Using Genetic Algorithms to Fine-Tune Instruction-Scheduling Heuristics

Steven J. Beaty
Hewlett-Packard
3404 East Harmony Road
Fort Collins, Colorado, 80525
Phone: (970) 229-6480
beaty@fc.hp.com

Scott Colcord
Computer Science Department
Michigan Technological University
1400 Townsend Drive
Houghton MI 49931-1295
Phone: (906) 487-2209
sacolcor@cs.mtu.edu

Philip H. Sweany
Computer Science Department,
Michigan Technological University,
1400 Townsend Drive
Houghton MI 49931-1295
Phone: (906) 487-2209
sweany@cs.mtu.edu

Abstract

Instruction scheduling is an NP-complete problem that involves finding the fastest sequence of machine instructions from an abstract program representation. List scheduling is a method often used for instruction scheduling when producing code for instruction-level parallel processors and can produce excellent results when appropriate heuristics are chosen. We have investigated a method of experimentally determining good scheduling heuristics and found that it does indeed provide an easy way to tune instruction-scheduling heuristics.

1 Introduction

Computer manufacturers are continually striving to make faster computers with a combination of faster circuitry and increasing the amount of simultaneous computation (parallelism) in their architectures. One popular method of increasing the degree of simultaneous computation is instruction-level parallelism (ILP.) ILP computers exploit the implicit parallelism that most programs contain [25]. They overlap the execution of operations[1] that do not depend on one another. For example, a memory address can be calculated while the value is computed to store there. Today, typical ILP processors have a memory address, an integer, and several floating point computational units. Future processors will have more of each of these. While high-performance architectures have included some ILP for at least 25 years [20], recent computer designs have exploited ILP to a larger degree. This trend shows no sign of reversing. Indeed, as most of today's multiprocessors are built with individual nodes exhibiting significant ILP, multiprocessor systems need to make efficient use of ILP hardware. Effective use of ILP hardware requires that the instruction stream be ordered such that, whenever possible, multiple low-level operations can be in execution simultaneously. This ordering of machine operations to effectively use an ILP architecture's parallelism is typically called *instruction scheduling* (IS.)

While instruction scheduling has been shown to be an NP-complete problem [8] practitioners have long achieved good results in polynomial time using *list scheduling*. List scheduling (LS) derives its name from the fact that a list of data-ready nodes (those with no unscheduled predecessors) is maintained. The essence of LS is to heuristically determine which data-ready node to schedule next. Many LS implementations use a weighted sum of key scheduling parameters to assign a priority to each data-ready node. This pri-

[1] We define an *operation* as an atomic computational function, such as an add, multiply, or memory access. An *instruction* is an abstract representation of the operations that can be issued during a single machine cycle. An instruction might contain more than one operation, and the operation(s) may not be performed in the order they are presented in the instruction.

ority is then used to order data-ready nodes. Allan [1] calls such a method *discriminating polynomial selection.* She suggest that the weights to be used for each term of the polynomial be experimentally determined. It is our contention that using a genetic algorithm to search the space of possible heuristic weights allows the weights to be easily tuned to the specific needs of a scheduler/architecture pair.

Genetic algorithms (GAs) provide a robust method to search an exponential search space. GAs rely on manipulation of populations of strings analogous to biological operations on chromosomes. Genetic Algorithms have been used successfully to perform the traveling salesperson problem [29, 30], job shop [30], and flow shop [5, 21] optimization problems. Encouraging results from these problems have led us to postulate that GAs are an effective technique for optimizing the heuristic weights used in list scheduling's discriminating polynomial selection.

To evaluate our contention the GAs can be useful in experimentally determining the weights for LS's discriminating polynomial selection, we have used the Rocket compiler [23, 2, 22]. Rocket is a highly-optimizing compiler retargetable for ILP architectures which uses descriminative polynomial selection to order nodes in list scheduling's data-ready list.

The remainder of this paper first describes instruction scheduling in Section 2 before discussing genetic algorithms in Section 3. In Section 4 we turn our attention to our experimental method for using GAs to determine appropriate heuristic weights for list scheduling and then discuss our experimental results in Section 5.

