Instruction Scheduling Using Simulated Annealing

Philip H. Sweany
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
sweany@cs.mtu.edu
Phone: (906) 487-3392
FAX: (906) 487-2283

Steven J. Beaty
Metropolitan State College of Denver
Department of Mathematical and Computer Sciences
Campus Box 38, P. O. Box 173362
Denver, CO 80217-3362
beatys@mscd.edu
Phone: (303) 556-5321
FAX: (303) 556-5381

Abstract

Most nodes of modern massively-parallel computing systems contain processors that use instruction-level parallelism to increase the speed of the individual processor. In order to achieve the greatest speedup possible, the compiler must perform instruction scheduling so that instructions are presented to the processor in the order that is most efficient. Instruction scheduling is a compiler problem that, due to its NP-complete nature, requires heuristic solutions for any significant programs.

One promising stochastic search technique that shows promise in the realm of instruction scheduling is the use of simulated annealing (SA.) Simulated annealing can be thought of as modified hill-climbing that includes "occasional perturbations" in the search space under investigation to avoid the problem of getting stuck in a local minimum or maximum. The process gets its name from the fact that it closely follows the physical process of annealing which is gradual cooling of a liquid until it solidifies.

We implemented an SA-driven instruction scheduler in our compiler for instruction-level parallel (ILP) architectures. This allows us to compare our SA instruction scheduler with a more traditional list scheduling approach. Experimental comparison of 114 data dependence graphs efficiency improvement of more than 6% when using an SA-driven scheduler over results obtainable with the standard list scheduling technique.

1. Introduction

Most modern massively-parallel computing systems (MPCS) contain computers that support instruction-level parallelism (ILP) in each of the nodes of the connected system. Such systems are then used to solve large problems, problems that can require large amount of execution time. For such applications, significant compile time can be tolerated in order to achieve maximal execution efficiency. This flexibility in compile time allows the opportunity to investigate alternatives that would be too computationally expensive in a compiler for most other applications. This paper investigates one technique to allow tradeoffs between compile-time and run-time efficiency for massively-parallel systems with hardware support for instruction-level parallelism.

Computer manufacturers are continually striving to make faster computers by 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 inclusion of ILP. ILP computers exploit the implicit parallelism that most programs contain [13]. They overlap the execution of operations[1] that do not depend on one another. For example, a memory address can be evaluated while the value to store there is computed. Today, typical ILP processors have multiple memory address, integer, and 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 [13], recent computer designs have exploited ILP to a larger degree. This trend shows no sign of reversing. Effective use of ILP hardware requires that the instruction stream be ordered such that, whenever possible, multiple low-level operations can be in execution

[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 or may not be performed in the order they are presented in the instruction.

simultaneously. This ordering of machine operations to effectively use an ILP architecture's parallelism is typically called *instruction scheduling* (IS.)

Instruction scheduling is a compiler problem that, due to its NP-complete nature [7], requires heuristic solutions for any significant programs. List scheduling (LS) [6] is a general scheduling method often used for instruction scheduling. It takes its name from the fact that, during scheduling, a priority list of available tasks is consulted to see which task should be scheduled next. While list scheduling cannot guarantee optimal schedules, it provides generally adequate instruction scheduling while requiring relatively little ($O(N^2)$) compile time. As such, list scheduling is a popular scheduling method in compilers for ILP architectures. While compilation for even embedded systems applications cannot tolerate the exponential time required to guarantee an optimal schedule, such compilers may be able to accept the extra compile time needed to perform a more thorough search of the possible scheduling space than is provided by standard list scheduling.

[4] has shown that stochastic search methods can be adapted to instruction scheduling allowing improved schedules at the expense of compile time. Stochastic search techniques are those that use a controlled amount of randomness in order to broaden the area of the solution space searched, with the hope of finding higher-quality results in those regions. One stochastic search technique that shows promise in the realm of instruction scheduling is the use of simulated annealing. Simulated annealing (SA) can be thought of as modified hill-climbing that includes "occasional perturbations" in the search space under investigation to avoid the problem of getting stuck in a local minimum or maximum. The process gets its name from the fact that it closely follows the physical process of annealing which is gradual cooling of a liquid until it solidifies.

We have implemented an SA-driven instruction scheduler in our compiler for ILP architectures. This allows us to compare our SA instruction scheduler with a more traditional list scheduling approach. We have found that using simulated annealing to help drive a standard list scheduler can lead to significant improvements in the obtained schedules at the cost of considerable compilation time necessary for simulated annealing.

