Practical Loop Generation

Z. S. Chamski M. F. P. O'Boyle
Centre for Novel Computing
The University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom
email: {chamskiz,mob}@cs.man.ac.uk

Abstract

This paper describes the integration of a formal loop generation technique into an auto-parallelizing compiler, MARS. A brief survey of loop generation techniques is given and is followed by the description of the loop generation strategy employed in our implementation. We describe the necessary input and output representations required for formal loop generation and describe how such a transformation fits into a complete compiler strategy. Given MARS' extended linear algebraic program representation and the constraints of a global compiler strategy, we have successfully integrated a formal tool into a FORTRAN compiler and have shown that combined they can outperform an existing commercial compiler.

1 Introduction

The goal of optimizing parallel compilers is to minimise the execution time of the compiled program. This is usually tackled by exploiting program parallelism and minimizing the impact of several overheads, typically communication, synchronisation and load balancing, and has been extensively studied ([2]).

Other overheads can be equally important in determining program execution time, e.g., local cache misses and array address translation [17]. In this paper we examine the impact of loop management overhead, introduced by evaluating the bounds of a loop and determining whether a loop body or statement should be executed. This overhead becomes particularly significant in a parallel environment, as parallelizing or partitioning a program introduces additional complexity into loop descriptions.

The research in this area has been targeted mainly at the complexity of algorithms used in a compiler, which were described in isolation rather than as components in an overall compilation strategy. In this paper, we first critically examine recent work in the area of loop generation techniques. Based on the example of MARS ([6]), an optimizing FORTRAN compiler, we then describe the integration of a systematic loop generation strategy into a compiler, and illustrate its effects on code fragments from FORTRAN application programs. Finally, we provide experimental evidence of the impact of loop generation techniques on overall program performance, further highlighted by comparing the achieved performance against that offered by a commercial parallelizing compiler.

2 Loop generation techniques: a critical review

The aim of the loop generation (LG) process is to produce a loop-based iterative code as efficient as possible, while preserving the runtime order of statement executions and synchronisation points of the program. This goal can be achieved in several ways of increasing complexity, both in terms of computations involved, and of restructuring transformations applied to the program being (re-)generated.

2.1 Basic requirements

Functionalities. Given a formal description of an iteration set (or domain) $\mathcal{IS}$, a loop generation technique must provide the information necessary to build a loop structure whose execution will enumerate all points in $\mathcal{IS}$ in a given lexicographic order.

The target loop structure can be either a perfect loop nest, in which case only the bounds (and possibly, the strides) of each loop have to be produced, or some form of imperfectly nested loops, which will require in addition the provision of the overall layout of various constructs (loops, conditionals, assignments) in the resulting syntactical structure. Hybrid solutions can also be used, in which the perfect loop nest approach is retained while the set of statements included in the body of the innermost loop is revised in connection with the bounds of the loop nest.

The perfect loop nest approach is easily amenable to a single-step operation in which the complete set of bounds is generated in one execution of the LG algorithm, and may lead to excessive run-time overheads. On the other hand, more flexible loop organisations require an incremental approach ([9]) at the risk of impairing the scalability of the technique if the compiler cost associated with the determination of each loop's bounds is too large. Therefore, the properties of the algorithm used as the core of the technique will be critical to its viability.

Input information. The description of an iteration set $\mathcal{IS}$ as supplied to the LG stage usually consists of a list of iterators $\mathcal{I}$ and a set of constraints $\mathcal{C}$ which are all satisfied only by points in $\mathcal{IS}$.

The iterators are a generalisation of loop indices and encompass any variable or formal parameter that can be used in delimiting iteration sets. Therefore, the number of iterators can be significantly larger than the nesting depth of the loop itself. The iterators which are not loop indices are naturally interpreted as loop parameters. The total number of iterators determines the complexity of loop generation and the complexity of resulting bound expressions or conditional guards on individual statements.

The fact that all constraints in $\mathcal{C}$ must be satisfied by the points in the iteration set implies that if all constraints are affine, the iteration set will be easily manipulated by linear programming techniques. However, this is not always the case; e.g., indirect references (as in "`do j = l(i),u(i)+2`") introduce expressions which can not be directly expressed in a purely affine framework and must be treated separately when using formal LG techniques. Furthermore, non-affine constraints can be disjunctions of other constraints, affine or not: $\texttt{j} \geq \texttt{min(i, f(i))}$ translates as "$\texttt{j} \geq \texttt{i}$ *or* $\texttt{j} \geq \texttt{f(i)}$", implying the presence of non-convex iteration sets.

In order to provide a simple description of $\mathcal{IS}$ at any time during the compilation process, the set $\mathcal{C}$ may contain redundant constraints. Such constraints will have to be eliminated during loop generation, so as to reduce the bound computation overhead.

