Some Experiments on Tiling Loop Programs for Shared-Memory Multicore Architectures

Armin Größlinger

Universität Passau, Department of Informatics and Mathematics
Innstraße 33, 94032 Passau, Germany
armin.groesslinger@uni-passau.de

Abstract. The model-based transformation of loop programs is a way of detecting fine-grained parallelism in sequential programs. One of the challenges is to agglomerate the parallelism to a coarser grain, in order to map the operations of the program to the available cores in a multicore architecture. We consider shared-memory multicores as target architecture for space-time mapped loop programs and make some observations concerning code generation, load balancing and cache effects.

Keywords. Multicore, automatic parallelization, loop transformations, polyhedron model

1 Introduction

The polyhedron model has been used to model loop programs and describe transformations on them for a long time [1,2,3] in which the expressiveness of the model has been increased successively (see, e.g. [4,5]). The usual parallelisation process consists of three phases. First, in an analysis phase, the loop bounds and data dependences are extracted from a given input program and represented as polyhedra and affine functions. Second, the model representation is transformed using mostly linear algebra to yield a (model of a) program with optimised characteristics, for example by enhancing the cache behaviour of the program or infusing parallelism. The third and last phase is to generate a program from the transformed model which executes the program as desired on a real machine. As it turns out, this code generation phase has remained a hard problem over the years. It is now largely solved for the sequential case (i.e., the target architecture is a sequential machine) [6,7], but remains a major challenge in the parallel case.

In this paper, we will consider two examples of loop programs which we will transform into parallel codes. In this case, the transformation phase is the application of a so-called space-time mapping to the model of the original program to infuse parallelism. As is well-known, we find way to much parallelism in doing so, i.e., the fine-grained parallelism has to be coarsened such that the number of parallel units of work matches the number of available cores. We consider two archetypical examples in Section 2, discuss code generation, load balancing and the cache behaviour of the generated code in Section 3, and present the results of some experiments in Section 4.

Dagstuhl Seminar Proceedings 07361
Programming Models for Ubiquitous Parallelism
http://drops.dagstuhl.de/opus/volltexte/2008/1374

2 Tiling

Tiling is a well-established technique to enhance data locality or coarsen the grain of parallelism in loop programs [8,9,10]. The iteration space of the program is covered with (usually congruent) tiles and the enumeration of the iteration points is changed so as to enumerate the tiles in the outer dimensions (i.e. loops) and the points within each such tile in the inner dimensions.

The shape and size of the tiles is usually chosen dependent on the dependence vectors [11,12,13] to minimise communication startups and the volume of the data communicated, especially in the context of distributed-memory architectures. For shared-memory systems, the number of startups and the volume are less of a concern, as long as the transfer time of the data between cores stays small compared to the computation time for each tile. It is, as we indicate here, not less important to address load balancing and select tile shapes and sizes which distribute an equal amount of work across the cores.

