A Scheme for Detecting the Termination of a Parallel Loop Nest

Max Geigl, Martin Griebl and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau

{geigl,griebl,lengauer}@fmi.uni-passau.de
http://www.fmi.uni-passau.de/~lengauer

Abstract

One central problem in the execution of parallel nested loops with non-affine bounds is the precise scanning (i.e., enumeration) of the points in their iteration space and the detection of their termination. Scanning schemes have been proposed for both shared-memory and distributed-memory implementations. However, these schemes work only for perfectly nested while loops. We propose a scheme which works for imperfectly nested while loops on shared memory. This scheme has been incorporated in our loop parallelizer LooPo.

Index terms: **loop parallelization, for loops, while loops, termination detection, non-speculative execution, code generation.**

1 Introduction

Loop nests in programs have the highest potential for speedup through parallelization, and there is a well developed body of work on the automatic parallelization of for loop nests with affine bounds [11, 20]. However, many practical applications require loops with more general bounds. Examples are while loops in convergence algorithms or for and while loops in programs for irregular data structures such as general sparse data structures, where the loop bounds depend on the distribution of the input data. Recently, loop parallelization techniques have been extended to these application domains.

The parallelization of while loops has been a topic of research for a number of years [5, 26, 29, 31]. The general approach has been to pipeline the successive iterations where possible (e.g., [26, 31]). However, this yields at most constant speedup.

Standard parallelizing techniques, e.g., [3, 30], or automatic parallelizers, e.g., [1, 4, 22, 23, 28], do not contain special static techniques for while loop parallelization. Since we want to derive the parallelization at compile time, dynamic methods, e.g., [24], are not usable either.

Other approaches [29, 31] handle *specific cases* in which the parallelization of while loops is possible, especially for while loops which are actually disguised for loops. But none of these approaches offers a way of parallelizing nests with while loops in the *general case*, even though there often exists potential parallelism.

Two recent dissertations have laid more general foundations of the parallelization of loop nests with while loops: Collard [7] explored speculative execution and Griebl [13] conservative execution. In *speculative execution*, one is permitted to "overshoot the mark" of termination in the interest of increased parallelism and fix things up later. In *conservative execution*, one commits to respecting every loop limit from the outset, i.e., one never executes iterations that lie beyond the termination point of a while loop.

We pursue here the conservative approach, specifically, the question of how to detect the global termination of the parallel loop nest. We propose a scheme for the termination detection of imperfectly nested loops with dynamic bounds (for loops or while loops) in shared-memory parallelism.

Our scheme is based on the polytope model for loop parallelization [11, 20]. In this model, the execution space is embedded into the n-dimensional integer lattice: each loop spans one dimension. The coordinates of a fixed iteration in this space are given by the values of the iteration variables of each of the loops. Parallelization is achieved by a (piecewise) affine coordinate transformation, a so-called *space-time mapping*, usually defined as a matrix, which skews the execution space to place each resulting dimension either exclusively in time or exclusively in space. In this *target execution space*, dimensions in time correspond to sequential loops and dimensions in space to parallel loops.

Initially, the space-time mapping method was developed for convex execution spaces. The execution spaces of for loop nests with affine bounds are polytopes, which are convex by definition. In the case of non-affine bounds or while loops, the execution space can become highly irregular—in the case of while loops, its precise shape evolves even only at run time [16, 21]. Still, we would like to parallelize nests with general for loops and while loops also at compile time. The formal work on which this paper is based can be found in [12].

We are incorporating the termination detection technique described in our loop parallelizer LooPo [18], which is based on the polytope model. In this paper, we illustrate the benefits and limits of the scheme at the level of the model.

2 Execution Spaces

An important step in the parallelization of a loop nest is the determination of the target execution space and the identification of a nest of target loops (space and time loops) which enumerates it. In this paper, we investigate the correlation between two different types of execution spaces and their images. We restrict ourselves to piecewise affine transformations, since present schedulers and allocators yield only piecewise affine space-time mappings.