2 Instruction Scheduling

Instruction scheduling involves the placement of atomic machine operations into machine instructions. A DDD (data dependence directed acyclic graph (DAG)) is often used to describe the necessary operations and their order. The nodes in a DDD contain the operations, and the arcs denote a partial order on the nodes. This partial order is used to guarantee program semantics.

2.1 Local Scheduling

List scheduling is a general scheduling method [6] often used for both instruction scheduling (IS) [16] and processor scheduling (PS) [26]. LS builds a data-ready list that contains all jobs that are not waiting on the results of another job. This ready list is then heuristically ordered, and the highest priority node is placed in the final schedule. This process repeats until there are no more nodes to place. In using a data-ready list, LS performs a topological sort of the DDD, thereby reducing the search space of the scheduling problem and increasing the chances of finding a valid schedule.

Of course to achieve efficient schedules, LS needs to have useful heuristics to order data-ready nodes so that the "most important" nodes are placed first. While researchers have extolled the virtues of determining these heuristics experimentally, most list schedulers seem to rely on ad hoc techniques for choosing heuristics. Allan and Mueller's discriminative polynomial selection technique provides a framework to combine several heuristics into a single heuristic by assigning weights to each individual heuristic term and summing the weighted terms. The key then is to determine what weights yield the most efficient schedules for a wide range of programs and architectures.

2.2 Global Scheduling

A considerable body of research has shown that exploitation of significant amount of ILP requires global scheduling techniques [24, 18, 25]. Several global scheduling techniques make use of a local scheduler, however, to reorder code for meta-blocks. A meta-block is a group of basic blocks [2] that a global scheduler treats as a single block. Trace scheduling [11, 9] is an example of such a global scheduler. It attempts to optimize the most frequently-executed paths of a control flow graph (CFG), possibly at the expense of the less frequently-executed paths. Blocks that are included in the frequently-executed paths are called *on-trace* and those in the less frequently-traveled paths are called *off-trace.* The basic idea is to use a local scheduling algorithm to move operations between blocks in an on-trace path to reduce the number of instructions executed within that trace. Other global scheduling algorithms that similarly rely on a local scheduler to reorder code in meta-blocks include Bernstein's Global Scheduling [3], Hwu's Sentinel Scheduling [17], and Sweany's Dominator-Path Scheduling [23]. Such use of a local scheduler on DDDs that contain control flow complicates the choice of appropriate discriminating polynomial weights and might well require different weights for meta-block scheduling than for basic block scheduling in order to achieve best results.

3 Genetic Algorithms

Genetic algorithms (GAs) manipulate populations of strings that represent the parameterization of the optimization problem. The strings correspond to chromosomes or genotypes in biological terms. There is a mapping from this representation to the phenotype of the actual solution. GAs use a form of selective pressure to encourage over-achieving

[2] A basic block is a straight-line sequence of instructions with single entrance and exit points which may contain a branch only as the last instruction in the block.

and discourage under-achieving strings in the population. A string's chances of reproducing correspond to its performance in the current environment. This is an easily understandable method and it produces robust searches of difficult parameter spaces as demonstrated by Holland and others [13, 7, 12].

Parameters are usually encoded into some form of binary representation. This representation is then used for subsequent operations and evaluations. Consider the string:

```
1100101001110110001
```

This could represent an integer, a fixed or floating point real, or any other relevant model of the parameters to be optimized. Multiple parameters are simply appended together. The initial population is usually generated by creating random strings.

To perform recombination, the basis for most genetic adaptation in nature, consider also the string:

```
xyyxxyxyxyxxxyyyxxy
```

(with `x` for `0` and `y` for `1`) and some number of break points. The genetic material from one string is then swapped between those break points with the corresponding material from the other. An example with two break points is:

```
11001 \/ 01001110110 \/ 001
xyyxx /\ yxyxyxxxyyy /\ xxy
```

resulting in the two children:

```
11001yxyxyxxxyyy001
xyyxx01001110110xxy
```

Although a single break point is usually used in discussions of GAs, two have been empirically shown by Booker [4] to produce better results.