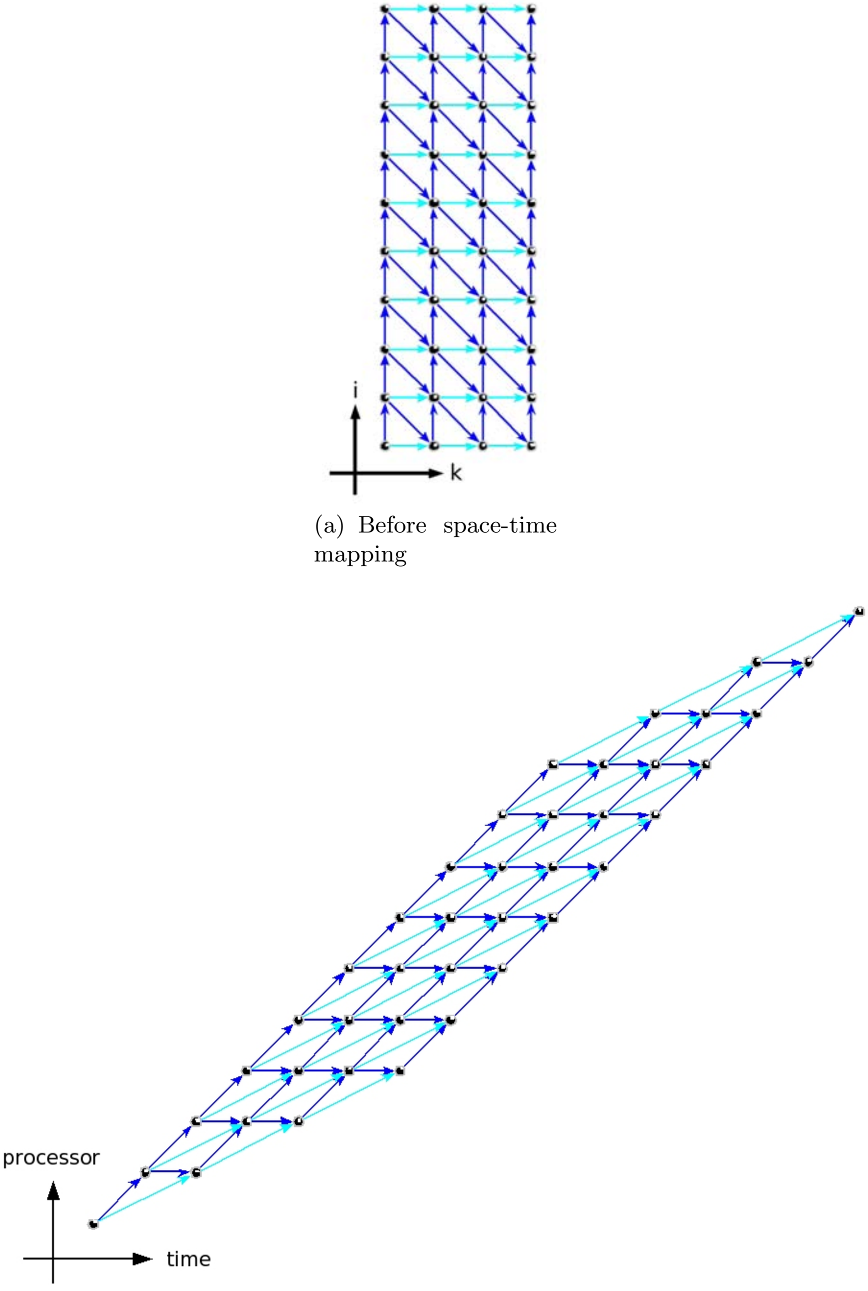
2.1 Parallel bounds

Tiling is simplest when opposite bounds of the index space are parallel. As an example, let us consider one-dimensional successive over-relaxation (SOR):

```
for k=1 to m
  for i=2 to n-1
    A[i] = (A[i-1] + A[i+1]) * 0.5
```

Here, `n` is the size of the array being processed, and `m` is the number of sweeps accross the array. A valid space-time mapping is to assign each loop iteration the execution time $t(k, i) = 2 \cdot k + i - 4$ and the virtual processor $p(k, i) = k + i - 3$. We do not want to go into the details here; we just note that, to be a correct mapping and one that enables tiling, the dependences after space-time mapping must point forward in time and forward (or to the same virtual processor) in space [14]. Figure 1 shows the index space before and after space-time mapping. The usual choice of tile shape in the space-time mapping community is rectangular, since intuitively, time and space are orthogonal after space-time mapping. But, since the bounds of the index space are parallel, an obvious choice of tile shape is a parallelogram whose bounds are parallel to the index space bounds. To achieve load balancing among the processors, the size of the tiles depends on the number NC of available cores.

Figure 2 shows a part of the space-time mapped index space with rectangular and parallelogram tiles. Tiling is described in the model by doubling the number of dimensions. Each index point is described by a coordinate of the space of tiles, and a "local" coordinate within the tile. Care has to be taken as to which tiles can be executed in parallel. Before tiling, all the index points (k, i) with the same $t(k, i)$ value are parallel. After, tiles with the same time coordinate in tile space *cannot* be executed in parallel, since the tiles span, in general, more than one time coordinate of the index space and, hence, dependences between these