Static and Dynamic Execution Spaces The shape and extent of a static execution space is known at compile time or, at least, it can be approximated by a superset whose shape and extent is known at compile time. Static execution spaces are characterized by loop bounds to which no data dependences are "pointing". We call loops with static execution spaces *static*.

while loops can only belong to a static execution space if the values occurring in their conditions are not changed by any statement. This renders only while loops that have either no iterations (the while condition is initially false) or infinitely many iterations (the while condition is always true). Which case applies can be determined at compile time. So, usually, execution spaces determined by one or more while loops are not static but *dynamic*.

On the other hand, also for loops may cause dynamic execution spaces, e.g., if the upper bound contains an array access which is indexed by surrounding loop indices. Since the values in the array are determined at run time, the execution space cannot be static. For a more detailed classification see [17].

3 The Scanning Problem

The main complication introduced into the polytope model by dynamic loops is that execution spaces become irregular. As a consequence, their precise parallel enumeration may become a problem. A target execution space for which no precisely enumerating parallel loop nest exists is called *unscannable* [16]. The case for the necessity of unscannable target spaces is made in [21].

In general, we can only enumerate a superset of an unscannable target execution space. In doing so, we have to prevent iterations that are not in the target execution space from executing; we call this *execution determination*. Also, we have to take care of the termination of the target loops; we call this *termination detection*. As it turns out, termination detection is an extension of execution determination.

4 Execution Determination

Whether a point in the iteration space must be executed or not is determined by a predicate, named *executed*, which is the conjunction of all conditions which determine whether an operation (an instance of a statement at a given iteration) is executed in the source domain:

1. The bounds of a for loop restrict the range of the index.
2. A valid termination condition of a while loop disables further iterations.
3. The guard of a surrounding if statement prohibits the execution of a specific instance.

Note that, in contrast to Conditions 1 and 3, it is not sufficient for Condition 2 just to add the while condition into the statement's *executed* predicate; the reason is that no iteration of the while loop is executed after the termination condition has been found true—even if, in a later iteration, the termination condition would be false again. This means that an iteration can have one of two states: *running* indicates that it represent a point inside the execution space, *terminated* that the point is outside the execution space. This state is propagated between the iterations of one while loop and is changed once from running to terminated as soon as the while condition evaluates to false. This leads to a formal recursive definition of *executed* for those dimensions which are spanned by while loops [12]. Note that the sequentiality of the while loop iterations is not introduced by our recursive definition of *executed*, but is inherent in the semantics of the while loop.

5 Termination Detection

5.1 Scanning Execution Spaces of Dynamic for Loops

Essentially, two things change in the transition from static to dynamic for loops:

1. There is a new statement which recomputes the new upper bound of the time loop(s) (see below).
2. We guard the original body with predicate *executed*.

Let us expand on item 1. A dynamic loop in the source nest may influence the upper bound of the time loop(s) in the target nest. Since we have to enumerate time until the iteration with the maximal value which the schedule prescribes, we need to know the maximal number max_ub of iterations, over all instances of a dynamic loop. Since the upper bounds UB_r of these instances are computed at run time, we can determine the maximum only iteratively at run time by computing $max\_ub := \max(max\_ub, UB_r)$. This maximum determines the upper bound of the time loop, and as it changes at run time, time must be enumerated by a while loop.

The use of max_ub instead of the real values of UB_r is responsible for the inexact iteration space but has the benefit of forcing the upper bound to be monotonic: the upper bound of any instance of the loop is at least as high as the maximum of the upper bounds of all previous instances. This monotonicity is one of the reasons why affine bounds are so suitable for parallelization. It enables us to compute the minimum and maximum easily: we find the extrema of a monotonic function at either one or the other end of the source domain.

We pay a two-fold price for a convenient shape of the new index space. Firstly, there is the additional effort that must be taken to prevent the execution of operations which do not belong to the execution space. By guarding the body with the original execution condition, we ensure that its execution space remains unaltered, although the number of iterations of the loop has changed. Secondly, the computation of the maximum, as just described, introduces a dependence which may not exist in the original source program.

