Towards Defining A Fitness Function for Genetic Compiling

Kenneth P. Williams and Shirley A. Williams
Department of Computer Science,
University of Reading,
PO Box 225, Whiteknights, Reading, UK, RG6 6AY
{K.P.Williams,Shirley.Williams}@reading.ac.uk
http://www.cs.reading.ac.uk/cs/people/{kpw,saw}
and http://www.cs.reading.ac.uk/cs/research/pedal/index.html

January 10, 1996

1 Abstract

In the last three decades a number of compiler transformations for optimising sequential programs for execution on vector or parallel architectures have been implemented. Optimisations for high performance architectures maximise parallelism and memory locality with transformations based on extensive control and data dependency analysis - with particular emphasis on loop-transformations. Current optimising compilers however lack an organising framework that allows the direct calculation of the optimal sequence of loop transformations to be applied.

We present a fast and efficient technique using genetic algorithms to optimise the compilation function. The technique involves translating the source program into a scaled-down, 'skeletal' form to which loop transformations can be applied as part of a genetic search. The resulting code can then be executed to quickly evaluate the 'fitness' of the sequence of loop-transformations applied. Eventually, the fittest sequence of loop-transformations found will then be applied to the original source program to produce the optimised executable code [KWSW96]. In this paper we examine issues involved in defining a fitness function for use in a genetic compiler.

2 Parallelising Compilers and Genetic Compilation

Numerous studies evaluating the effectiveness of parallelising compilers have pointed out the limitations of current techniques. The criteria against which parallelising compilers are evaluated varies, the most common approaches are to test their effectiveness against a set of programs restructured by other parallelising compilers, or manually by a programmer. The results are not encouraging. Some of the major problems identified include : (i) loop transformation techniques are not applied to the most important 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 incorrectly decide that a technique is not worth applying (iv) the inappropriate application of some techniques may have negative effects on performance (v) timeouts in the compilers' search for the optimal parallelisation. We will show how these problems are overcome by genetic compilation.

Genetic algorithms have proved suitable for solving problems where direct analysis is ineffective or impossible. Our research initially set out to tackle the interference problem. A sequence of loop transformations may be encoded in genetic form in the following manner: given a set of loop transformations T, we can assign a letter to each transformation in T thus:

$$T = \{A,B,C, ...\}$$

where: A = *loop-skewing*, B = *strip-mining*, C = *loop-interchange*, etc. Next, we encode a sequence of transformations into a character string, *Ch*, called a *chromosome*:

$$Ch = \{A,B,Z,A, ...\}$$

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

A genetic search for the ordering of loop transformations which will produce the parallelised 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) can be initiated in the following manner. First, we create an initial *population* of parallelising compilers, each of which has its own chromosome encoding which represents the order of loop transformations it applies. Next, we give one copy of our original program S to each individual in our population and allow them to apply their sequence of transformations. The resulting output from each individual is a parallelised version of S. We then execute each of these programs on the target machine and record their execution times. Next, we create a new population by performing genetic operations such as *crossover*, fitness proportionate *reproduction* and *mutation* on the individuals whose fitness has just been measured. Finally we discard the old population and iterate using the new population. Although the method specified above will produce a parallelised version of a sequential program, three problems become apparent:

1. If the only way we can test the effectiveness of the parallelised 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 once, or only a few times) then the genetic compilation process may be too expensive (in terms of machine time, etc).

2. The whole process will be slow.

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.

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. It will be noted that control and data dependency information is of vital importance in the automatic parallelisation 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 control and data dependencies remain the same.

If we are applying a sequence of transformations to loops in a program, we can simply reduce the indices of each loop in order to speed up program execution. This preserves the vital control and data dependency information we need while significantly speeding up the fitness evaluation function (see Fig. 1). Most importantly, even though the skeletal version of the loop executes fewer iterations, *the control and data dependencies for the two loops are identical.* Namely: $S_1\delta^a_{(0)}S_1$, $S_1\delta^t_{(+1)}S_2$ and $S_3\delta^a_{(+1)}S_1$.

```
        for (i=1; i<10000; i++) {          for (i=1; i<10; i++) {
S1:        A[i] = A[i] + Z[i];                A[i] = A[i] + Z[i];
S2:        B[i] = X[i] + A[i-1];              B[i] = X[i] + A[i-1];
S3:        C[i] = X[i] + A[i+1];              C[i] = X[i] + A[i+1];
        }                                  }

       Original loop                          Skeletal loop
```

Figure 1: The original loop and its intermediate 'skeletal' form have identical control and data dependencies.

