Using Genetic Algorithms to Optimize the Automatic Parallelization Function*

Kenneth P. Williams

Dept. of Computer Science
University of Reading
Whiteknights, Reading
UK, RG6 6AY
K.P.Williams@reading.ac.uk

Shirley A. Williams

Dept. of Computer Science
University of Reading
Whiteknights, Reading
UK, RG6 6AY
Shirley.Williams@reading.ac.uk

Abstract

The limited ability of compilers to find the implicit parallelism in sequential programs is a significant barrier to the improved use of high performance computers. Optimizations for high performance architectures aim to maximize parallelism and memory locality with transformations based on extensive control and data dependency analysis - with particular emphasis on loop-transformations. However, current optimizing compilers lack an organizing framework that enables the direct calculation of the optimal sequence of loop transformations to be applied.

Our proposed solution involves transforming the source program into a scaled-down representation, called 'skeletal' form to which loop transformations can be applied as part of a genetic search. The resulting population of skeletal programs can then be executed to quickly evaluate the 'fitness' of the sequence of loop-transformations applied to each. Ultimately, the fittest sequence of loop-transformations will be found which can then then be applied to the original source program to produce the optimized executable code. In conclusion we compare genetic compiling (GC) with existing techniques.

1 Introduction

Extensive studies evaluating the effectiveness of parallelizing compilers have highlighted the limitations of current techniques. Various criteria against which parallelizing compilers are evaluated are used, the most common approaches being to test their effectiveness against a set of programs restructured by (i) other parallelizing compilers, or (ii) manually by a programmer. The results are disappointing. One study of four Perfect benchmark programs compiled on an Alliant FX/8 produced speedups between 0.9 (that is, a slowdown) and 2.36 out of a potential 32; when the applications were tuned by hand, the speedups ranged from 5.1 to 13.2 [4].

Some of the major problems identified include: loop transformation techniques not being applied to the most computationally intensive loops in the program (ii) the application of one technique may prevent the application of another (called *interference*) (iii) the heuristics used by the compiler may decide incorrectly that a technique is not worth applying (iv) the inappropriate application of some techniques may have detrimental effects on overall performance (v) timeouts in the compilers' search for an optimal sequence of transformations. We will show how these problems are tackled by genetic compiling (GC).

2 Genetic Compiling

Genetic algorithms (GAs) have been shown to be successful in solving problems where direct analysis is ineffective or impossible. Our research initially set out to tackle the problem of loop transformation interference. In our proposed model, a sequence of optimizations and transformations may be encoded into gene form in the following manner: given a set of loop transformations *Trans*, we assign a letter to each member of the set thus:

*Part of this work is funded by the Engineering and Physical Sciences Research Council (EPSRC), UK.

Trans = {*A,B,C,* ....}
A = *loop-fusion*
B = *loop-interchange*
C = *strip-mining*
etc ...

Next, we encode a sequence of transformations into a character string, *Ch*, called a *chromosome*:

Ch = {*A,C,B,Z,A,* ....}

The next step is to apply the sequence of transformations specified in *Ch* to a sequential program *S* to produce a parallelized version *P* which can then be compiled and run on a target machine. Clearly, the order in which the transformations are applied has a significant impact on the performance of *P*. Interestingly, we can create another chromosome *Ch2*, which specifies a different sequence of transformations and then apply *Ch2* to *S* so we create another parallelized program *P2*. By noting the times programs *P* and *P2* take to execute on the target machine (and possibly other factors such as memory usage, too), we can say that if *P* executes faster than *P2* then the sequence *Ch* is 'fitter' than *Ch2*. Similarly, if program *P2* executes faster than *P*, we can infer likewise for the sequence *Ch2*. If the execution times for the two programs are approximately the same then we can say the two chromosomes *Ch* and *Ch2* have approximately equal fitness.

2.1 The Compilation Search

We can initiate a genetic search for the ordering of loop transformations which will produce the parallelized version of *S* which has the shortest execution time on a designated target machine, (or more precisely, the chromosome which achieves the highest fitness rating), in the following manner.

