fourier-Motzkin test.For time critical applications, expensive tests like Omega test can be used.

III. LOOP TRANSFORMATIONS

The various loop transformations such as loop interchange, reversal, skewing and blocking are useful for two important goals: parallelism and efficient use of the memory hierarchy.A survey of the various loop transformation is given as follows:

A. SCALAR EXPANSION:

To overcome from the dependencies we use scalar expansion for removing false dependencies by introducing extra storage.Scalars introduce S2_aS1 dependence in loops. They can manifest as compiler generated temporaries.

```
do i = 1, n
c = b[i]
a[i] = a[i] + c
end do
```

This dependence can be eliminated by expanding the scalar into an array, effectively giving each iteration a private copy

```
real T[n]
do all i = 1, n
T[i] = b[i]
a[i] = a[i]
```

B. LOOP PERMUTATION:

Loop interchange simply exchanges the position of two loops in a loop nest. One of the main uses is to improve the behavior of accesses to an array. It is also known as loop interchange. Loop interchange simply exchanges the position of two loops in a loop nest.

For example, given a column-major storage order1, the following code accesses a[] with astride of n. This may interact very poorly with the cache, especially if the stride is larger than the length of a cache line or is a multiple of a power of two, causing collisions in set-associative caches.

```
do i = 1, n
do j = 1, n
b[i] = b[i] + a[i,j]
end do
```

Interchanging the loops alters the access pattern to be along consecutive memory locations of a[], greatly increasing the effectiveness of the cache.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

```
do j = 1, n
do i = 1, n
b[i] = b[i] + a[i,j]
end do
```

However, loop interchange is only legal if the dependence vector of the loop nest remains lexicographically positive after the interchange, which alters the order of dependencies to match the new loop order. For example, the following loop nest cannot be interchanged since its dependency vector is (1, −1). The interchanged loops would end up using future, uncomputed values in the array.

```
do i = 2, n
do j = 1, n-1
a[i,j] = a[i-1,j+1]
end do
```

Similarly, loop interchange can be used to control the granularity of the work in nested loops. For example, by moving a parallel loop outwards, the necessarily serial work is moved towards the inner loop, increasing the amount of work done per fork-join operation.

C. LOOP REVERSAL:

Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations. Also, certain architectures utilize looping constructs at Assembly language level that count in a single direction only (e.g. decrement-jump-if-not-zero (DJNZ)).
For example, the following code cannot be interchanged or have its inner loop parallelized because of (1, −1) dependencies.

```
do i = 1, n
do j = 1, n
a[i,j] = a[i-1,j+1] + 1
end do
```

Reversing the inner loop yields (1, 1) dependencies. The loops can now be interchanged and/or the inner loop made parallel.

```
do i = 1, n
do j = n, 1, -1
a[i,j] = a[i-1,j+1] + 1
end do
end do
```

D. LOOP SKEWING

Loop skewing takes a nested loop iterating over a multidimensional array, where each iteration of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

Let's illustrate the process with the following simple loop:

```
do i = 1, n
a[i] = a[i] + c
end do
```

Let's assume that strips of 64 array elements are desirable. The first new line computes the multiple of 64 closest to n. The outer loop iterates towards this multiple in increments of 64. A new inner loop performs the original loop on the current strip. Finally, a fix up loop may be required if n is not a multiple of 64.
Note that this inner loop could also be converted into a do all loop.

```
TN = (n/64)*64
do i = 1, TN, 64
do j = 1, 64
a[i+j-1] = a[i+j-1] + c
end do
end do
do i = TN+1, n
a[i] = a[i] + c
end do
```

E. LOOP BLOCKING:

Loop tiling reorganizes a loop to iterate over blocks of data sized to fit in the cache.

```
for(i=0;i<N;i++)
for(j=0;j<N;j++)
c[i]=c[i]+a[i,j]*b[j];
```

Loop blocking is a common loop transformation which consists in breaking the entire loop into chunks. This is mainly done on the iteration space and can be seen as a task partitioning.

```
for(i=0;i<N;i+=p)
for(j=0;j<N;j+=p)
for(ii=I;ii<min(i+p,N);ii++)
        for(jj=j;jj<min(j+q,N);jj++)

        c[ii]=c[ii]+a[ii,jj]*b[jj];
```

F. LOOP (NODE) SPLITTING:

Cyclic dependencies in a loop prevent loop fission (Section 4.8). For example, the following loop has a flow dependence S1_f0S2 and an anti-dependence S2_a1S1 bothon a[], forming a cycle.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

```
do i = 1, n
S1: a[i] = b[i] + c[i]
S2: d[i] = (a[i] + a[i+1]) / 2
end do
```

However, anti-dependencies can be eliminated by copying to a new name. Thus we can create a shifted copy of the original contents of a[] as T[] to replace the a[i+1] reference in S2.