(a) Before space-time mapping

(b) After space-time mapping

Fig. 1. 1d SOR: Index space with dependences

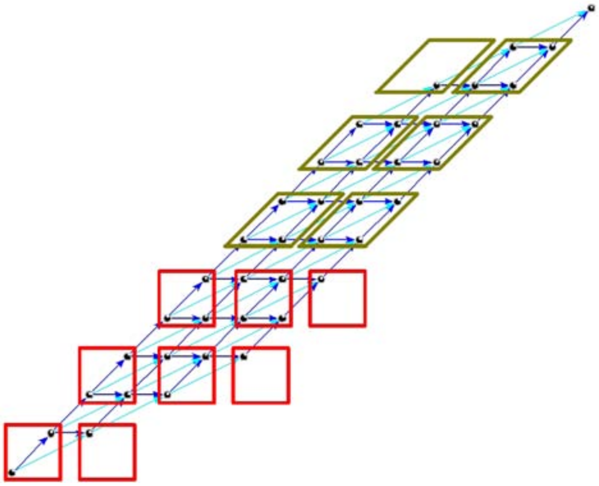


Fig. 2. 1d SOR: Space-time mapped index space with tiles

tiles can exist. A so-called *skewing* [10] has to be performed and the tiles with time coordinate t_T and processor coordinate p_T with the same $t_T + p_T$ value can be executed in parallel (cf. Figure 3).

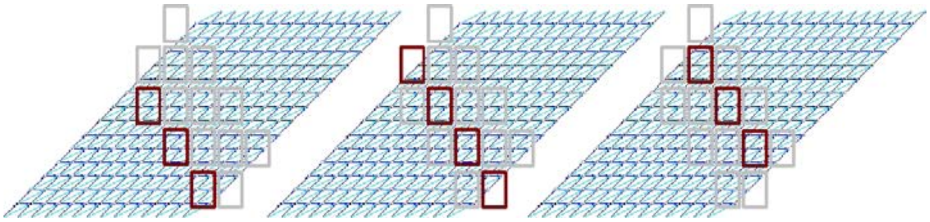


Fig. 3. Skewing: Tiles which can be executed in parallel (highlighted) in successive time steps (from left to right)

Due to this skewing, the number of tiles which can be executed in parallel is not determined by the height of the tiles, but by their width (in case of the parallelogram tiles) or their width and height together (in case of rectangular tiles), respectively. The relations are given by

$$\text{width} = \frac{m}{f \cdot \text{NC}}, \quad \text{height} \geq 1 \qquad \text{(parallelograms)}$$

$$\text{width} = \frac{m}{f \cdot \text{NC}} - \text{height}, 1 \leq \text{height} < \frac{m}{f \cdot \text{NC}} \qquad \text{(rectangles)}$$

where $f \geq 1$ is an arbitrary integral factor denoting the number of tiles to be assigned to one core. It may seem that a rectangular tiling cannot achieve load balancing because there are incomplete tiles along the borders of the index space. But, by closer inspection (cf. Figure 3), it becomes clear that, with the given choice of the tile width and height, there are either NC complete tiles in parallel, or NC + 1 tiles of which the first and the last (the tiles crossing the borders of the index space) together form one complete tile. Therefore, if we distribute these tiles cyclically across the cores, each core will execute one complete tile (in terms of the number of operations performed).

2.2 Triangular index space

A slightly more complicated case than an index space with parallel bounds is an index space of triangular shape. An example is the backward substitution phase of Gaussian elimination:

```
for i=1 to n-1
  for k=0 to i-1
    B[i] = B[i] - A[i][k]*B[k]
```

The index space after space-time mapping with $t(i, k) = k$ and $p(i, k) = i - 1$ is shown in Figure 4. To achieve load balancing, one would have to choose "growing" tiles, but this is illegal because it would generate tiles with cyclic dependences. Due to the dependences, which are of the form $(t, p) \mapsto (t + 1, p + a)$ for $a \geq 0$, only a rectangular tiling[1] is legal and easy to describe.