The first step is to create an initial *population* of parallelizing compilers, each of which has its own chromosome encoding which the type and order of loop transformations it will apply. Next, we give one copy of our original program *S* to each compiler in our population and allow them to apply their sequence of transformations. The resulting output from each individual is a parallelized version of *S*. We then execute each of these programs on the target machine and record their execution times to evaluate their 'fitness'. Next, we generate a new population by performing genetic operations such as *mutation*, *crossover* and fitness proportionate *reproduction* on the individuals whose fitness has just been measured. Finally we discard the old population and iterate with the new population. The full process is shown in Fig.4.

The first iteration of this loop may work on a population of randomly generated individuals. From there on, fitness testing, in concert with the genetic operations, will work to improve the performance of the overall population (i.e. 'survival of the fittest' for sequences of optimizations).

2.2 Problems with Encoding

The basic method described above will successfully produce a parallelized version of a sequential program, however, three problems are evident:

1. If the only way to test the effectiveness of the parallelized code is to compile and execute it on the target machine, then for some programs (e.g. computation programs that are only likely to be executed only a few times) then the genetic compilation process may be too expensive (in terms of machine time, etc).

2. The whole process will require extensive computing resources.

3. Simply encoding a sequence of transformations is too vague in that it does not specify *how* the transformations are to be applied (e.g. to each loop individually, or just to some loops, or to all loops together in some way, etc).

We solve problems (1) and (2) by translating the sequential program *S* into an intermediate form (called *skeletal form*). We overcome problem (3) by creating different *species* of compiler that co-exist within our population.

2.2.1 Skeletal Form

In order to perform the genetic search as quickly as possible it is essential that each member of the population has their fitness assessed quickly. Therefore any technique which can speed up the execution of the parallelized code on the target machine will be of interest to us. It will be noted that control and data dependency information is of vital importance in the automatic parallelization process. It will also be noted that the dependencies for any given loop hold true for that loop regardless of the number iterations it performs. In other words, whether a loop executes

10 times or 10,000 times, the inherent structure of the control and data dependencies of the loop, remain the same. If we are applying a sequence of transformations to loops in a program, we can reduce the indices of each loop in order to speed up program execution. This preserves the vital dependency information we need while significantly speeding up the fitness evaluation function.

To illustrate, in Fig.1, even though the skeletal version of the loop executes fewer iterations, *the structure of the control and data dependencies for the two loops are the same*, namely: $S_3\delta^t_{(+1)}S_2$, $S_1\delta^a_{(+1)}S_3$ and $S_3\delta^a_{(0)}S_3$

```
/* Original loop    */

            for (i=1; i<=10000; i++) {
      S1:      X[i] = C[i] + Z[i+1];
      S2:      Y[i] = C[i] * Z[i-1];
      S3:      Z[i] = Z[i] - A[i];
            }

/* Skeletal loop    */

            for (i=1; i<=10; i++) {
      S1:      X[i] = C[i] + Z[i+1];
      S2:      Y[i] = C[i] * Z[i-1];
      S3:      Z[i] = Z[i] - A[i];
            }
```

Figure 1: The structure of the control and data dependencies of the original loop and its intermediate 'skeletal' form are the same.

In order to maintain the global relationships between loops in a program we define a *reduction function*, R, which represents a scaling factor by which the indices of each loop can be reduced while maintaining the ratios of iterations performed between loops and also the control and data dependency information.

An initial definition of R involves noting the number of iterations to be performed by each loop in the program and storing them in vector form (I^1,I^2,...,I^n). We then calculate the greatest common divisor of these values:

$$R = \gcd(I^1,I^2,...,I^n)$$

Once R has been computed the number of iterations performed by each loop is scaled down accordingly. The final indices for each loop will also have to take into account the size and type of data structures involved. The resulting program is said to be in 'skeletal form'. These concepts are illustrated in Figs. 2 & 3. The technique also extends to doubly nested loops and beyond.

```
   /* Original program     */

  for (i=1; i<=10000; i++) {
    .......        /* 10000 iterations */
  }

  for (i=0; i<5000; i++) {
    .......        /* 5000 iterations */
  }

  for (z=3000; z>=1500; z--) {
    .......        /* 1500 iterations */
  }

 /* R = gcd(10000,5000,1500) = 500 */
```