Output information. In addition to the layout of the loop structure, the bound functions and index strides, a loop generation technique can also identify the conditions under which the entire loop or any particular statement in it is executed. This information can then be propagated *upwards* in the syntax tree and used to further reduce loop-related overheads.

2.2 Existing loop generation techniques

In the following, we present a voluntarily restricted discussion of several loop generation techniques described in the literature. The different solutions are grouped according to the underlying linear programming algorithm. This approach is motivated by the fact that the properties of the core algorithm determine to a large extent the relevant characteristics of each technique.

Fourier-Motzkin pairwise elimination. The Fourier-Motzkin pairwise elimination [13] is conceptually the simplest linear programming algorithm used for loop generation. It was proposed by Irigoin in [18], further explored in [1, 28] and extended in [3, 22, 25].

The Fourier-Motzkin pairwise elimination directly addresses the issue of loop bound overhead, and is used in [19] as the core of an algorithm which builds nested loop sequences ([9]) from unions of polytopes (bounded polyhedra), which can then be used to discriminate between different statement sets associated with each of the polytopes. An alternative technique for building nested loop sequences from non-convex iteration sets based on the Fourier-Motzkin elimination has been proposed in [29] but, as opposed to the approach of [19], it cannot handle the case of sets of polytopes labeled with distinct statement sets.

Extensions to the basic version of the algorithm, to be found in [3, 22, 25], deal with the case of iteration sets defined over non-dense integer lattices (Z-modules) leading to loops with non-unit strides. Pugh's Omega-test [24] is an extension of the Fourier-Motzkin's elimination designed to handle the case of integer solutions, useful to identify and eliminate empty iteration sets.

The single-step operation of the Fourier-Motzkin elimination can be a limitation to its intensive use, and does not allow the decomposition of the bound computation into independent steps.

The combinatorial operation of the algorithm can cause tractability problems when operating on higher-dimension spaces. Experiences show that while the performance of the Fourier-Motzkin for low dimensions and constraint numbers is indeed very good ([22, 4]), it degrades significantly for larger problems in terms of both execution time and memory usage ([8, 20]) and faces arithmetic precision problems if the constraint sets contain large coefficients.

PIP. The *Parametric Integer Programming* (*PIP*) algorithm was devised by Feautrier in [14]. It is an extension of the Simplex algorithm ([12, 26]) aimed at handling the case of parametric inequalities. The algorithm uses the ascending lexicographic ordering at the cost function. This makes it suitable for solving problems arising from program transformation and guarantees the uniqueness of the optimum.

Applications of PIP in the context of loop generation have been discussed in [15, 16, 7, 11, 9, 8, 10]. While it was initially used to compute the bounds of perfect loop nests, PIP serves also as the basic building block of a nested loop sequence generator for domains defined by means of both single polytopes ([9]) and arbitrary unions of polytopes ([10]). Also, PIP can conveniently compute piecewise linear bounds of "jagged" dense iteration sets [10].

An important characteristic of PIP is its possibility of determining the bounds independently for each iterator. This provides a means of decomposing the loop generation process into elementary steps which can be carried out in parallel ([8]) and allows an incremental operation in which the amount of redundant computations at each step is significantly reduced.

In terms of scalability, the PIP algorithm (as most Simplex implementations) trades space requirements for high absolute values of coefficients handled during the computations, and requires an implementation based on arbitrary-precision arithmetic. The general complexity characteristics of the Simplex make this approach viable ([8]) and remove any limitations linked to the values of input coefficients.

"Chernikova's" algorithm. A technique based on the the dual representation of polyhedra by constraints and rays/vertices has been proposed in [21]. It is primarily intended for the generation of bounds in perfect loop nests and operates in a single pass from the outermost to the innermost iterator.

The implementation of this technique is based on the Polylib toolkit ([27]) and, while being fast, suffers problems with fixed-precision arithmetic on larger domains. An arbitrary-precision version of this toolkit is currently under development, and will provide a testbed for a more systematic experimentation.

3 Integrating advanced loop generation into a realistic compiler

In the light of the above review, we decided to explore the integration of approaches based on PIP into the MARS compiler, jointly developed by Irisa (Rennes, France) and the Centre for Novel Computing (Manchester, Great Britain). Our choice of the loop generation framework is based on the flexibility and the scalability of the algorithm, and the expressive power of the nested loop sequence construct ([9]).

3.1 Compiler structure

MARS is an optimizing compiler targeted at single address space architectures such as KSR-1, Meiko CS-2 and SGI Power Challenge. It is built on top of Sage++ ([**?**]) and consists of approximately 20,000 lines of C++. The overall structure of the compiler is shown in figure 1.

