Genetic Algorithms and Instruction Scheduling

Steven J. Beaty *
Department of Mechanical Engineering
Colorado State University
Fort Collins, Colorado, 80523
beaty@longs.lance.colostate.edu

Abstract

Many difficulties are encountered when developing an instruction scheduler to produce efficacious code for multiple architectures. Heuristic-based methods were found to produce disappointing results; indeed the goals of validity and length compete. This lead to the introduction of another method to search the solution space of valid schedules: genetic algorithms. Their application to this domain proved fruitful.

1 Introduction

This paper discusses some of the breadth of the work reported in depth in [Bea91]. Parts of the verbiage are common. It also extends the information contained in [BWJ90]

Instruction scheduling involves choosing, from the large solution set of possible concurrent instructions, one packing that hopefully reduces both the execution time and the space of the program. The solution space may be viewed as an incomplete n-dimensional hypercube, where n is the number of operations to be performed. Each operation can be executed at a variety of locations in the code, and each dimension represents the range of instructions that operation can be placed.

Most existing instruction scheduling methods rely on heuristics to remove the examination of parts of the search space that appear fruitless. Using heuristics can be difficult when attempting to arrive at an efficient yet efficacious compactor. This difficulty is compounded by several factors. The heuristics generally must be regenerated for each machine targeted. The heuristics themselves are not in a form easily understood by humans, thus making it difficult for humans to correctly guess and modify a compactor's behavior. It is also possible that the heuristics do not address an issue that has great import on the final code. Heuristics that work well for one ordering of MOs may not work well for another. Heuristics are also picked before the execution of the instruction scheduling routine and remain static throughout. They have no ability to learn from previous runs or to take advantage of anomalous situations existing in specific instruction scheduling situations that lead to shorter code sequences. It would be desirable to use a search technique that does not involve heuristics yet provided a robust and efficient examination of the problem space.

*The author is currently employed by NCR Microelectronics.

1.1 DDDs and Orders

The instruction scheduling problem is often represented using data dependence dags (DDDs.) Nodes in the DDD contain information about the operations that must be performed; the edges constrain the order of execution. Edges provide a *partial order* on the nodes such that an edge between nodes specifies when nodes can execute relative to each other.

Given a partially ordered dag, a question that arises is determining the *total orders* consistent with the partial order. That is, to embed the partial order in a linear order, i.e., to arrange the objects into a linear sequence $a_1, a_2, \ldots, a_n$ such that whenever $a_j \prec a_k$ we have $j < k$ [Knu73]. A method for producing this result is called a *topological sort*. Hecht [Hec77] gives an algorithm for topologically sorting a dag.

A desirable property of the topological sort algorithm is that the operation is *possible* for every partial ordering. This means it will always produce a total order, of the possibly many available, given a partial order. Topological sorting is one method of creating total orders from partial orders, but certainly not the only method.

The number of edges in a dag are a concern because it limits the number of different possible total orders. A lower and upper bound can be calculated to demonstrate this. In a completely inter-connected dag, the number of different possible orderings is

$$(D(N,E)) = \prod_{l=0}^{l<=levels} (\#N \in l)!$$

The *level* of a node in a dag is defined to be length of the longest path from the roots to the node. This formula can be derived by observing that all nodes at level l must be chosen before any nodes in level $l + 1$. The number of different orderings at any level is the number of permutations for the nodes at that level. This results in a lower bound for a dag. The upper bound may be calculated by using a completely unconnected graph. The function is then simply

$$(D(N, E)) = N!$$

This is simply the number of permutations of all the nodes. This is the upper bound for a dag and represents the ultimate in flexibility. As the number of orderings increase, the number of different final schedules increase, allowing a scheduler more opportunities to create good schedules. A difficulty with reducing the number of edges is the resultant increase in the size of the search space.

