An Evaluation of Existing Heuristics for Register Bank Partitioning Using Genetic Algorithms

By

Dineel D. Sule

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Michigan Technological University

2001
This thesis, “An Evaluation of Existing Heuristics for Register Bank Partitioning Using Genetic Algorithms”, is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT of Computer Science

Thesis Advisor       Dr. Philip H. Sweany

Department Chair     Dr. Linda M. Ott

Date
Contents

1 Introduction ................................................................. 3

2 Related Work ............................................................. 8
   2.1 Bottom Up Greedy Approach ...................................... 8
   2.2 Capitaino et al ....................................................... 10
   2.3 Nystrom et al ....................................................... 11
   2.4 Unified Assign and Schedule ..................................... 12
   2.5 Partial Component Clustering .................................... 13
   2.6 University of Dortmund ......................................... 14

3 Genetic Algorithms ....................................................... 16
   3.1 Approach ............................................................ 17
   3.2 Properties .......................................................... 20
   3.3 An Example ........................................................ 23

4 Research on Clustering at MTU ......................................... 27
   4.1 Register Component Graph ....................................... 27
4.2 Saurabh Jang’s Thesis .................................................. 28
4.3 A Greedy Approach to Clustering .................................. 30
4.4 Work on Register Cloning .............................................. 31
4.5 Jason Hiser’s Thesis ..................................................... 33
4.6 The Kernighan-Lin Partitioning approach ......................... 36

5 The Experimental Methodology ........................................ 38
  5.1 Experimental Setup .................................................. 39
     5.1.1 Machine Configurations ..................................... 42
     5.1.2 Copy Models .................................................. 43
     5.1.3 Instruction Latencies ......................................... 44
     5.1.4 Benchmark C Programs ..................................... 45
     5.1.5 Benchmark Fortran Loops ................................... 48
  5.2 Genetic Algorithm based Register Partitioning .................. 48
     5.2.1 The Search Scheme ......................................... 49
     5.2.2 The Package .................................................. 51
     5.2.3 The Implementation ......................................... 56

6 Experimental Results and Analysis ................................. 65
  6.1 Whole Program Compilation ...................................... 66
  6.2 Software Pipelined Fortran Loops ................................ 88

7 Conclusions and Future Work ........................................ 98
List of Tables

3.1 Initial GA Population ......................................................... 23
3.2 New Solutions obtained as a result of Reproduction and Crossover .... 24
3.3 GA Population as a result of worst solution replacement ............... 25
3.4 Final GA Population at the end of the first iteration .................... 26
4.1 Performance Degradation in percentage for the LPK Algorithm .......... 29
4.2 Average II over tested benchmarks ....................................... 32
4.3 Average number of copies inserted ....................................... 32
4.4 AGPS degradation in percentage for Software Pipelined 16-wide VLIW . . 35
5.1 Machine Configurations tested ............................................. 42
6.1 First set of data relating to the performance comparison of the GA and AGPS in terms of percentage degradation seen over ideal schedules .......... 68
6.2 Second set of data relating to the performance comparison of the GA and AGPS in terms of percentage degradation seen over ideal schedules .......... 71
6.3 Third set of data relating to the performance comparison of the GA and AGPS in terms of percentage degradation seen over ideal schedules .......... 72
6.4 Data summary displaying the percentage degradation differences calculated for the three earlier data sets ........................................ 74

6.5 Evaluation of the performance of AGPS utilizing the best results seen using GA. ................................................................. 75

6.6 Comparison of the average percentage degradation in II over the II in the case of the Ideal Architecture ........................................... 96
## List of Figures

3.1 A Simple Genetic Algorithm .............................................. 21

5.1 The Near-Optimal Clustering Genetic Algorithm ..................... 50

5.2 A sample dataflow graph .................................................. 59

5.3 Block Diagram representing the Experimental Methodology followed .... 64

6.1 Comparison of the degradation over ideal schedules seen in the case of the Embedded Copy model .............................................. 78

6.2 Comparison of the degradation over ideal schedules seen in the case of the Copy Unit model .................................................. 79

6.3 Histogram showing the percentage degradation difference seen in the case of the Embedded Copy and Copy unit models ............................. 81

6.4 Comparison of the degradation over ideal schedules seen in the cases of the embedded copy and copy unit models when using the GA ............. 85

6.5 Functional Distribution of Degradation in the case of the Embedded copy model .............................................. 87

6.6 Functional Distribution of Degradation in the case of the Copy Unit model ............................. 87
6.7 Distribution of the degradation in II in the case of an embedded copy model, 2 wide with 8 FUs per cluster .................................................. 89

6.8 Distribution of the degradation in II in the case of a copy unit model, 2 wide with 8 FUs per cluster .................................................. 90

6.9 Distribution of the degradation in II in the case of an embedded copy model, 4 wide with 4 FUs per cluster .................................................. 91

6.10 Distribution of the degradation in II in the case of a copy unit model, 4 wide with 4 FUs per cluster .................................................. 91

6.11 Distribution of the degradation in II in the case of an embedded copy model, 8 wide with 2 FUs per cluster .................................................. 93

6.12 Distribution of the degradation in II in the case of a copy unit model, 8 wide with 2 FUs per cluster .................................................. 93
**ABSTRACT**

The problem of partitioning registers in a clustered VLIW architecture has critical impact on the quality of code generated. As a result, several clustering schemes to allocate registers to the various register banks have come into existence over the past few years.

However, since the clustering problem is a NP-complete problem all schemes proposed to date are heuristics or approximations to the solution. Furthermore, all of these schemes report their results in comparison to the ideal but un-practical, non-partitioned VLIW machine. As a result, a real world estimate of the status of clustering as it exists today has never been generated.

Presented in this document, is an innovative scheme leveraging the capabilities of an off-line, stochastic, modern heuristic: namely a Genetic Algorithm, to help generate near-optimal solutions to the clustering problem. The solutions generated are then compared to the solutions generated by means of an existing state-of-the-art heuristic developed for partitioning registers. In this way an attempt was made to evaluate the state of clustering as it exists today. Also presented is some very interesting data dealing with the effects of copy architecture variation on the quality of the final code generated and the ability of a genetic algorithm to cope with a highly volatile solution search space. The results obtained show that while the targeted heuristic, Hiser’s Advanced Greedy Partitioning Scheme[4, 5, 6], performs adequately when the available parallelism in the program is high, it does show some promise for improvement when dealing with code where the level of parallelism is inherently low.
Chapter 1

Introduction

The increasing need for computing power has been a dominant driving force in the field of computer science for some time now. *Instruction-level parallel computing* is one of the realms which came into existence to meet this ever-increasing demand for computational speed. It is also a domain which has seen an enormous increase in popularity in recent years.

The work described here is an attempt to evaluate the performance of an existing *state-of-the-art* technique developed for partitioning registers when generating code for instruction-level parallel architectures with partitioned register banks [4, 5, 6]. Register partitioning has a critical impact on the quality of code generated for instruction-level parallel architectures. As shall be seen the contribution of the work described here addresses the issue of how close one of today’s better partitioning techniques [4, 5, 6] is to generating a possibly optimal partition: a major issue for instruction-level parallel computing.

*Instruction-level parallelism* (ILP) refers to the ability of an architecture to execute multiple, independent instructions in parallel at an intra-processor level. Traditionally, archi-
 Architectures were designed to execute only one instruction at a time. When compared to these older architectures, ILP provides a very attractive option to dramatically increase computing speed.

Two kinds of ILP architectures have been developed to take advantage of parallelism at an intra-processor level. These architectures are called *superscalar* and *very long instruction word* (VLIW) architectures.

- **Superscalar Architecture**: These are architectures which are designed to contain multiple functional units (FU) capable of executing in parallel. The distinguishing feature of this kind of ILP architecture is that the binding of instructions to functional units is done at runtime and is decided using on-chip hardware. Compaq’s Alpha 21164 is one example of a popular superscalar processor available today.

- **VLIW Architecture**: This category of ILP architectures again refers to architectures which contain multiple functional units capable of simultaneous execution. However this architecture lacks the capability to decide which instructions to execute in parallel. It instead relies on the compiler to tell it which instructions it should execute in parallel. Texas Instruments TMS320C6x architecture is an example of a VLIW architecture in use today.

While both of the above defined architectures attempt to effectively exploit parallelism at an intra-processor level they each have their own set of advantages and disadvantages. The ability of superscalar architectures to bind instructions and FUs at runtime comes at the cost of expensive hardware and loss of scalability. Comparatively, the low cost yet scalable VLIW architecture has the disadvantage of requiring a highly sophisticated compiler capable
of binding instructions to FUs statically at compile time when little runtime dependence information is available.

Another problem faced by ILP architectures is that an increase in the parallelism of the processor by means of an increase in the FU count has a detrimental effect on the overall clock speed of the processor [11]. This is because with an increase in the number of FUs there is an increase in the number of read and write ports accessing the register file. Consider the example of an ILP architecture that has eight FUs capable of executing simultaneously. Each FU would require at least three ports (two read and one write) to execute efficiently, giving a total requirement of 24 register ports.

A solution to this problem of decreasing clock cycle speed with an increase in the intra-processor parallelism is to divide the single monolithic register file (RF) into multiple register files, each one being local to only a subset of the available FUs. Such an architecture is known as a clustered architecture where a cluster refers to a RF and its associated FUs. Thus, considering the earlier example, instead of having a single register file with 24 ports it would be possible to design a clustered architecture made up of four clusters, each cluster requiring only 6 ports for register access.

While the above solution provides a window of opportunity to build a fast, practical ILP processor, it has a side effect. FUs in clustered architectures can only execute instructions whose operands are available in the register file local to the FU. Thus, if an instruction is scheduled to execute on a FU and even one of its operands is not present in the local RF, that operand has to be copied to the local RF before the instruction can commence execution. These copy instructions not only increase the number of instructions that need
to be executed but they also tend to, at times, serialize the code. Allocating instructions and operands to the correct clusters so that these copies can be minimized thus becomes an issue of paramount importance. This is what is referred to as the register partitioning or clustering problem. It is this register partitioning problem that the work being described here had focused on.

On a superscalar architecture the allocation of instructions to the appropriate clusters so as to minimize the number of inter-bank copies is done using hardware. In the case of a VLIW architecture even this register partitioning problem has to be resolved by the compiler at compile time, a non-trivial task.

Many schemes have been proposed to generate good quality code for VLIW architectures with partitioned register banks \([2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 17]\). The register partitioning problem discussed above falls into the list of critical problems that need to be resolved to meet this objective. Unfortunately, this problem of allocating instructions and operands to the appropriate clusters as described earlier falls into the category of NP-complete problems. As a result, an exhaustive search for an optimal solution to the problem becomes unadvisable.

All the schemes which have to date been proposed for generating code for clustered VLIW architectures rely on heuristics to try to search for acceptable solutions to the register partitioning problem. Two important factors to consider when heuristics are involved in the search for a solution are the shape of the search space and the problem’s natural categorization.

In the case of the clustering problem the search space involved is non-linear by nature
and the problem being considered is a *minimization* problem. These factors increase the chance that the heuristic used might get caught in a localized area during its search for a solution, thereby resulting in unacceptably bad solutions. Intuitively then, a heuristic capable of escaping from such a localized search area might be capable of providing a better, wider and possibly optimal solution set. *Genetic algorithms* (GA) is one such category of heuristics which is capable of escaping from a localized solution space. The work described here attempts to evaluate the effectiveness of an existing, acceptably good, greedy register partitioning scheme [4, 6, 5] used for generating code for VLIW architectures with partitioned register banks by comparing it to a register partitioning scheme that uses a modern stochastic technique, namely a Genetic Algorithm, to generate partitions which could represent the *reasonably close* to optimal solution set.
Chapter 2

Related Work

As described earlier, the clustering problem has critical impact on the quality of code generated. It is therefore not surprising that this problem has been the focus of several studies in recent years. Since the work described here attempts to evaluate one of the available techniques for register partitioning [4, 6, 5] it is important to understand some of the other similar techniques which are now available and how they compare in relation to the one which is the focus of this study. This section takes a look at some of these other clustering techniques which have come into existence over the past few years.

2.1 Bottom Up Greedy Approach

Ellis was one of the first to deal with the clustering problem [3]. He implemented a solution to the problem in his compiler, called Bulldog, which was targeted towards VLIW architectures.

His approach for generating code was quite simple. The compiler initially selects a trace or a set of basic blocks which are to be scheduled together. The trace is next converted into
a directed acyclic graph (DAG) to represent the instruction dependences. Each operation in the DAG is then assigned to some candidate functional units. Finally scheduling and register assignment is performed. When this register assignment is being performed the copy operations, if and when needed, are added to the DAG on the fly. Also if an operation is due to be scheduled but registers or FUs are not available the operation is simply delayed until an appropriate register or FU becomes available. Ellis called this approach bottom-up greedy (BUG).

While BUG was an approach to solving the clustering problem it had several failings. The most critical one was the problem of copy operations being added on the fly. Copy operations are a side effect of code generation. Where possible, they should be hidden amongst idle FUs. The approach that Bulldog takes, of inserting copies on the fly, not only leads to schedule length increase but it also causes code serialization, something which should never be allowed to happen on VLIW architectures. Another problem is that the FU assignment is done in a greedy manner. This means that even if not needed, initial instructions will end up being spread out over FUs causing excessive copy requirements. The third problem is the problem of Bulldog delaying operations until an appropriate FU is found. Conditions may arise where this may never happen and spilling may be required to break the deadlock. Since this is never done, the BUG algorithm can fail causing the compiler to crash. Another limitation of BUG is that it is implemented for traces which represent straight line code. A more global approach should yield better results.
2.2 Capitanio et al

An interesting approach was developed at the University of California - Irvine to solve the register partitioning problem [11]. Their approach attempts to generate code for partitioned register bank architectures by improvising from the code generated for an ideal VLIW architecture having only a single register file.

More specifically, their algorithm first accepts ideal VLIW code built by their compiler. It then converts the code into a directed cyclic graph (DCG) representing the dependences and control flow. It is this DCG that is then partitioned to make it suitable for partitioned register bank architectures. The partitioning is done by means of a stochastic algorithm similar to simulated annealing. The partitioning algorithm tries to ensure that the number of copies required across register banks is minimal. After the DCG is partitioned, required inter bank copies are added where needed and the code is then compacted to reduce the empty slots available in parallel with the copy instructions. An important thing to note here is that the ideal VLIW code described above is not the optimal VLIW code. The problem of scheduling is itself NP-complete so generating an optimal schedule is improbable, if not impossible. Instead the ideal VLIW code refers to the code their scheduler builds for an ideal, unpartitioned VLIW machine.

Their approach to partitioning does have some drawbacks. First, it is limited to only straight line loops. Second, their approach gets caught in the classical phasing problem faced by VLIW compilers: should partitioning precede scheduling or vice versa. Their approach of scheduling first might yield poor results since partitioning could cause changes in some of the conditions which existed when scheduling was done, causing the profitability analysis
done during scheduling to fail. By forcing their final scheduler, which compacts code, to work locally they add constraints which tend to limit the effectiveness of their algorithm.

2.3 Nystrom et al

Nystrom and Eichenberger present a technique for assigning clusters to instructions in loops such that modulo scheduling can be effectively done on those loops [12]. A big advantage of their technique is that since it is geared towards improving a loop’s performance through modulo scheduling, increases in schedule size because of clustering can be hidden by the performance improvement gained through the process of software pipelining of the loop.