Another operation in the reformation of strings is mutation. This is accomplished by randomly toggling some of the bits in the offspring. This creates genetic diversity. It has been found, in the general case, that mutation rates should be kept low (less than 5%) for best exploitation and least disruption of the information present.

In standard GAs, all the strings in the population are reformed during a generation. Parents are crossed on the basis of their performance in comparison to the average fitness of the population and mutation is allowed to occur on the offspring. The selective pressure is provided by the fitness measure; the differential need not be great to achieve good results. Both selective pressure and initial population sizes may be tuned to match the problem space. The type of crossover and rate of mutation needs to be selected based on the problem type.

To relate the encoding with the sampling of hyperspace, consider a string of length three. With this we get the ability to represent a three-dimensional hypercube. The string `011` represents a corner of the hypercube. Edges have one of the bits as a "don't care", i.e. `01*`. Faces have two "don't cares": i.e. `0**`. The entire space can be expressed by a complete "don't care string": i.e. `***`. Strings that contain a "don't care" in some position are termed schemata. In general, each binary encoding corresponds to one corner in the hypercube and samples $2^L - 1$ different hyperplanes in the search space where L is the length of the binary encoding. This is the idea of "intrinsic parallelism" whereby one string samples the productivity of many hyperplanes [13]. The schema theory indicates that individual hyperplanes will increase or decrease their representation in a population based upon their relative fitness in that population when reproduction and recombination are applied.

The more diverse the original population, the more global the search. The search does not avoid or escape from local minima; it does a global search where local minima are ignored in favor of higher-valued strings. If a local minima is found to be best, it will tend to be competitive with all areas of the space searched. It has been shown that if an area in hyperspace has above average performance and is sampled by a schema in the population, that area's representation will increase within the population. It has been calculated that for the processing of N structures per generation approximately N^3 schemata are sampled (intrinsic parallelism).

The ability to sustain search is dependent upon the genetic diversity in the population. When a population lacks diversity, new areas of the space are not examined. Mutation can be used to drive the search into these unexamined areas. However, a fixed level of mutation has been shown to disrupt the search early and then fail to provide enough diversity in the later stages. Thus, adaptive mutation increases the mutation rate based on the homogeneity of the population and gives better performance.

The GENITOR GA program, developed by Whitley [27, 28], has some differences with "standard" GAs that appear to increase performance. It does not replace the entire population with each generation. Instead it probabilistically chooses two parents to reform into two offspring. Recombination and mutation occur, then one of the offspring is discarded randomly. The remaining offspring is placed in the population according to its fitness in relation to the rest of the strings. The lowest-valued string is discarded. This keeps high-valued strings within the population, directly accumulating high-performance hyperplanes. It also bases the reproductive opportunity upon rank with the population, not upon a string's fitness value in comparison with the average of the population, reducing the impact of selective pressure fluctuation. It also reduces the importance of choosing a proper evaluation function for fitness in that the difference in the fitness function between two adjacent strings is irrelevant.

Target	Local	Global
URM	110	95
RS6000	119	95
I860	110	33

Table 1: Number of Data Dependence DAGs Used for Evaluation

4 Experimental Method

To evaluate our contention that GAs are a good way to tune LS heuristics, we've taken the terms of the discriminating polynomial used in the Rocket compiler and used a GA to optimize the weights on those terms. Note that positive terms in the polynomial denote that a heuristic is useful in creating a efficacious schedule, while negative terms denote a heuristic that hurts the final schedule. Section 4.1 lists the 25 terms available in Rocket's list scheduler.

In addition to experimentally finding good heuristic weights for LS, we wanted to compare heuristic weights for differing machines and scheduling methods to determine the weights useful for different scheduling contexts. As Rocket is a retargetable compiler, we ran GAs for each of three different architectures: the IBM RISC RS6000 [14], a computer based on an Intel i860 chip [15], and a hypothetical ILP machine called the Unlimited Resource Machine (URM) developed at Michigan Technological University [19]. We made use of both Rocket's local and global instruction scheduling (dominator-path scheduling.) This allowed us to determine heuristic weights for both local and global scheduling and compare the two.