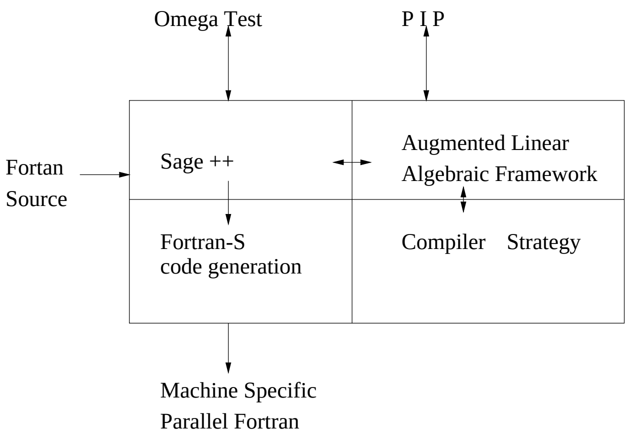

Figure 1: MARS structure

Fortran77 is read in and stored as an annotated syntax tree in Sage++, then translated into an augmented affine form. This format allows high level transformations to be applied, simplifying the development of a compiler strategy. It also allows formal tools such as PIP to be interfaced to a general purpose compiler allowing exploitation of linear properties, while providing a means of representing all legal Fortran programs.

The compiler strategy consists of a sequence of phases based on a cost model of the target architecture [6], and is summarised in table 1. Each phase transforms and updates the linear algebraic representation of the program. The updated representation is translated back into the Sage++ format. The Fortran-S [5] compilation system wraps the node program in machine-specific directives before it is unparsed as a parallel Fortran program.

Pre-Partitioning	Parallel Detection
	Significant Regions
Partitioning	Global Alignment
	Global Data Partitioning
Post-Partitioning	Local Realignment
	Work Scheduling
	Synchronisation Placement
	Mapping
	Hierarchical Locality Optim'n
	Loop Optimization

Table 1: Compiler Phases

3.2 Interfacing LG with other compilation stages

Loop generation is constrained in the choice of possible program transformations, as otherwise it may undo the work of previous compiler stages. Given that inter-processor costs (such as communication and synchronisation) are considered more significant than node overheads (cache misses and loop overhead), they take higher priority if conflicting transformation requirements arise.

Size parameters, processor numbers and partitioning. In loop generation theory it is often assumed that the equations or inequalities expressing partitioning relations are always written in affine form, as shown in equations 1, 2 and 3:

$$(z-1)\times m+1 \leq \quad j \quad \leq z\times m-1 \qquad (1)$$

$$mp \leq \quad N \quad < (m+1)p \qquad (2)$$

$$1 \leq \quad z \quad \leq p \qquad (3)$$

where N denotes the total number of elements along the partitioned dimension, p is an *integer constant* corresponding to the number of processors used for partitioning along this dimension, m denotes the size of the resulting blocks, z is the processor number, and j is the partitioned index.

Loop generation techniques presented in the literature are usually restricted to the context of polyhedral iteration sets delimited by affine constraints. However, this is not always directly applicable, and may require major modifications to meet the requirements of the end user. In particular, the number of available processors may be supplied dynamically at the beginning of each execution to avoid systematic recompilations when the program is to be run on a different machine configuration.

With the number of processors p becoming a free parameter, the definition of block size as given by inequality 2 is no longer affine. To work around this, one can note that the bounds of the block allocated to a processor are invariant during execution. Hence, we introduce two new symbolic iterators lo_i and hi_i, local to each processor and representing the lower and upper bounds of the values of an array index i which are mapped to each processor. If the array is partitioned over k dimensions, this will introduce $2k$ additional parameters, defined as follows :

$$lo_i = (z_i - 1) \times \lceil \frac{\upsilon_i - \lambda_i + 1}{p_i} \rceil + 1,$$

$$hi_i = z_i \times min(\lceil \frac{\upsilon_i - \lambda_i + 1}{p_i} \rceil, \upsilon_i),$$

where z_i is the processor id in partitioned dimension i, υ_i and λ_i are the upper and lower bounds of the array elements local to processor z_i in partitioned dimension i and p_i is the number of processors in dimension i. These parameters are considered as the outermost iterators and together with array bounds and loop indices form an *affine* framework for expressing iteration set constraints related to partitioning.

Integrating other non-affine constraints. The general "non-affine" case corresponds to bounds expressed using indirections and appears, e.g., in sparse system solvers. To proceed with the static analysis and optimization of loop structures in presence of indirect index expressions, it is necessary either to re-express these in an affine framework by substituting each redirected bound term with a new iterator, or to skip over the non-affine bound expressions and consider only the purely affine subset of constraints. We apply the latter approach and use the bound computation decoupling offered by PIP ([8]) to skip over the computation of bounds which appear unbounded due to lack of associated affine constraints.