With respect to clustering, their approach is to assign nodes to clusters in a manner whereby the impact on the initiation interval (II) of the loop is minimal. They also allocate the largest strongly connected components (SCC) of the graph to clusters first so that these SCCs do not get spread out across register banks due to the lack of resources. In addition to this, they try to minimize the number of copies required by assigning data dependent nodes to the same cluster. The logic behind their approach is that even if the schedule length is longer due to the serialization introduced by their technique, when the II is kept minimal the loop’s overall throughput would definitely increase when multiple iterations of the loop execute in parallel due to software pipelining.

Their approach is actually quite good. The only drawback is that when the natural II of the loop is itself large, then due to code serialization the loops overall throughput will worsen degrading the final quality of code generated due to wasted/idle FUs.
2.4 Unified Assign and Schedule

Another scheme to solve the clustering problem was developed at North Carolina State University by Ozer et al [13]. The argument they make to promote their algorithm, known as the Unified Assign and Schedule (UAS) algorithm, is that separating the scheduling and partitioning phases in a compiler leads to a degradation in the final quality of code generated. So, they approach the problem by doing clustering and scheduling hand in hand within a single phase.

In their scheme, unscheduled, data ready operations are picked and an attempt is made to try to schedule those operations. The decision of cluster allocation is taken by checking clusters for availability and is based on a priority function. In case an operation cannot be scheduled another data ready operation is selected and the operation whose scheduling attempt failed is simply delayed until later. In case a cluster is found for scheduling it is checked if any copies are required and if they can be scheduled in prior instruction slots. If it is possible, the operation is scheduled with its requisite copies else another cluster is selected for the operation and the entire process is repeated.

UAS has its own set of drawbacks. Like BUG, it adds copy instructions on the fly, that is when needed. This means that the scheduler is not taking full advantage of the fact that it can find out what copies would finally be required and schedule accordingly. In addition, its worst case time complexity is $O(n^3)$. This means that the time for partitioning and scheduling will drastically degrade as the number of operations to schedule increases. Also the priority function deciding which cluster gets evaluated first for operation allocation plays a critical part in deciding the final quality of code generated. This means that a
priority function unsuited for the kind of code being scheduled will degrade the algorithm’s performance.

2.5 Partial Component Clustering

The partial component clustering (PCC) technique was developed at HP-Labs for dealing with the problem of generating code for partitioned register bank architectures. The scheme [2] attempts to do a better job of clustering by trying to push copies off the critical path in the code.

The PCC method starts off by generating a DAG representing the operations being scheduled and partitioned. It then traverses the longest path in the DAG which represents the critical path. It iterates this process generating several sets of sub-graphs each representing a sub-path in the DAG. PCC then schedules the code for each of the sub-graphs in the order of decreasing number of nodes in the sub-graph. The logic here relates to the fact that the critical path is represented by the longest path in the DAG. By scheduling the sub-graphs in decreasing order of the number of nodes, most of the copies will be placed in sub-graphs which are off the critical path thus generating a much better schedule. The initial cluster assignment is done by attempting to balance the number of operations across the clusters with a secondary priority given to minimize the number of copies. After the initial assignment is done, the scheme iterates over the entire DAG applying a list scheduler locally to see if the schedule length or copies required can be further reduced.

PCC’s biggest strength is its ability to leave the critical path of execution comparatively free of performance degradation due to resource unavailability. However in those cases where
the DAG is comparatively unbalanced in size some of the initial decisions made by the algorithm might be wrong leading to worse final performance. Also in the case where the categories of operations present in the partial components being scheduled are themselves not balanced in number, incorrect decisions could be made regarding the assignment of clusters.

2.6 University of Dortmund

Rainer Leupers developed a technique [10] for generating code for clustered VLIW architectures. His technique is interesting in that it is stochastic by nature and works by attempting to maximize the utilization of free FUs for copying purpose.

His scheme iterates between partitioning and scheduling trying to find the best balance between the two. It starts off by randomly allocating nodes in the data flow graph (DFG) to clusters. This random partitioning provides a starting point for the algorithm. Once this random clustering is done, the algorithm schedules the code generated and evaluates it for quality and cost involved. It then repeats the process by switching a subset of the nodes present in the DFG amongst clusters and re-evaluating the new schedule obtained and so on. At each step, it is evaluated if the new schedule is better than the old one and if it is then it is retained; otherwise the old cluster assignment is maintained. Also, with a certain probability worse solutions are accepted so as to help in escaping from a local optima. This probability of accepting worse solutions decreases over time. The stochastic technique he uses is simulated annealing.

Leupers's technique's biggest drawback is that like most stochastic schemes it fails to guarantee that it will ever get the best solution. This is of course compounded by the fact
that it starts off with an absolutely random initial partition which means that it can never guarantee a worst case lower bound on the quality of code generated. Another drawback is the algorithm’s time complexity. Because of the need to reschedule every time for the purpose of calculating the cost of partitioning, the amount of time needed by the algorithm to execute is extremely large. In addition to this, the scheduling algorithm he uses is list scheduling which is local by nature and a global scheduling technique might intuitively give better results. His scheme however has an advantage most heuristics do not: the ability to escape from a local optima.

As can be seen, the clustering problem has been the target of intensive research for a long time now. Several very interesting schemes have been proposed over the past few years and each has its own strengths and weaknesses. However, all of these approaches are united in the goal that they try to generate the best possible results in an environment that is extremely volatile and whose structure cannot be predicted. Overall, they are all just approximations to the best solutions that might ever be obtained.
Chapter 3

Genetic Algorithms

Heuristics are approximations made to reach a solution when the problem being investigated is of non-polynomial time complexity. Since a heuristic is only an approximation, the quality of the final solution obtained is very often driven by the shape of the search space involved. When the search space involved is non-linear the chance that the heuristic will get caught in a local optima and the final result obtained will be sub-optimal is very high. New categories of search techniques [16] have evolved in recent years which have the capability of escaping from such local optima thus generating potentially better solutions when compared to traditional search techniques. This category of heuristics is stochastic by nature. However, it is this property of randomness that gives these heuristics the requisite strength to escape from local optima and generate a better solution set. Simulated annealing, tabu search and genetic algorithms (GA) are examples of such modern heuristics. In this chapter an overview of one of these modern heuristic search techniques, namely Genetic Algorithms, is presented.
3.1 Approach

Genetic algorithms (GAs) are a comparatively newer heuristic approach to problem solving. It is modeled along the lines of Darwin’s theory of genetic evolution. Within a short span of time it has also become an extremely popular heuristic within its category. A GA has a very simple approach to problem solving. It is basically a neighborhood search technique. So like most other neighborhood search schemes it scans through the search space looking for solutions that are better than the one it has currently found. The difference between a GA and traditional neighborhood searches is that unlike traditional approaches which scan only one locality, a GA works with a set of solutions and thus has the capability of exploring multiple regions in the search space simultaneously. In addition to this the region in the search space that is scanned more vigorously is the region which shows a better subset of solutions. All of this is done by means of operations modeled on genetic evolution.

The way a GA proceeds when exploring a search space is that it generates an initial set of solutions which form the base population, essentially representing a starting point for the search. Let \( S = \{s_1, s_2, s_3, ..., s_n\} \) represent the set of solutions a GA initially generates. The set \( S \) is called the population on which the GA operates. As can be noted, \( S \) consists of points in the search trajectory and can be extremely diverse if needed. Each solution present in the population is called a chromosome and is generally represented by a string encoding the information pertaining to the position of the solution in the search space and its degree of optimality.

Once a population has been generated, the GA iteratively applies its basic operators, namely crossover, reproduction and mutation, to the population thereby allowing it to scan
through the search space looking for better solutions. It is these genetic operators that
determine the GA’s ability to converge to a good solution within a limited time.

**Crossover:** Crossover is the operator which helps the GA generate a new solution. For
this reason it is also the operator that determines, to a large extent, the next point that
will be looked at in the search space during the scanning process. Crossover can best be
explained by an example as follows. Let

\[
s_1 = \{a_1, a_2, a_3, \ldots, a_n\} \quad s_2 = \{b_1, b_2, b_3, \ldots, b_n\}
\]

represent two chromosomes in the population. Crossover operates by selecting points within
the chromosomes at which the crossover would occur and possibly switching the contents of
the two chromosomes between the crossover points. For example, if in the above chromo-
somes points two and three were selected for crossover, after applying the operator two new
solutions representing two possibly new points in the search space would be generated.

\[
s_3 = \{a_1, b_2, b_3, \ldots, a_n\} \quad s_4 = \{b_1, a_2, a_3, \ldots, b_n\}
\]

There are several other ways in which crossover can operate [16]. However, in essence what
it tries to do is generate new *candidate* solutions which can be evaluated to see if a better
solution has been obtained.

**Reproduction:** Reproduction is the operator which selects the chromosomes from the
population for Crossover to operate on. It is also the operator that decides which chromo-
somes continue to stay on in the population and which chromosomes get replaced. Thus,
reproduction is the operator which decides to a large extent the major direction in which
the GA proceeds when exploring the search space. Reproduction, essentially, is the operator that builds the next generation of solutions for the GA to examine. The decisions regarding which chromosomes stay on in the population and which chromosomes are used to generate newer solutions are guided by a measure known as the *fitness function*. The fitness function basically evaluates to a value that represents the quality and cost of the chromosome being analyzed. It helps in ensuring that only the best of the solutions continue to stay on in the population and that the neighborhood of the best solutions is explored more vigorously. The way this process works is that when candidates are selected for crossover, they are selected with a certain probability. Chromosomes with higher fitness are assigned a higher probability and so have a higher chance of being selected for crossover. In addition to this, when a new generation is being constructed, chromosomes with the least fitness are replaced with a higher probability. The reproduction operator thus generally follows the rule, *survival of the fittest*, when building a new generation for the GA to examine.

**Mutation:** While crossover and reproduction build newer population sets for the GA to explore they are designed in ways such that the algorithm finally converges to a small area within the search space. Mutation is the operator that brings about the necessary diversity in a localized population allowing the GA to possibly escape from local optima. It is the operator that gives a GA the ability to explore regions in the search space that the GA would normally have never been able to reach. The way mutation generally works is that with a certain probability parts of a chromosome are randomly changed. This alters the properties of the chromosome and the chromosome then represents a new, stochastically determined, solution in the search space. For example if mutation were to decide that in a chromosome
represented as $s_1 = \{a_1, a_2, a_3, \ldots, a_n\}$ position three was to be randomly changed to $c_1$ the new representation of the chromosome would be $s_1 = \{a_1, a_2, c_1, \ldots, a_n\}$. This new representation has a completely new, possibly better fitness and represents a totally different position in the search space. Mutation is generally applied with a very low probability so that the diversity involved is kept under control and there is a convergence to some extent to a more promising region within the search space.

The three operators, crossover, reproduction and mutation represent the basic tools that the GA uses to explore the search space for a solution. By iteratively applying a combination of all of these operators to the population, the GA is able to simultaneously scan diverse regions in the search space while at the same time being able to explore good solutions locality in detail where needed. In this way it generates a fairly good solution to the problem at hand. A simple algorithm representing a GA is shown in Figure 3.1.

Another interesting thing to note in the case of GAs is that the generation count is one of the driver functions deciding the quality of the final solution. In simpler words, the longer a GA is allowed to operate, the better is the quality of the solution obtained. This is because as the generation count increases the GA is able to scan through larger portions of the search space thus being able to locate better solutions if and when they exist.

### 3.2 Properties

Compared to most heuristics available today, GAs have several attractive properties which account for their increasing popularity.
Algorithm GeneticAl()
Input: Problem space
Output: Solution
Start
   Initialize Generation Count;
   population = BuildInitialPopulation();

   While(Generation Count has not been reached) do
      // Apply Crossover to build the new population
      newpopulation = Reproduction(population);
      // Evaluate Fitness of both populations
      population = BuildNextGeneration(newpopulation, population);
      Mutate(population);
      Increment Generation Count
   enddo

   Solution = EvaluateFinalPopulation(population);
End

Figure 3.1: A Simple Genetic Algorithm
• The most important property of a GA is its ability to scan non-linear search areas with ease. The design of a GA is such that it can generate good results irrespective of the shape of the search area involved. This means that as long as the basic operators of a GA are properly built it can traverse a search space of any shape without facing any problems.

• The mutation operator gives a GA the capability to escape from a region within the search space if and when needed. This means that the probability of a GA getting stuck within a particular locality in the search space is negligible.

• Unlike most other heuristics, a GA is a parallel search technique. By operating on a set of solutions instead of a single solution, a GA is able to cover diverse regions in the search space simultaneously. This is a property that almost no other heuristic has.

• Inspite of its capability to reach over vast regions of the search space, a GAs ability to scan promising neighborhoods more vigorously when compared to less promising regions of the search space is not diminished. In simpler words, a GA does not lose its quality of basically being a neighborhood search technique.

A GA, however, does have some limitations. Its biggest drawback is that it often requires more time when compared to traditional heuristics to generate good quality solutions. In addition to this, it has to be kept in mind that a GA is finally just a heuristic. So, like other heuristics, the solution obtained cannot be guaranteed to be optimal. What can however be said is that it is a better heuristic when compared to most other heuristics since it has properties which, given time, make it capable of producing solutions other heuristics would
<table>
<thead>
<tr>
<th>String</th>
<th>Value of $x$</th>
<th>Representation</th>
<th>Fitness Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>string 1</td>
<td>3</td>
<td>0011</td>
<td>-2</td>
</tr>
<tr>
<td>string 2</td>
<td>6</td>
<td>0110</td>
<td>-2</td>
</tr>
<tr>
<td>string 3</td>
<td>11</td>
<td>1011</td>
<td>38</td>
</tr>
<tr>
<td>string 4</td>
<td>1</td>
<td>0001</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.1: Initial GA Population

never be able to generate.

### 3.3 An Example

Perhaps the best way to understand the progression of a *genetic algorithm* based search for a solution is by means of an example. Consider the example of trying to maximize the value of an expression. Let $E$ represent the expression whose value is to be maximized with $x$ ranging from 0 to 15.

\[ E = x^2 - 9x + 16 \]

To solve this problem using a GA the first thing that would be needed is a population for the GA to operate on. Consider a population which is represented by the binary representation of the value of $x$. As an example consider a population of size four. Thus the GA would start off by randomly generating four binary strings representing the values of $x$. Let Table 3.1 represent the initial population generated by the GA.

Once the GA generates the initial population it can then iterate over the population applying the standard genetic operators of crossover, reproduction and mutation. However,
Table 3.2: New Solutions obtained as a result of Reproduction and Crossover

<table>
<thead>
<tr>
<th>Parent 1</th>
<th>Parent 2</th>
<th>Child 1</th>
<th>Fitness of child 1</th>
<th>Child 2</th>
<th>Fitness of child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>0001</td>
<td>1001</td>
<td>16</td>
<td>0011</td>
<td>-2</td>
</tr>
<tr>
<td>1011</td>
<td>0011</td>
<td>1011</td>
<td>38</td>
<td>0011</td>
<td>-2</td>
</tr>
</tbody>
</table>

before it can proceed it has to identify the *fitness function* dictating the direction in which to proceed. In the case of the example here consider a fitness function represented by the value of the expression $E$. Since the problem is a maximization problem the higher the *evaluated value* of $E$, the better will the fitness of the selected candidate be. Column four of the Table 3.1 shows the fitness evaluation of the initial population.