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 by adjusting their indices. The resulting program is said to be in 'skeletal form'. Further refinements to the reduction function take into account loop-carried dependencies, and non-linear loop dependencies. Because the genetic compiler will have access to run-time memory usage information, this transformation may also be extended to handle dynamic memory allocation. Further scaling-down transformations may be possible.

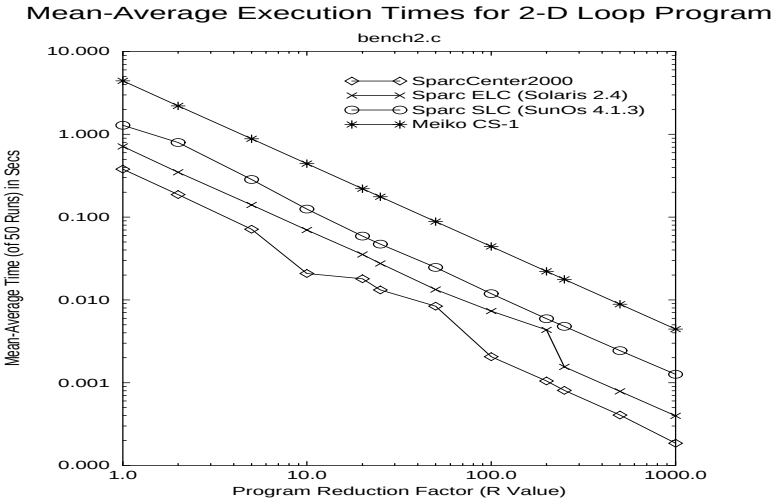

Figure 2: Mean Averages of 50 Runs of 'bench2.c'.

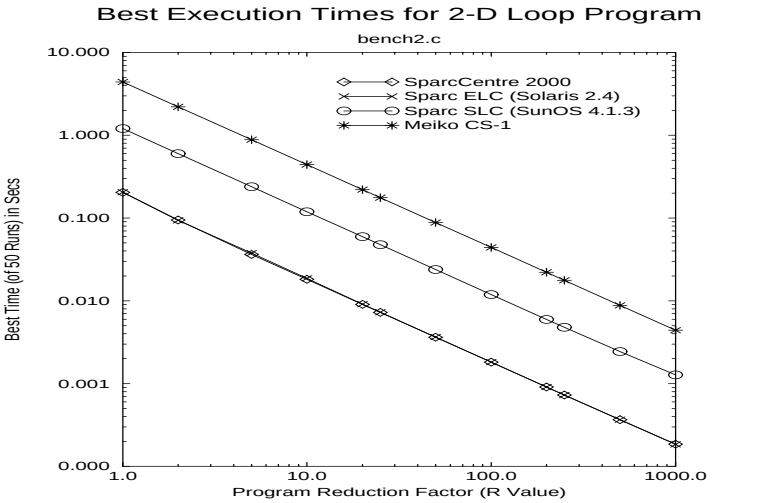

Figure 3: Best Execution times of 50 Runs of 'bench2.c'.

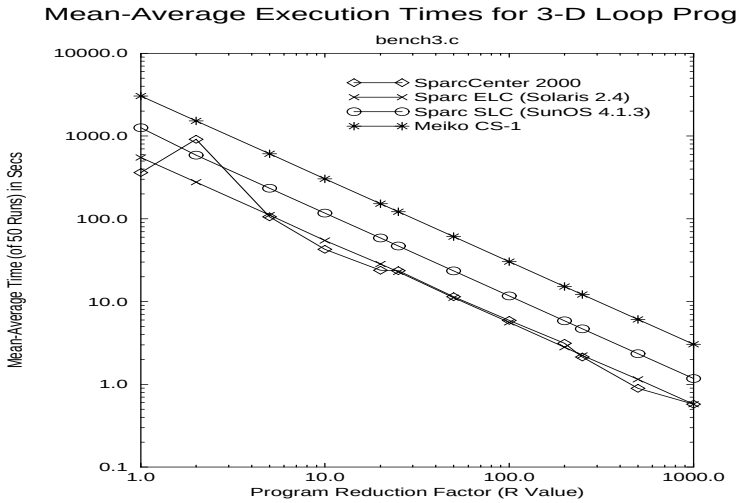

Figure 4: Mean Averages of 50 Runs of 'bench3.c'.