Recent work by Brightwell and Winkler [BW90] has shown that determining the actual number of total orders in a dag, given a partial ordering, is #P–complete. That is, as Garey and Johnson state [GJ79], the problem is at least as hard as finding all the Hamiltonian circuits that exist in a graph. #P–complete enumeration problems are thought to be "harder" than their corresponding NP–complete existence problems. For example, if P=NP, and it could be shown in polynomial time that an arbitrary graph *contains* a Hamiltonian circuit, it is not apparent that this would provide a polynomial time method of knowing *how many* Hamiltonian circuits exist.

2 Genetic Algorithms

Realistic scheduling problems are difficult to represent using traditional mathematical techniques. As a result, more traditional optimization methods are difficult to apply. GAs are capable of searching ill-structured spaces and also provide a global method of search [Gol89, Hol75].

Genetic algorithms have been recently applied to three areas of interest with good result: the traveling salesperson problem (TSP), job-shop scheduling, and flow shop scheduling [WSF89, WSS90, CS89, SMM$^+$91, Sys90]. High quality solutions have been found for each of these problems. The results are not based upon heuristics or local optimization information. As is common with GAs, a method of ranking the current population is required. In these problems, this is a simple task of summation to find the length of each member of the population. This task is, if anything, easier for instruction scheduling because the length of scheduled code is trivial to find.

The encouraging results from the similar problems drove the use of GA's for instruction scheduling. With the increasing complexity that expressing parallelism in both the source code and in the architecture places on scheduling, and the concomitant increase in the importance of producing good code, previous methods were found too weak. The search space is certainly discontinuous, with good schedules adjacent to bad ones, so that methods of local improvement (such as hill-climbing) cannot be relied upon to find globally competitive solutions. Another important feature provided by the removal of local methods is the time reduction realized by not performing the analyses required to drive these methods. In the case of instruction scheduling, the need to analyze the DDD and produce the metrics the heuristics use is obviated. Finally, the time taken by genetic algorithms can be directly controlled. The time taken by other global search methods, such as exhaustive search, cannot be.

A distinction before some specifics. In Coffman [Cof76] the problem of *scheduling* is stated as that of placing tasks with precedence constraints on processors for execution. The precedence constraints must not be violated by the scheduler. Given a method of placing tasks in a schedule that ensures validity, the problem may be changed into one of *sequencing*. Sequencing specifies the *order* tasks are scheduled; another method generates the schedule based on that order.

2.1 Methodology

To use genetic algorithms for a particular problem domain, only a few ancillary functions must be defined. One of these is an evaluation function that ranks the fitness of a string from the population. Choosing a proper function, i.e., one that represents a string's relative worth in the population without inordinate bias, is important. For instruction scheduling, a minimization problem, the result of the evaluation function must reflect the length of the final schedule that a member of the population generates. A difficulty encountered is that not all strings will produce valid final schedules. Failures will occur when a conflict arises (i.e. timing, resource, or field) due to a string's ordering. It is not surprising that certain orders will fail to produce valid schedules for any given DDD; in fact, the impact of ordering on the production of valid schedules is emphasised in previous methods of instruction scheduling.

One possible solution is to give failing strings some predetermined "bad" (large) value. Difficulties with this method include

- all failing strings will be given the same value, no matter how close they got to producing a valid schedule, and
- the evaluation function will produce undesirable bias among the poorly performing strings, i.e., they will all appear equivalent.

This will interfere with the natural selection process of the genetic algorithm because actual performance is not reflected by the evaluation function. After consideration, the method selected performs a "worst-case" evaluation when a string fails to produce a valid schedule. This evaluation is produced by assuming all unscheduled operations have no parallelism available in them, necessitating their serial placement. The calculation of the evaluation function is then trivial; it is the number of instructions that contain operations so far, plus the length of the path containing the serial ordering of all the unscheduled operations. This produces a good estimate in the event of schedule failure; those schedules with more operations placed will receive a better evaluation. It also produces an exact evaluation in the presence of a valid schedule.

Six different recombination operators were studied. These are described in Starkweather et al. [SMM+91] and include two order crossovers, partially mapped crossover, cycle crossover, position-based crossover, and edge recombination. Starkweather et al. demonstrate that each operator will perform differently for each given problem domain. The performance difference can be measured in the speed of convergence to a good solution. For example, edge recombination finds good solutions more rapidly on the TSP while performing more poorly than the others on scheduling problems.