Note that instances of a dynamic for loop at level q can still be executed in parallel, if this is also possible in the original program. The additional dependence is affecting the loop at level $q-1$.

Note that, within the polytope model, we must assume that the maximum of n elements is computed sequentially in n steps. However, the dependence from an instance of the update of max_ub to the next is not strictly necessary. We could calculate all values of UB_r at the same time. Since we need not impose any order on computing the maximum, the time for computing the maximum need only be logarithmic. However, since this solution falls outside the polytope model, in which no sublinear execution times can be achieved, our implementation performs the sequential computation which runs in linear time.

If the target program is intended to run on an asynchronous machine, we have to ensure that all updates of max_ub which are scheduled at the same time are performed before the new upper bound is used.

5.2 Scanning Execution Spaces of while Loops

While the number of iterations of an instance of a for loop is known before the first operation is executed, the number of iterations of a while loop is determined in the course of its execution. This is the main difference between non-affine for loops and while loops. The consequences are examined in this subsection.

In contrast to for loops, the upper bound of a while loop is not given as a value but as a condition. When we are dealing with a while loop, we have to compute an upper bound (a number) that is increasing monotonically over all instances of this loop.

Similarly to non-affine for loops, we need a new variable which expresses the upper bound and which is shared by all instances of the while loop. For consistency reasons, let us call it also max_ub.

Initially, max_ub is set to the lower bound, i.e., every instance of the while loop evaluates predicate *executed* at least once. As long as the condition $LB \leq i \leq max\_ub$ holds, the respective instance of the loop is iterating. Because of $i \leq max\_ub$, every instance is iterating at least as often as the previous instance. If the current instance turns out to execute more iterations than the previous one, max_ub is incremented at every further iteration. Thus, at any given point in time, max_ub is the maximum number of iterations which an instance of the loop has executed up to that point.

Since every instance of the loop runs at least as long as the previous one, some instances might run longer than they do in the source program. This is the reason for the difference between the iteration and the execution space and, therefore, we have to apply execution determination, governed by the truth value of *executed*.

5.3 Bounds and Termination

We have established the correctness of this termination detection scheme by proving the following five lemmata:

Enumerating enough Time. The termination scheme will not prevent any operation in the source program from being executed by the target program. (Lemma 40 in [12].)

Termination of Time Loops. The time loops terminate after a finite number of additional steps if the source loops terminate. (Lemma 41 in [12].)

Lower Bounds for Space Loops. The lower bounds of the space loops can be computed as in the polytope model. (Lemma 42 in [12].)

Upper Bounds for Space Loops. The upper bounds of the space loops can be computed by allocators based on the polytope model. (Lemma 43 in [12].)

Enumerating enough Space. The space coordinates of every target iteration fall within the bounds of the parallel target loops. (Lemma 44 in [12].)

5.4 Options in Termination Detection

The basis of the termination detection described in this paper is the calculation of the maximum *max_ub* of all upper bounds. Two other methods have been proposed: the *signaling scheme* for distributed-memory computers [15] and the *counter scheme* for shared-memory computers [14]. Their goal is to avoid as much communication and overhead as possible (within the scope of the respective machine model).

We concentrate on synchronous shared-memory parallelism, since this model is closest to the usual sequential way of thinking and since we wanted to learn about parallel execution at an abstract level.

In this section, we compare the counter scheme to the termination detection scheme presented here, which we call the *maximum scheme*.

In the counter scheme, a global shared counter is incremented before a new "tooth" (an instance of a loop in any arbitrary dimension) starts executing and is decremented when a tooth terminates. The program terminates when every tooth has terminated, which is the case when the counter is decremented to 0 for the first time. There are also optimizations possible to prevent the counter from becoming a bottleneck [14].

The main advantage of the counter scheme is in the reduction of synchronization by one dimension. The management of the counter does not affect the innermost dimension, which causes a considerable decrease in run-time overhead; the more iterations the instances of the innermost loop execute, the bigger the gain.