The next thing a GA would have to do would be to apply the reproduction and crossover operators to try and generate new candidate solutions to add to the population. Suppose the GA was to proceed by trying to generate two new solutions every iteration. The way this would happen is that reproduction would first select two pairs of solutions on whom crossover would then operate and generate *possibly* better solutions. In the example let the GA’s reproduction operator select the string pairs *three and four* and *three and one*. These selected string pairs would then be handed over to the crossover operator for *mating*. Let the crossover operator select position three to be crossed over in the string pair three and four and position one for the string pair three and one. Table 3.2 shows the results obtained as a result of applying the reproduction and crossover operators.

The reproduction operator would then take the two best solutions obtained as a result of crossover and replace the two worst solutions present in the population with these newly generated solutions. Table 3.3 shows the new population set generated as a result of this
<table>
<thead>
<tr>
<th>String</th>
<th>Value of $x$</th>
<th>Representation</th>
<th>Fitness Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>string 1</td>
<td>11</td>
<td>1011</td>
<td>38</td>
</tr>
<tr>
<td>string 2</td>
<td>9</td>
<td>1001</td>
<td>16</td>
</tr>
<tr>
<td>string 3</td>
<td>11</td>
<td>1011</td>
<td>38</td>
</tr>
<tr>
<td>string 4</td>
<td>1</td>
<td>0001</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.3: GA Population as a result of worst solution replacement

*survival of the fittest* rule based replacement policy.

As can be seen, the new population on which the GA will operate from this point on is much more fitter than the original population. By iterating over the population using the reproduction and crossover operator in a manner similar to the one described here the GA generates some very good solutions. However, there is a downside to this kind of progression too. As already said the population at this point is much more fitter than the original population. But, as can also be seen from Table 3.3, the population has converged to a small set of values. If only reproduction and crossover are applied this trend will continue unabated till all solutions will represent the value 11 which represents the best solution. This means that the GA might get caught in a local optima and be no better than traditional heuristic searches. It is to avoid this kind of situation that the GA always includes the mutation operator to bring in diversity before the solution set can converge to a single set of values.

Consider the example here. Suppose the mutation operator was to randomly decide to change bit one in string three from one to zero. The resulting string representation would then become 0011 which has a fitness value of -2. Thus a solution with a poor fitness is now accepted into the solution with the logic that sometimes it might actually lead to a better solution. The final population obtained at the end of this first iteration after applying
<table>
<thead>
<tr>
<th>String</th>
<th>Value of $x$</th>
<th>Representation</th>
<th>Fitness Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>string 1</td>
<td>11</td>
<td>1011</td>
<td>38</td>
</tr>
<tr>
<td>string 2</td>
<td>9</td>
<td>1001</td>
<td>16</td>
</tr>
<tr>
<td>string 3</td>
<td>3</td>
<td>0011</td>
<td>-2</td>
</tr>
<tr>
<td>string 4</td>
<td>1</td>
<td>0001</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.4: Final GA Population at the end of the first iteration

reproduction, crossover and mutation is shown in Table 3.4.

Generally, mutation is applied with a very low probability so as to keep diversity under control and prevent the GA from degenerating into a completely random search. The description of the GA operations above are only for a single GA iteration. The solution set obtained is the first generation solution set. These same steps will be repeated over and over again by the GA till a certain number of generations have been evaluated. The larger the number of generations evaluated, the better will the obtained final solution be.

Of course it goes without saying that all of these operator selection mechanisms are stochastic by nature and so while a GA cannot guarantee an optimal solution, it can be expected that given time the population will indeed converge to a near-optimal solution.
Chapter 4

Research on Clustering at MTU

The register partitioning problem has been extensively studied at Michigan Technological University (MTU). What makes the contribution of these studies even more interesting is that all of this work has been done within the context of trying to build a *retargetable* ILP compiler, namely Rocket. As a result of this focus on retargetability, all algorithms developed within the context of this investigation into the clustering problem have been designed in a machine independent manner. The ability to abstract away machine dependent details from the clustering problem has allowed the investigation to proceed smoothly on a wide variety of ILP architectures. This chapter looks at these previous studies relating to the clustering problem conducted at MTU.

4.1 Register Component Graph

At the core of all of the algorithms developed at MTU for looking into the problem of generating code for VLIW architectures with partitioned register banks is a structure known as the
register component graph (RCG). The RCG was developed to represent register interferences so that the algorithms could rely on a single data structure to provide information pertaining to the advantages and disadvantages of assigning a registers to a particular register bank. Basically, a RCG is a graph in which each node represents a symbolic register and each edge represents the interaction between the nodes/registers it connects. Every node and edge in the RCG has weights attached to it. These weights are calculated in a way such that they represent the advantage or disadvantage of assigning the symbolic registers involved to the same or to different register files. Thus from a single graph not only is it possible to glean information pertaining to register interferences but it is also possible to generate the requisite profitability analysis dealing with a particular register’s cluster assignment.

4.2 Saurabh Jang’s Thesis

Jang was the first graduate student at MTU to utilize the concept of a RCG when generating code for VLIW architectures having partitioned register banks. His approach [8] when trying to solve the clustering problem was simple. It consisted of four phases.

- **Phase 1: Build an ideal VLIW schedule.** This is the initialization phase of his clustering algorithm. It builds a schedule for a VLIW architecture having only one register file. It is an important phase because it decides what the RCG will actually look like.

- **Phase 2: Build the RCG.** This phase looks at the ideal schedule, identifies the register interferences and encodes the information correctly within the RCG structure. It is perhaps the most important phase in the algorithm since all the remaining processing
<table>
<thead>
<tr>
<th>Program</th>
<th>Instructions</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>8q</td>
<td>11</td>
<td>40</td>
</tr>
<tr>
<td>bubble</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>gauss</td>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>livermore</td>
<td>6</td>
<td>37</td>
</tr>
<tr>
<td>matrixmult</td>
<td>12</td>
<td>63</td>
</tr>
<tr>
<td>nsieve</td>
<td>11</td>
<td>30</td>
</tr>
<tr>
<td>whetstone</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>10</strong></td>
<td><strong>33</strong></td>
</tr>
</tbody>
</table>

Table 4.1: Performance Degradation in percentage for the LPK Algorithm

is done based on the information captured during this phase.

- **Phase 3: Partition the RCG.** At this stage the RCG is partitioned and the symbolic registers are assigned to appropriate clusters. The algorithm he uses to partition the RCG is the Lee, Park and Kim (LPK) graph partitioning algorithm. The LPK algorithm attempts to minimize the edge-cut generated due to partitioning. Since in the case of the RCG each edge being cut signifies a required copy instruction this phase of Jang’s clustering algorithm attempts to minimize the number of copy instructions which have to be added as a result of clustering.

- **Phase 4: Insert copies and re-schedule.** This is the final phase of Jang’s clustering algorithm. In this phase the copies required as a result of RCG partitioning are added to the instruction stream and the code is re-scheduled to incorporate these copy instructions.

The results Jang reports for his algorithm are summarized in Table 4.1. These tests were conducted over a three wide clustered VLIW machine having general purpose FUs. The
latency involved for integer operations was one cycle and was two cycles for floating point operations. The copy latency was two cycles for copying integer operands and was four cycles for copying floating point operands.

As can be seen, he reports excellent numbers related to the code size degradation. From this it is possible to draw the conclusion that his LPK algorithm [8] does indeed succeed in minimizing the number of copy instructions that it has to add to the final code. When looking at the degradation related to the execution time, it can be seen that the results are no longer impressive. However, looking closely there are some results in that column which are in the *acceptable* range. These contradicting numbers can be explained. In cases where the results relating to the degradation of execution time are within an acceptable tolerance, the code involved naturally has a high level of ILP. As a result, Jang's LPK algorithm manages to generate some parallel streams of instructions which improves the code's overall performance. However in those cases where the test code naturally has a lower level of ILP the algorithm's inability to generate parallel instruction streams results in unacceptably long schedules and the degradation related to the execution time worsens. In short, the LPK algorithm partitions the RCG such that the final code generated gets sequentialized.

### 4.3 A Greedy Approach to Clustering

Jang’s research [7, 8] showed that while the LPK algorithm minimized the number of copies required, it had the drawback of serializing the code. To overcome this and other deficiencies in Jang’s work [7, 8], Sweany and his colleagues designed a new greedy algorithm for partitioning the RCG [4].
Their partitioning approach, like the LPK algorithm, attempts to minimize the number of copy instructions required in the schedule. However, in addition to this copy minimization, their approach also considers the effects of a particular register’s allocation to a particular cluster on the length of the schedule being generated. The greedy approach thus attempts to overcome the deficiencies in Jang’s partitioning algorithm while at the same time trying to maintain its strengths related to copy minimization.

The results of the greedy approach were good for code where the level of ILP available was low. In those cases it managed to extract the necessary parallelism from the code resulting in acceptably good schedules. However in those cases where the code naturally had higher levels of ILP, it resulted in unacceptably long schedules. The reason for this was that the greedy approach to clustering resulted in an excessive number of copy instructions. In code where ILP is low such copies can be hidden by utilizing empty issue slots. However when the ILP increases idle instruction slots are no longer freely available and this causes the final schedule length to increase drastically.

4.4 Work on Register Cloning

Darla Kuras’s Masters Thesis [9] explored the option of cloning registers which were used across multiple register files. The logic behind cloning was that provided free registers were available on the target register bank, cloning would eliminate the need for some copy instructions while maximizing resource usage on those register banks. Kuras targeted her work [9] to software pipelined loops. The instructions generating the register clones were placed in the loop prelude. As a result their effect on code degradation was minimal. At
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Ideal</th>
<th>Greedy</th>
<th>Greedy-Clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC</td>
<td>29.77</td>
<td>42</td>
<td>39.85</td>
</tr>
<tr>
<td>FMM</td>
<td>12.33</td>
<td>14</td>
<td>13.83</td>
</tr>
<tr>
<td>Kernel</td>
<td>11.53</td>
<td>16.47</td>
<td>14.67</td>
</tr>
<tr>
<td>Other</td>
<td>11.27</td>
<td>17.40</td>
<td>15.40</td>
</tr>
</tbody>
</table>

Table 4.2: Average II over tested benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Greedy</th>
<th>Greedy-Clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC</td>
<td>46.92</td>
<td>37.77</td>
</tr>
<tr>
<td>FMM</td>
<td>14.83</td>
<td>13.83</td>
</tr>
<tr>
<td>Kernel</td>
<td>18.73</td>
<td>14.20</td>
</tr>
<tr>
<td>Other</td>
<td>27.53</td>
<td>22.80</td>
</tr>
</tbody>
</table>

Table 4.3: Average number of copies inserted

the same time the work they did eliminated some of the copy instructions resulting in better schedules. In addition to these commonly used registers she targeted loop induction variables for cloning. The logic in this case was that induction variable updates take only a single cycle whereas copy instructions took longer to execute. Tables 4.2 and 4.3 summarize Kuras’s reported results. As can be seen, cloning does make an impact on the quality of code finally generated. From Table 4.2 it can be seen that on an average the II for the tested loops slightly improved when cloning was used. Table 4.3 shows that on an average the number of copies added as a result of partitioning decreased when cloning was used. Both of these tables show that combining cloning with the Sweany’s greedy partitioning algorithm has better results than using the greedy algorithm by itself.
4.5 Jason Hiser’s Thesis

Sweany’s greedy partitioning algorithm performs better than Jang’s LPK algorithm. However, it adds too many copies in the code it generates. As a result, when the ILP available in the code is naturally high it results in unacceptably long schedules. Looking at these problems, Jason Hiser modified Sweany’s greedy algorithm to try and improve its performance [4, 6, 5]. He called it the Advanced Greedy Partitioning Algorithm [4]. His technique consists of four phases.

- Phase 1: Build the Ideal Schedule. This is the initialization phase of his algorithm. It is similar to the initialization phase in Jang’s partitioning scheme [8]. This phase basically generates a schedule for a VLIW machine having a monolithic register bank. The ideal schedule generated by this phase will provide the information to be encoded within the RCG.

- Phase 2: Build the RCG. When investigating the failings of Sweany’s greedy algorithm Hiser realized that better performance could be achieved if the RCG was modified to include some other factors [4, 5, 6]. He found that the original RCG only encoded information pertaining to the advantages of allocating specific registers to the same cluster. A separate graph was needed to track register interference. He modified the RCG so that the edges between the RCG nodes and the RCG nodes themselves could now encode weights in them. These weights could be both positive or negative and could therefore be used to identify registers whose allocation to the same cluster was relatively more advantageous than other register pairs. As a matter of fact, by using
negative weights it was possible to clearly identify registers which were *good candidates* for parallelism. A single graph structure could thus encode both the important factors related to clustering: relative merits of allocation to the same cluster and the available parallelism. This phase of his algorithm performs this critical job of building the *modified* RCG from the ideal instruction schedule generated in Phase 1.

- Phase 3: Partition the RCG using the Advanced Greedy Partitioning Scheme (AGPS). This is Hiser’s core partitioning algorithm. As has already been seen, his modification of the RCG resulted in the encoding of some very important information within a single data structure. His partitioning scheme attempts to maximize the usage of this information. When assigning clusters he basically orders the nodes of the RCG in decreasing order of their weights. This ensures that registers that are related to a larger number of registers are allocated first. The logic behind such a move was that when a node that is connected to a large number of registers is allocated first the other connected nodes could be allocated in a much more intelligent fashion resulting in better cluster allocation. This would in turn minimize the number of copies required between register banks. In addition, the allocation is done keeping in mind the allocation patterns pertaining to individual clusters. This helps in spreading out the instructions across clusters so that the necessary level of ILP can be generated, albeit at some copying cost. Analyzing the profitability of allocating a particular register to a cluster and identifying an individual node’s priority during the cluster allocation process is basically what AGPS does to partition the RCG.
<table>
<thead>
<tr>
<th>Machine Configuration</th>
<th>Embedded Model</th>
<th>Copy Unit Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 clusters</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>2 clusters, SP</td>
<td>11</td>
<td>50</td>
</tr>
<tr>
<td>4 clusters, SP</td>
<td>26</td>
<td>22</td>
</tr>
<tr>
<td>8 clusters, SP</td>
<td>62</td>
<td>33</td>
</tr>
</tbody>
</table>

SP: Software Pipelined

Table 4.4: AGPS degradation in percentage for Software Pipelined 16-wide VLIW

- Phase 4: Reschedule the Code. This is the final phase of his algorithm. Here the partitioned RCG is taken and the code is re-scheduled so as to meet the partitioning requirements imposed by AGPS.

Hiser tested AGPS over a variety of machine models. More specifically, he tested his scheme over machines with varying cluster size and varying inter-cluster copy implementations. Some of his results are summarized in Table 4.4. As can be seen, he reports extremely good results for his partitioning scheme [4, 6, 5]. He reports a 10% degradation in performance on an average over the entire test suite of non-software-pipelined code. This degradation is with respect to the ideal but un-practical VLIW machines. With respect to software pipelined loops the degradation worsens and is on an average 25% worse than the code generated for an ideal VLIW machine. Also, the kind of machine model used also strongly affected the quality of code generated. This was as expected. In the embedded machine model the copy instructions compete with other instructions for FUUs. In the case of the copy unit machine model while other instructions are left free of competition from copy instructions, the number of copies that can be scheduled simultaneously is limited. Therefore, depending
on the number of FUs attached to a particular cluster the performance improves or degrades. This is adequately reflected in Hiser’s results.