The conditions for stopping search must be examined. Halting is certainly predicated on finding one valid schedule, a possibly non-trivial requirement. Because this requirement is present in all forms of instruction scheduling, it is not viewed as a detriment to a GA-based approach. Another condition for stopping can be the closeness to the theoretical best for the DDD. On "easy" DDDs (with simple timings and dependencies), the theoretical best can be achieved with regularity. Therefore, this condition should be checked to determine the credibility of continuing the search. Another bound easily placed is the number of recombinations performed. Time constraints certainly can be used to generate a value for this bound, unfortunately, a direct relationship does not exist between time spent and the quality of the resulting schedule; an optimal schedule may be found in the first generation, or not found after innumerable generations. Such is the case with all non-deterministic optimizers; unless a solution matches a known theoretical best, a stopping condition cannot be reliably specified [1]. Genetic algorithms will generate solutions competitive with all areas of the solution space searched. Increasing the number of recombinations, and to a lesser degree the population size, will tend to produce "more optimal" solutions. This provides motivation to perform as many recombinations as possible.

The number of generations should therefore be related to the relative difficulty of producing an optimal schedule for a given DDD. DDDs with few simple operations do not require as many generations to find good schedules as do those with many complex operations. Basing the number of generations on a low-order polynomial has proven effective. The size of the population is based on a different lower-order polynomial. The homogeneity of the population is an indirect indicator of convergence. If all solutions are similar, either a competitive solution has been found or unvisited areas of the search space contain better answers. In the first case, the search may be halted. In the second case, mutation should be used to expand the area of the search space covered. It is difficult to differentiate these two conditions.

Two genetic approaches to the instruction scheduling problem are outlined in the following sections. Both are based on the manipulation of strings of non-repeating integers. This representation is consistent with those used in the TSP and shop scheduling problems previously mentioned. All populations are randomly initialized. For each member of the initial population, the evaluation function must be executed in order to create a sorted gene pool.

It is possible to vary various parameters of a genetic algorithm to increase its effectiveness on a particular problem although no such effort was made in these studies. The selection bias was 1.5. There was no mutation, adaptive or otherwise. Each population was initialized with different (random) values.

2.2 With List Scheduling

The first approach used a genetic algorithm in concert with a list scheduler. Several immediate benefits accrued due to this method:

1. the use of existing list scheduling technology (requiring no additional scheduling code),
2. the use of an already-understood scheduling method, and
3. the use of a method similar to both Syswerda's job-shop scheduling and Starkweather et al.'s brewery scheduling approaches.

These benefits allowed comparisons to be made against existing results, both to verify the validity and efficacy of the resultant schedules.

The strings in the population represented the priority ordering of the nodes in the DDD. As nodes in ROCKET's [2] DDDs are numbered consecutively from $1 \ldots n$, where n is the number of nodes, a node number at a given place in the string has priority over all that appear later. This information is used to pick which node, of all those possible in the data ready list, will be placed next. This removes all heuristic judgements based on node attributes.

Most of the schedules produced were valid. This is attributed to the power of the implicit heuristic of list scheduling, placing only data ready nodes. In the present implementation, integrating the GENITOR genetic algorithm

[1] Remember that theoretical best and optimal are not the same. The former specifies the best without the presence of any additional constraints, the latter considers those extra constraints.

[2] The name of our current compiler.

[WK88, Whi89, WS90] routines took less time than tuning the list scheduling heuristics in order to achieve a high degree of reliability in the production of valid schedules. Each machine targeted requires the re-tuning of these heuristics, replicating this time spent. Using a genetic algorithm, nothing needs modification to target to another architecture. This can greatly increase the flexibility of an instruction scheduler, useful both for pre-production performance studies where architectural features may be changed and their impact studied, and where the production of code for many disparate machines is desired.

