Classifying Loops for Space-Time Mapping

Martin Griebl and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
email: {griebl,lengauer}@fmi.uni-passau.de
WWW: http://www.uni-passau.de/~lengauer

Abstract. We propose a class hierarchy for loops in a loop nest. Its purpose is to help identify the proper code generation methods for a space-time mapped nest. We illustrate the hierarchy and its use on a loop nest for computing the reflexive transitive closure of a graph.

1 Introduction

Traditional methods of space-time mapping apply to nests of for loops [15]. Given a for loop nest, an optimizing search can identify at compile time a space-time mapping which is minimal according to some stated metric (like the number of execution steps, processors, communication links, etc.). This is so because all information necessary for the search is static, i.e., available at compile time.

Lately researchers in loop parallelization have become interested in dynamic properties of loop nests. One useful generalization is to admit while loops [16]. The upper bound of a while loop is, in general, not known before the loop starts executing. However, it is not true that every space-time mapped for loop nest can be treated with traditional code generation methods and every nest containing while loops cannot—e.g., every for loop can be coded trivially as a while loop.

We propose a classification of loops and outline which code generation methods are necessary in each case. The crucial factors in the classification are when the bounds of the loop can be determined and which form they take. In the case of a (perceived) dynamic loop bound, different kinds of control dependences in the loop nest's dependence graph must be considered, which has repercussions on the potential parallelism and on the form of the target code. The data dependences and space-time mappings need to be (piecewise) affine for all classes. As in the Chomsky hierarchy of formal languages, the larger the class, the lower the number we give it. We comment on ways of parallelizing each class, and on the nature of the target code. In nests with loops of varying classes the general rule is: consider the biggest class (the one with the lowest number). We suspect that in many cases optimizations of this rule are possible.

2 Classification of Loops

We introduce five classes. As illustrating example imagine a double loop nest whose outer loop is on i between 0 and, say, some problem size parameter n. For each class, we shall give an example inner loop on j.

Class 4: Affine Loops. The bounds of these loops are affine expressions in the indices of the outer loops and in the structure parameters (i.e., the parameters which define the problem size). Nests with only affine loops can be treated well by traditional methods [15], which are realized in a number of systems [2, 14, 22, 23]. Inner loop: for $j := 0$ to $i + 5$ do.

Class 3: Convex Loops. If the loop, together with the loops enclosing it, enumerates a (discrete) convex set (the *execution space*), then there must be a loop nest which enumerates precisely the points of the set's image (the *target space*) under the space-time mapping; we call this fact *scannability* [11]. But there is no general mathematical framework (similar to Fourier-Motzkin elimination for Class 4 [1]) for identifying this loop nest.

The requirement that the check for convexity must be possible at compile time restricts the loop bounds to functions in the outer loop indices and structure parameters. Inner loop: for $j := 0$ to $\sqrt{i}$ do.

Class 2: Arbitrary for *Loops.* The next larger class of loops contains loops whose number of iterations is not known at compile time, but is known when the execution of the loop commences. The bounds are closed expressions in arbitrary variables and parameters. (Here, we assume that the upper bounds are evaluated once before the execution of the loop as in Pascal and Modula, not before every iteration as in C.) These loops are usually written as for loops, even though the bounds must be calculated at run time. Inner loop: for $j := 0$ to $A[i]$ do, for some array A.

If a loop of Class 2 is contained in a loop nest, then the image of the nest's index set is, in general, unscannable [11]. Therefore, we must scan a superset of the image and prevent the points which are not in the image from execution. For this purpose, we consider control dependences with dependence vector $\mathbf{0}$ from the computation of the loop bound to all statements of the loop body. These dependences reflect that the maximal number of iterations can and must be calculated before the operations of the body are executed.

For Classes 3 and 4 such control dependences need not be considered since the transformed loop bounds capture all required information. However, if the space-time mapped bounds of convex loops cannot be computed precisely but only estimated at compile time, then enumerating a superset of the image and taking explicitly care of the control dependences becomes necessary to exclude the points from execution which are not in the image.