Figure 2: The loops of the original program.

```
   /* Skeletal program (R = 500) */

  for (i=1; i<=20; i++) {
    ........       /* 20 iterations */
  }

  for (i=0; i<10; i++) {
    .......       /* 10 iterations */
  }

  for (z=3000;z>=2997;z--) {
    .......       /* 3 iterations */
  }
```

Figure 3: The 'skeletal' version of Fig.2. Note how the number of iterations each loop performs has been scaled down by 500 (R) while preserving the structure of the program control and data dependencies [‡].

To summarize, the reduction transformation should not change the fundamental structure of the program dependencies, it simply *reduces* the quantities (i.e. the

[‡]Preliminary results indicate that the speedup obtained by executing the skeletal program are almost linear with R. This means the fitness of a sequence of optimizations can be evaluated very quickly (up to 500 times faster than executing the original program).

number of iterations) involved.

2.3 Related Work

Early program restructuring for supercomputers goes back to the ILLIAC-IV project and the work of David Kuck, *et al.* at the University of Illinois. Most restructuring transformations are under-pinned by the data dependency work done by Wolfe [11] and Banerjee [2] as well as Ferrante *et.al* [5]. More recently code re-arrangement with rescheduling techniques have been developed by William Pugh [6] and notably unimodular loop-transformations by Utpal Banerjee [3]. Banerjee's approach has been an attempt to escape the 'one optimization at a time' paradigm but has only been applied to nested loops for the loop-interchanging, loop-reversal and loop-skewing transformations. Further work using unimodular transformations with attention to optimizing for locality of reference within tightly nested loops has been undertaken within the SUIF compiler project at Stanford University [10]. A more formal, axiomatic approach to ordering of optimizations is presented by Whitfield and Soffa [8], and recently a genetic programming approach to automatic parallelization was demonstrated by Walsh and Ryan [7].

Most of these approaches have been restricted in their ability to deal with imperfectly nested loops and complicated or transitive dependencies. Much of this work however, is seen as complimentary to the development of genetic compiling since it is simply a way of scheduling optimizations [9], genetic compiling itself does not define any new optimizations.

3 Summary and Conclusions

A high-level diagram of the genetic compiler process is presented in Fig.4. It follows the regular GA procedure, adapted to make it suitable for optimizing the automatic parallelization function. The final output of the genetic search will be a 'fittest' sequence of optimizations and transformations, which are then applied to the original program to produce a new optimized source program, ready for compilation and execution on the target machine.

The stages of the process are numbered 1-7 in the diagram. A description of each stage follows:

1 The input to the compiler is a sequential program, *prog.seq*, written in a high-level language, such as C, or Fortran. The program is firstly translated into an intermediate form called three-address code (*TAC*), producing *prog.tac*. This representation is more amenable to optimization [1].

2 The Reduction Transformation is applied to *prog.tac* producing the skeletal form, *prog.skel*, of the TAC code.

3 A population of parallelizing compilers exists, each of which is able to parallelize *prog.skel* in a different way (as specified in the chromosome encodings). Given a population of size N, N copies of *prog.skel* are created, each member of the population then parallelizes its' copy of *prog.skel* in its' own way, thus producing parallelized skeletal programs *skel1.par*, *skel2.par*, ..., *skelN.par*.

4 Each parallelizing compiler is then evaluated for its' fitness by executing the parallel code it produced. Factors such as speed of execution and memory usage are taken into account to determine how well a sequence of optimizations has performed. Each member of the population is assigned a figure indicating how well it performed.

5 Having evaluated all the members of the population, we have a choice of whether to continue the optimization search (go to stage **6**), or to terminate the process by simply applying the best sequence of optimizations we have found so far, i.e. stage **7**.

6 We continue the optimization search by creating a new population of compilers. This is done by applying the usual genetic operators (crossover, mutation, and reproduction) at the chromosome level to selected pairs of compilers taken from the population. By careful selection, we can propagate the attributes of the fittest members of the population into the next generation while eliminating undesirable elements. The mutation operator in particular, will create novel sequences of optimizations which will help raise the fitness level of the overall population.