3.3 A global approach to loop generation

In MARS, information relevant to loop generation is supplied in two forms: loop nest constraints, and statement constraints. Loop nest constraints are the image of the original iteration spaces augmented with partitioning relations. Statement constraints are introduced, e.g., by the "owner-computes" scheduling and data partitioning.

The loop and statement constraints can be exploited in several ways to reduce the overhead of loop generation, depending on the allowed degree of program restructuring:

1. by attaching loop-invariant constraints on a single-statement loop body to the loop nest in which they are invariant at some level;
2. by computing the simplest possible bound expressions without altering the overall layout of loops and conditionals;
3. by eliminating redundant statement constraints and unnecessary iterations of outer loops;
4. by eliminating iterator-dependent statement constraints which are invariant *over a region of the iteration set*;
5. by reducing/minimising the bound evaluation overhead through building appropriate nested loop sequences out of perfect loop nests;
6. by replacing loops reduced to a single iteration by their bodies in which the occurrences of the index are substituted with its actual value.

Strategy. The relocation of loop-invariant constraints can be thought of as a preprocessing step in the LG process, and requires only simple symbolic manipulation. As a result, each loop nest is given a (possibly empty) set of additional constraints restricting its iteration set. In the general case, this is only carried out for single-statement loop bodies. However, global array alignment ensures that write accesses sharing a common access function will have identical statement constraint sets.

Bound recomputation is directly carried out using PIP, from the outermost loop inwards. The input information supplied to PIP is the union of the affine subsets of constraints $\mathcal{C}_{\mathcal{IS}}$ describing the nest's iteration set $\mathcal{IS}$ and additional constraints $\mathcal{C}_{extra}$ propagated from inside the loop body.

As the generation of nested loop sequences from unions of domains is defined only for dense iteration spaces (i.e., for programs in which all loops have unit strides), the loop generation process is restricted for programs which do not satisfy this property. In such cases only redundant bounds, unnecessary iterations and redundant statement constraints can be removed.

Statement constraints which could not be eliminated as redundant after being propagated to higher levels in the tree must be represented by explicit conditional constructs in the final program.

The NLS generation algorithm used here differs from the version in [10] in that the constraint systems delimiting the iteration sets of individual statements are built on the fly, whenever an assignment statement is reached when exploring the program tree, rather that being given at the beginning of the computations. This in turn implies that some branches of the resulting loop tree may exhibit empty iteration sets under some constraints on outer indices. If so, the constraints are used to decompose the appropriate outer loop(s) and when applicable, the longest straight branch containing the offending node is pruned off the loop tree as it corresponds to a loop nest with no actual iterations.

The overall loop generation strategy is summarized in figure 2.

1. Annotation pass: $\forall$ statements S subject to a statement-specific constraint set C_S, propagate C_S and a pointer to S to the outermost parent loop L in which C_S remains invariant.

2a. Non-restructuring loop generation:

1. $\forall$ loop L, generate irredundant bounds for the affine part of $\mathcal{IS}(L)$. Reconstruct directly bounds corresponding to non-affine constraints.
2. Eliminate all redundant statement-specific constraint sets C_S.
3. $\forall$ loop L annotated with a constraint set C_S applicable to all statements in L, replace L by

   IF C_S THEN L ENDIF.
4. For each remaining guard annotation C_S, introduce a new boolean variable B_S and generate the sequence

   IF C_S THEN B_S = true
   ELSE B_S = false
   ENDIF
   code for L with S replaced with
   IF B_S THEN S ENDIF

2b: Restructuring loop generation:

1. Build a global nested loop sequence from each top-level loop tree using constraint sets C_S as iteration *sub*-space identification.
2. Replace each loop L reduced to a single iteration by the appropriate instance of the body of L.

Figure 2: Global loop generation strategy.

4 Application examples

This section presents a few examples to illustrate some of the techniques described in the previous section. Consider the loop shown in fig. 3*a*. Without modifying the loop layout, it is possible to rewrite automatically its bounds into the form shown in fig. 3b, by taking into account the fact that block bounds lo_1, hi_1, lo_2 and hi_2 are by construction contained in the index range, in this case

$$1 \leq \quad lo_1 \leq hi_1 \quad \leq 512$$
$$1 \leq \quad lo_2 \leq hi_2 \quad \leq 512.$$

```
 do  j = max (1,lo2),min (512,hi2)
    do  i = max (1,lo1),min (512,hi1)
       psi(i,j) = amp * sin ((float (i) -&
&0.5) * di) * sin ((float (j) - 0.5) * dj)
    enddo
 enddo
```

a) intermediate code: redundant bound functions

```
 do  j = lo2,hi2
    do  i = lo1,hi1
       psi(i,j) = amp * sin ((float (i) -&
&0.5) * di) * sin ((float (j) - 0.5) * dj)
    enddo
 enddo
```

b) optimized code: one function per bound

Figure 3: Elimination of redundant bound functions (origin: `Shallow`).