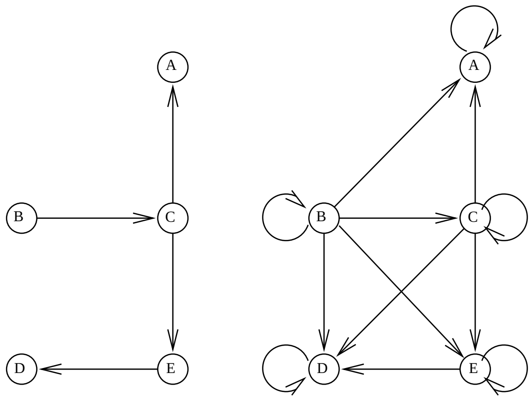
Class 1: Static while *Loops.* In the most wide-spread case of while loops, the upper bound is also fixed when the while loop starts its execution—however, it is not given explicitly as a closed expression but as a while condition which does not hold in some iteration. Consequently, there is a while dependence, i.e., a control dependence from one iteration to the next iteration of the while loop. Obviously the target loop bounds must be computed at run time. Inner loop: for $j := 0$ while $A[i, j] > 0$ do, where array A is not modified in the body.

Class 0: Dynamic while *Loops.* In the most general case of loops, the number of iterations may be changed by the iterations of the loop body. The difference to loops of Class 1 is a data dependence from a statement in the loop body to the while condition. This has no consequences for the code generation. Inner loop: for $j := 0$ while $A[i, j] > 0$ do, where array A is modified in the body.

In the literature, a popular way of parallelizing loops of Class 1 is to execute the while loop—hopefully avoiding must of the computations in the loop body—in order to evaluate the number of iterations, and then to insert this number as the upper bound of an equivalent for loop [24]. This approach can also be applied to loops of Class 0 which means dividing them into a "control" and a "rest" part. We claim that the space-time mapping approach unifies and generalizes other approaches to the parallelization of while loops [20, 24], and that it yields the same pipelined solutions—or better ones, since one does not add unnecessary data dependences and provided one uses the fastest available by-statement scheduler [8, 9].

3 Example Problem: Transitive Closure

Our illustrating example is a loop nest which computes the reflexive transitive closure of a directed acyclic graph that is given by its adjacency list. More formally, a graph is represented by a set *node* of nodes and, for every node, by the number *nrsuc* of its successors and the set *suc* of successor nodes. *rt* of n is the adjacency list of node n in the reflexive transitive closure. Figure 1 depicts a graph and the data structure representing it.



n	0	1	2	3	4
node	A	B	C	D	E
nrsuc	0	1	2	0	1
suc		C	A E		D
rt	A	B C A E D	C A E D	D D	E

Fig. 1. A graph (left), its reflexive transitive closure (middle) and the adjacency lists representing both (right)

4 Source Program

4.1 Algorithm

The following source algorithm computes the reflexive transitive closure, under the assumption that the resulting adjacency lists *rt* are initially empty:

```
for every node n do
    add n to rt of n
    while there is a node m not yet considered in rt of n do
        for every successor ms of m do
            add ms to rt of n
```

Note that this algorithm may produce adjacency lists which contain some node more than once. This is a suboptimal representation, but enforcing lists with unique elements spoils the parallelism.

4.2 Implementation

Since the polyhedron model offers no methods for dealing with sets or lists (not yet, anyway) but excels on arrays, we use arrays in our concrete representation. *node* and *nrsuc* are one-dimensional arrays, *suc* and *rt* are two-dimensional. For the computation of the reflexive transitive closure we need an auxiliary one-dimensional array *nxt* which, for every n, provides a pointer to the next free entry in the list of n's successors. Initially all undefined array elements contain the value $\perp$; *rt* and *nxt* are totally undefined. Here is the source program:

```
for n := 0 while node[n] ≠ ⊥ do
    rt[n,0] := node[n]
    nxt[n] := 1
    for d := 0 while rt[n,d] ≠ ⊥ do
        if ¬tag[n,rt[n,d]] then
            tag[n,rt[n,d]] := tt
            for k := 0 to nrsuc[rt[n,d]] − 1 do
                rt[n,nxt[n]+k] := suc[rt[n,d],k]
            end
            nxt[n] := nxt[n] + nrsuc[rt[n,d]]
        endif
    end
end
```

The range of array *node* exceeds the number of nodes by 1 in order to accommodate the undefined element which forces termination of the outer while loop.