7 A genetic optimization process (such as this) is usually terminated either because of time constraints, or else because there has been no improvement in the level of the fittest member of the population for the last few generations. Once terminated, we simply take the best sequence of optimizations we have found so far and apply them to *prog.tac* to produce an optimally restructured parallel program.

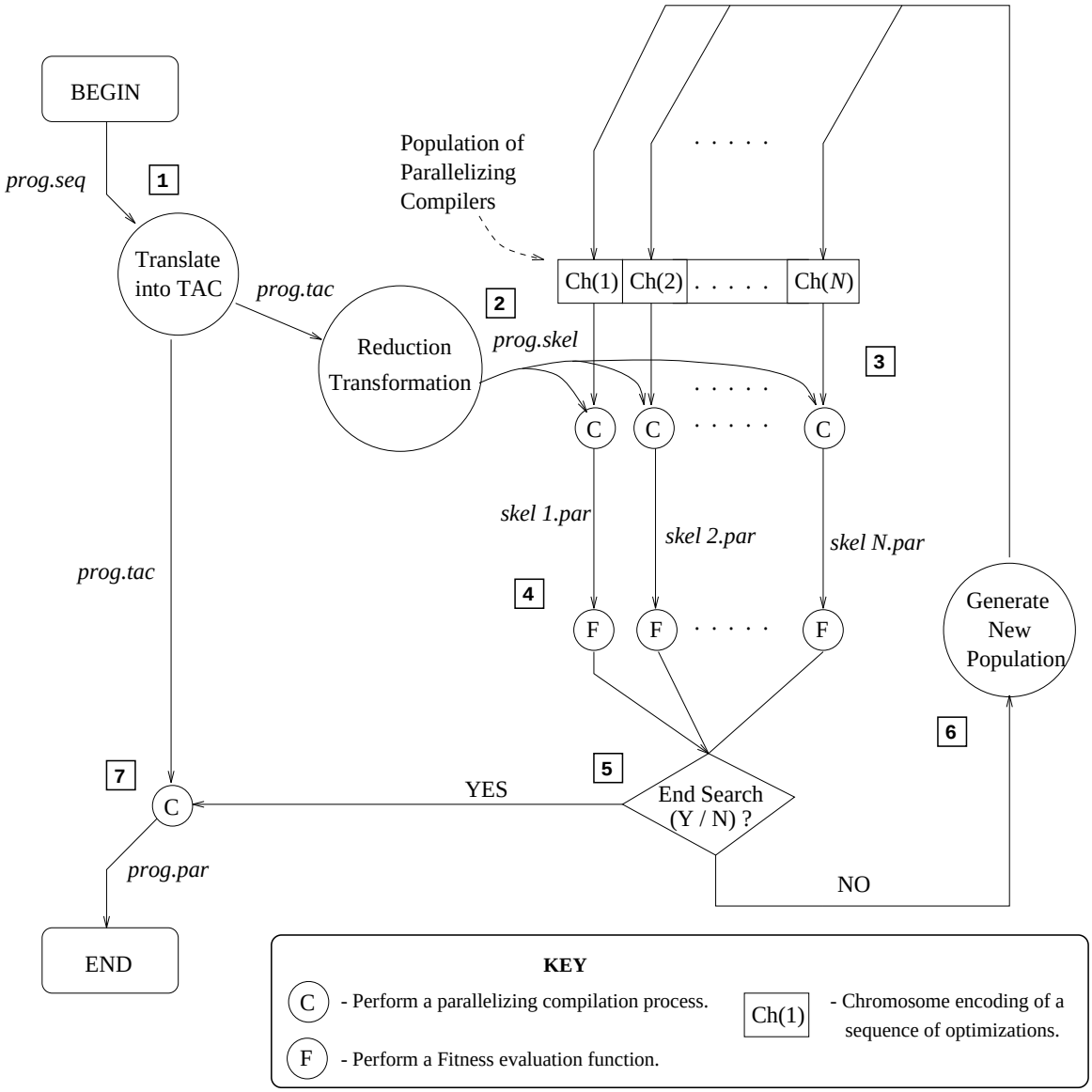

Figure 4: High-Level Overview of the Genetic Compiling Process.