In our GA, we chose a population size of 1000 192-bit strings and ran for 2500 generations. We treated each group of 8 bits within the 192-bit strings as an integer weight (-128 to 127) and mapped the bit strings directly to an array of 24 term weights used in Rocket. To evaluate strings, we instruction scheduled a representative number of data dependence DAGs for each architecture tested. Table 1 shows the number of DDDs evaluated for each combination of architecture and scheduling method. To build the test suites of DDDs, examples were taken from a mixture of scientific and system code. Of course scheduling DDDs to measure the fitness of strings generated by Genitor required a considerable time. Typically, each run required two to three days to complete on a Sparc10 workstation. It is our contention, however, that investing two or three days of computer time to fine tune a list scheduling heuristic to a particular combination of architecture and scheduling method is time well spent.

4.1 Discriminating Polynomial Terms

We have investigated many heuristics in attempts to achieve valid schedules for a variety of architectures. Typically, considerable testing goes into choosing heuristics for a particular target, but some heuristics, such as critical path are almost always used.

Here is a list of discriminating polynomial terms we have found useful for some architecture we have targeted:

1. `height` – the maximum number of arcs from this node to any sink node.
2. `on_critical_path` – the operation is on the longest path in the DDD.
3. `on_schedule_critical_path`. – the operation is on the longest path in the DDD when operation timings are considered.
4. `lexical_order` – ordering of nodes from source. Fisher [10] suggests that program lexical order is not a good metric for list scheduling priority, but it can be used for non-ILP architectures to produce a default ordering.
5. `branch_node` – the node is a branch node, especially useful in the presence of delayed, restricted branching mechanisms. This has been used to increase the chance a branch node will be placed before other operations.
6. `resource_usage_of_this_type` – the amount of use of this node's resource in this DDD. The more contention for resources, the earlier a node should be placed in order to free the resource as soon as possible for reuse.
7. `used_and_defined_resources` – as above, nodes that use more resources than others should be scheduled so they do not interfere with others needing those resources.
8. `least_recently_used_resource` – a method of forming round-robin reference to resources.
9. `field_usage_of_this_type` – as with resources, try to minimize instruction field conflicts.
10. `fields_used`.
11. `least_recently_used_field`.
12. `successors` – the more successors a node has, the earlier it should be scheduled, allowing its successors to become data ready as early as possible. This exposes more parallelism to the scheduler.

13. `restricted_successors` – the more restricted successors [3]. a node has, the earlier it should be scheduled so timing is more flexible within the DDD.

14. `total_restricted_successors` – total of the timings for all restricted successors

15. `shortest_restricted_successor`

16. `distance_from_successors` – a measure of how restricted the edges to the successors are.

17. `predecessors` – the more predecessors a node has, the later it should be scheduled, allowing its predecessors to become data ready as early as possible. This exposes more parallelism to the scheduler.

18. `restricted_predecessors` – the more restricted predecessors a node has, the later it should be scheduled so timing is more flexible within the DDD.

19. `total_restricted_predecessors` – total of all restricted predecessors.

20. `shortest_restricted_predecessor`

21. `distance_from_predecessors` – a measure of how restricted the edges to the predecessors are.

22. `schedule_spread` – the number of instructions in which an operation can be placed. The greater the spread, the more flexibility for placement.

23. `average_restricted_successor_gap` – the average of $\Delta(e)$ for all the restricted successors.

24. `average_restricted_predecessor_gap` – the average of $\Delta(e)$ for all the restricted successors.

5 Experimental Results

The data obtained from GA searches for useful heuristic weights demonstrate several intesting features. For example, we found that each of the six populations was able to find strings which evaluated about 5% better than the best of the randomly generated strings in the initial population. This suggests that tuning list scheduling's heuristics may have a limited effect on the schedules generated.