Determining how to apply a sequence of transformations to a program is crucial to the performance of the final optimised code. There are numerous ways in which transformations can be applied. The question then is 'which method is best for applying a sequence of transformations' ? Our proposed solution is to have several *species* of compiler within our population - each of which will apply their transformations in their own way. The final output of the genetic search will be a 'fittest' sequence of optimisations and transformations which will then be applied to the original program. This will then produce an new source program, optimally restructured and ready for compilation and execution on the target machine.

There are two parameters connected with the evaluation of the fitness of a program P, namely: execution time, and efficiency of memory usage. The optimisation of these two parameters produces highly fit programs. A pre-condition of fitness evaluation is that parallelised programs P^n are *semantically equivalent* to the original sequential program S. This is guaranteed by the semantic validity of the individual optimisations and transformations applied. To help define a fitness function $\mathcal{F}$ we created two simple test programs `'bench2.c'` and `'bench3.c'`. The former consisting of a single assignment statement within a doubly-nested (2D) loop, the latter within a triply-nested (3D) loop. Each program was then run on four target machines: a Meiko CS-1; a Sparc ELC; a Sparc SLC; and an 8 processor SparcCenter 2000.

```
#define SIZE 10000

for (i=0; i<SIZE; i++)
  for (j=0; j<SIZE; j++)
    A[i][j] = 0;     ;

   2D - loop
```

```
for (i=0; i<SIZE; i++)
  for (j=0; j<SIZE; j++)
    for (k=0; k<SIZE, k++)
      A[i][j][k] = 0;

   3D - loop
```

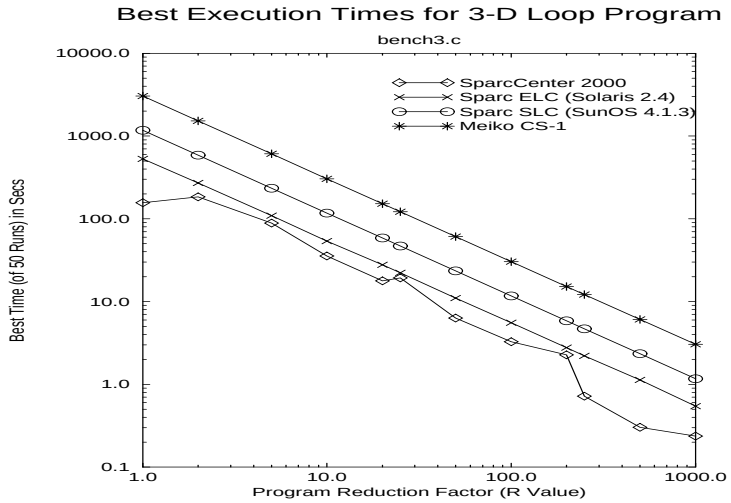

Figure 5: Best Execution times of 50 Runs of 'bench3.c'.

Starting with an initial Reduction factor (R) value of 1 (i.e. the original program) each program was executed 50 times and the Mean Average ($\bar{x}$) and Best execution times were recorded. The R value was then increased to 2, the loop indices amended accordingly, and the program executed 50 times with the results recorded as before. This evaluation process was repeated with R values of 1, 2, 5, 10, 20, 25, 50, 100, 200, 250, 500 and 1000. The primary aim of this was to confirm that linear increases in the program's Reduction factor, R, resulted in linear decreases in the program's execution time. This hypothesis was confirmed by our results (Figs 2, 3, 4 and 5).

A further issue we investigated was whether it was better to use the Mean Average or the Best execution time, of a number of runs (n), in the $\mathcal{F}$ function. In theory it appears neither option has any large advantage over the other. In practice however, determining the Best execution time of a set of runs is significantly more efficient than calculating the Mean Average. This becomes particularly important as the value of n increases. In conclusion then we can say that the fitness function $\mathcal{F}$ of a parallelised program P, can be defined as: $\mathcal{F}$ = Best_Execution_Time from n runs of P.

It will be noted that at this stage, the definition of $\mathcal{F}$ is based only on the *time* factor of program execution. A future definition of $\mathcal{F}$ will also have to take into account the program's efficiency of memory usage. Genetic compiling represents a new and powerful design for compilers and is particularly suited to automatically parallelisation. It builds on the existing substantial body of work (summarised in [Wolfe96]) while opening up a promising new area of research.

References

[KWSW96] 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, *to appear*, (1996).

[Wolfe96] Michael J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, (1996).