From the comparisons seen so far, it is possible to adequately conclude that Hiser’s AGPS is by and large the best of the clustering algorithms developed at Michigan Technological University. Even more fascinating is the fact that when compared to clustering algorithms developed elsewhere, AGPS still turns out to be one of the best register partitioning schemes in existence today. The algorithm’s strongest point is its machine independence. Since the RCG used in AGPS abstracts away machine details, investigation of the register partitioning problem can proceed smoothly over a variety of architectures. In addition, scheduling and register allocation are phases separated from the register partitioning phase. Thus, it is possible to do partitioning independent of the constraints imposed by scheduling and register allocation. Most of the other algorithms [3, 11, 7, 8, 9, 4, 5, 6, 13, 12, 10, 2, 17] as seen earlier lack this property which limits their applicability specifically to the models they were targeted for. And despite this level of machine independence, AGPS still manages to provide results which are similar to if not better than those reported by other studies. Hiser’s AGPS is the partitioning scheme whose performance is being evaluated in the work being described here.

4.6 The Kernighan-Lin Partitioning approach

J. Zhang in his Master’s Thesis explored the possibility of an alternative partitioning scheme yielding better results when compared to AGPS [17]. His argument was that since AGPS is greedy, it may not yield the best possible results. In simpler words, the AGPS essentially
sacrifices better schedules for speed of execution. To see if better schedules could be obtained he investigated the use of the Kernighan-Lin (KL) partitioning algorithm to partition the RCG. The choice of the KL algorithm was made because of its ability to provide least edge-cut partitions. Zhang in the end concludes that it is sometimes possible to improve upon the AGPS partitions by using the KL partitioning algorithm. However, his partitioning scheme also has a worse time complexity.

This chapter provided a brief overview of the amount of effort invested by Michigan Technological University in trying to address the clustering problem. It also showed the infrastructure readily available there which could be effectively leveraged to address the issues involved when effectively trying to evaluate the state of clustering as it exists today. The rest of this document focuses on the detailed experimental methodology followed in order to address the state of clustering evaluation and an analysis of the results obtained during the process.
Chapter 5

The Experimental Methodology

_Clustering_, as seen earlier, is an issue of significant importance to the ILP community. Also seen earlier is the fact that this problem has been the focus of several studies over the past few years. Every study has proposed a new approach to try to generate good quality code for partitioned register bank VLIW architectures. Since all of these proposed approaches are heuristics they are generally fine tuned to meet a fixed criteria. As a result, they perform impressively for certain sets of code but fail miserably over some other kinds of code. In addition to this, given the NP-complete nature of the clustering problem even the best of the heuristics report some amount of degradation related to the final quality of code generated. To compound matters, since in general an exhaustive search for a solution is unpractical, most of the existing register partitioning measures report results by comparing themselves to code generated for the currently unrealizable, monolithic register bank VLIW machine. All of these factors give rise to a small but important set of questions.

- _Question 1:_ How far away from the _real_ optimal solution are today’s _acceptably_ good
clustering schemes? Here the real optimal solution refers to the solution that would be optimal not for a monolithic register bank VLIW machine but for the actual partitioned register bank VLIW machine that is required to be dealt with in the real world.

- **Question 2:** Is it feasible to design a general purpose scheme that generates near real optimal solutions to the clustering problem? The general purpose scheme refers to a scheme that could be expected to generate good solutions over the entire data set not over specific parts of the data set.

- **Question 3:** At what cost would such a good quality solution generating scheme be successful?

The work described here was an attempt at finding answers to this set of questions listed above. The remainder of this chapter focuses on the details pertaining to the approach followed when finding answers to the questions listed above and an analysis of the results obtained in the process.

### 5.1 Experimental Setup

To find the best possible answers to the questions under investigation the most important thing that was needed to be done was the selection of a good benchmark with which to compare the results obtained in the course of this work. It is our opinion that Hiser’s Advanced Greedy Partitioning Scheme \[4\] is one of the better clustering schemes in existence today. Not only does he report acceptably good results but his scheme is also fast and yet machine independent. Hiser’s AGPS thus became the benchmark partitioning scheme of
choice, for the purpose of this study. As shall be seen ahead, using AGPS as a benchmark also provided the necessary flexibility to investigate a variety of machine models and their effects on register partitioning. In addition to this, since Hiser’s reported results are quite good, AGPS was by and large the ideal choice as a benchmark in the search for an answer to Question 1 listed earlier.

The next important thing to consider was the vehicle of delivery or tool to be used during this investigation. There were a couple of factors that influenced this decision. The first factor was the need to cover a large enough architecture set so as to make the obtained results reasonably acceptable. The second factor affecting this decision was the need to use a tool that was not only well established but also incorporated qualities that would place it amongst the best tools available today. All of these factors lent themselves to the idea of using a good quality, retargetable compiler for the purpose of implementing and testing concepts during this study. Sweany’s Rocket [15], was just the right compiler that met those requirements. Not only is it a well established, academic, retargetable compiler but it also incorporates some extremely advanced concepts such as software pipelining that could, at times, possibly affect the final outcomes. In addition to this, Rocket was the compiler used by Hiser to implement AGPS [4] which in turn had been selected as the benchmark to be used for the purpose of this study. This was also an important factor since it provided an opportunity to compare the results obtained from this study to Hiser’s results in an extremely simple and straightforward manner.

As described earlier, this study had hoped to generate results for a wide variety of machine models. This was by no means an easy task. It brought in the requirement for a machine
modeling software which had to be flexible enough to accomodate a varying number of clusters, varying cluster sizes and at the same time have the ability to model machines in the presence or absence of dedicated units for copying purposes. Poplawski's *Unlimited Resource Machine* (URM) [14] was therefore the natural choice as the target for Rocket for the course of this investigation. The URM not only allows modeling machines that have an arbitrary number of FUs, clusters and differing copy models but it is also comparatively easy to use and target. By using Rocket and targeting it to the URM this investigation was able to generate results which helped answer all of the questions posed earlier in a straightforward manner and at the same time allowed the application of these results to a wide range of architectures.

The discussion so far describes how for the purpose of this investigation Hiser's AGPS [4, 6, 5] was selected as the benchmark in order to evaluate the generated results. Also discussed are the motivations behind this, the primary one being the assertion that AGPS is by and large one of todays best *available* clustering algorithms. In order to make the results obtained *exactly comparable* to Hiser's reported results it was decided to retain and use the same experimental setup Hiser used for his work.

This brings forth another important decision taken in order to fulfil the requirements of this study. This decision related to the choice of the test suite to be used to gather the requisite results. Here again it was decided to implement the algorithm generating *near-optimal* solutions and test it using the same machine models and test programs Hiser had used for the purpose of testing AGPS. There were a couple of reasons for this. The most important factor driving this particular decision was the fact that AGPS had been tested
<table>
<thead>
<tr>
<th>Total Machine Width</th>
<th>Cluster Size</th>
<th>Cluster Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.1: Machine Configurations tested

on a wide variety of models and programs, all representing some of the best benchmarks available for evaluating the quality of code generated. The second reason was that using the same test suite provided an opportunity to compare the results obtained during the course of this study to Hiser’s results in an extremely straightforward and simple manner. Using the exact same test suite helped do away with any need to interpret code generation side-effects which otherwise might have come into existence if different benchmarks had been used for testing the two approaches. In this way a completely transparent evaluation of the results obtained was achieved. The details pertaining to the machine models and the benchmark programs\(^1\) used for testing the algorithms are described below.

5.1.1 Machine Configurations

Several cluster configurations were evaluated in the course of this work. Machine widths of 8 and 16 functional units grouped as clusters of size two, four, and eight were tested. Table 5.1.1 shows the exact clustering methods that were used. In all machine models, each functional unit was general purpose and capable of executing any instruction.

\(^1\)All machine and benchmark descriptions have been taken from Hiser’s thesis [4] document and modified appropriately
Machines with less than eight functional units shouldn’t require partitioned register banks. On the other extreme, machines with more than 16 functional units will likely be infeasible for a long time. The cluster configurations shown for the 16 wide machine are the most likely to be implemented and demonstrate scenarios from an almost ideal machine (two clusters) to a highly partitioned machine (eight clusters).

5.1.2 Copy Models

Within these configurations, three variations were tested that differ only by the method that copy instructions are represented. The first model is called the embedded model. In this method, copy instructions take one issue slot in a functional unit associated with the cluster of the destination register. By using this model, up to \( N \) copy instructions can be issued in a single cycle, if there are \( N \) functional units, regardless of how many clusters are in the machine. However, if a copy operation is issued to a functional unit on a given cycle, no other operation can be issued to that functional unit. Thus, copy instructions compete for valuable issue slots when the machine is fully utilized.

The second model is the copy unit model 1. In this model, copy instructions don’t take any issue slots in the machine. Instead, there is a limit on the number of copy instructions that can take place per cycle. In a machine with \( N \) clusters, we allow up to \( \log_2(N) \) copy instructions to be issued along with an operation per functional unit. This has the advantage that copy instructions don’t take up valuable issue slots when the machine is fully utilized. Unfortunately, only a limited number of copy instructions can be issued per cycle. This could slow a program down because it may not be possible to move data quickly enough.
The last model, copy unit model 2, is the variation of the copy unit 1 model. Like model 1, no issue slots are used for copies. However, up to \( N \) copies may be issued in a single cycle, so long as no unit has more than \( \log_2(N) \) registers read or written. This has the advantage that more copy instructions can be issued in a single cycle, without requiring too many ports per register bank dedicated to copy instructions.

It should be noted, that the clustering algorithms are not dependent on any of these methods of copying values, only that copy instructions exist in some form. Any form of copy instruction could be used within this framework.

5.1.3 Instruction Latencies

All machine models used the following operation latencies:

- Loads take 2 cycles,
- Integer Multiplies take 5 cycles,
- Integer divides take 12 cycles,
- Other integer instructions take 1 cycle,
- Floating Point Multiplies take 2 cycles,
- Floating divides take 2 cycles,
- Other floating point instructions take 2 cycles,
- Stores take 4 cycles, and
- Integer copy instructions take 2 cycles, while floating copies take 3 cycles to complete.

Instruction latencies for current architectures are similar to the ones presented above. The difference, however, is that the latencies above are slightly lower than can be expected
for real architectures now, and into the future. The reason for this is to provide a worst case scenario for the algorithms. If latencies are low, a partitioning algorithm will have to perform optimally in order to minimize delays due to inter-cluster copy operations.

The one exception to the “low latency makes partitioning difficult” rule is with copy operations. Instead, they have a large latency. The large latency on copy operations will cause any partitioning algorithm to perform worse than with shorter latencies for copy operations because data movement will take longer. Yet again, this generates the worst case scenario to evaluate a partitioning algorithm

5.1.4 Benchmark C Programs

To test the algorithms for whole programs, 20 C programs were used. The 20 C programs are listed below, along with a short description of each and minimal execution metrics. For each program, the number of functions included in the program and the number of instructions required to run the program on the “ideal” 4-wide ILP machine, that being one with a single register bank, and the latencies listed in the Section 5.1.3.

KillCache  KillCache is a program designed to make small caches perform poorly. It creates random data, manipulates it in a way that inhibits temporal and spatial locality of memory, and print the results. The single function in this program executes approximately 44,000K instructions.

MergeSort  The MergeSort program allocates space for a moderately sized array, initializes the array to a set of randomly generated numbers, performs a merge sort on the array, and prints the unsorted and sorted arrays. The four functions in this program execute
approximately 31K instructions.

**Dice** The Dice program simulates rolling a pair of dice using a random number generator. Its function is to ensure that the random number generator provides a good distribution of numbers over for large number of calls. The four functions in this program execute approximately 48K instructions.

**Pascal** The Pascal program recursively calculates and prints out each number in the first 10 rows of Pascal’s triangle. The three functions in this program execute approximately 23K instructions.

**IsPrime** The IsPrime program determines which of the first 1000 integers are prime. The two functions execute approximately 47K instructions.

**ListTest** This program creates, appends, destroys, and prints lists. It demonstrates the power of the algorithm on small, modular code. The six related functions take approximately 555K instructions.

**MatrixMult** This program initializes two arrays, multiplies them together, and prints out the resulting array. This single function related to executing this program takes 227K instructions.

**Malloc** This program allocates an array using malloc, initializes it, and prints out the array. Only one function is used to implement this program and it executes a total of 9000K instructions.
**BinarySearch**  This program allocates an array, fills it with random numbers, sorts it, and then repeatedly searches the array using a binary search. The three necessary functions for this execute a total of 644K instructions.

**Hanoi**  This program recursively solves the five disk Towers of Hanoi problem. The two related functions take a total of 0.6K instructions to execute.

**8Queens**  This program enumerates every solution to the 8 Queens on a chess board problem. The three functions used to implement this algorithm take a total of 248K instructions to execute.

**BubbleSort**  This program runs the bubble sort algorithm. The three functions that comprise this program take 80K instructions to execute.

**Nsieve**  This program is the Sieve of Eratosthenes program by Al Aburto. The three functions used to implement this algorithm take 81M instructions to execute.

**Heapsort**  This program is the Heapsort program for variable sized arrays by Al Aburto. The two functions that comprise this program execute 25M instructions.

**Dmxpy**  This program performs floating point calculations on several global arrays. The single function takes approximately 4K instructions to execute.

**Gauss**  This program solves the Gaussian elimination problem for a moderate array. The ten functions that comprise this program take approximately 91K instructions to execute.

**TSP**  This program solves the Traveling salesman problem for a small set of cities. The exponential algorithm takes approximately 11K instructions to execute the four functions
used.

**Livermore** The Livermore loops, which includes kernels taken from scientific code. The three functions take 1307K instructions to execute.

**Whetstone** This program is the whetstone benchmark written in C. It takes approximately 981K instructions to execute the five functions.

**Frac** This program finds rational approximation to floating point value and is written by Robert Craig, AT&T Bell Labs - Naperville. The two functions used to do this take approximately 0.5K instructions.

### 5.1.5 Benchmark Fortran Loops

In order to examine how the *near-optimal* solution generating algorithm performed with respect to software pipelined loops, it was used in the context of software pipelined Fortran loops. The loops to be tested had been extracted from Spec95 and were unrolled four times to allow adequate parallelism usage. The loops were software pipelined for the ideal machine, and then the RCG was built from this pipelined schedule. After partitioning, the loops were re-pipelined, the achieved Initiation Interval for both pipelines was recorded, and the percentage degradation calculated.

### 5.2 Genetic Algorithm based Register Partitioning

As seen earlier, the clustering problem is classified as a NP-complete problem. Generating an *optimal* solution to this problem by exhaustively searching the solution space is therefore
unpractical. For the purpose of this study however, there was a need to generate solutions which were either optimal or as close to optimal as possible. These factors disqualified the use of ordinary heuristics as the mechanism to use, since they tend to get caught within local optima and thus result in sub-optimal solutions quite regularly. Of all the schemes available today perhaps the only techniques that rival exhaustive searches with respect to the quality of solutions generated are the so called modern heuristic search schemes [16]. These schemes, as seen earlier, are subset searches like ordinary heuristics but have a wider span and the capability to escape from local optima if and when needed. This gives them the capability to generate solutions of exceptional quality almost regularly. It was natural therefore to choose one of these modern heuristics to generate the near optimal solutions required to evaluate Hiser’s AGPS. The search scheme chosen to implement the search for a near optimal register partition was a Genetic Algorithm.