We also found that the heuristic weights for well-performing strings did indeed differ considerably based upon the architecture compiled for and the scheduling method used. Using both local and global scheduling on each of three machines (URM, RS6000, i860), six populations of strings were generated to choose heuristic weights. Table 2 shows the means and standard deviations of the weights obtained from each of the 100 top performers from the three GA pools for local scheduling. Table 3 shows the analogous data for global scheduling. In each case, individual heuristics are identified by same number in which they are listed in Section 4.1. By looking at the top 100 strings from each pool after 2500 generations we can begin to determine how important each of the 24 heuristic factors is to creating efficient schedules. A large average weight, combined with a relatively small standard deviation suggests that an individual heuristic term must be important as it was included in most of the high-performance bit strings.

Tables 2 and 3 show several interesting trends. These data can be used to test the hypothesis (using confidence analysis) that the "optimal" weight is different than zero for any heuristic factor. When we do this we discover that, at the 60% confidence level, all of the 24 heuristics are different from zero for at least one combination of scheduling method and architecture. This suggests that each heuristic is indeed useful in ordering data-ready nodes for some scheduling context. However, only one heuristic, lexical_order (number 4), was found to be significantly different from zero at the 75% confidence level for all six combinations of architecture and scheduling method. This is certainly counter-intuitive as popular wisdom suggests that critical path heuristics are the most important and that lexical order is relatively unimportant. It is curious, however, that the weight for lexical order was positive in two populations (both local scheduling) and negative for the other four. Both critical path heuristics (2 and 3) in contrast, were positive in all populations, though only significantly so (at the 75% level) for five out of six populations for weighted critical path and four out of six for unweighted critical path. Note also that for local scheduling, unweighted critical path is at least as important in determining good schedules as weighted critical path, while for global scheduling the weighted critical path seemed roughly twice as important a heuristic, judging from the observed weights of the high-performance strings.

6 Conclusions

We have successfully used genetic algorithms (GAs) to determine appropriate weights for the heuristic terms used in list scheduling's discriminating polynomial. We have found that GAs allow us to tune our list scheduler to a particular architecture and scheduling method in a short amount of time.

It should not be surprising that our GA experiments showed that effective discriminating polynomial weights differed considerably from architecture to architecture as well as when moving from local to global scheduling. This

[3] a successor operation is restricted if there is a maximum time the result of the dependent operation is valid

Heuristic	URM		RS6000		I860	
Number	Weight	S.D.	Weight	S.D.	Weight	S.D.
1	-36.5	22.5	-56.4	23.7	-44.0	34.7
2	93.6	24.6	58.3	44.6	70.0	58.1
3	108.0	15.7	23.9	49.7	60.9	58.2
4	16.6	11.7	71.9	22.2	-11.5	5.95
5	-24.9	60.3	-6.73	49.5	0.38	75.1
6	-5.21	62.3	24.2	61.4	-1.92	64.5
7	-20.7	78.1	-16.1	72.2	-0.05	82.7
8	-23.3	73.5	-3.08	91.8	-8.24	67.8
9	-8.64	66.6	-24.2	84.5	17.7	78.1
10	-40.0	71.5	-1.87	71.8	-19.0	88.4
11	-1.25	81.0	17.6	55.8	3.75	55.1
12	23.1	26.1	104.9	18.3	60.9	26.0
13	-53.3	67.6	31.0	72.6	69.6	52.0
14	-32.1	70.3	37.1	60.5	72.2	39.5
15	-23.8	71.0	-27.2	72.8	9.6	71.0
16	-18.6	49.0	0.99	76.5	-0.24	79.0
17	4.82	25.1	64.3	53.6	51.5	50.8
18	3.80	65.3	19.9	75.8	22.7	74.4
19	15.7	89.2	32.0	66.6	0.72	78.7
20	19.7	69.4	1.52	86.7	-16.6	75.6
22	5.96	40.9	-11.7	67.9	8.68	63.8
22	6.88	71.5	22.3	69.3	4.40	65.5
23	-29.2	71.4	32.5	60.7	68.9	47.1
24	-11.4	84.5	28.4	66.3	-7.29	70.7