Given a loop nest, bound optimization helps reducing the overhead related to loop start-up time at any level in the nest. However, it does not address the cost of executing constructs others than loops, and especially, assignment statements guarded by data-independent conditionals. Such conditionals can only be eliminated by restructuring the body of the loop in conjunction with its bounds, as shown in figure 4, where a new conditional guards the execution of the modified loop.

Consider the loop outline in figure 5, based on a wing modelling code used by a European aircraft manufacturer. Given a block partition on index 1 and 2, each processor will compute array elements satisfying

$$lo_1 \leq \quad index1 \quad \leq hi_1 \tag{4}$$
$$lo_2 \leq \quad index2 \quad \leq hi_2 \tag{5}$$

giving the following per-statement constraints:

$$S_1 \quad : \quad lo_1 \leq j \leq hi_1 \wedge lo_2 \leq k+1 \leq hi_2 \tag{6}$$
$$S_2 \quad : \quad lo_1 \leq j \leq hi_1 \wedge lo_2 \leq k+2 \leq hi_2 \tag{7}$$
$$S_3 \quad : \quad lo_1 \leq j \leq hi_1 \wedge lo_2 \leq k+1 \leq hi_2 \tag{8}$$
$$S_4 \quad : \quad lo_1 \leq j+1 \leq hi_1 \wedge lo_2 \leq k+1 \leq hi_2 \tag{9}$$

```
 do  i = 1,512
   if (lo2 .le. 1) then
     vs(i,1) = v(i,1) + alpha *&
&(vnew(i,1) - 2.0 * v(i,1) + vsold(i,1))
   endif
 enddo
```

a) intermediate code: condition is loop-invariant

```
 if (lo2 .le. 1) then
   do  i = 1,512
     vs(i,1) = v(i,1) + alpha *&
&(vnew(i,1) - 2.0 * v(i,1) + vsold(i,1))
   enddo
 endif
```

b) optimized code: condition placed outside the loop

Figure 4: Moving a loop-invariant condition outside the loop (origin: `SHALLOW`).

```
                do  k = 1,ix-2
                 do  j = 2,iy-1
                   do  n = 1,iz
S1               U(n,j,k+1)   = ...
S2               W(n,j,k+2)   = ...
S3               V(n,j,k+1)   = ...
S4               X(n,j+1,k+1) = ...
                     enddo
                    enddo
                   enddo
```

Figure 5: Multi-statement loop nest (origin: `WING`).

As a result, only statements S_1 and S_3 share a common set of constraints, those associated with statements S_2 and S_4 defining two additional regions in the iteration space of the loop. The set of the separate regions over which the same subset of the original sequence S_1; S_2; S_3; S_4 is executed is shown in fig. 6, the shades of grey indicating the number of distinct statements being executed (lightest: 1, darkest: 4). The code resulting from the elimination of single-iteration loops is shown in figure 7.

Although this code is more complex, the four conditionals per statement have been eliminated, reducing the cost of each iteration and the total number of evaluated conditionals by an order of magnitude.

5 Experiments

The loop generation strategy outlined in section 3 has been implemented in MARS. Some programming

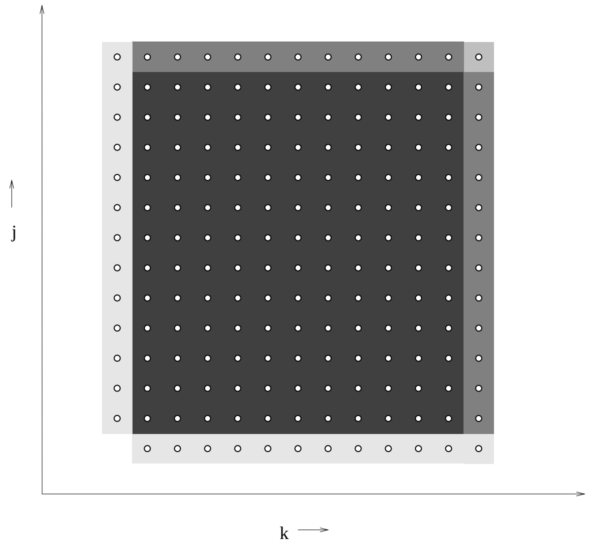

Figure 6: Overlapping local iteration spaces (origin: WING).

```
do  j = lo_1,hi_1
  S_2 with k = lo_2-2
enddo
do  k = lo_2-1,hi_2-2
  S_4 with j = lo_1-1
  do  j = lo_1,hi_1-1
    S_1; S_2; S_3; S_4
  enddo
  S_1; S_2; S_3 with j = hi_1
enddo
S_4 with k = hi_2-1 ∧ j = lo_1-1
do  j = lo_1,hi_1-1
  S_1; S_3; S_4 with k = hi_2-1
enddo
S_1; S_3 with k = hi_2-1 ∧ j = hi_1
```

Figure 7: Final nested loop sequence for WING.

effort was spent in converting MARS affine representation to the form used by PIP, but the majority of the time was spent in implementing the restructuring within MARS based on the output of PIP.