This changes the dependencies to S3_a1S1, S1_f0S2, and S3_f0S2, breaking the cycle.

```
do i = 1, n
S3: T[i] = a[i+1]
S1: a[i] = b[i] + c[i]
S2: d[i] = (a[i] + T[i]) / 2
end do
```

Without a cycle and since S2 has no loop-carried dependencies to itself, it can be fissioned-off into its own loop (placed after the first loop to honour S1_f0S2) and made parallel. The same could be done to S1 and S3 if desired.

```
do i = 1, n
S3: T[i] = a[i+1]
S1: a[i] = b[i] + c[i]
end do
do all i = 1, n
S2: d[i] = (a[i] + T[i]) / 2
end do
```

G. LOOP FUSION:

Another technique which attempts to reduce loop overhead, when two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data.

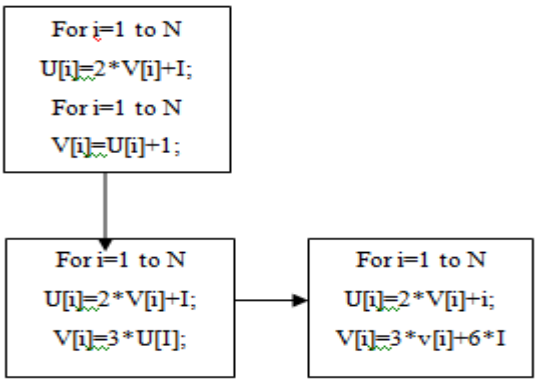

Fig.1: Loop Fusion

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

H. LOOP FISSION:

Loop fission/distribution attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.
The following code shows this by having a distance zero flow dependence from the first to second statement.

```
do i = 1, n
a[i] = a[i] + c
x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

The first statement can be moved to its own copy of the loop, where it can execute in parallel. The writes of a[] must still occur before the reads in the second statement, so the new loop must precede the second one. Consequently, if circular dependencies exist between two statements, they cannot be separated by fission.

```
do all i = 1, n
a[i] = a[i] + c
end do all
do i = 1, n
x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

This transformation would also not be possible if the dependence distance was non-zero. For example, if the first statement was a[i+**1**] = a[i] + c, then the second statement, after loop fission, would not be able to access the original value of a[i].

I. LOOP UNROLLING:

Duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which may degrade performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead (except multiple instruction fetches & increased program load time), but requires that the number of iterations be known at compile time (except in the case of JIT compilers). Care must also be taken to ensure that multiple re-calculations of indexed variables is not a greater overhead than advancing pointers within the original loop.

```
do i = 2, n-1
a[i] = a[i] + a[i-1] * a[i+1]
end do
```

The following loop shows an unrolling of factor 2. The upper loop bound must be altered to stay in its original range and a small fix-up conditional statement or loop may be needed afterwards to finish the last nmod f statements.

```
do i = 1, n-2, 2
a[i] = a[i] + a[i-1] * a[i+1]
a[i+1] = a[i+1] + a[i] * a[i+2]
end do
if (mod(n-2,2) = 1) then
a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

J. LOOP UNSWITCHING:

Unswitching moves a conditional inside a loop outside of it by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional.The following example shows a loop with a conditional execution path in its body.Having to perform a test and jump inside every iteration reduces the performance of the loop as it prevents the CPU, barring sophisticated mechanisms such as trace caches or speculative branching, from efficiently executing the body of the loop in a pipeline. The conditional also inhibits do all parallelization of the loop since any conditional statement must execute in order after the test.

```
do i = 2, n
a[i] = a[i] + c
if (x < 7) then
b[i] = a[i] * c[i]
else
b[i] = a[i-1] * b[i-1]
end if
end do
```

Similarly to Loop-Invariant Code Motion, if the loop-invariant expression is a conditional, then it can be moved to the outside of the loop, with each possible execution path replicated as independent loops in each branch. This multiplies the total code size, but reduces the running set of each possible branch, can expose parallelism in some of them, plays well with CPU pipelining, and eliminates therepeated branch test calculations. Note that a guard may also be necessary to avoid branching to a loop that would never execute over a given range.