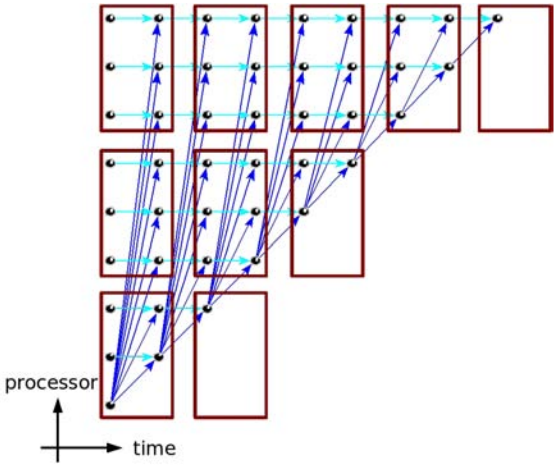


Fig. 4. Backward substitution: Index space after space-time mapping with rectangular tiles

[1] Or another parallelepiped tiling with an angle of at least $\frac{\pi}{2}$ between the spanning vectors, but this does not improve the situation.

A rectangular tiling poses two problems. Only on one border, the covering tiles are incomplete, such that they cannot find another tile with which to form a complete tile, and the number of parallel tiles is not constant in time. The space-time mapping suggests that the program can start with a high degree of parallelism which then shrinks towards the end of the program. But, due to the skewing required by tiling, the parallelism has a growing phase first, followed by the shrinking phase. To address the problem of incomplete tiles, we would have to use a tiling with two different tile shapes, namely two triangles, one with the same orientation as the index space and the other such that both triangles together form a rectangle. With suitably chosen tile sizes, the first triangles will line up with the diagonal border of the index space and no point inside the tiles will be wasted. This approach leads to complex target code, and we have not pursued it so far.

To handle the varying number of parallel tiles, we suggest to use a cyclic mapping of tiles to cores. Figures 5 shows the aspired distribution of tiles for a part of the execution. Executing tiles on different diagonals in parallel, as shown in the figure, is legal for diagonals with at least NC tiles, since there are no data dependences between the tiles executed in parallel. In the beginning and the end of the execution, the number of parallel tiles is less than NC and we cannot utilise all cores, but the biggest part of the index space (provided that NC is small compared to the number of tiles) can be executed using all cores.

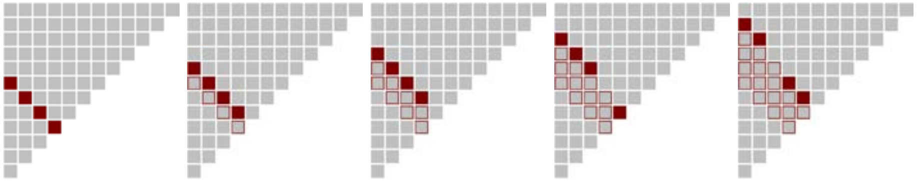


Fig. 5. Cyclic parallel execution: Highlighted tiles are executed in parallel, bordered tiles are completed in preceding time steps.

A simple model for estimating the performance of this cyclic tiling and comparing it with the standard execution, is to count the number of index points within each tile (as an estimate for its execution time) and computing the execution time of the whole program from this, taking delays imposed by synchronisation into account. This simple model predicts the following speed-ups:

NC	2	4	8
speed-up standard	1.83	3.08	4.62
speed-up cyclic	1.83	3.41	5.22

Due to the incomplete tiles along one of the borders and the smaller degree of parallelism in the beginning and the end, the expected speed-ups are sub-linear and, for two cores, almost identical.

3 Code generation

The program transformation and code generation is fully automatic. We use the tools of LooPo [15] for dependence analysis, computing and applying space-time mappings, and applying tiling to the model. CLooG [7] generates sequential loops from the polyhedral description, and a postprocessing phase (developed by LooPo team members) annotes the generated loops with OpenMP pragmas and inserts the transformed loop bodies into the code. The only manual part in the whole process is to select spanning vectors of the tiles, by which the tile shape and size are determined.