Two sample programs were chosen to investigate the impact of loop generation optimization on parallel performance. WING is part of a program used by a European aircraft manufacturer to aid in the design of certain aircraft parts, while SHALLOW is a well known implementation of the shallow water equations and is widely used as benchmark.

We compared the execution times of the programs produced by the original MARS compiler, the updated version of MARS. The performance reference was the code produced by KAP, a commercial optimizing compiler. The original MARS compiler generated guards for each statement within a loop after partitioning. In the case where there was just one statement, the guard was incorporated into the loop bounds. The KAP compiler uses an entirely different compilation strategy: on the KSR-1, it uses a fork&join model and defers until runtime a part of the bound generation for tiled loops.

The programs were run on a KSR-1 for varying data sizes and processor numbers. The 1/time ratio was plotted against processor number to show the speedup of each version of the program, with the results shown in figures 9 to 14, at the end of this paper.

From the figures, it can be seen that the old version of MARS is superior to KAP over all data and processor sizes for the WING program. It performs particularly well for small grain sizes, i.e., when the number of processors p is large compared to problem size n. This is due to the amount of additional synchronisation placed by KAP which grows linearly with n, while the amount of work is $O(\frac{n^3}{p})$. Hence, for relatively small values of n and large values of p, this synchronisation overhead shows to be significant. The optimized loop generation shows a major improvement over the original code produced by MARS (up to 2.5 times) and results in a code up to 10 times faster than the code generated by KAP. Once again, the biggest improvement is for smaller grain sizes.

A similar scenario occurs for SHALLOW. In this case however, KAP does much better wrt. the old MARS compiler, and in one case ($n = 512, p = 32$) slightly outperforms MARS. On closer examination, the bottleneck appeared to be due to the compiler-generated code, shown in figure 8a.

The automatic elimination of redundant statement guards produced the code shown in figure 8b, with an overall gain in performance ranging from 19 to 56% (33% avg.) across all test cases.

6 Conclusion

This paper describes how a parallelizing compiler can use loop generation techniques to enhance program performance. Using an affine representation of the program, we have successfully integrated a formal loop generation tool into a FORTRAN compiler, while respecting the constraints of a global compiler strategy. We show that the resulting system significantly outperforms an existing commercial solution.

```
do  j = 1,512
   do  i = 1,512
      if (lo0 + (-1) * i .le. 0) then
          if ((-1) * hi0 + i .le. 0) then
             if (lo1 + (-1) * j .le. 0) then
                if ((-1) * hi1 + j .le. 0) then
                   p(i,j) = pnew(i,j)
                endif
             endif
          endif
       endif
      if (lo0 + (-1) * i .le. 0) then
          if ((-1) * hi0 + i .le. 0) then
             if (lo1 + (-1) * j .le. 0) then
                if ((-1) * hi1 + j .le. 0) then
                   u(i,j) = unew(i,j)
                endif
             endif
          endif
       endif
      if (lo0 + (-1) * i .le. 0) then
          if ((-1) * hi0 + i .le. 0) then
             if (lo1 + (-1) * j .le. 0) then
                if ((-1) * hi1 + j .le. 0) then
                   v(i,j) = vnew(i,j)
                endif
             endif
          endif
       endif
   enddo
enddo
```

a) intermediate code: unnecessary iterations and guards

```
do  j = lo1,hi1
   do  i = lo0,hi0
      p(i,j) = pnew(i,j)
      u(i,j) = unew(i,j)
      v(i,j) = vnew(i,j)
   enddo
enddo
```

b) optimized code: conditional-free and restricted to useful iterations

Figure 8: Elimination of invariant guards and unnecessary iterations (origin: SHALLOW).

Although encouraging, the results at this stage are only preliminary and further experimentation, possibly on different architectures, will be needed to fully validate our approach. In particular, trade-offs between static and dynamic loop generation will be investigated, as compile-time loop generation costs induced by complex program transformations can dominate the overall compilation time ([?]).

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50. ACM, June 1991.