2. List Scheduling

Sequencing is defined by Ashour [1] as being "concerned with the arrangements and permutations in which a set of jobs under consideration are performed on all machines." That is, what is the order the jobs will be performed? What is the priority of each job? Sequencing thereby ranks the jobs to be executed. Baker [2] states "scheduling is the allocation of resources over time to perform a collection of tasks." Scheduling usually places already-prioritized jobs into slots, often accounting for conflicts in resource usage. The combined sequencing/scheduling (order/place) process produces the desired outcome: jobs placed on machines capable of performing the desired tasks in the correct order at a correct time.

Instruction scheduling (IS) involves the placement of atomic machine operations into machine instructions. A data dependence DAG (DDD) is often used to describe the necessary operations and their order. The nodes in a DDD contain the operations, and the edges denote a partial order on the nodes. Typically, DDD edges are weighted so that the edge ($O_i \rightarrow O_j$, *min*) indicates that node O_j must follow node O_i by at least *min* instructions in the final schedule. This partial order is used to guarantee program dataflow requirements. The edges of a DDD do not constrain the order nodes are placed in the schedule, only the order they appear in the final schedule.

List scheduling (LS) [6] is a general scheduling method often used for instruction scheduling [12]. In general list scheduling, an ordered list of tasks, called a *priority list*, is constructed. The priority list takes its name from the fact that tasks are ranked such that those having the highest priority will be chosen first. When a machine (or functional unit in the context of instruction scheduling) becomes free, the priority list is scanned for the first unexecuted task ready to be executed. A task is ready when all its predecessors have been executed. If not enough resources are available, the priority list is scanned further until a task is found that can be executed or, if no such task can be found, the machine remains idle until a task on the list is ready.

When using list scheduling in an instruction scheduler, the inputs are typically

1. A priority list of operations that are data- and timing-ready (i.e.: all predecessor operations have completed.) This priority list is typically called a data-ready set (DRS) in the context of instruction scheduling.

2. The DDD.

3. Resource limitations of the target architecture.

and the output is a list of instructions, where each instruction is a set of operations. This list is initially empty; when all the operations are placed in instructions, the schedule is *final*, and scheduling is complete. An instruction holds all the operations that start during that machine cycle; the operations may end in different cycles.

Instructions are numbered 1 through k where k is the length of the schedule. Let I_{O_i} represent the instruction number in which O_i is scheduled. A valid schedule requires that for each arc ($O_i \rightarrow O_j$, *min*), the resulting schedule ensures $min \leq I_{O_j} - I_{O_i}$. In addition, there must be sufficient resources to execute all operations that overlap in time.

By adding each DDD node, N, to the DRS only after all of N's predecessors have been scheduled, we reduce the search space of the scheduling problem and increase the chances of finding a valid schedule. This restriction forms an implicit heuristic for list scheduling: placing nodes with no predecessors more often results in valid total orderings (from the partial ordering represented by the DDD) than does scheduling nodes with as-yet unscheduled predecessors. Note that placing nodes only after all predecessor nodes have been placed is, however, only a heuristic. We might well be able to achieve shorter schedules if "important" nodes could be placed in the schedule even before some predecessor node(s).

List schedulers are often categorized as being one of two types: operation-driven or instruction-driven. In instruction-driven scheduling, instructions are considered in order and each instruction is filled as much as possible before moving on to consider the next instruction. Instruction-driven scheduling fills one instruction at a time such that all operations in instruction I_j are placed before any operation is placed in instruction I_{j+1}. In the instruction-driven model, operations are only added to the priority list once they are timing-ready for scheduling. Thus, if the DDD contains an arc, ($X \rightarrow Y$, 3) and X is scheduled in instruction *n*, then Y cannot be added to the priority list until instruction *n+3* is being considered. In contrast, operation-driven scheduling does not "fill" an instruction at a time, but rather places each DDD node at an "appropriate" location in the schedule whenever that DDD node is to be placed. Operation-driven scheduling makes use of the relative timing between nodes (represented by DDD edges) and the schedule location of whatever nodes have been scheduled to date in order to find an appropriate spot for the DDD node being placed.

In fact, it is the use of operation-driven scheduling that makes possible the simulated-annealing driven scheduling described in the next section.