4.3 Classification of Loops

Let us classify the loops in this program.

The outermost loop is a typical member of Class 1. If we had stored the number of nodes in some variable, we would get a loop of Class 3, and if the number of nodes were a structure parameter known at compile time, it would even be a loop of Class 4. Target code enumerating the transformed index space precisely can be generated, since it is convex whether the outermost loop is a for or a while loop. However, if we convert this loop to Class 3 or Class 4, we can omit the unit and null control dependence vectors, which must be cited in loops of Class 1. This may result in a better schedule.

The loop on d is of Class 0 since list $rt[n]$, which determines its termination, becomes longer as execution proceeds.

The innermost loop is of Class 2 since its number of iterations is fixed when the loop starts, but is not known at compile time.

5 Space-Time Mapping

To find a valid space-time mapping we compute all data dependences and insert the necessary control dependences according to the analysis of the loops in the previous section: all loops of Classes 4, 3 and 2 get a zero dependence vector from the loop-controlling statement to every statement in the body, and all loops of Classes 1 and 0 get, additionally, a unit dependence vector from the loop-controlling statement to itself, modelling the while dependences.

Then, standard techniques can be used to determine a valid space-time mapping; techniques which do not consider the loop bounds (e.g., [6]) can be applied to any loop nest without change, whereas more precise methods considering the loop bounds (e.g., [8, 9]) must be adapted so as to deal with while loops (Classes 0 and 1).

6 Target Program

One of the most intricate problems in parallelizing general loops is to generate code for the transformed program. Even for a set of loops of Class 4 which are not perfectly nested the target code may become very complex, but there are algorithms for an automatic generation [13, 21].

In general, there may not even be a target program which enumerates precisely all transformed points of the execution space: the central problem is to find computable bounds for the loops enumerating (a superset of) the execution set.

6.1 Synchronous Parallelism

In general, a synchronous target program cannot be created with standard methods since, at compile time, there is no boundary on the outer, sequential loop without knowledge of the maximal extent of some inner loop—in general, there need not be a scannable transformation in the synchronous case [16]. Target code enhancements which deal with this typical problem in both shared and distributed memory systems are given in [4, 12]. One can also employ a speculative approach as described in [3]. However, these complex schemes are not necessary for all classes of loops.

For Class 4, the computation space is known at compile time to be a polytope. Therefore, code enumerating the target space can be generated easily with standard methods like Fourier-Motzkin elimination [1] or PIP [7]. In principle, loops of Class 3 can also be treated at compile time since their execution space is convex but, at present, no code generation methods are known.

Loops of Class 2 result in non-convex execution spaces, but the compiler can easily generate a guard for the execution of every iteration. The simplest version of such a guard for a loop of Class 2 at some level r is:

$$lb_r \leq (\mathcal{T}^{-1} * \mathit{target\_coordinates})|_r \leq ub_r$$

where lb_r and ub_r are the expressions for the lower and upper bound of loop r, resp., and $|_r$ denotes the projection of the rth coordinate of a vector. The main property of this guard is that its applications in different iterations are independent of each other and can therefore be executed simultanously.

For loops of Class 1 and 0, the full scheme must be used because the guard can only be determined iteratively at run time.

6.2 Asynchronous Parallelism

For asynchronous programs one can always find a scannable transformation. This transformation yields target code without any overhead—for any class. If, for some reason, one prefers an unscannable transformation (e.g., in order to obtain space optimality), the comments of Section 6.1 apply.

However, an asynchronous program can only be written in a very abstract model, where we allow some outer loops in space to enumerate an infinite number of processors. We know that the number of processes is given by an affine function of time, i.e., the number of used processors grows affinely with time. But, since the time coordinate is not known in the outer spatial loops, one must allocate infinitely many processors initially.