For the one-dimensional SOR example, the generated code looks like this:

```
#define S1(i, k) { A[i]=(A[1+i]+A[i-1])/2; }

int upperBound1 = floord(5*M+3*N-14,1500);
for (glT1=-1; glT1 <= upperBound1; glT1++) {
  int lowerBound2 = max(ceild(375*glT1-1499,1875),
    max(ceild(750*glT1-M-1498,2250), max(ceild(750*glT1-2*M-N+5,750),0)));
  int upperBound2 = min(floord(1500*glT1+2*N+1493,7500),
          min(floord(M+N4,1500),min(floord(1500*glT1+1499,4500),
                                    floord(1500*glT1+1499,1500))));
  #pragma omp parallel for schedule(static,1) private(vT1,vP1)
  for (rp1=lowerBound2; rp1 <= upperBound2; rp1++) {
    int upperBound3 = min(floord(1500*glT11500*rp1+1499,2),
             min(3000*rp1+2998, min(1500*rp1+M+1498,2*M+N-5)));
    for (vT1=max(750*glT1-750*rp1,max(3000*rp1-N+3,max(1500*rp1,0)));
             vT1 <= upperBound3; vT1++) {
      int upperBound4 = min(1500*rp1+1499,min(floord(vT1+N-3,2),vT1));
      for (vP1=max(1500*rp1,max(ceild(vT1,2),vT1-M+1));
           vP1 <= upperBound4; vP1++) {
         S1(vT1-vP1+1, -vT1+2*vP1+2);
      }
    }
  }
}
```

Note that there are four `for` loops, the outer two enumerating the tiles, the inner two enumerating the points within the respective tile. The second loop is marked `omp parallel for`, since it enumerates the parallel tiles. The body of the loop has become more complex compared to the original program, because the original loop indices `i` and `k`, in which the statement is expressed, have to be reconstructed from the new indices.

A fast execution of the program may seem unlikely due to the complex bounds in the loops and the necessary reconstruction of the original indices in the body. But it turns out that `i` (and hence the addresses of `A[i-1]`, `A[i]`, `A[i+1]`) is an induction variable even after the transformation and compilers can detect this. GCC 4.2.1 generates the following x86 assembly code for the innermost loop:

```
.L87:
        fldl    (%eax)
        addl    $1, %edx
        faddl   -16(%eax)
        fmul    %st(1), %st
        fstpl   -8(%eax)
        addl    $16, %eax
        cmpl    %edx, %ecx
        jge     .L87
```

The four floating-point instructions (`fldl`,`faddl`,`fmul`,`fstpl`) perform the computation `A[i]=(A[i+1]+A[i-1])*0.5` and the other four instructions deal with the loop counter and updating `%eax` which holds the address of `A[i+1]`. This implies that the loop bounds are only computed rarely compared to the execution of the body, provided that the innermost loop has a sufficient number of iterations.

The cyclic tile distribution among the cores described in Section 2.2 is not generated automatically at present. To get the code for this mode of operation, we first generate the code as described above, but then replace the `omp parallel for` pragma by an `omp parallel` pragma such that every core enumerates every tile. We let every core execute the inner loops only for every NCth iteration with different offsets for each core, with suitable synchronisation statements (`omp barrier`) inserted.

4 Experiments

We ran our experiments on a machine with four cores, consisting of two Dualcore AMD Opteron processors with 2.2 GHz and 2 GB of RAM. Since this is a NUMA (non-uniform memory access) architecture, we have to take care not to spoil the benchmarks with local vs. remote memory effects. As we will see, memory locality does not play a significant role in the one-dimensional SOR, because of its cache behaviour (cf. Section 4.2).

4.1 Tiling overhead

As has been outlined in Section 3, the complex loop bounds and the reconstruction of the original loop indices still permit the innermost loop to be small. We now look at the question of how many iterations the innermost loop must have to make the effort of computing the complex loop bounds negligible. Figure 6 shows the execution times for $n = 10^6$, $m = 9,000$ and varying tile heights (the innermost loop enumerates the height dimension of the tile). The tile width has been chosen according to the formulas presented in Section 2.1 with $f = 1$. The execution time converges towards about 12.38 seconds. We observe that the parallelogram tiling has a slightly higher overhead (the loop bounds are more complex). With the rectangular tiling, the tile height can only be increased up to