3. List Scheduling with Simulated Annealing

Instruction scheduling may be viewed as a search-space problem involving an incomplete n-dimensional hypercube, where n is the number of operations (DDD nodes) to be scheduled. Each operation might be executed at a variety of locations in the code, and each dimension represents the range of instructions in which that operation might be placed. Instruction scheduling is complicated by both the inherent dataflow ordering between operations in the DDD and the complexities of the target architecture. The architecture may have complex timings between operations, a number of different field encodings, and a limited number of resources that can perform any given operation.

Most existing instruction schedulers rely on heuristics to remove from examination those parts of the search space that appear fruitless. Identifying appropriate heuristics can be difficult, however, when attempting to arrive at an efficient yet efficacious scheduler. 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 choose and modify a scheduler's behavior.
- It is possible that the heuristics do not address an issue that has great influence on the final code.
- Heuristics that work well for one ordering of operations 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 situations that lead to shorter code sequences.

Considering these difficulties in generating schedulers, several researchers have proposed using stochastic methods to solve the scheduling problem. In [11], the Metropolis Monte Carlo technique was used. [9] used a Boltzmann Machine approach to good effect. In [3, 4], Genetic Algorithms were shown to produce good results, and in [5] they were compared with Tabu search.

In order for stochastic search techniques to work on a given problem, two things are usually needed: an ability to express the problem as a set of values that are to be optimized, and a way to compare solutions accurately to delineate fruitful areas of search. In most problems, the set of values are used to denote independent variables in the equation being optimized.

For instruction scheduling, we need to map the problem to a set of values that can be permuted to produce a valid solution. In classic list scheduling, the sequence the data-ready list controls the overall efficacy of the schedule. The straightforward mapping then, is to have the optimization method produce values with its internal method, and use these values to sequence of the data-ready list for the scheduler. That is, instead of having the heuristics choose the sequence of operations to schedule in classic list scheduling, have the stochastic search technique pick the sequence.

This is where operation-driven scheduling comes into play. Because operation-driven scheduling allows (potentially) valid schedules to be generated by placing DDD nodes into the schedule in an arbitrary order, we can use the stochastic technique of choice generate "random" orderings (permutations) of the DDD nodes for any DDD to be scheduled. We must then have some mechanism to evaluate a given sequence of DDD nodes, thus ranking the fitness of different possible permutations of nodes. Choosing a useful evaluation function, i.e., one that represents a sequence's relative worth 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 given sequence generates. A difficulty encountered is that not all sequences will produce valid final schedules. Failures occur when a conflict arises (e.g. timing, resource, or field) due to the order of scheduling the operations. It is not surprising that certain orders fail to produce valid schedules for a given DDD; the impact of ordering on the production of valid schedules is emphasized in all previous instruction scheduling methods.

After consideration, the evaluation function we selected performs a "worst-case" evaluation when a node-ordering 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.

Armed with an evaluation function we can turn our attention to stochastically generating different DDD nodes orderings. For this study, we used the ASA code provided by Lester Ingber [10], an implementation of adaptive simulated annealing. The word adaptive refers to the ability of the algorithm to modify its behavior on the fly. It essentially "decides" the number of iterations spent at a certain temperature level, how to reduce the temperature and alter the random selection of the neighbors.

In the current implementation, we use ASA to find the permutation π of of the n jobs that produces a schedule with minimum number of instructions when this permutation is used as the ordering for the list scheduling heuristic.

4. Experimental Evaluation

In order to evaluate the possible usefulness of instruction scheduling based upon simulated annealing, we compared, for a test suite of 114 data dependence DAGs (DDDs), the scheduling results generated by our standard list-scheduling implementation to those obtained using our simulated-annealing driven scheduler. Overall, we found that the simulated-annealing driven scheduler generated better schedules, but at the cost of significant compile time.

Name	Description
8q	a recursive solution to the 8 queens problem
gauss	a Gaussian elimination program
whetstone	the standard benchmark
Livermore	a C implementation of the Livermore loops
Cut	the Unix cut program
matrix	standard matrix multiply code
bubble	bubble sort
factorial	compute factorials
clinpack2	a C implementation of linpack
dhrystone	the standard benchmark
nseive	sieve of Eratosthenes

Table 1. C Programs in Test Set

4.1. DDDs

To compare our two scheduling methods we needed DDDs for which to generate schedules. The 114 DDDs used in this study all come from a test suite of the eleven C programs listed in Table 1. The compiler [14] was used to build a DDD for each basic block in the test suite of C programs. Only DDDs consisting of ten or more nodes were included in the suite of DDDs used for the experimental evaluation of simulated-annealing driven scheduling.