5.2.1 The Search Scheme

The selection of a Genetic Algorithm as the technique to use in order to implement the search for a near-optimal solution was driven by a couple of factors. The most important factor was that a GA based search scheme is what is describe as a parallelized serial search for a solution. A parallelized serial search means that unlike similar schemes like simulated annealing that serially search a solution space and then heavily rely on the escape mechanism to get them out of local optima, a GA progresses by looking at multiple regions in the solution space simultaneously. While it still needs the escape mechanism at times to get out of tight corners, its reliance on this mechanism is greatly reduced because its movement is generally
Algorithm NearOptClusteringGeneticAl()
Input: Unpartitioned RCG
Output: Near-Optimal Partitioned RCG
Start
  Initialize Generation Count;
  population = BuildInitialPopulation();
  Partition RCG Using a AGPS
  Add the solution obtained to population

While(Generation Count is not high enough)
do
  Schedule individual solutions in the population
  Evaluate Fitness of the population from the schedules
  // Apply Crossover to build the new population
  newpopulation = Reproduction(population);
  Schedule individual solutions in newpopulation
  Evaluate Fitness of the new population from the schedules generated
  population = BuildNextGeneration(newpopulation, population);
  Mutate(population);
  Increment Generation Count
endo

// Schedule and Evaluate the solutions in the final population
Solution = EvaluateFinalPopulation(population);
End

Figure 5.1: The Near-Optimal Clustering Genetic Algorithm

not restricted to a limited area in the solution space at any given point in time. This ability to span all across the search area also gives it a wider reach when compared to other similar schemes. This was another issue that drove the decision in its favor.

Figure 5.1 shows a brief outline of the GA used during the course of this investigation. Amongst the important points to take note of in this approach is the fact that of the initial set of solutions generated, at least one always represents a solution obtained using a variant of Hiser’s AGPS. The reason for this is the belief that the AGPS variant does generate
good quality solutions and so represents at least one good starting point for the search. In addition to this, while it was expected that the algorithm not be caught in a local optima, using AGPS as a starting point ensured that the solution search always started in at least one of the more promising regions of the search space.

5.2.2 The Package

As seen from the previous section, a decision was taken to use Genetic Algorithms as the framework within which to implement the near-optimal register partitioning solution. While we firmly believe that this was a good decision, it is also a very important fact that genetic algorithms have a very large number of independently varying parameters. All of these parameters interact with each other in sometimes very surprising ways. This interaction does sometimes lead to the failure of an implementation. In order to discount the possibility of such a failure with respect to the implementation, it was decided to use an available, already proven genetic algorithm package to build the partitioning scheme. A very important side effect of this particular decision was that it helped standardize the implementation and thus has helped make the implementation even more widely acceptable.

In order for a GA package to be selected, the following set of requirements had to be met:

- It had to be a properly Software Engineered package.
- The package had to be well documented so as to avoid any “unpleasant” surprises along the way.
• The package had to have a clean interface that would allow its easy inclusion within
the Rocket framework.

• It was a requirement that the package had to have been commonly used earlier to solve
a similar kind of problem.

• The genetic operators defined in the package had to be amongst the best available
today.

Darrell Whitley’s Genitor [1] turned out to be a GA package that completely fulfilled all
of the above listed requirements. Not only is Genitor a well written piece of software but it
is also extremely flexible and includes a host of very useful utility functions. Add to this the
fact that Genitor had already been used earlier to solve somewhat similar problems such as
the Travelling Salesman Problem and Genitor became the package of choice to use. Genitor’s
versatility as a GA package can also be gauged by the fact that at its core it includes a variety
of datatype independant operators each of which can be tuned individually to the needs of
any class of problems being investigated.

But above all of these advantages, Genitor has two capabilities that allow it to be placed
amongst the list of the best GA packages available. These are the two core operators a
GA relies on when traversing a search area. The operators refered to here are the crossover
and mutation operators. At Colorado State University these two operators have been closely
studied and powerful variants of their traditional forms have been developed and incorporated
into the Genitor framework. These powerful variants are known as the Reduced Surrogate
Crossover operator [1] and the Adaptive Mutation operator [1].
The Reduced Surrogate Crossover operator is actually based on the same idea as the traditional crossover operation. For example consider two genes:

\[ s_1 = \{a_1, a_2, c_3, c_4\ldots, a_n\} \quad s_2 = \{b_1, b_2, c_3, c_4\ldots, b_n\} \]

A traditional crossover operator would proceed by first selecting the two points in the genes at which to exchange information and then proceeding with the actual information exchange. This is done so as to generate two new children which might then be added to the population if they are deemed fit enough. For example suppose in gene \( s_1 \) points \( a_2 \) and \( c_4 \) were selected and in gene \( s_2 \) points \( b_2 \) and \( c_4 \) were selected as the crossover points. The new children generated would then be

\[ s_3 = \{a_1, b_2, c_3, c_4\ldots, a_n\} \quad s_4 = \{b_1, a_2, c_3, c_4\ldots, b_n\} \]

From the above example it can be seen that the aim of a crossover operator is to actually generate a new set of genes which might then, depending on their fitness, be added to the population. While the traditional crossover operator succeeds in fulfilling this requirement in a lot of the cases, it does fail miserably in some other cases. For example consider what would happen if points \( c_3 \) and \( c_4 \) were to be selected as crossover points in both genes \( s_1 \) and \( s_2 \). The children generated would then be

\[ s_3 = \{a_1, a_2, c_3, c_4\ldots, a_n\} \quad s_4 = \{b_1, b_2, c_3, c_4\ldots, b_n\} \]

which are exactly identical to the selected parents. This means that the exchange of information between the genes was wasted in this case. This kind of situation can be expected to occur more and more frequently as the GA progresses and the population starts stabilizing with most of the population slots getting filled up with the same, fittest gene. It is this
kind of wasted effort that the Reduced Surrogate Crossover operator specifically attempts to avoid.

The Reduced Surrogate Crossover operator, like the traditional crossover operator, aims to generate *newer* and possibly better solutions to add to the population. However unlike the traditional crossover operator’s technique of blindly selecting the crossover points, Reduced Surrogate Crossover first checks to ensure that there are differences in the information that is being exchanged between the two genes. If there are no differences then a new set of points is chosen for crossover and the process is repeated. Inspite of all of this, it is still possible that the set of parents chosen may not have any differences between them. In such an extreme condition the Reduced Surrogate Crossover operator gives up and proceeds like the traditional crossover operator. However, it does indicate the number of position differences that existed in the information exchanged. A failure of the Reduced Surrogate Crossover operator generally implies that a population stagnation state has been reached and so this position difference number it reports can be utilized as a trigger to intensify mutation so as to allow sufficient diversification of the population.

Mutation is the other operator that greatly affects the quality of solutions which a GA generates. Since it is the operator which the GA relies on to escape from local optima, it has the ability to very often make or break a GA. Traditionally, the use of mutation has been restricted to times when a GA has a stagnating population and there really is a very pressing need to diversify. Even then, a lot of times it is recommended that the mutation rate be small. The reason for this traditional approach is that mutation is basically a disruptive operator. A high mutation rate can virtually reverse all of the gains made by a GA during its
progression and might actually end up placing the GA in a randomly selected worse solution area if it is applied in an uninhibited manner. However, inspite of all of these shortcomings, a mutation operator is a necessity if a genetic algorithm is to succeed.

In order to overcome these shortcomings seen during mutation, a variant of traditional mutation was developed at Colorado State University and incorporated into Genitor. This variant is known as the Adaptive Mutation operator. It is an operator that basically attempts to limit the level of disruption that traditional mutation induces and does so in an educated manner. The adaptive mutation operator, before actually mutating a gene, checks to see if there is really a need to mutate that particular gene. One of the ways this can be done is by checking the parents of the new gene to identify the number of differences they have in their chromosomes. This can be used to identify the level of uniqueness of the population. The more unique candidates a population has, the lesser is the need for mutation. Genitor’s Adaptive Mutation operator implements just this philosophy [1]. For example if a gene is found to have parents that are about 80% unique, the child might be mutated at only a 10% rate. On the other hand if the parents are found to have only about 10% unique chromosomes in them, then the child might actually be mutated at a higher rate, maybe say 80%. In this way the application of the highly disruptive mutation operator can be more or less limited to only the most deserving candidates.

Seen from the above discussion is the fact that Genitor is an extremely versatile package with some very cutting edge operators incorporated in it. It was for these specific reasons that Genitor was selected as the package of choice in order to implement the search for a near-optimal register partition.
5.2.3 The Implementation

Discussed so far are the details pertaining to the environment selected to conduct the experiment in, decisions relating to the tools to use and some of the important factors influencing those decisions. This section discusses details relating to the actual implementation put in place in order to gather the results necessary to fulfil the requirements of this particular study.

In order to gather the requisite numbers relating to the performance of the genetic algorithm vis-a-vis Hiser’s AGPS and analyze them the following things were needed to be done.

- **Step 1**: The platform over which Hiser had implemented AGPS needed to be reassembled so as to recreate the environment in which AGPS had been tested.

- **Step 2**: Sweany’s Rocket had to be modified so as to allow the inclusion of the GA based solution generator.

- **Step 3**: Already discussed are the reasons pertaining to how and why Genitor was chosen as the GA package to help implement the solution to the clustering problem. However, while the package was available, an interface had to be built to allow the transfer of information out of Rocket and into the GA and then once the GA completed its operation to communicate the results back to Rocket for final processing. At the same time it was also important to clearly identify what kind of input would benefit the GA the most and decide how to squeeze this information out of Rocket.

- **Step 4**: A system had to be put in place to gather the final results generated at the end
of this experiment, filter the useful information from this entire data set and arrange it separately in a useful manner to ease analysis.

Of all the above listed steps, the importance of Step one is very often underestimated. In the case of any experimental work like this one, it is of utmost importance that the sanctity of the environment in which the base case has been tested be maintained. The reason for this is simple. Even a slight variation in any of the parameters can often provide a mighty blow to the very foundations of the work being evaluated by making a lot of the assumptions made in that work useless and thereby rendering all of the newly generated results in turn useless. So in order to provide a transparent evaluation of Hiser’s work a lot of time was invested in replicating the environment in which Hiser had tested AGPS to within a very rigorous tolerance level. This involved things like setting up the exact same test suite he had used and rolling back any changes made to Rocket to the time when AGPS was incorporated into Rocket.

Once the necessary environment had been set up it was time to proceed with the core algorithm implementation. Steps two and three described earlier represent the things actually needed to be done to incorporate the GA based scheme into Rocket. However, before progress could actually be made there were some other implementation related decisions that were needed to be taken.

The first of these decisions related to the kind of information a particular gene had to encapsulate. In the case of the problem being tackled here, the job of the GA was to come up with possible sets of separated operands that could extract the best performance from the program. This lends itself to the idea of a gene encapsulating information pertaining
to every operand seen in the ideal schedule. It should not only include information about its allocated bank but also its position in the entire ideal schedule. Keeping this in mind it was decided that the gene would basically be the list of operands seen in the ideal schedule with every chromosome being indexed by the operand’s position in the ideal schedule. Thus basically each chromosome in the gene would be an operand from the ideal schedule and it would encapsulate information about the bank allocated to it.

To understand in detail the implications of the genetic representation chosen, consider the following example schedule.

\[
\begin{align*}
r_1 &= r_2 + r_3 \\
r_4 &= r_5 + r_1 \\
r_5 &= r_1 + r_2
\end{align*}
\]

Considering the format described earlier, the genetic representation of this schedule would become

\[
\text{Gene} = r_1 - r_2 - r_3 - r_4 - r_5 - r_1 - r_5 - r_1 - r_2
\]

As can be seen, every instance of an operand has a separate identity in the representation. This generates much more flexibility and helps in locating a better performing partition.

This brings forth the next important decision taken: How should an individual chromosome be represented? The Genitor framework allows the chromosomes to represented as either bit patterns, integers or floating point numbers. The decision in this case was very closely tied to an earlier decision about what a gene should actually encapsulate. Since the chromosomes here encapsulate the allocated bank information for an operand it was decided that an integer representation would best suit this particular implementation.
A very important factor in the success of any genetic algorithm based implementation is the representation of the worth or fitness of a particular gene. This implementation was no exception. So the selection of a possible fitness representer had to be done extremely carefully. However, the targets of the implementation were quite diverse. On one side whole programs had been targeted where the overall schedule length could identify a solution’s fitness. On the other hand software pipelined loops had been targeted in which case a shorter schedule length does not necessarily represent the best possible schedule. However in this case the II of the loop does indeed identify the best possible solution. The implementation done takes care of both these factors by handling loops and whole programs seperately, a continuation of the way Rocket handles these cases.

The selection of schedule length as the fitness representer for whole programs was itself not simple. The reason for this is that the parameter schedule length is itself extremely deceiving. To understand this consider the dataflow graph shown in Figure 5.2.

In the graph B1, B2, B3 and B4 represent blocks of instructions. Suppose that a statement S needs to be scheduled and that its addition causes an increase in the overall static
cycle count of the entire function by a single cycle. This is a quite easily possible scenario if there are no holes available in the schedule where an instruction can hide. Suppose the candidate blocks where this instruction can be scheduled are blocks B1 and B2. In such a case the static schedule length will not differ depending on the choice of the block since the selection of either of the blocks causes an equivalent increase. Under such circumstances the compiler might select either of the blocks and might even end up breaking the tie in a random fashion. However, such a solution need not always be the best solution. Consider what would happen if at run time block B1 were executed 10000 times and block B2 only about 10 times. This is also a possible scenario and in such a case choosing block B1 to schedule the instruction in would be a very bad choice since the runtime performances of the two schedules differ drastically in favor of block B2.

The important thing to learn from the above example is that such conditions where schedules with equal static cycle counts show drastically different dynamic behaviours are rampanty present in today’s available code segments. Representing the fitness of the genes by means of the static cycle count might therefore actually have ended up being a bad choice. It might actually have caused a premature demise of some really good quality genes from the population. On the other side, dynamic block execution data is generally not available until after the program has completed execution, by which time it is already too late.

To overcome this problem and to prevent a premature destruction of good quality partitions a profiler is used to gather block execution statistics and then utilize these statistics to generate fitness information for the genes. However, to guard against the absence of profile data the implementation switches to using the static cycle counts as the fitness indicator
when block execution statistics are unavailable. Thus the implementation, for whole prog-
gram compilation purpose, uses dynamic instruction count to represent a particular genes
worth when profile data is available and uses the static cycle count when this profile data is
absent.

The discussion in this section so far details some of the most critical decisions taken as
progress was made with the implementation of the GA. Most of these decisions were design
decisions affecting the very progress of the GA. Once these decisions had been taken Rocket
was modified and the Genitor framework was included within it. However, to facilitate faster
data generation there were some other schemes and utilities that were incorporated as part
of the implementation.

A close look at the GA implementation shows that while on the whole the operator
application and the new generation transition can be expected to be done fairly quickly,
the evaluation of a particular gene’s worth is an extremely tedious job. It is therefore not
surprising that one of the utilities referred to above relates to the fitness evaluation of all
new genes.

From the discussion so far, it can be seen that the fitness evaluation was related to the
cycle counts generated when the coloring scheme encapsulated within the gene was used to
actually schedule the code. This cycle count identification is easier said than done. Every
genetic evaluation involves the following phases:

- Rolling back the compilation to just after the ideal schedule has been generated.
- Actually partitioning and scheduling based on the properties of the gene being evalu-
ated.
• Assigning the evaluated gene its new worth.

As can be seen, having to partition and schedule for every genetic evaluation is a non-trivial task and is extremely costly in terms of the time involved. However, it is a necessity that cannot be ignored if the GA is to be successful. So, to lower these demands on the time needed, a data structure called as the *history table* was designed and implemented. It is a simple structure that just maintains the history of the GA’s progression through the search area. The structure essentially stores an arbitrary number of genes and their evaluated worth. It is maintained as a circular queue with the latest seen gene displacing the oldest gene in the list if the list ever gets filled up. The genes in the history table have to be unique with no repetition allowed for maximum efficiency.