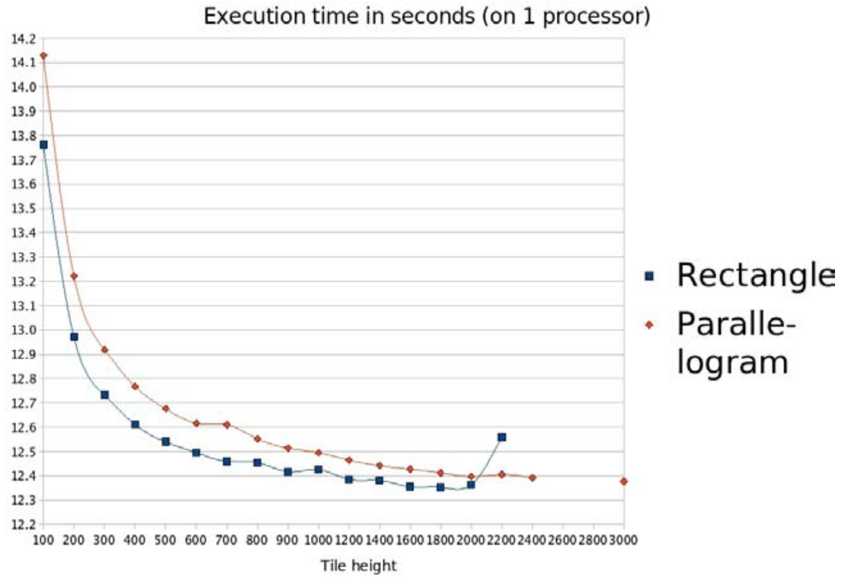


Fig. 6. Tiling overhead in 1-dim SOR for $n = 10^6, m = 9,000$ (x-axis non-linear)

about 2000, since the tile width has to be set to $\frac{m}{\text{NC}} - \text{height}$, i.e. $2250 - \text{height}$ in our case. Therefore, the tiles become very narrow for heights greater than 2000 and, accordingly, the second innermost loop has only a few iterations which causes noticable overhead for height greater than 2000. The parallelogram tiling does not suffer from such a restriction; the height can be arbitrarily increased as the width is fixed to, in this example, 2250. On the other hand, we have to note that the run-time evaluation of the loop bounds can suffer integer overflows, e.g., the program does not execute correctly for tile heights 2400 and 2600 (which are missing in the diagram for this reason).

4.2 Cache behaviour

The one-dimensional SOR example also demonstrates that space-time mapping and tiling can enhance the cache behaviour and, hence, reduce the execution time. Compared to the original execution, the tiled program is a vast improvement. The original program takes 75.4 seconds to execute. Even with space-time mapping applied (and no tiling), the execution time is still 23.6 seconds, but only with tiling the time reduces to about 12.4 seconds. Using the Cachegrind tool from the Valgrind suite [16], almost no cache misses occur (Table 1). For the backward substitution example, Cachegrind reports a cache miss rate of about 6.1% for both the standard parallel and the cyclic parallel execution.

	Reads		Writes		
Instructions	Total	Misses	Total	Misses	
18k	16k	2			`for glT1=`
102k	88k				`for rp1=`
42,235k	21,099k	18k			`for vT1=`
36,056,066k	14,035k				`for vP1=`
35,999,928k	17,999,964k	3,973k	8,999,982k	0	`A[i]=...`

Table 1. 1d SOR: cache misses (L1 and L2 combined)

4.3 Speed-up

As mentioned, the one-dimensional SOR example has an original execution time of 75.4 seconds, which is reduced to 23.6 seconds through space-time mapping. Table 2 shows the rectangular tiling with the best execution time on one and on four cores, respectively, and the best (on one and four cores) parallelogram tiling. The parallelogram tiling shows slightly better scaling behaviour, so it overtakes the rectangular tiling which has a slightly better execution time on one core.

	Rectangular Tiling			Parallelogram Tiling		
Cores	1	2	4	1	2	4
	Width=850, Height=1400					
Time in secs	12.38	6.33	3.23			
Speed-up	1.00	1.95	3.82			
Efficiency*	99.76%	97.55%	95.59%			
	Width=650, Height=1600			Width=2250, Height=3000		
Time in secs	12.35	6.32	3.26	12.38	6.28	3.17
Speed-up	1.00	1.95	3.79	1.00	1.97	3.90
Efficiency*	100.00%	97.71%	94.71%	99.76%	98.33%	97.40%

Table 2. Speed-ups for 1d SOR (efficiency relative to best execution)

For the backward substitution example, the proposed cyclic distribution of the tiles among the available cores does not lead to the expected speed-up, at least with the implementation strategy we have chosen. Table 3 shows a clear difference in speed-up on two cores, although the prediction for both parallel executions are the same (on two cores). Why this difference exists, although the cache behaviour (studied with Cachegrind) is almost identical for both programs and both programs have the same execution time on one core, is unclear at this point. We made sure that the two cores used to execute the program are on the same physical processor, i.e. all the memory accesses are to local memory. It remains to be investigated why the standard parallel and the cyclic parallel execution behave differently and how it is possible to exploit the load balancing promised by the cyclic parallel execution.