To generalize the counter scheme for the execution of imperfect, general loop nests, essentially, for every tooth of a for loop (that is not the innermost loop), the counter must be incremented by the number of iterations the loop will execute. This is the number of teeth of the next inner loop. When an instance of a for loop terminates, the counter must be decremented just as for while loops.

However, the target execution space is not the same for all statements with the same source execution space (we focus on this aspect in Section 6). So it is not possible to state precisely when an instance of a loop terminates. Consider two statements in the same loop r of a loop nest. An instance of the surrounding loop terminates only after both statements have been executed. It must be ensured that the counter is decremented at the point of termination. We suspect that, for imperfect loop nests in which every statement has its own transformation, it is hard to identify this point at compile time.

Another drawback is the reduced information the counter scheme stores. Only the number of currently executing loop instances is known, which leads to a quite inexact scan of the space dimensions. Since partitioning (see also Section 7) divides a space dimension into a space dimension for the real processors and a time dimension, this rough approximation also costs time.

We illustrate the differences between the spaces scanned by the counter and the maximum scheme by means of the following figures. Consider the spaces shown in Figure 1.

The left part shows the source index space of statement s in the program

```
for i := 0 to 3 do
    for j := 0 to a[i,0] do
        s;
```

The right part shows the target index space determined by the transformation $T_s = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

Note that the j-dimension of the source index space on the left of Figure 1 has no upper bound

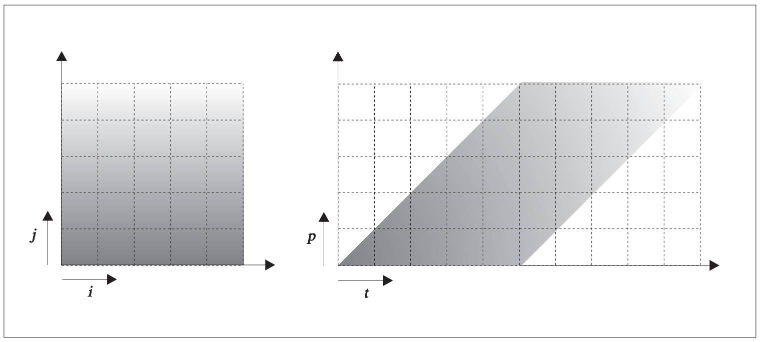

Figure 1: The index spaces

in the polyhedron model. Accordingly, both dimensions of the target index space have no upper bounds, either (expressed by the shade of grey turning to white).

The following three figures show different sample target execution spaces with their respective scanned target spaces.

In the case of Figure 2, the counter scheme enumerates one more (virtual) processor than the maximum scheme, but the latter enumerates considerably more time.

The region containing the horizontal dashed lines contains the points which would be scanned additionally by the maximum scheme if the target index space (known at compile time) were not considered. Since our maximum scheme considers the intersection of the target space and the space that would be scanned if only the max_ub values were used (without concerning the index space), the dashed region is not part of the scanned target space.

This is an example which makes the difference between both schemes very clear and for which the counter scheme delivers better results. The reason is that the first tooth is the longest and the upper bound of j is decreasing monotonically.

Figure 3 shows a case that is more common.

Because the longest tooth is close to the end of the outer loop, the maximum scheme is about as good as the counter scheme; it requires only one additional time step. The space dimension is scanned more precisely by the maximum scheme. The additional information given by the max_ub values leads to the jagged—and, thus, more precise—left border of the target space scanned by the maximum scheme. Again, the dashed region is not scanned.

Figure 4 depicts the kind of target space which is supported best by the maximum scheme. Both methods yield minimal time, but only the maximum scheme scans the space dimension precisely. The maximum scheme yields the same run time as the counter scheme if the upper bound of the j-dimension is increasing monotonically. In this case, the maximum scheme always scans a smaller subset of the target index space than the counter scheme.