The success of the history table data structure is based on its ability to avoid the need to evaluate a gene’s worth altogether. If a gene has been evaluated previously and it exists in the table then its worth can just be looked up thus speeding up the GA’s progression drastically. The requirement of course is that the gene should have been evaluated at some point in time earlier and it should continue to exist in the current table. These conditions are fulfilled more and more often as the GA’s population converges to within a specific area in the search space. This is just part of a GA’s natural behaviour and it was found that the speedup achieved was considerable once this particular data structure was used.

The other important utility implemented was meant to again alleviate the problems generated as a result of the computational intensity necessary to successfully complete the operation of any GA based approach towards problem solving. Already discussed in detail is how costly it is to generate solutions using a GA. The cost refered to here is not only in

62
terms of time. It is also related to the resources required to complete a run of the GA. The need to roll back to the ideal schedule every time an evaluation dealing with the fitness of a gene is required to be done puts enormous demands on the amount of resources needed to successfully complete a single run of the GA.

The coupling of vast resource requirements and bad time complexity of the genetic algorithm prompted the inclusion of a state save utility in the implementation. The utility makes use of some “dump” components available with Genitor to dump out the current state of the GA every arbitrary number of generations and also at the end of the GA run. The GA state is made up of its population members along with the worth of each population member. The generation counter is also dumped to disk. In addition, at the end of a GA run the history table is also saved to disk. A component to read this data back into the algorithm when needed was also incorporated within the implementation. This way the ability to perform a state save and a state restore when needed was added to the implementation. The utility of this approach is evident in the case where an execution of the GA is halted after about N generations and it is found that the results have not yet stabilized. In the absence of this state save utility it would have been necessary to re-run the GA from the start for N+M generations and hope that the population has at that point stabilized. With the state save utility all that is needed to be done is to start the GA run again for M generations and the GA will automatically bootstrap itself using the state restore component. Not only does this provide the ability to continue executing from the last stop point but the latest history is also available for immediate use.

All the ideas discussed to this point were incorporated into the Rocket framework. Fig-
Figure 5.3: Block Diagram representing the Experimental Methodology followed

Figure 5.3 shows a block diagram schematic representing the experimental methodology followed in order to complete this experiment. This new implementation was able to generate some pretty interesting data sets. Presented in the next chapter is this newly generated data and its analysis.
Chapter 6

Experimental Results and Analysis

As indicated earlier in this document, the main aim of this work was to evaluate the performance of one of today’s better known register partitioning schemes by utilizing the capabilities of one of the so called modern heuristic search schemes. This chapter presents the data generated when the GA implementation was used to solve the clustering problem. This data is compared with the data generated using Hiser’s AGPS and an analysis of that comparison is presented. The chapter is arranged in two parts. The first part compares and contrasts the findings related to whole program compilation. Whole program compilation represents that area of compilation where Hiser’s AGPS was highly successful and therefore this section represents the most important part of the findings presented in this document. The second part details the results obtained when compiling for software pipelined Fortran loops.
6.1 Whole Program Compilation

Using a GA to solve a particular problem often entails having to go through the process of fine tuning the GA so that it can adequately explore the search space. The GA implemented was driven by means of three critical parameters whose interaction facilitated the process of gathering some very interesting and surprising data relating to the progression of the GA over its search trajectory. The three parameters were Mutation Rate, Selection Bias and Number of Generations. Mutation Rate is the rate at which the GA diverges. The Number of Generations is the number of times the GA iterates over its population. The Selection Bias identifies the likelihood with which a particular solution is selected as a parent for crossover purpose. For example, a Selection Bias of 3.0 means that the best solution from the population is three times more likely to be selected as a parent in relation to the next best available solution which in turn is three times more likely to be selected when compared to its successor in the fitness hierarchy. The results generated, in addition to measuring the performance of AGPS also measured the effects of these three parameters on the quality of the solutions seen.

The required tests were run over the whole programs test suite while maintaining a population size of 20 genes with 20 offspring being generated every generation. Of the parameters used to drive the GA in order to generate these initial results, perhaps the most controversial one is the selection of such a low generation count. However, this value of generation count was selected after several close observations were made during some dummy test runs whereby it was found out that, rather surprisingly, the GA for most of the cases converged to a specific area within the search space within a span of about 10
to 20 generations. After that time all of the new offspring seen were having their worth looked up from the history table. This meant that they represented genes that had already been evaluated before. The results obtained and the actual parameters used are listed in Table 6.1. The other two parameters, as can be seen, had conservative values assigned to them to ensure that errors if they occurred would occur on the safe side.

As can be seen the results generated were pretty mixed. On one hand no improvement over AGPS generated schedules were visible. However, a closer look shows that these were the cases where no further improvement was really possible because AGPS schedules were as good as ideal schedules. Then there were some extremely good results where some very large positive percentage degradation differences could be seen. The percentage degradation difference refers to the difference in the degradation over ideal schedules seen when comparing the GA and AGPS. This percentage degradation difference is calculated as % degradation relative to the ideal schedule reported by AGPS - % degradation relative to the ideal schedule reported by the GA. Positive percentage degradation differences therefore refer to the differences in degradation over ideal schedules in favor of the GA. The largest positive percentage degradation difference achieved was to the tune of 11.9% in the case of the embedded copy model and 9.5% in the copy unit model’s case. Finally there were results, which shockingly enough, show a massive negative percentage degradation difference. Negative percentage degradation differences refer to the differences in degradation over ideal schedules in favor of AGPS, that is, differences showing that AGPS had outperformed the GA. While this worse performance might seem unsettling at first it can be explained.

As mentioned earlier a variant of AGPS was used to seed the GA’s initial population.
Generation Count: 40  
Selection Bias: 3.0  
Mutation Rate: 0.1

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Embedded Copies</th>
<th>Copy Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AGPS</td>
<td>GA</td>
</tr>
<tr>
<td>Pascal</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IsPrime</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>List/Test</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NSieve</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Hanoi</td>
<td>2.5</td>
<td>0.0</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>19.9</td>
<td>8.0</td>
</tr>
<tr>
<td>MergeSort</td>
<td>32.4</td>
<td>24.6</td>
</tr>
<tr>
<td>KillCache</td>
<td>18.9</td>
<td>9.6</td>
</tr>
<tr>
<td>Dice</td>
<td>3.9</td>
<td>17.3</td>
</tr>
<tr>
<td>Matrix Mult</td>
<td>1.3</td>
<td>24.8</td>
</tr>
<tr>
<td>Malloc</td>
<td>22.2</td>
<td>44.4</td>
</tr>
<tr>
<td>8Queens</td>
<td>18.9</td>
<td>32.0</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>14.2</td>
<td>2.6</td>
</tr>
<tr>
<td>Heapsort</td>
<td>28.9</td>
<td>30.7</td>
</tr>
<tr>
<td>DMXPY</td>
<td>14.9</td>
<td>8.2</td>
</tr>
<tr>
<td>Gauss</td>
<td>20.5</td>
<td>13.6</td>
</tr>
<tr>
<td>TSP</td>
<td>7.1</td>
<td>13.1</td>
</tr>
<tr>
<td>Livermore</td>
<td>13.4</td>
<td>19.4</td>
</tr>
<tr>
<td>Whetstone</td>
<td>5.3</td>
<td>9.8</td>
</tr>
<tr>
<td>Frac</td>
<td>29.0</td>
<td>19.5</td>
</tr>
</tbody>
</table>

Table 6.1: First set of data relating to the performance comparison of the GA and AGPS in terms of percentage degradation seen over ideal schedules
This variant generates some solutions that are worse when compared to AGPS. Therefore the “worse than AGPS” results reported in Table 6.1 are worse when compared to the original AGPS but are actually better than the points from where the GA actually started executing. Their shortcoming is that they are not good enough to beat AGPS’s performance.

However, this explanation of the worse results still does not explain why the GA was not able to outperform Hiser’s AGPS as had been hoped. To understand this better it is important to realize that despite of all its capabilities a GA is still only a heuristic. It is a better kind of approximation to a solution but because of the parameters used it had failed to generate results which were all better than the results generated by Hiser’s AGPS.

So, in order to generate more acceptable results it was decided to analyze and change the current set of parameters defining the progression of the GA. Of the three parameters, the first one suspected for causing the GA’s failure was its mutation rate. The fact that the results seen were, in some cases, worse than AGPS lent credibility to the fact that the GA seemed to be getting stuck in some kind of local optima and was not being able to escape. To improve the GA’s chances of success it was decided to drastically increase its rate of divergence. With this in mind the value of the mutation rate was set to 0.7 and the tests were re-run. Surprisingly, even with this upped mutation rate of 70% there was little or no change in the results obtained earlier.

How could one explain this seemingly uncharacteristic behavior of the Genetic Algorithm? Increasing the mutation rate should have forced the GA to start diverging to a great extent. A failure inspite of such high divergence rates meant that low divergence was not the primary problem. Further analysis showed that, perhaps, solution convergence was happening too
fast for even the divergence parameter to be of much help. This suggested that slowing down the descent of the GA might help improve its performance. With this in mind the selection bias of the GA was decreased to a value of 2.0. Sure enough, almost immediately, a lot of the “worse than AGPS” results started looking up. This second set of data that collected is listed in Table 6.2.

As can be see, this new data set is a major improvement over the initial data set. A lot of the “worse than AGPS” results have improved to a satisfactory level. However, even in this second set there are entries which show a negative percentage degradation difference. This negative percentage degradation difference is less than the first set but it still is a degradation in performance showing that the GA was still having trouble converging to the correct area within the search space and that there still was room for improvement.

Looking at the encouraging trend visible from Table 6.2 it was decided to further explore the effects of varying the most critical parameters affecting the GA’s traversal of the search space. It was hoped that further fine tuning would provide reasonably acceptable results. Various combinations of values were assigned to the Selection Bias and the Mutation Rate parameters and finally the GA was successfully fine tuned. The combination at which the GA provides the overall best results, that is, best results amortized over the entire test suite, was with a selection bias of 1.4 and a mutation rate of 70%. These results are presented in Table 6.3.

This third set of data presented in Table 6.3 is very interesting. It represents the best amortized results the GA produces. However, a closer observation shows that it does not necessarily represent the best possible individual results seen. An explanation is therefore
Generation Count: 40
Selection Bias: 2.0
Mutation Rate: 0.5

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Embedded Copies</th>
<th>Copy Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Embedded Copies</td>
<td>Copy Unit</td>
</tr>
<tr>
<td></td>
<td>Raw Diff.</td>
<td>Raw Diff.</td>
</tr>
<tr>
<td></td>
<td>Percentage Impr/ Degr. over AGPS</td>
<td>Percentage Impr/ Degr. over AGPS</td>
</tr>
<tr>
<td></td>
<td>AGPS</td>
<td>GA</td>
</tr>
<tr>
<td>Pascal</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IsPrime</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ListTest</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NSieve</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Hanoi</td>
<td>2.5</td>
<td>0.0</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>19.9</td>
<td>11.6</td>
</tr>
<tr>
<td>MergeSort</td>
<td>32.4</td>
<td>22.7</td>
</tr>
<tr>
<td>KillCache</td>
<td>18.9</td>
<td>9.5</td>
</tr>
<tr>
<td>Dice</td>
<td>3.9</td>
<td>14.8</td>
</tr>
<tr>
<td>Matrix Mult</td>
<td>1.3</td>
<td>25.0</td>
</tr>
<tr>
<td>Malloc</td>
<td>22.2</td>
<td>22.1</td>
</tr>
<tr>
<td>8Queens</td>
<td>18.9</td>
<td>30.1</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>14.2</td>
<td>14.1</td>
</tr>
<tr>
<td>Heapsort</td>
<td>28.9</td>
<td>17.8</td>
</tr>
<tr>
<td>DMXPY</td>
<td>14.9</td>
<td>10.5</td>
</tr>
<tr>
<td>Gauss</td>
<td>20.5</td>
<td>12.8</td>
</tr>
<tr>
<td>TSP</td>
<td>7.1</td>
<td>13.1</td>
</tr>
<tr>
<td>Livermore</td>
<td>13.4</td>
<td>13.4</td>
</tr>
<tr>
<td>Whetstone</td>
<td>5.3</td>
<td>5.3</td>
</tr>
<tr>
<td>Frac</td>
<td>29.0</td>
<td>16.9</td>
</tr>
</tbody>
</table>

Table 6.2: Second set of data relating to the performance comparison of the GA and AGPS in terms of percentage degradation seen over ideal schedules
Generation Count: 40  
Selection Bias: 1.4  
Mutation Rate: 0.7

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Embedded Copies</th>
<th>Copy Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AGPS</td>
<td>GA</td>
</tr>
<tr>
<td>Pascal</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IsPrime</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ListTest</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NSieve</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Hanoi</td>
<td>2.5</td>
<td>0.0</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>19.9</td>
<td>17.6</td>
</tr>
<tr>
<td>MergeSort</td>
<td>32.4</td>
<td>21.4</td>
</tr>
<tr>
<td>KillCache</td>
<td>18.9</td>
<td>9.5</td>
</tr>
<tr>
<td>Dice</td>
<td>3.9</td>
<td>3.7</td>
</tr>
<tr>
<td>Matrix Mult</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Malloc</td>
<td>22.2</td>
<td>22.1</td>
</tr>
<tr>
<td>8Queens</td>
<td>18.9</td>
<td>18.9</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>14.2</td>
<td>4.4</td>
</tr>
<tr>
<td>Heapsort</td>
<td>28.9</td>
<td>19.3</td>
</tr>
<tr>
<td>DMXPY</td>
<td>14.9</td>
<td>9.8</td>
</tr>
<tr>
<td>Gauss</td>
<td>20.5</td>
<td>4.5</td>
</tr>
<tr>
<td>TSP</td>
<td>7.1</td>
<td>7.1</td>
</tr>
<tr>
<td>Livermore</td>
<td>13.4</td>
<td>13.4</td>
</tr>
<tr>
<td>Whetstone</td>
<td>5.3</td>
<td>6.1</td>
</tr>
<tr>
<td>Frac</td>
<td>29.0</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Table 6.3: Third set of data relating to the performance comparison of the GA and AGPS in terms of percentage degradation seen over ideal schedules
required here to understand this confusing behaviour of the GA whereby the set of best overall results is not actually a collection of the best individual results.

To understand the above behaviour it is required to once again go back to a fact repeatedly stressed throughout this document: “A genetic algorithm is just another heuristic”. The essence of this statement explains the behaviour of the GA seen so far. The fact of the matter here is that a GA is driven by several key parameters. The success of these parameters in helping the GA find the best possible results is directly related to the search terrain being negotiated. The search terrain seen when trying to solve the clustering problem is highly volatile. In simpler words this means that every program being compiled has a search area with its very own shape and size and so every program has its very own set of best fitting parameters. No single set of parameters can therefore be expected to provide the best possible results for all kinds of programs possible. To generate the best possible results it is necessary to individually tune the driving parameters on a per program basis. This important fact is further brought out clearly by the data displayed in Table 6.4 which showcases a summary of the data shown in Tables 6.1, 6.2 and 6.3. The table shows the percentage degradation differences seen in the case of all three earlier seen data sets side by side.

When the above presented facts are understood it can be seen that, actually, the GA is doing really well when generating a set of best amortized results. However, for the purpose of this study, to evaluate the performance of AGPS, Table 6.5 is presented which compares AGPS generated degradation with the best individual results obtained using the GA.