4.2. Experiment

Using the compiler, we scheduled each of the 114 DDDs constructed as described in Section 4.1, generating instruction schedules for the DEC Alpha 21164 architecture [8] which allows up to two integer and two floating point operations to begin execution in a cycle. Latencies for integer and floating point computations as well as the load and store latencies were assigned according to the 21164 timing model. All loads were assumed to be cache hits. With this machine model, we compared schedules by counting the number of instructions that each DDD would require to execute.

Table 2 list the schedule data obtained in the study. The first column of each table gives the DDD number. This number has no special significance and is used only to differentiate the DDDs. The second column indicates how many nodes that DDD contains. The third column shows instructions required to schedule the DDD using traditional list scheduling. The fourth column indicates how many instructions required to schedule the DDD using simulated-annealing driven scheduling. Notice that many of the entries in column four are "same". This indicates that the simulated-annealing driven scheduler failed to find a better schedule than the list scheduler.

Table 2 does not show the compile time needed to schedule the DDDs, but while traditional scheduling scheduled all 114 DDDs in slightly less than four minutes on a Sun Sparc10, simulated-annealing driven scheduling required almost five days (117 hours) to schedule all 114 DDDs on the same machine. Thus, as expected, simulated-annealing driven scheduling was considerably slower than traditional scheduling. However, it was not so slow, requiring a little more than 1 hour per DDD, that it should not be considered if excellent run-time efficiency is required.

For those 75 DDDs where simulated-annealing driven scheduling and traditional scheduling differed, the simulated-annealing scheduler produced shorter schedules 60 times (80% of the time) while traditional scheduling produced the better schedule 5 times and there were 10 DDDs for which both schedules required the same number of instructions. Summing all of the instructions for the 75 DDDs under comparison shows that, even counting the simulated-annealing losses, simulated-annealing driven scheduling required 6.7% fewer instructions. This can lead to a substantial savings in execution time for those DDDs.

A likely scheduling strategy would be to first perform traditional list scheduling for each DDD of an application and then to perform simulated-annealing driven scheduling for each DDD (or maybe only for those which profiling indicates constitute a significant percentage of total execution time.) This would allow use of the better of the two schedules. If such a policy were used, the "combined" tradition-SA scheduler would produce results (for our test suite) that results in 7.3% fewer instructions that the traditional scheduler.

The negative correlation between our simulated-annealing scheduler being better than simple LS and DDD size should not be surprising when you consider that our strategy for using simulated annealing is to evaluate different "random" permutations of integer 1 to N where N is the number of nodes in a DDD. Given that we would expect only a few permutations to lead to legal schedules, scheduling for larger DDDs (with substantially larger numbers of possible permutations) would be less likely to encounter any legal schedules during the stochastic search. Still, since the 114 DDDs used in this study come from "real" code and since the only bias based upon DDD size discarded small DDDs (less than 10 nodes) we feel that our approach to simulated-annealing scheduling should lead to significantly improved schedules for local (single-basic block) DDDs.

Note that we did not choose to "seed" the ASA routines by initializing the search space with a DDD node-ordering from standard list scheduling. This would have the effect of removing all the "longer" schedules from the SA-driven scheduler, since we would always have at least one "short" schedule, namely the one generated by list scheduling. We chose to accept random initial permutations instead, to ensure that the search space visited was not biased by the original list-scheduled result.

5. Conclusions

This paper has shown the benefit of applying a simulated annealing (SA), a stochastic search technique, to the problem of producing high-quality code. With a combined traditional/SA instruction scheduling method, over 7% fewer instructions were needed when compared to a traditional list scheduler. This can lead to both decreased code size and runtime. There is no doubt that using SA for optimization requires more compilation time, but this is well amortized over the length of time a program is in production.

A concern of the current implementation is the number of times SA fails to generate a shorter schedule. This high percentage has not been noted in previous stochastic IS techniques and needs to be studied in order to find the root cause. It may be that this is a cost that must be paid in order to achieve the highest level of optimization possible. With this caveat, we would recommend this method of IS to practitioners in the field.

References

[1] ASHOUR, S. *Sequencing Theory*. Springer-Verlag, New York, 1972.