[2] J.M. Anderson and M.S. Lam Global Optimizations for Parallelism and Localiy on Scalable Parallel Machines, In *ACM/SIGPLAN Symposium on Programming Languages Design and Implementation*, ACM Press, 1993

[3] M. Barnett and Ch. Lengauer. Unimodularity considered non-essential. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Parallel Processing: Conpar 92 - VAPP V*, pages 659–664. Springer-Verlag, September 1992. LNCS 634.

[4] A. J. C. Bik and Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report 94–42, Department of Computer Science, Leiden University, The Netherlands, 1994.

[5] F,Bodin, L. Kervella and T. Priol *Fortran-S: A Fortran Interface for Shared Virtual Memory Architecture*, In Supercomputing, November 1993

[6] F. Bodin and M. O'Boyle. A Compiler Strategy for SVM In *Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computing* Kluwer Press, May 1995.

[7] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* PhD thesis, Université de Rennes I, February 1993.

[8] Z. Chamski. Fast and efficient generation of loop bounds. In G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans, editors, *Parallel Computing: Trends and Applications*, pages 265–272. North Holland, 1994.

[9] Z. Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. In *27*th *Hawaii International Conference on System Sciences*, volume II: Software Technology, pages 14–22, 1994.

[10] Z. Chamski. Beyond convexity: Scanning 'nonconvex polyhedra'. In *28*th *Hawaii International Conference on System Sciences*, volume II: Software Technology, pages 73–82. IEEE, January 1995.

[11] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. Research Report 93-15, École Normale Supérieure de Lyon, Lyon, France, May 1993.

[12] G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347, New York, 1951. Wiley.

[13] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Computational Theory*, 14:288–297, 1974.

[14] P. Feautrier. Parametric integer programming. *Recherche Opérationnelle / Operations Research*, 22(3):243–268, 1988.

[15] P. Feautrier. Semantical analysis and mathematical programming. Application to parallelization and vectorization. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms*, pages 309–320. North-Holland, 1989.

[16] P. Feautrier and N. Tawbi. Résolution de systèmes d'inéquations linéaires ; mode d'emploi du logiciel PIP. Rapport technique 90-2, MASI, Univ. Pierre et Marie Curie, Paris, France, January 1990.

[17] S. Hiranandani, K. Kennedy, J. Mellor-Crummet and A. Sethi, Compilation Techniques for Block-Cyclic Distributions In *ACM/SIGPLAN International Conference on Supercomputing*, ACM, Manchester, July 1994.

[18] F. Irigoin. *Partitionnement des boucles imbriquées. Une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Univ. Pierre et Marie Curie, Paris, France, June 1987.

[19] P. M. W. Knijnenburg and A. J. C. Bik. On reducing overheads in loops. Technical Report 95–07, Department of Computer Science, Leiden University, The Netherlands, March 1995.

[20] M. Le Fur. Parcours de polyèdre paramétré avec l'élimination de Fourier-Motzkin. Publication interne 858, Irisa, Rennes, France, September 1994.

[21] H. Le Verge, V. van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Publication interne 830, Irisa, Rennes, France, May 1994.

[22] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In U. Banerjee, D. Gelertner, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 391–405. Springer Verlag, 1993. LNCS 757.

[23] M.F.P. O'Boyle , L.Kervella and F. Bodin, Synchronisation Minimisation in a SPMD execution model, *to appear* in Journal of Parallel and Distributed Computing special issue on Distributed Shared Memory Systems.

[24] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[25] J. Ramanujam. Non-unimodular transformations of nested loops. In *Supercomputing '92*, pages 214–233. IEEE, 1992.

[26] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley Interscience Series in Discrete Mathematics. John Wiley and Sons, 1986.

[27] D. Wilde. A library for doing polyhedral operations. Publication Interne 785, Irisa, Rennes, France, December 1993.

[28] M. J. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[29] J. Xue. Construction of DO loops for non-singular transformations of non-convex iteration spaces. Submitted to *Parallel Processing Letters*, 1994.

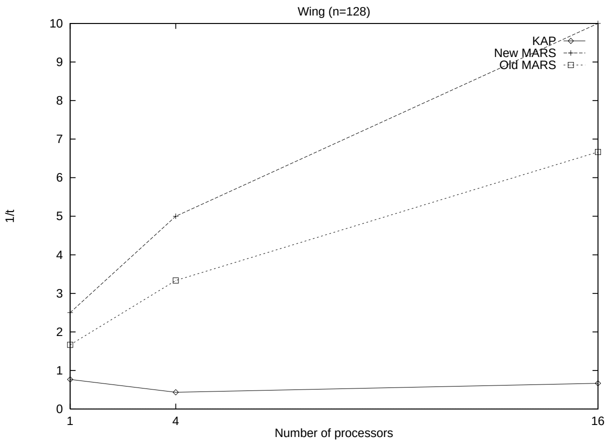

Figure 9: Performance of 128×128 `WING`.