Let us summarize our results:

- The performance of both schemes depends heavily on the actual target execution space. If there is statistical knowledge about the values occurring at run time, then it is possible to choose the most suitable scheme. Monotonically increasing upper bounds are supported better by the maximum scheme whereas, with virtual processors, the time dimensions are scanned more precisely by the counter scheme.

- An increase of the number of processors has a greater effect with the counter scheme than with the maximum scheme. But, of course, both schemes increase performance.

- No method can be said to be best. A combination of both methods is better than any of the two alone. It scans the intersection of both scanned target spaces, which is smaller than any individual scanned target space.

Merging both schemes leads to complex target code. We chose to use the maximum scheme for termination detection, because it constitutes a more consequent generalization of the existing method for affine for loops than the counter scheme. Additionally, we avoid the necessity of finding the statement of all statements with the same execution space whose schedule yields the largest values (see Section 6).

Further studies will have to deliver more concrete and measurable results.

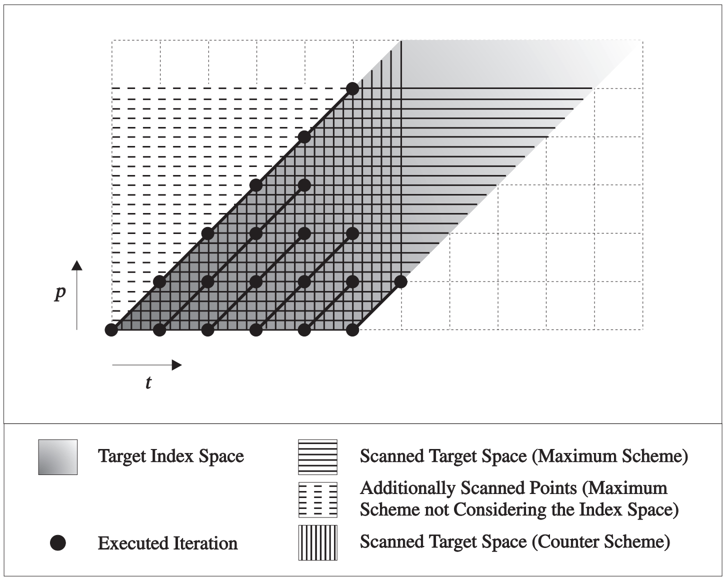

Figure 2: The scanned target spaces: counter vs. maximum (values of $a[i]$: $5, 3, 2, 2, 1, 1$)

6 Mapping every Statement Individually

Our deliberations so far—except at points where we have referred to the present section—have been concerned with perfect loop nests with a body all of whose statements are subject to one and the same coordinate transformation. This is how the scheduling and allocation method by Lamport [19] can treat bodies of perfectly nested loops. Thus, all operations of different statements at the same iteration are mapped to the same instant of time and the same processor, and the body of the iteration is executed sequentially. Only different iterations are executed in parallel.

Modern scheduling and allocating techniques [8–10] calculate an affine schedule and allocation for each statement individually. This implies that operations of different statements and different iterations can be executed in parallel and that loops do not have to be perfectly nested.

The disadvantage is that, in general, every statement has a different target execution space—even if the source execution spaces are equal. These target execution spaces must be scanned by one common target loop nest, i.e., they must be merged.

Wetzel [27] offers a method for merging target execution spaces of statements in the body of loop nests with only affine loops.

Since we base our code generation technique on the output of Wetzel's code generator, we must explain briefly its basics here without considering special cases.

Roughly, the transformation of the source execution spaces yields as many d-dimensional target polytopes as there are statements in the source program. One dimension of the union of all these target polytopes is enumerated by one single target loop, which is sequential for a time dimension and a parallel for a space dimension. The lower bound of the loop for a given target dimension is determined by the minimum of the minima of the respective dimension. Analogously, the upper bounds are determined by the maximum of all maxima.

Example 1 (Merging the Target Spaces).