This combined list scheduling/genetic algorithm performed well. In simple DDDs, it easily found solutions as good as list scheduling alone. These were instances where timing was flexible, and placement order was basically irrelevant. The combination also found some new best-known solutions to difficult DDDs. While this may appear surprising, consider the great lengths required to generate a set of heuristics that would usually produce valid schedules. The heuristic with most impact on final schedule length, critical path, has been shown to produce erroneous results in simple cases. Other heuristics, such as counting the number of restricted edges a node has, must be emphasised in an attempt to assure schedule validity. The emphasis therefore shifts (correctly) from the possibility of generating shorter schedules to the probability of producing valid schedules. The use of GAs tends to place nodes in an order that produces shorter results, and with an evaluation function that reflects failure as a longer result, it will place nodes in an order that also produces valid results.

A detraction from these encouraging results is the time complexity of the combined algorithm. As shown before, list scheduling is at least $O(n^2)$, and repeatedly performing this operation to evaluate a given string increases the time complexity. If the number of generations is linear with respect to the number of nodes in the DDD, the complexity becomes $O(n^3)$. Increasing the population size also results in similar complexity increases.

2.3 Without List Scheduling

The time complexity of the GA-based list scheduling method motivated an exploration into other forms of scheduling. No existing methods had less complexity and fit within the framework developed. The goal became the placement of operations in less than $O(n^2)$ complexity. It was noticed that, with the application of the absolute timing algorithm, each node already "knew" approximately where in the final schedule it must be placed. If the order of placement could be performed intelligently, there was no reason for a top-down (list scheduling) priority in node placement. Top-down priority had been used as a pseudo-intelligent form of ordering; it cannot adapt to vagaries of individual graphs. With genetic algorithms an intelligent, adaptable method is present. The scheduling mechanism therefore became using the GA to pick the order operations are placed in the schedule. This works because each operation "knows" where it can be placed. Certainly, failures can occur due to choosing an improper order, creating an invalid schedule.

To increase the likelihood of valid schedules, limited lookahead scheduling is employed. Here, "limited" denotes lookahead packing only those nodes with $(op) = (n, n)$ absolute timing. That is, only nodes with no choice in their placement, due to the placement of another by the GA, are scheduled. Nodes with $(op) = (n, m)$ absolute timing are ignored by lookahead. This retains the greatest amount of flexibility and problem knowledge for the genetic algorithm to work with. If the GA places nodes toward the end of the DDD first, lookahead scheduling may do most of the work in scheduling the DDD. This is not a detriment to the process as it only properly reflects the affects of the GA choice of node-placement order. As a by-product, this lookahead effort decreases the time spent by the entire scheduling process, as the GA is required to examine fewer nodes for placement. Most of the time spent scheduling then resides in the absolute timing and lookahead algorithms.

A difficulty with this method is the increased potential of generating invalid schedules. As node placement order is critical for success in creating a valid final schedule, the original random generation of orders the genetic algorithm provides tend to fail often. Several methods were tried to increase the number of valid schedules generated:

1. increasing the number of generations,
2. increasing the population size, and
3. seeding the initial population with a known good order. This order may be generated with another form of scheduling.

All three of these methods produced an increase of valid schedules for difficult-to-schedule DDDs. The first method provides the genetic operator more diverse material upon which to operate. The second enlarges the area of the search space represented in the initial population. The third refines an initially correct schedule while also searching for better solutions.

Genetic operators emphasizing order converged faster than those emphasizing adjacency. This comes as no surprise; all previously effective methods also emphasized order. This evidence does however shed additional light on the nature of of the instruction scheduling process by providing more controlled, empirical evidence. The ordering of the placement of nodes by the genetic algorithm mirrors the approach used by human coders. The nodes with the greatest impact on final schedule length are placed first, with those having lesser impact placed later. The order of placement that ensures validity is also reflected.

By removing list scheduling, with its associated execution time and FCFS ordering, more schedules can be generated in a given amount of time. This provides for a more thorough search of the problem space, given a similar amount of time. Given the facts that the quality of the solutions generated is likely enhanced, and schedule validity is assured, this approach bears more fruit than the combination of GAs and list scheduling.