Consider these results in detail. As can be seen, there is a rough 50-50 split between the cases with no improvement over AGPS’s performance and the cases with some measurable
<table>
<thead>
<tr>
<th>Program Name</th>
<th>GC: 40</th>
<th>GC: 40</th>
<th>GC: 40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EC</td>
<td>CU</td>
<td>EC</td>
</tr>
<tr>
<td>Pascal</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IsPrime</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ListTest</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NSieve</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Hanoi</td>
<td>2.5</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>11.9</td>
<td>6.9</td>
<td>8.3</td>
</tr>
<tr>
<td>MergeSort</td>
<td>7.8</td>
<td>0.1</td>
<td>9.7</td>
</tr>
<tr>
<td>KillCache</td>
<td>9.3</td>
<td>9.5</td>
<td>9.4</td>
</tr>
<tr>
<td>Dice</td>
<td>-13.4</td>
<td>-5.8</td>
<td>-10.9</td>
</tr>
<tr>
<td>Matrix Mult</td>
<td>-23.5</td>
<td>-8.4</td>
<td>-23.7</td>
</tr>
<tr>
<td>Alloc</td>
<td>-22.2</td>
<td>-22.2</td>
<td>0.1</td>
</tr>
<tr>
<td>8Queens</td>
<td>-13.1</td>
<td>-11.1</td>
<td>-11.2</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>11.6</td>
<td>-4.8</td>
<td>0.1</td>
</tr>
<tr>
<td>Heapsort</td>
<td>-1.8</td>
<td>-3.4</td>
<td>11.1</td>
</tr>
<tr>
<td>DMXPY</td>
<td>6.7</td>
<td>2.1</td>
<td>4.4</td>
</tr>
<tr>
<td>Gauss</td>
<td>6.9</td>
<td>2.0</td>
<td>7.7</td>
</tr>
<tr>
<td>TSP</td>
<td>-6.0</td>
<td>2.9</td>
<td>-6.0</td>
</tr>
<tr>
<td>Livermore</td>
<td>-6.0</td>
<td>-9.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Whetstone</td>
<td>-4.5</td>
<td>-3.1</td>
<td>0.0</td>
</tr>
<tr>
<td>Frac</td>
<td>9.5</td>
<td>6.2</td>
<td>12.1</td>
</tr>
</tbody>
</table>

Table 6.4: Data summary displaying the percentage degradation differences calculated for the three earlier data sets
<table>
<thead>
<tr>
<th>Program Name</th>
<th>AGPS</th>
<th>GA</th>
<th>Raw Diff.</th>
<th>Percentage Improv. over AGPS</th>
<th>AGPS</th>
<th>GA</th>
<th>Raw Diff.</th>
<th>Percentage Improv. over AGPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IsPrime</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ListTest</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NSieve</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Hanoi</td>
<td>2.5</td>
<td>0.0</td>
<td>2.5</td>
<td>100.0</td>
<td>2.5</td>
<td>0.0</td>
<td>2.5</td>
<td>100.0</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>19.9</td>
<td>8.0</td>
<td>11.9</td>
<td>59.8</td>
<td>13.7</td>
<td>6.8</td>
<td>6.9</td>
<td>50.4</td>
</tr>
<tr>
<td>MergeSort</td>
<td>32.4</td>
<td>21.4</td>
<td>11.0</td>
<td>34.0</td>
<td>22.0</td>
<td>21.9</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>KillCache</td>
<td>18.9</td>
<td>9.5</td>
<td>9.4</td>
<td>49.7</td>
<td>18.9</td>
<td>9.4</td>
<td>9.5</td>
<td>50.3</td>
</tr>
<tr>
<td>Dice</td>
<td>3.9</td>
<td>3.7</td>
<td>0.2</td>
<td>0.0</td>
<td>3.9</td>
<td>3.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Matrix Mult</td>
<td>1.3</td>
<td>1.3</td>
<td>0.0</td>
<td>0.0</td>
<td>1.3</td>
<td>0.2</td>
<td>1.1</td>
<td>84.6</td>
</tr>
<tr>
<td>Malloc</td>
<td>22.2</td>
<td>22.1</td>
<td>0.1</td>
<td>0.0</td>
<td>22.2</td>
<td>22.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>8Queens</td>
<td>18.9</td>
<td>18.9</td>
<td>0.0</td>
<td>0.0</td>
<td>15.0</td>
<td>14.6</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>14.2</td>
<td>2.6</td>
<td>11.6</td>
<td>81.7</td>
<td>16.0</td>
<td>15.8</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td>Heapsort</td>
<td>28.9</td>
<td>17.8</td>
<td>11.1</td>
<td>38.4</td>
<td>18.9</td>
<td>18.8</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>DMXPY</td>
<td>14.9</td>
<td>8.2</td>
<td>6.7</td>
<td>45.0</td>
<td>14.9</td>
<td>7.9</td>
<td>7.0</td>
<td>47.0</td>
</tr>
<tr>
<td>Gauss</td>
<td>20.5</td>
<td>4.5</td>
<td>16.0</td>
<td>78.0</td>
<td>16.4</td>
<td>9.2</td>
<td>7.2</td>
<td>43.9</td>
</tr>
<tr>
<td>TSP</td>
<td>7.1</td>
<td>7.1</td>
<td>0.0</td>
<td>0.0</td>
<td>7.1</td>
<td>4.2</td>
<td>2.9</td>
<td>40.8</td>
</tr>
<tr>
<td>Livermore</td>
<td>13.4</td>
<td>13.4</td>
<td>0.0</td>
<td>0.0</td>
<td>9.4</td>
<td>9.4</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Whetstone</td>
<td>5.3</td>
<td>5.3</td>
<td>0.0</td>
<td>0.0</td>
<td>5.1</td>
<td>5.0</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>Frac</td>
<td>29.0</td>
<td>5.5</td>
<td>23.5</td>
<td>81.0</td>
<td>29.2</td>
<td>13.9</td>
<td>15.3</td>
<td>52.4</td>
</tr>
<tr>
<td>Average</td>
<td>12.0</td>
<td>7.4</td>
<td>4.6</td>
<td>28.4</td>
<td>10.0</td>
<td>8.1</td>
<td>1.9</td>
<td>23.5</td>
</tr>
<tr>
<td>Improvables Average</td>
<td>15.0</td>
<td>9.2</td>
<td>5.8</td>
<td>35.5</td>
<td>12.5</td>
<td>10.1</td>
<td>2.6</td>
<td>29.3</td>
</tr>
</tbody>
</table>

Table 6.5: Evaluation of the performance of AGPS utilizing the best results seen using GA.
improvement over AGPS. It should also be noted that this kind of improvement can be seen in the cases of both of the test models: embedded copy model and copy unit model.

Considering the embedded copy model, it can be seen that the maximum improvement relative to AGPS was seen in the case of the program Hanoi which showed a 100% improvement. Interestingly, the largest positive percentage degradation difference between AGPS and the GA is seen in the case of the program Frac which shows it to be 23.5 points. This means that when compared with respect to the ideal schedules an improvement of around 23 points can be generated whereas relative to AGPS the largest improvement seen is a full 100 percentage points. In addition there are several other cases like Bubblesort, Gauss, and Binarysearch that show a significant improvement over AGPS. In these cases the improvement level seen relative to AGPS is over 50%. Finally there are roughly half of the test cases that show miniscule or no improvement when compared to AGPS.

In the case of the copy unit model the improvement shown appears to be much more subtle. The maximum positive percentage degradation difference is seen in the case of the program Frac which shows an approximate 15 point improvement. Again, when comparing the performance relative to AGPS, the largest improvement seen is to the tune of 100 percentage points in the case of the program Hanoi. There are some other significant improvement cases such as Matrixmult, KillCache and DMXPY but their effects appear much less prominent when compared to the improvement levels seen with the embedded copy model. Again even in this particular case there are several test cases that show only a miniscule improvement possibility.

The last two rows presented in Table 6.5 are very interesting. They detail the amortized
performance level of AGPS in relation to the GA. As can be seen, the average positive percentage degradation difference the GA was able to achieve was of the order of 4.6 points in the case of the embedded copy model and 1.9 points in the case of the copy unit model. The numbers show that AGPS seems to be performing quite nicely since the average positive percentage degradation difference the GA was able to generate was not too impressive. However, looking at the average percentage improvement the GA was able to generate over AGPS, it can be seen that it is of the order of 28% in the case of the embedded copy model and of the order of 23% in the case of the copy unit model. These amortized percentage improvements by themselves show that quite a significant level of improvement is still possible over AGPS.

Looking more closely at the table it can be seen that actually, in a lot of cases, AGPS is showing no degradation over ideal schedules. Improvement over these test cases is not really feasible. So in order to show the average improvements generated for only those cases where any scale of improvement is really possible, the “Improvables Average” are shown in the last row of Table 6.3. It can be seen that the average improvement, relative to AGPS, in the case of the embedded copy model is to the tune of 35% and in the case of the copy unit model is approximately 29%. The difference in the numbers presented here shows that very often numbers can be extremely misleading and that differences in performance might actually get hidden amongst the deluge of numbers.

So, presented in Figures 6.1, 6.2 and 6.3 are a much more visible form of these same results. Figure 6.1 shows a histogram depicting the degradation over ideal schedules seen in the case of AGPS side-by-side with the degradation over ideal schedules seen with the
Figure 6.1: Comparison of the degradation over ideal schedules seen in the case of the Embedded Copy model

GA in the case of the embedded copy model. Figure 6.2 shows a similar comparison but for the copy unit model case. Finally, Figure 6.3 depicts a histogram showing the improvement obtained over AGPS in each of the tested cases for both of the test models.

The histogram in Figure 6.1 clearly underlines the improvement generated in the case of the embedded copy model when using the GA. As can be seen, the difference between the heights of the bars is quite significant wherever it exists. In addition, looking closely at the histogram, it can be seen that wherever there was little or no improvement in performance, that is, no difference in the heights of the bars, the degradation of AGPS over ideal schedules was either absent totally or generally not too significant.
Figure 6.2: Comparison of the degradation over ideal schedules seen in the case of the Copy Unit model
Looking at the histogram in Figure 6.2 somewhat similar conclusions can be drawn. Wherever the improvement is seen it is extremely significant. In most of the cases where improvement is absent the scope for improvement is in any case quite small.

The observations made above in the cases of both of the test models seem to suggest two things:

- The GA has been fairly successful in negotiating a search space that is extremely volatile.
- There is still a significant room for improvement in performance over Hiser’s AGPS in the case of whole program compilation.

The first conclusion vindicates the stand that a GA can indeed generate solutions that might be as good as they might ever get to be. At the same time, it shows that the ideal schedules may not actually be the optimal schedules in the case of partitioned register bank architectures. To clearly indicate and understand the implications of this second conclusion it is important to look at the histogram in Figure 6.3.

That histogram shows the specific percentage degradation differences the GA obtained when compared with Hiser’s AGPS. Some pretty interesting conclusions can be drawn from the histogram. The first observation made is that there seems to be a widespread potential for improvement over AGPS. This is not very surprising since AGPS is by nature a greedy algorithm. A second, more important conclusion that can be drawn is that in the case of the copy unit model AGPS seems to be performing quite impressively. However it seems to be clearly lagging behind in the case of the embedded copy model. This second observation brings out some very important questions: Why is there a discrepancy in performance be-
Figure 6.3: Histogram showing the percentage degradation difference seen in the case of the Embedded Copy and Copy unit models
tween the two models? Does this fact reflect a shortcoming in AGPS? Is it a shortcoming in the GA? Why is there still such a large room for improvement over AGPS generated partitions?

In order to answer these questions the design of AGPS was analyzed and compared to the design of the GA. The answers, it was found out, were a blend of both a shortcoming in AGPS and a problem with restrictions caused by the architecture of the copy models. To understand the answers it is important to first explain a subtle difference in the way AGPS operates and the way the GA builds its best solution set.

The design of Hiser’s AGPS dictates that in order to decide the bank allocation for the operands in the ideal schedule it has to use a data structure known as the Register Component Graph (RCG). This RCG is in turn designed such that every node in the RCG represents a particular virtual register (VR) from the original schedule. The problem here is that in the original schedule there might be multiple operands having the same VR number because of there being the possibility of a VR definition and multiple VR uses. This dictates that most of the uses of the defined operand will end up being allocated to the same register bank. This decision is correct when it is considered that this was meant to ensure that most of the uses of a VR be localized to the same bank thus minimizing the copy requirement, thereby improving the quality of the schedule. The decision was based on the belief that, as far as possible, all members of a single def-use chain be allocated to the same cluster. A problem with this approach however is that it tends to add a restriction on the algorithm’s ability to spread the uses out over the various available banks. For example consider building a schedule for a two wide ILP architecture with the following hypothetical code snippet:
\[
\begin{align*}
  r_1 &= r_2 + r_3 \\
  r_{10} &= r_2 + r_6 \\
  r_4 &= r_1 + r_6 \\
  r_5 &= r_1 + r_4 \\
  r_8 &= r_1 + r_7
\end{align*}
\]

It can be seen that here the VR r1 is repeatedly used. When AGPS builds a schedule it will allocate all of the instructions having r1 as their operands to the same cluster. The schedule it builds might then be:

<table>
<thead>
<tr>
<th>BANK1</th>
<th>BANK2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r2 + r3</td>
<td>NOP</td>
</tr>
<tr>
<td>r_{10} = r2 + r_6</td>
<td>NOP</td>
</tr>
<tr>
<td>r_4 = r_1 + r_6</td>
<td>NOP</td>
</tr>
<tr>
<td>r_5 = r_1 + r_4</td>
<td>NOP</td>
</tr>
<tr>
<td>r_8 = r_1 + r_7</td>
<td>NOP</td>
</tr>
</tbody>
</table>

This however is not necessarily the best possible solution. The best solution will be where the different uses of r1 can be spread out over the clusters so that the free resources available can be best utilized. The best schedule in such a case might be

<table>
<thead>
<tr>
<th>BANK1</th>
<th>BANK2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r2 + r3</td>
<td>NOP</td>
</tr>
<tr>
<td>r_{10} = r2 + r_6</td>
<td>COPY r1, r20</td>
</tr>
<tr>
<td>r_4 = r_1 + r_6</td>
<td>r_8 = r_20 + r_7</td>
</tr>
<tr>
<td>r_5 = r_1 + r_4</td>
<td>NOP</td>
</tr>
</tbody>
</table>

As can be seen, in this case it was possible to hoist an instruction occurring later on in the schedule up the hierarchy along a different cluster. In the process the schedule quality improved as it was possible to utilize the free resources available on the second cluster which otherwise were being wasted. It goes to show that there is a little bit of parallelism that AGPS is missing out. The GA on the other hand has the ability to allocate every individual instance of an operand to a separate bank and therefore *might* eventually succeed in finding the above seen best solution. This approach of spreading instructions out by copying the
operands to a different cluster so as to better exploit the available parallelism has been explored earlier by Darla Kuras [9], though to a lesser extent. Her approach, known as cloning, does precisely this same thing. However, she mainly targets loops and induction variables.

The picture painted above, of successfully exploiting the parallelism available in the program however need not always be very rosy. A major requirement for the success of the approach described above is that sufficient empty space be available in the parallelized schedule to accommodate the copies that will end up being required to transfer operands from one bank to the other. If this space is unavailable then the schedule cannot be sufficiently parallelized and AGPS’s results will probably be very close to the best that might ever be obtained. This fact explains, to a large extent, the results obtained and presented in the histograms earlier.