In Figure 5 (left), we have different target iteration domains for two statements. They result from different transformations applied to the (identical) source domains of the two statements. In Figure 5 (right), we see that the convex scanned set contains additional points.

To prevent the execution of points that are not in the execution space of a statement, every statement in the target program is guarded by its transformed predicate *executed*.

Note the two different levels at which an inexact scanning of the target execution spaces can occur:

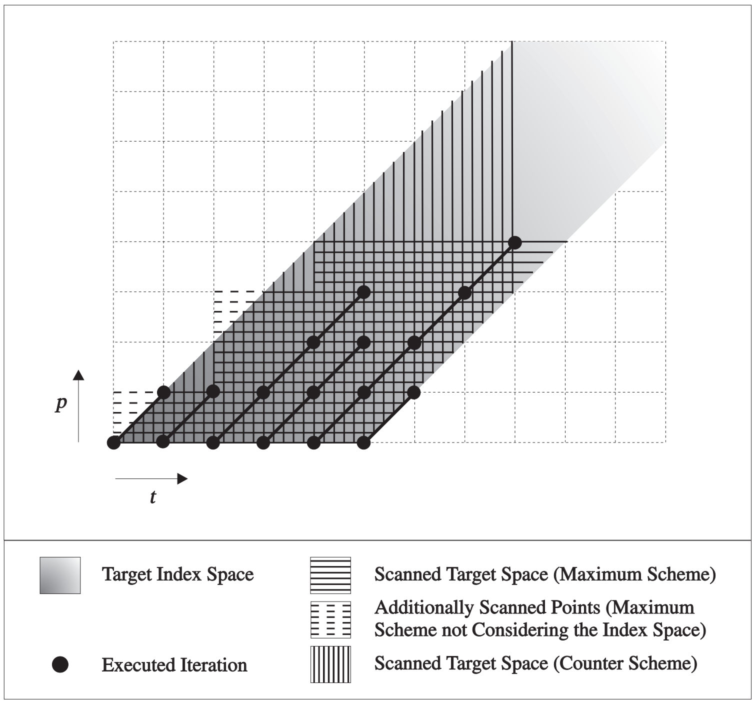

Figure 3: The scanned target spaces: counter vs. maximum (values of $a[i]$: $1, 1, 3, 2, 4, 1$)

1. The target execution space *of a statement* may not be precisely scannable.
2. The union of the target execution spaces of different statements is not precisely scannable—even if the individual target execution spaces are precisely scannable.

Since predicate *executed* of a statement describes its execution space exactly, it solves both problems in one go.

7 Implications of Finiteness

The example of Figure 1 illustrates that it is not always possible to generate an asynchronous parallel target program, because the upper bounds of the space loops may not be known at the beginning of the execution. On the other hand, there are machines that work asynchronously and, consequently, need asynchronous programs. The logical implication would be that, in general, only parallel synchronous or sequential asynchronous target programs are derivable.

However, this scenario does not take into account that we have only a finite number of processors—say, n—and must apply a partitioning algorithm for mapping the virtual processor coordinates onto real processors. The only requirement is that the partitioning algorithm must be of the type "locally parallel, globally sequential" (LPGS); in the absence of upper bounds at compile time, LSGP would not be able to derive a limited number of processors [2, 6, 25, 32].

Essentially, after partitioning, one virtual processor loop has become a nest of two loops. One enumerates all n processors in some dimension and the other counts the number of steps each of the n processors has to make in order to scan the entire (virtual) dimension.

Example 2 (Partitioning).

Let the target program

for $p := 0$ while *cond* do *body*

enumerate a space dimension that could be executed in parallel if its upper bound were known. Further, let n be the number of (real) processors in the respective dimension. Partitioning would change the loop as follows:

```
for pp := 0 to n − 1 do
    for pt := pp while cond step n do
        p = pp + pt
        body
```

The while loop still does not offer the opportunity for more parallelism—but the new for loop does.