A brief comparison of the traditional static compilation and new genetic compiling approaches may now be undertaken.

- A parallelizing compiler which employs purely static analysis techniques only has access to the information explicitly stated in the code - this may not always specify how the program performs at run-time and the compiler must use heuristics to guess the impact of some optimizations and transformations. A genetic compiler however, because the program is executed numerous times during the compilation process, also has access to all run-time information - most importantly memory allocation, profiling information, pointer/aliasing usage, etc. These will help create more accurate estimations of transformation effectiveness.

- A static compiler can only apply optimizations and transformations in the way it was programmed, usually only one way. A genetic compiler on the other hand can apply optimizations and transformations in several ways and compare the performance of the code produced by each to see which method is best.

- One limitation of static analysis techniques is that it is not always possible to determine the exact data dependence distance of a loop at compile-time. This of course is no problem to our genetic search since we execute the program numerous times during the compilation process are thus able to determine the dependence distance precisely.
- Another situation where static compilers have been thus far restricted in their ability to generate efficient parallel code is where the iteration space of a loop is not known at compile-time, as with `while` and `do` loops with conditional exits. The run-time techniques used in genetic compiling are suitable for analyzing such loops.
- A static compiler is smaller in size and requires less computing resources to perform its function. A genetic compiler requires greater computational power to restructure the code, however its results are likely to be more impressive.
- The genetic compiler model is extensible in that new transformations and optimizations can be easily added and their effectiveness, alone, and in combination with other optimizations/transformations can be quickly and easily established.

The technique of genetic compilation represents a new and powerful design for compilers and is particularly suited to the purpose of automatically parallelizing sequential programs. It builds on the existing substantial body of work (summarized in [12]) while opening up a promising new area of research.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (2nd Edition), (1986).

[2] Utpal Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Press, (1988).

[3] Utpal Banerjee, *Unimodular Transformations of Double Loops*, in Advances in Languages and Compilers for Parallel Processing, (Ed. A. Nicolau), Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge: Mass, Chap. 10, (1991).

[4] Rudolf Eigenmann, Jay Hoeflinger, Z. Li, & D.A. Padua, *Experience in Automatic Parallelization of Four PERFECT Benchmark Programs*, in Proc. 4th International Workshop on Languages and Compilers for Parallel Computing, CSRD-TR-1193, (1991).

[5] Jeanne Ferrante, Karl, J, Ottenstein & Joe D. Warren, *The Program Dependence Graph and its' Use in Optimization*, ACM-TOPLAS, 9, 3, July (1987).

[6] William Pugh, *Uniform Techniques for Loop Optimization*, in Proc. of ACM Conf. on Supercomputing, Cologne, Germany, June, (1991).

[7] Paul Walsh & Conor Ryan, *Automatic Conversion of Programs from Serial to Parallel Using Genetic Programming - The Paragen System*, in Proceedings of ParCo 95 (Parallel Computing 1995), Universiteit Gent, Belgium, 19 - 22 September, (1995).

[8] Debbie Whitfield and Mary Lou Soffa, *An Approach to Ordering Optimizing Transformations*, "Proceedings of 2nd ACM Symposium on Principles and Practice of Parallel Programming (ACM-PPOPP), Seattle, Washington, March 14-16, 1990", pgs 137-147, Mar (1990).

[9] Kenneth P. Williams and Shirley A. Williams, *Genetic Compilers: A New Technique for Automatic Parallelisation*, 2nd European School of Parallel Processing Environments (ESPPE '96), L'Alpe d'Huez, France, April 1-5, (1996).

[10] Michael E. Wolf and Monica S. Lam, *A Loop Transformation Theory and an Algorithm to Maximise Potential Parallelism*, IEEE Trans. on Parallel and Distributed Systems (IEEE-PADS) pgs 1-38, Oct (1991).

[11] Michael J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, (1989).

[12] Michael J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, (1996).