In a real implementation, however, the problem of allocating an infinite number of processors at some time step is obsolete since, in general, all processors must be allocated before the parallel program starts its execution. There, the unboundedness is solved by standard partitioning or folding techniques [5, 18, 19].

This explains the non-existence of a whileall construct (a parallel while loop with an upper bound given by an arbitrary boolean expression); whileall would have to activate a set of processors in one time step (like forall) but would have to test a linearly large set of conditions, which cannot be done in constant time.

7 An Alternative Classification

Our classification is based on the question of how one can decide whether or not at some given point (in a superset of the target space) a computation must be applied. An alternative criterion, just as important for target code generation as ours, is whether target loop bounds can be determined which are functions in the outer target loop indices (if any) and structure parameters only.

This is feasible for loops of Class 4 with Fourier-Motzkin and for while loops (Classes 0 and 1), since their while dependences allow termination detection at run time [10, 12]. Thus, these classes would not change.

Classes 2 and 3, however, would in the alternative classification be divided orthogonally: in both classes there are loop nests for which target loop bounds can be found at compile time (e.g., any source loop nest under scannable transformations) and loop nests for which this is not possible—either because of a lack of mathematical methods or because of theoretically unsolvable problems at compile time (e.g., bounds depending on variables whose values vary during the program's execution). In the latter case, mathematical methods might (hopefully) provide conservative estimates of the loop bounds, (e.g., for monotonic functions); in the worst case, the loop bounds must be computed at run time.

We prefer our classification, since the classification of loops should only be based on properties of the source loop nest—but scannability is determined by the shape of the space-time matrix [11].

8 Conclusions

We hope to have demonstrated that space-time mapping methods, which in their original form from systolic design [17] can handle only a subset of Class 4 (perfect nests with uniform dependences), are becoming more and more generally applicable. Recent extensions are pushing the limit of current data dependence analysis technology: dependence tests for data structures other than static arrays are required.

Acknowledgements

The first author is grateful to Max Geigl for numerous extremely helpful discussions and careful readings of the paper. This work is part of the DFG project RecuR and received travel funds from the DAAD exchange program PROCOPE.

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1993.
2. P. Boulet, M. Dijon, E. Lequiniou, and T. Risset. Reference manual of the Bouclettes parallelizer. Technical Report 94-04, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, October 1994.
3. J.-F. Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.
4. J.-F. Collard and M. Griebl. Generation of synchronous code for automatic parallelization of while loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EURO-PAR '95 Parallel Processing*, Lecture Notes in Computer Science 966, pages 315–326. Springer-Verlag, August 1995.

5. A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION*, 12(3):293–304, December 1991.
6. A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, September 1994.
7. P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.
8. P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.
9. P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, October 1992.
10. M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EURO-PAR '95*, Lecture Notes in Computer Science 966, pages 315–326. Springer-Verlag, 1995.
11. M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.
12. M. Griebl and C. Lengauer. A communication scheme for the distributed execution of loop nests with while loops. *Int. J. Parallel Programming*, 23(5):471–495, 1995.
13. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994.
14. H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Processing*, 3:173–182, 1991.
15. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.
16. C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.
17. P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice-Hall, 1990.
18. J.-P. Sheu and T.-H. Tai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 2:430–439, 1991.
19. J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION*, 14(3):297–332, 1993.
20. P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of while loops on pipelined architectures. *J. Supercomputing*, 5:119–136, 1991.
21. S. Wetzel. Automatic code generation in the polytope model. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.
22. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *Proc. Fourth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 31–37. ACM Press, 1994.
23. M. Wolfe. The Tiny loop restructuring research tool. In H. D. Schwetman, editor, *Proc. Int. Conf. on Parallel Processing*, volume II, pages 46–53. CRC Press, 1991.
24. Y. Wu and T. G. Lewis. Parallelizing while loops. In D. A. Padua, editor, *Proc. Int. Conf. on Parallel Processing*, volume II, pages 1–8. Pennsylvania State University Press, 1990.