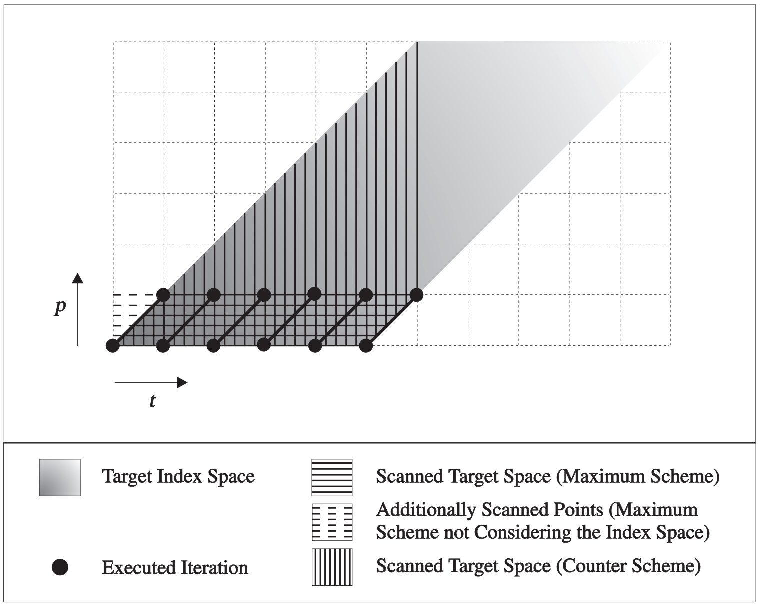

Figure 4: The scanned target spaces: counter vs. maximum

Since all statements in the body may be executed in parallel, we can change the for loop into a forall loop and exploit the maximum parallelism our machine offers. The second loop would be sequential in any case (also if it were a for loop), because we do not have more than n processors in the respective dimension. Thus, no more parallelism is possible.

With a finite number of processors, we can have partially parallel while loops with a speed-up of n.

The theoretically interesting consequence of this section is that, in practice, we can also derive asynchronous target programs that exploit the maximum parallelism offered by a given multi-processor machine. It does not exploit the maximum theoretical parallelism offered by the source program.

8 Conclusions

Loop nests with while loops occur frequently in applications, and they have just as high a potential for parallelization as for loop nests. Recently, methods for their automatic parallelization have become available. One problem that arises in the code generation phase is that of distributed termination detection.

The maximum scheme for termination detection presented here is a modification of the counter scheme presented in [14]. The strong and weak points of both schemes could be illustrated quite easily. The modifications enabled the easy integration of imperfectly nested loops and statement-dependent transformations. This was the main reason for incorporating these methods in our parallelization framework LooPo [18]. As of now, we have no comparative data from performance measurements, since we are still working on the parallel code generation component of LooPo.

One interesting implementation detail is that all central modules (dependence analysis, scheduler, allocator, partitioner), which were originally intended for affine for loops only, did not require any modification. This was made possible by using (unrestricted) structure parameters as upper bounds for dynamic loops, thus "sneaking" them into the core modules as if they were affine for loops. Only the module generating the target loops was changed, in which these unrestricted structure parameters are replaced by the real expressions which come from the maximum scheme.

Acknowledgements

This work was supported financially by the DFG through project *RecuR*, and by the DAAD through a PROCOPE exchange project.

References

[1] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot, and R. Keryell. PIPS: A framework for building interprocedural compilers,