The results presented in Figures 6.1, 6.2 and 6.3 show that large scale improvement is indeed possible over Hiser’s AGPS. At the same time, the improvements seem to be much more visible in the case of the embedded copy model. The copy unit model has a limit on the number of copies it can schedule simultaneously. On the other hand, if sufficient holes are present, the embedded copy model would support the simultaneous scheduling of a fairly large number of copies and therefore would be successful in better exploiting the parallelism available in the program. This means that the lower copy scheduling power of the copy unit model might actually hamper its ability to exploit parallelism any further. This is precisely what appears to be happening and explains the comparatively lower performance improvement numbers the GA reports for the copy unit model.
Figure 6.4: Comparison of the degradation over ideal schedules seen in the cases of the embedded copy and copy unit models when using the GA

Hiser’s work [4, 6, 5] had reported that while there were a lot of cases where the performance of both the embedded copy and copy unit models were similar, there were also a lot of other cases where the copy unit model was being able to outperform the embedded copy model. Figure 6.4 graphically presents a side by side comparison of the performances of both of the machine models when using the GA.

As can be seen, the results obtained are pretty interesting. The data presented shows that both the models tested show a nearly similar performance degradation in most of the cases. There are still some cases in which one model outperforms the other but these are more or less evenly divided and as a matter of fact the more successful model, defined as the model
which was more often able to show an improvement of over 5% over the other model, seems to be the embedded copy model, not the copy unit model. This, rather surprising, observation seems to suggest that a presence of a larger number of copy units in an architecture is essential to effectively exploit all of the parallelism available in the program. It also appears that since the embedded copy model turned out to be the more successful of the two models there are a sufficient number of holes available in the schedules which if effectively exploited can lead to better performance.

Other information that might be of interest is the gene size and structure. The gene structure, as has already been described, consisted of a list of VRs. The list size was variable. Over the entire test suite the smallest gene size observed consisted of 12 chromosomes. The largest gene was observed to have been made up of 1267 chromosomes. This information is significant in that it shows the versatility and flexibility of the GA in dealing with varying size environments and also brings forth the amount of volatility present in the environment within which the GA was expected to operate.

Figures 6.1 and 6.1 show some of the other data successfully obtained during the course of this investigation. These histograms show a functionwise distribution of the degradation seen when using the GA compared to the functionwise distribution Hiser reported. The results are very much in line with what has been seen so far.

In the cases of both of the test models the number of functions showing smaller amounts of degradation increased in comparison to AGPS. Conversely, there is also a marked decrease in the overall number of functions showing an over 20% degradation relative to the ideal schedules. At the same time this trend in improvement is much more clearly visible in the
Figure 6.5: Functional Distribution of Degradation in the case of the Embedded copy model

Figure 6.6: Functional Distribution of Degradation in the case of the Copy Unit model
case of the embedded copy model when compared to the copy unit model. However, looking closely at both Figures 6.1 and 6.1 it can be seen that, surprisingly, in the case of the copy unit model the number of functions showing no degradation relative to the ideal schedule is larger than the number of similar functions in the case of the embedded copy model. It can also be seen that there is a correspondingly larger number of functions showing a degradation of over 20% in the case of the copy unit model than in the case of the embedded copy model. Intuitively then, it makes sense that the embedded copy model is showing a better performance compared to the copy unit model. These trends seem to underline the fact that the model employing embedded copies is much more successful in exploiting the parallelism available in the program on the whole but at the same time the copy unit model seems to be having a higher success in making partitioned schedules perform like ideal schedules. This, as shown earlier, is because of the higher copying power available in the embedded copy model. It also goes to show that limiting the number of copy units adds a restriction on the amount of parallelism that can be exploited by a partitioned register architecture.

6.2 Software Pipelined Fortran Loops

With respect to whole program compilation it can be seen that there is still ample scope for improvement over Hiser’s AGPS. So far the conditions making improvements on such a scale feasible have also been analyzed. Whole program compilation however can be expected to have only a limited amount of parallelism in it. To evaluate the performance of AGPS when compiling for more intensely parallel code its performance was compared in relation to the
Figure 6.7: Distribution of the degradation in II in the case of an embedded copy model, 2 wide with 8 FUs per cluster

GA when compiling for Software Pipelined loops.

The results in this category have been obtained by using machine models incorporating copy schemes similar to those seen earlier, that is, models utilizing embedded copies and models having dedicated copy units. The machine models however differ in the size and scale of the clusters they incorporate. The first set of results presented here has been obtained for machine models having two clusters with each cluster bearing eight FUs. Figures 6.7 and 6.8 show the histograms detailing these results.

As can be seen, in the case of the embedded copy model results from Figure 6.7 the improvements obtained were quite small, often of the order of a few percentage points. However there is a trend in the histogram of an improvement towards lower degradation levels although not on a major scale.
Figure 6.8: Distribution of the degradation in II in the case of a copy unit model, 2 wide with 8 FUs per cluster

In the case of the copy unit model results shown in Figure 6.8 the trend in improvement over AGPS is again quite similar in nature. In this case however, there is no change in the number of functions showing no degradation over ideal schedules. There is however a big increase in functions showing around 20-40% degradation over ideal schedules. It accounts for the decrease in the number of functions showing degradation in performance of over 80%.

The second set of results obtained in this category are for machine models having four clusters, with each cluster bearing four FUs. It represents the more balanced machine model. Figures 6.9 and 6.10 show the histograms depicting these results.

For the embedded copy model, in this case, as can be seen from Figure 6.9, the improvement is even more limited than the improvement seen earlier in the case of a similar model having two clusters and eight FUs. While there does still appear to be room for improvement
Figure 6.9: Distribution of the degradation in II in the case of an embedded copy model, 4 wide with 4 FUs per cluster

Figure 6.10: Distribution of the degradation in II in the case of a copy unit model, 4 wide with 4 FUs per cluster
it appears to be quite miniscule in comparison, only about 2%. As a matter of fact, there was no change in the number of functions registering a degradation of over 80%.

The copy unit model again shows a somewhat similar trend in the case of this particular cluster configuration. There is however a decrease in the percentage of functions showing a degradation of over 80%. This decrease however is miniscule and seems to reflect in the increase in functions showing a degradation of between 40-80%. This intuitively goes to show the level to which it might be possible to improve over AGPS.

The final set of results presented are related to a machine model which is made up of eight clusters, each bearing only two FUs. This model basically represents a worst case scenario amongst the models seen so far and will probably never be actually implemented. The results obtained are shown in Figures 6.2 and 6.2. From the histograms it looks like the GA did have some trouble negotiating the solution space for this particular architecture model.

From Figure 6.2 it can be seen that in the case of the model using embedded copies there is actually a decrease in the number of functions showing no degradation over ideal schedules when compared to AGPS. This decrease is attributed to the extreme nature of the search space which the GA seems to have found very difficult to navigate. There is, however, an increase in the number of functions present at the very next tier in the hierarchy (20%-40%) and this increase is large enough to absorb decreases in numbers all across the remaining spectrum. Once again, the number of functions showing a degradation of over 80% is unchanged.

In the case of the copy unit model, Figure 6.2 shows that, again, the GA faced problems
Figure 6.11: Distribution of the degradation in II in the case of an embedded copy model, 8 wide with 2 FUs per cluster

Figure 6.12: Distribution of the degradation in II in the case of a copy unit model, 8 wide with 2 FUs per cluster
when overcoming barriers in its path. It can be seen that the GA was able to match AGPS in performance in relation to the number of loops showing no degradation. However, there is a significant increase in the number of functions showing a 60%-80% degradation in performance. At the same time, there is a decrease in the number of functions showing a degradation of more than 80% showing that at least some of the worse performing loops might actually have some room for improvement.

Before an analysis of the results presented so far in this section is done and conclusions drawn it is important to understand and acknowledge a very important factor limiting the success of the GA. In section 6 a set of ground rules which needed to be fulfilled before a genetic algorithm could be termed as entirely successful were established and laid out, at least in the case of whole program compilation. Amongst them was the observation that in the case of the clustering problem, the search space is highly volatile and therefore every program has its own particular set of parameters which contribute to the GA’s success. There is a parameter combination at which the GA is able to obtain the best overall results. However, for individual best performance numbers the parameters varied wildly and the GA needed to be fine tuned on a per program basis. The results that have been presented in this section have been obtained by setting the GA parameters to the “overall best solution generator” values. This means that on the whole when using this particular set of parameters the GA will generate results that are, as a set, better than those generated using any other parameter set when applied to the entire test suite. The results presented in this section, however, are not a collection of the best individual results. In simpler words, the results obtained here, in the case of compilation for software pipelined fortran loops, are not an
upper bound on what to expect from such intensely parallel code. However, looking at the
success seen when using this “overall best” parameter setting in Section 6 it can be said that
the results presented in this section will be within a reasonable tolerance level of a collection
of the individual best results that might ever be obtained.

Based on the assertions made so far, it is now possible to evaluate the performance of
Hiser’s AGPS when compiling for intensely parallel code. Looking at all of the histograms
presented in this section it can be seen that AGPS has been fairly successful in generating
some pretty good quality solutions. The improvement the GA was able to obtain was fairly
small in most of the cases. In the cases where the GA did improve over AGPS it was at the
lower end of the spectrum i.e. loops with greater than 80% degradation. These improved
loops, however, appear to be able to move only a little bit up the hierarchy, mostly to the
region showing between 40%-60% degradation over ideal schedules.

A very interesting picture emerges when the results just presented are seen in the light of
the conclusions drawn in Section 6. In that case it was seen that there was still ample scope
for improvement over AGPS, especially when it came to using the embedded copy model.
In the case of the Fortran loops however, all of this improvement opportunity just seems
to have vanished. As a matter of fact in most of the cases where improvement is seen it
seems to be more impressive in the case of the copy unit model. This is an understandable
condition when the conclusion drawn in Section 6 that the embedded copy model generated
more improvement in the case of whole program compilation because it was able to utilize
free slots available in the schedule to spread instructions out across the register banks is
considered. Since the Fortran loops tested here are software pipelined, this supply of free

95
<table>
<thead>
<tr>
<th>Machine Model</th>
<th>Embedded Copies</th>
<th>Copy Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 clusters, 8 FUs each</td>
<td>16.7</td>
<td>32.4</td>
</tr>
<tr>
<td>4 clusters, 4 FUs each</td>
<td>28.2</td>
<td>22.8</td>
</tr>
<tr>
<td>8 clusters, 2 FUs each</td>
<td>46.5</td>
<td>31.5</td>
</tr>
</tbody>
</table>

Table 6.6: Comparison of the average percentage degradation in II over the II in the case of the Ideal Architecture

slots to schedule instructions and copies in could be expected to dry up if AGPS was able to utilize all of the remaining parallelism available in the code. From the results presented in this section it can be concluded that this does seem to be happening in the case of the embedded copy model and so the schedules generated by AGPS are probably very close to optimal in the case of intensely parallel code.

In the case of the copy unit model there does seem to be a little bit of improvement possible over AGPS. Again this improvement is mostly seen in the area of the worst degradation candidates and it appears to be moving only a little bit up the degradation hierarchy. This is a significant observation because it goes to show that again AGPS was able to generate some very impressive schedules and in the cases where it missed out, in any case the improvement likely to be generated is not very large.

Table 6.6 compares the performance of the GA and AGPS by displaying the average percentage degradation in II over the Ideal schedules. As can be seen, no major improvements are visible. In the case of the architecture consisting of 4 clusters with 4 FUs per cluster which is most likely to represent the next generation of clustered architectures there is an about three percent degradation difference improvement visible in comparison to AGPS for
the embedded copy model when using the GA. In the case of the copy unit model again little improvement can be seen in the case of this architecture. This kind of a situation is visible in the case of the other two tested architectures too. Overall this fact underlines the trend that AGPS appears to be performing in a satisfactory manner when used in the context of intensely parallel code.

Another important aspect that comes out when the sets of results obtained here are compared is that the presence or absence of operand copying ability virtually makes or breaks the ability of the machine model to exploit parallelism. In the absence of a significant copy unit availability the level to which parallelism can be exploited ends up being limited. However, architectural limitations notwithstanding, AGPS does seem to be performing very impressively when allocating clusters to operands in the case where the available parallelism in the code being partitioned is high.
Chapter 7

Conclusions and Future Work

This thesis presented the results of an experiment conducted in order to gauge the performance of an existing, state-of-the-art, register partitioning scheme. In the process, this study also unravelled some very interesting aspects of compiling for architectures with partitioned register banks and also indicated the effectiveness of using genetic algorithms to solve the clustering problems.

The results obtained show that inspite of AGPS being one of the best known partitioning schemes today the genetic algorithm was able to generate some very impressive improvements in performance when compiling for whole programs. The average improvement relative to AGPS that was seen was of the order of 28% for the model employing embedded copies and around 23% for the model having dedicated copy units. Also shown was the fact that this scale of improvement was made possible because Hiser’s AGPS, in some instances, failed to spread instructions out across comparatively less dense clusters. The results also show that when compiling for whole programs, the embedded copy model, on account of the presence
of a large number of holes in the schedule is actually able to leverage the freely available functional units to spread instructions across clusters and is able to perform at par with the copy unit model. In comparison, when using the copy unit model, the algorithm is hampered by its limited copy unit availability and is thus not able to achieve major improvements in performance over AGPS.

However, when compiling for code having a higher level of parallelism it was found that AGPS was highly successful in generating high quality schedules. The GA was unable to achieve any major improvement in performance. The fact that AGPS, which is a greedy scheme, was able to achieve such an impressive performance in the case of densely parallel code shows its versatility in exploiting the available parallelism.

In addition, also provided in this document is a detailed description dealing with the operation of the Genetic Algorithm that was used. In the process of gathering results dealing with the performance of AGPS, other extremely important data dealing with the progression of this particular modern heuristic was found and documented. Using this new data it has been possible to stipulate a framework of ground rules that could be useful when using genetic algorithms to solve other similarly complex problems.

The results obtained also go on to show that a GA is quite successful when it comes to adapting itself to a highly complex and volatile search space. They also show that when seeded with a good quality approximate solution, the GA is capable of converging to a solution which might be close to an optimal solution in a fairly short generation span. At the same time, the readings show that the GA has to be fine tuned on an individual basis before the best quality solution can be obtained.
In the future it might be interesting to see if it is possible to devise a heuristic that is as successful as AGPS in the realm of compiling for intensely parallel code and is still successful at extracting all of the available parallelism when it comes to dealing with code where the available parallelism is itself limited. The results presented in this document also show that limiting the number of copy units usually limits the exploitation of the parallelism available in the program. So, it might be very interesting to see the effects of an increase in the copying abilities of the test models used during the course of this investigation.

Overall, it has been shown that a genetic algorithm can be adapted to help evaluate heuristics dealing with problems as complex as the clustering problem and having search spaces that are as volatile as in the field of compilation. The cost of this achievement is quite high since search space volatility dictates a fine tuning requirement on a per test case basis. However, inspite of this cost it is still heartening to know that it is feasible to evaluate the true performance levels of such heuristics by looking at only a limited number of solutions from the entire solution set.

Finally, the analysis done as part of this work shows that the Advanced Greedy Partitioning Scheme performs very impressively when it comes to compiling for software pipelined loops, that is, code which has a high degree of parallelism in it. It however does need some improvement when it comes to extracting parallelism from code which does not contain loops. This shows the state of clustering as it exists today and points to areas that could be focused on when future efforts are directed towards solving the clustering problem.
Bibliography