3 Summary

The desire for efficient and efficacious scheduling of instructions motivated an effort to understand the search space of the problem. With a better understanding of the nature of the problem, genetic algorithms were applied to more effectively search the space of valid solutions. This application proved to give excellent results. The lack of retargeting adds to the scheduler's usefulness in research (and quick-turn development) settings. The ability to rapidly generate excellent code by thorough search extends this usefulness to areas requiring such code.

References

[AM88] V.H. Allan and R.A. Mueller. "Microcode compaction with general synchronous timing". *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, 14(5):595–599, May 1988.

[Ban88] U Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, Ma, 1988.

[Bea91] S.J. Beaty. *A Study of Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[BSKT79] U. Banerjee, S. Shen, D.J. Kuck, and R.A. Towle. "Time and parallel processor bounds for fortran-like loops". *IEEE Transactions on Computers*, C-28(9):660–670, Sep 1979.

[BW90] Graham Brightwell and Peter Winkler. "Counting linear extensions is #p-complete". DIMACS Technical Report 90-49, Bellcore, 445 South Street, Morristown, New Jersey, 07960, July 1990.

[BWJ90] S. Beaty, D. Whitley, and G. Johnson. "Motivation and framework for using genetic alogorithms for microcode compaction". In *Proceedings of the 23th Microprogramming Workshop (MICRO-23)*, Orlando, FL, November 1990.

[Cof76] E.G Coffman. *Computer and Job-Shop Scheduling Theory*. Jon Wiley & Sons, New York, 1976.

[CS89] Gary A. Cleveland and Stephen F. Smith. "Using genetic algorithms to schedule flow shop releases". In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. W.H. Freeman and Company, San Francisco, CA, 1979.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, NY, 1977.

[Hol75] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms. Addison-Wesley, Reading, MA, second edition, 1973.

[LDSM80] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallett. "Local microcode compaction techniques". *ACM Computing Surveys*, 12(3):261–294, September 1980.

[PKL80] D.A. Padua, D.J. Kuck, and D.H. Lawrie. "High speed multiprocessors and compilation techniques". *IEEE Transactions on Computers*, C-29(9):763–776, Sept 1980.

[PW86] D.A. Padua and M.J. Wolfe. "Advanced compiler optimizations for supercomputers". *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[SB90] P. Sweany and S. Beaty. "Post-compaction register assignment in a retargetable compiler". In *Proceedings of the 23th Microprogramming Workshop (MICRO-23)*, Orlando, FL, November 1990.

[SDX87] B. Su, S. Ding, and J. Xia. "Microcode Compaction with Timing Constraints". In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, Colorado Springs, CO, December 1987.

[SMM$^+$91] T. Starkweather, S. McDaniel, K. Mathias, C. Whitley, and D. Whitley. "A comparison of genetic sequencing operators”. In *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

[Sys90] Gilbert Syswerda. “Schedule optimization using genetic algorithms”. In L. Davis, editor, *The Genetic Algorithms Handbook*. 1990.

[Veg82] S.R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[Whi89] Darrell Whitley. “The GENITOR algorithm and selective pressure: Why rank - based allocation of reproductive trials is best”. In *Proceeding of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[WK88] D. Whitley and J. Kauth. “Genitor: a different genetic algorithm”. In *Proceeding of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130. Denver, Co., 1988.

[WS90] Darrell Whitley and Tim Starkweather. “GENITOR II: A distributed genitic algorithm”. *In Press: Journal of Theoretical and Experimental Artificial Intelligence*, 1990.

[WSF89] D. Whitley, T. Starkweather, and D. Fuquay. “Scheduling problems and traveling salemen: The genetic edge recombination operator”. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[WSS90] D. Whitley, T. Starkweather, and D. Shaner. “The traveling saleman and sequence scheduling quality solution using genetic edge recombination”. In L. Davis, editor, *The Genetic Algorithms Handbook*. 1990.