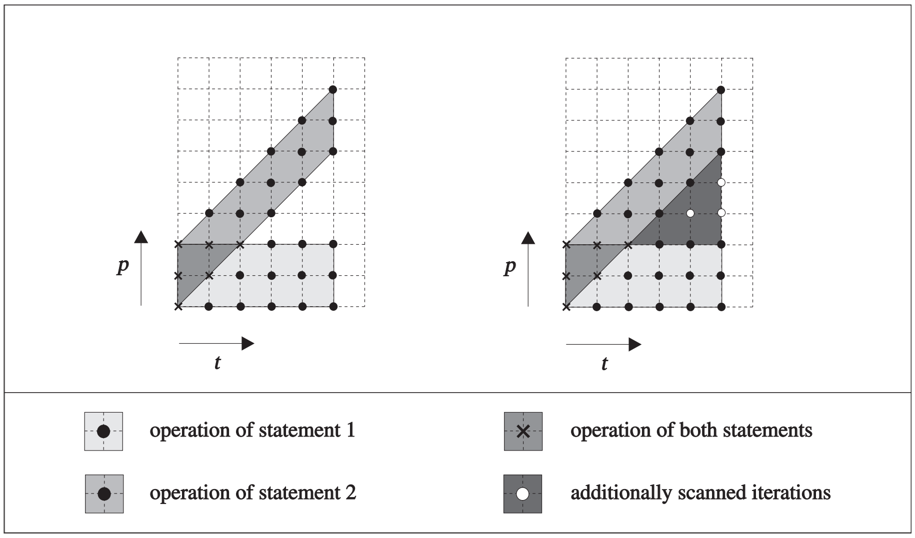

Figure 5: Target execution spaces (left) and scanned execution space (right)

parallelizers and optimizers. Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, Apr. 1996. http://www.cri.ensmp.fr/~pips/.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, 1991.

[3] U. Banerjee. *Loop Parallelization.* Series on Loop Transformations for Restructuring Compilers. Kluwer, 1994.

[4] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 141–154, Ithaca, New York, Aug. 8–10, 1994. Springer-Verlag.

[5] J. P. Bonomo and W. R. Dykson. Pipelined iterative methods for shared-memory machines. *Parallel Computing*, 11:187–199, 1989.

[6] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION*, 17:33–51, 1994.

[7] J.-F. Collard. *Parallélisation automatique des programmes à contrôle dynamique.* PhD thesis, Université Paris VI, 1995.

[8] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Technical Report 96-06, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Apr. 1996.

[9] M. Dion and Y. Robert. Mapping affine loop nests: New results. In B. Hertzberger and G. Serazzi, editors, *High-Performance Computing & Networking (HPCN'95)*, Lecture Notes in Computer Science 919, pages 184–189. Springer-Verlag, 1995.

[10] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991.

[11] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer-Verlag, 1996.

[12] M. Geigl. Parallelization of loop nests with general bounds in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, Mar. 1997. http://www.fmi.uni-passau.de/loopo/doc/geigl-d.ps.gz.

[13] M. Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1996. Technical Report MIP-9701.

[14] 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: Parallel Processing*, Lecture Notes in Computer Science 966, pages 315–326. Springer-Verlag, 1995.

[15] M. Griebl and C. Lengauer. On scanning space-time mapped **while** loops. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 677–688. Springer-Verlag, 1994.

[16] M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, Sept. 1994.

[17] M. Griebl and C. Lengauer. Classifying loops for space-time mapping. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96, Vol. I*, Lecture Notes in Computer Science 1123, pages 467–474. Springer-Verlag, 1996.

[18] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In M. Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96)*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[19] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, Feb. 1974.

[20] 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.

[21] C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In 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.

[22] C. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'90)*, Research Monographs in Parallel and Distributed Computing, pages 423–453. Pitman, 1990.

[23] PRiSM SCPDP Team. Systematic construction of parallel and distributed programs. http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html.

[24] L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. In *Proc. 9th Int. Parallel Programming Symposium (IPPS'95)*, pages 347–355. IEEE Computer Society Press, 1995.

[25] J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION*, 14(3):297–332, 1993.

[26] P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of while loops on pipelined architectures. *J. Supercomputing*, 5:119–136, 1991.

[27] S. Wetzel. Automatic code generation in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, Nov. 1995. http://www.fmi.uni-passau.de/loopo/doc/wetzel-d.ps.gz.

[28] 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. http://suif.stanford.edu/suif/.

[29] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[30] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[31] 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.

[32] J. Xue. Communication-minimal tiling of uniform dependence loops. *J. Parallel and Distributed Computing*, 42(1):42–59, Apr. 1997.