[2] BAKER, K. R. *Introduction to Sequencing and Scheduling*. John Wiley and Sons, Inc., New York, 1974.

[3] BEATY, S. Genetic algorithms and instruction scheduling. In *Proceedings of the 24th Microprogramming Workshop (MICRO-24)* (Albuquerque, NM, Nov. 1991).

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

[5] BEATY, S. A comparison of genetic algorithms and tabu search for instruction scheduling. In *International Conference on Neural Networks and Genetic Algorithms* (Innsbruck, Austria, Apr. 1993).

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

[7] DEWITT, D. *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.

DDD	Nodes	List	ASA	DDD	Nodes	List	ASA	DDD	Nodes	List	ASA
1	24	19	same	2	17	14	10	3	16	10	11
4	20	23	22	5	35	29	same	6	50	77	same
7	24	19	same	8	18	11	13	9	19	12	13
10	23	16	13	11	60	82	same	12	20	13	13
13	24	18	17	14	31	25	same	15	19	56	56
16	80	116	same	17	30	21	same	18	50	48	same
19	41	32	same	20	20	18	same	21	52	43	same
22	18	16	15	23	71	65	same	24	189	148	same
25	76	59	same	26	88	58	same	27	27	23	22
28	129	125	same	29	27	23	22	30	98	97	same
31	26	24	23	32	57	64	same	33	30	36	34
34	60	85	same	35	14	13	12	36	22	24	23
37	27	26	26	38	13	18	17	39	10	5	6
40	10	9	8	41	81	62	same	42	42	28	same
43	20	16	16	44	58	117	same	45	12	11	9
46	28	23	18	47	19	16	14	48	26	21	same
49	15	13	12	50	10	8	7	51	15	12	10
52	18	20	19	53	10	8	7	54	14	11	9
55	19	20	19	56	21	18	16	57	15	11	10
58	21	19	14	59	16	13	12	60	15	11	9
61	16	11	10	62	20	21	20	63	19	16	15
64	27	22	same	65	15	13	12	66	15	12	10
67	18	21	20	68	10	8	7	69	15	11	10
70	19	21	20	71	20	17	same	72	14	9	8
73	26	21	19	74	16	13	12	75	10	8	7
76	16	11	10	77	19	21	20	78	10	8	7
79	15	11	9	80	20	21	20	81	12	7	6
82	27	23	same	83	15	11	10	84	12	8	7
85	10	10	8	86	29	22	same	87	22	16	15
88	16	15	14	89	33	23	same	90	29	22	22
91	31	22	22	92	19	12	same	93	34	25	same
94	17	10	11	95	13	11	9	96	18	14	same
97	17	10	same	98	18	14	same	99	19	17	15
100	13	10	7	101	11	9	8	102	165	119	same
103	54	47	47	104	22	19	18	105	22	19	18
106	15	21	21	107	20	28	27	108	31	22	22
109	20	24	23	110	11	8	8	111	26	20	same
112	42	71	same	113	32	66	same	114	38	33	same

Table 2. Schedule Length in Instructions

[8] EDMONDSON, J., RUBENFELD, P., AND PRESTON, R. Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro 15*, 2 (Apr. 1995), 33–43.

[9] GLORIA, A. D., FARABOSCHI, P., AND OLIVIERI, M. A non-deterministic scheduler for a software pipelining compiler. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25)* (Portland, Oregon, Dec. 1992), pp. 41–44.

[10] INGBER, L. Adaptive simulated annealing (asa): Lessons learned. *Control and Cybernetics 25* (1996), 33–54.

[11] JACOBS, D., PRINS, J., SIEGEL, P., AND WILSON, K. Monte carlo techniques in code optimization. In *Proceedings of the 15th Annual Workshop on Microprogramming (Micro-15)* (Palo Alto, California, Dec. 1982), pp. 143–148.

[12] LANDSKOV, D., DAVIDSON, S., SHRIVER, B., AND MALLETT, P. Local microcode compaction techniques. *ACM Computing Surveys 12*, 3 (September 1980), 261–294.

[13] RAU, B. R., AND FISHER, J. A. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing 7* (1993), 9–50.

[14] SWEANY, P., AND BEATY, S. Post-compaction register assignment in a retargetable compiler. In *Proceedings of the 23th Microprogramming Workshop (MICRO-23)* (Orlando, FL, November 1990), pp. 107–116.