```
if (n > 2) then
if (x < 7) then
do all i = 2, n
a[i] = a[i] + c
b[i] = a[i] * c[i]
end do
else
do i = 2, n
a[i] = a[i] + c
b[i] = a[i-1] * b[i-1]
end do
end if
end if
```

K. LOOP INVERSION:

This technique changes a standard while loop into a do-while (a.k.a. repeat/until) loop wrapped in an if conditional, reducing the number of jumps by two for cases where the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the if guard can be skipped.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

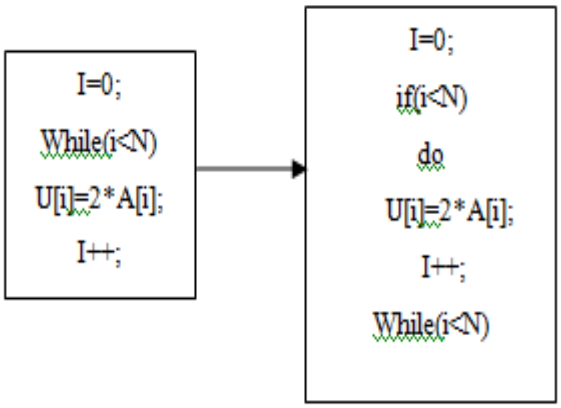

Fig. 2: Loop Inversion

L. LOOP VECTORIZATION:

Vectorization attempts to run as many of the loop iterations as possible at the same time on a multiple-processor system.Loop vectorization attempts to rewrite the loop in order to execute its body using vector instructions. Such instructions are commonly referred as SIMD (Single Instruction Multiple Data), where multiple identical operations are performed simultaneously by the hardware.

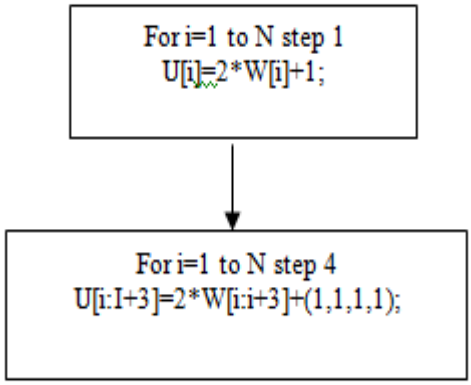

Fig. 3: Loop Vectorization

M. LOOP PARALLELIZATION:

Loop parallelization is a special case for Automatic parallelization focusing on loops, restructuring them to run efficiently on multiprocessor systems. It can be done automatically by compilers (named automatic parallelization) or manually (inserting parallel directives like OpenMP).

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

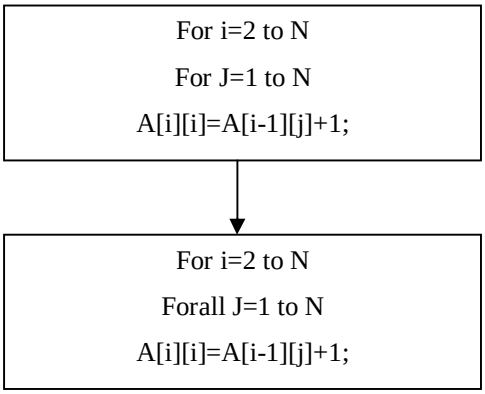

Fig. 4: Loop Parallelization

IV.CONCLUSION

The survey of the various transformations for general loop nests was carried out in this work. Different kinds of loop transformations can be applied in order to expose the parallelism before moving into explicit parallelization.In this paper we surveyed the loop transformation techniques and they have been shown useful for extracting parallelism from nested loops for a large class of machine, from vector machine and VLIW machine to multi processors architectures.

REFERENCES

1. Jean-Francois Collard, *Reasoning About Program Transformations*,, 2003 Springer-Verlag. Discusses in depth the general question of representing executions of programs rather than program text in the context of static optimization.
2. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. *Compiler transformations for high-performance computing*. Report No. UCB/CSD 93/781, Computer Science Division-EECS, University of California, Berkeley, Berkeley, California 94720, November 1993 (available atCiteSeer [1]). Introduces compiler analysis such as data dependence analysis and inter-procedural analysis, as well as a very complete list of loop transformations.
3. Steven S. Muchnick, *Advanced Compiler Design and Implementation*, 1997 Morgan-Kauffman. Section 20.4.2 discusses loop optimization.
4. R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures. Morgan and Kaufman, 2002.
5. Utpal Banerjee. Dependence Analysis (Loop Transformation for Restructuring Compilers). Springer; 1 edition (October 31, 1996).
6. F. Irigoin and R. Triolet. Supernode partitioning. POPL '88 Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages http://www.cri.ensmp.fr/classement/doc/A-179.pdf