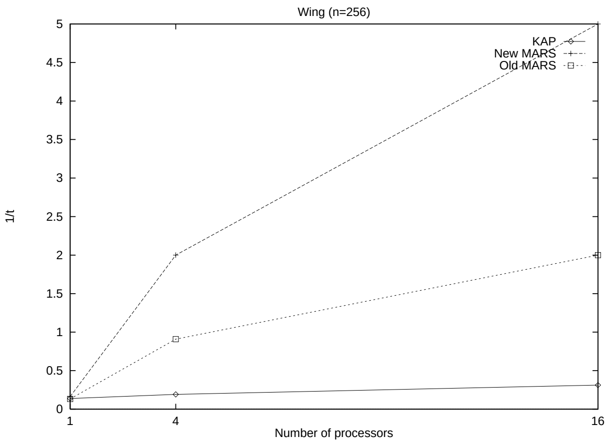

Figure 10: Performance of 256×256 `WING`.

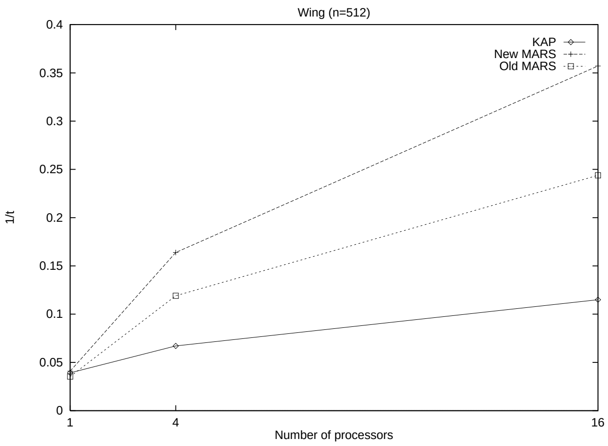

Figure 11: Performance of 512×512 `WING`.

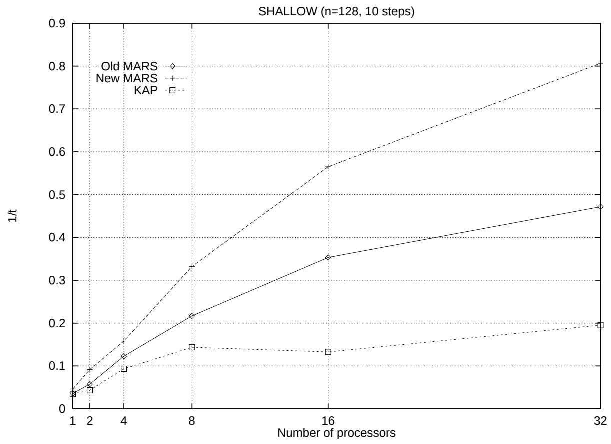

Figure 12: Performance of 128×128 `SHALLOW`.

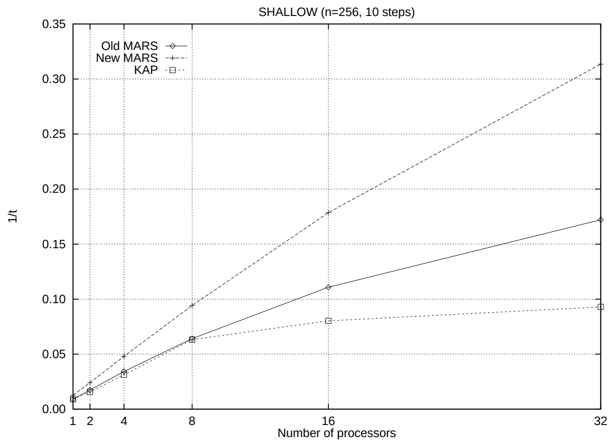

Figure 13: Performance of 256×256 `SHALLOW`.

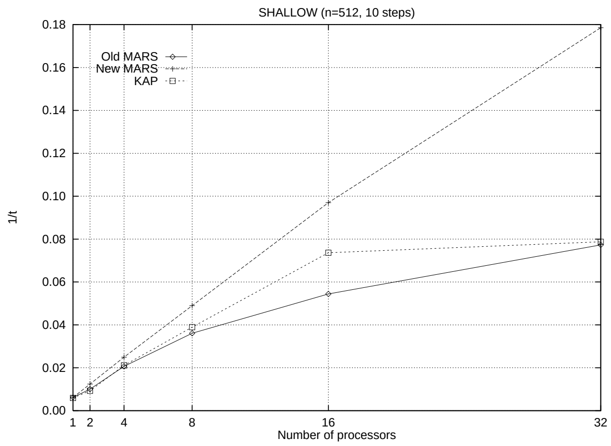

Figure 14: Performance of 512×512 `SHALLOW`.