Table 2: Heuristic Weights for Local Scheduling

Heuristic	URM		RS6000		I860	
Number	Weight	S.D.	Weight	S.D.	Weight	S.D.
1	-2.46	36.3	-65.1	36.7	-34.4	66.8
2	39.4	48.7	38.4	59.5	23.4	65.6
3	72.2	32.8	60.1	54.3	78.7	53.8
4	-17.5	7.42	-6.83	4.16	-19.1	23.3
5	23.0	78.8	16.6	74.7	-1.07	72.9
6	-16.9	81.6	-4.93	71.4	-7.20	84.7
7	-4.16	77.7	31.4	77.1	-6.19	85.7
8	-12.1	72.4	-28.9	77.3	2.08	67.1
9	11.1	83.6	-3.13	72.6	-11.2	64.3
10	-15.8	65.7	-14.9	76.8	-18.4	83.3
11	-8.11	84.6	-5.00	75.9	-2.1	60.4
12	-23.6	44.0	15.0	69.7	31.0	48.0
13	74.7	47.4	68.7	52.7	60.5	51.2
14	63.1	62.6	34.8	65.0	64.6	56.1
15	-11.0	69.9	30.6	74.8	-0.95	73.6
16	-8.66	63.3	-6.07	71.5	14.2	59.8
17	-40.8	63.3	-46.5	58.7	36.4	52.1
18	-29.4	82.1	-8.05	82.5	0.25	63.0
19	1.10	71.6	-4.63	74.3	9.24	72.5
20	-0.60	64.1	-9.93	80.5	19.4	64.1
22	-11.9	72.3	-23.0	61.2	-4.94	59.1
22	-5.85	64.7	0.24	72.5	17.8	73.5
23	74.2	39.7	56.7	63.4	67.5	42.5
24	-5.11	79.5	6.81	71.2	2.62	76.3

Table 3: Heuristic Weights for Global Scheduling

suggests that, as hypothesized, tuning heuristics to a particular scheduling context is an important consideration when building an instruction scheduler. Our study of 24 different heuristic terms indicated that all were useful in building a list scheduler for at least one of the six combinations of scheduling method and architecture tested.

A number of different areas for future work is indicated from this effort. The effect that different architectures play in chosing a set of heuristics could bear much fruit. Studying the effect of the application set used to derive the heuristics would be interesting. Some of the traditional ideas about which heuristics are important are brought into question, and these questions need addressing. The differences between global and local heuristics, while not surprising, could lead to other interesting research areas.

Acknowledgments

We would like to thank the National Science Foundation for helping to fund this work through Grant CRR-9308348.

References

[1] V. H. Allan. *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

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

[3] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, June 1991. SIGPLAN '91.

[4] L. Booker. Improving search in genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 61–73. Morgan Kaufmann, 1987.

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

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

[7] K. DeJong. *An Analysis of Reproduction and Crossover in a Binary - coded Genetic Algorithm*. PhD thesis, University of Michigan, Ann Arbor, 1986.

[8] D. J. DeWitt. *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[9] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1986. PhD thesis, Yale, 1984.

[10] J. A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, Oct. 1979.

[11] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

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

[14] IBM. *IBM Journal of Research and Development*, Jan. 1990.

[15] Intel. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

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

[17] S. A. Mahlke, W. Y. Chen, W. mei W. Hwu, B. R. rishna Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *asplos5*, volume 27, pages 238–247, Boston, MA, Oct. 1992.

[18] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word A rchitectures. *IEEE Transactions on Computers*, 33(11):968–976, Nov. 1984.

[19] D. A. Poplawski. The unlimited resource machine (URM). Technical Report CS-95-01, Department of Computer Science, Michigan Technological Univers ity, Houghton, Jan. 1995.

[20] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[21] 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.

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

[23] P. H. Sweany and S. J. Beaty. Dominator-path scheduling — A global scheduling method. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25)*, pages 260–263, Portland, Oregon, Dec. 1992.

[24] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, Oct. 1970.

[25] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Forth Internation Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, Apr. 1991.

[26] T. M. Watts, M. L. Soffa, and R. Gupta. Techniques for integrating parallelizing transformations and compiler based scheduling methods. In *Supercomputing '92 Proceedings*, 1992.

[27] D. 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.

[28] D. Whitley and T. Starkweather. sENITOR II: A distributed genitic algorithm. *In Press: Journal of Theoretical and Experimental Artificial Intelligence*, 1990.

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

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