	Cores		Real Speed-up	Theoretical Speed-up in a very simple model
	1	2		
Sequential	203ms	-		
Standard parallel execution	177ms	124ms	1.43	1.832
Cyclic parallel execution	177ms	140ms	1.26	1.829

Table 3. Backward subsitution: Speed-ups for $n = 10,000$

5 Conclusions

After many years of research, the polyhedron model has finally reached a stage at which the last remaining big challenge, parallel code generation, is being tackled and solved step-by-step. From our preliminary experiments, which we have presented here, we can gain confidence that code generated from the polyhedral description of transformed sequential programs can exhibit good cache behaviour and small enough execution overhead in the complex loop bounds such that good speed-ups are possible. Further experiments have to be carried out to explore the situation with other, more complex examples.

Two problems which often spoil performance are load imbalances due to a varying number of tiles in the parallel dimension and the presence of incomplete tiles at the borders of the index space. Both problems need to be researched further; we have only offered basic ideas of solutions. After these problems have been solved, the polyhedron model may be ready for use in mainstream compilers for multicore architectures.

References

1. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. J. Supercomputing **14** (1967) 563–590
2. Lamport, L.: The parallel execution of DO loops. Comm. ACM **17** (1974) 83–93
3. Lengauer, C.: Loop parallelization in the polytope model. In Best, E., ed.: CONCUR'93. LNCS 715, Springer-Verlag (1993) 398–416
4. Griebl, M.: The Mechanical Parallelization of Loop Nests Containing while Loops. PhD thesis, University of Passau (1996) also available as technical report MIP-9701.
5. Größlinger, A., Griebl, M., Lengauer, C.: Introducing non-linear parameters to the polyhedron model. In Gerndt, M., Kereku, E., eds.: Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004). Research Report Series, LRR-TUM, Technische Universität München (2004) 1–12

6. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. Int. J. Parallel Processing **28** (2000) 469–498
7. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004), IEEE Computer Society Press (2004) 7–16
8. Xue, J., Huang, C.H.: Reuse-driven tiling for data locality. In Li, Z., Yew, P.C., Chatterjee, S., Huang, C.H., Sadajappan, P., Sehr, D., eds.: Languages and Compilers for Parallel Computing, Springer (1997) 17–33
9. Xue, J.: On tiling as a loop transformation. Parallel Processing Letters **7** (1997) 409–424
10. Griebl, M.: On the mechanical tiling of space-time mapped loop nests. Technical Report MIP-0009, Fakultät für Mathematik und Informatik, Universität Passau (2000)
11. Xue, J.: Communication-minimal tiling of uniform dependence loops. J. Parallel and Distributed Computing **42** (1997) 42–59
12. Andonov, R., Balev, S., Rajopadhye, S., Yanev, N.: Optimal semi-oblique tiling. In: Proc. 13th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA 2001), ACM Press (2001) 153–162 Extended version available as technical report: IRISA, nr. 1392, dec. 2001.
13. Griebl, M.: The minimal number of communication startups when tiling space-time mapped programs. In: Ninth International Workshop on Compilers for Parallel Computers (CPC 2001). (2001) 117–126
14. Griebl, M., Feautrier, P., Größlinger, A.: Forward communication only placements and their use for parallel program construction. In: Languages and Compilers for Parallel Computing, 15th International Workshop, LCPC'02. Revised Papers. Lecture Notes in Computer Science 2481, Springer-Verlag (2005) 16–30
15. Griebl, M., Lengauer, C.: The loop parallelizer LooPo. In Gerndt, M., ed.: Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96). Konferenzen des Forschungszentrums Jülich 21, Forschungszentrum Jülich (1996) 311–320
16. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007). (2007)