Optimal Instruction Scheduling Using Integer Programming

Kent Wilken **Jack Liu** **Mark Heffernan**

Department of Electrical and Computer Engineering
University of California, Davis, CA 95616
{wilken,jjliu,meheff}@ece.ucdavis.edu

Abstract – *This paper presents a new approach to local instruction scheduling based on integer programming that produces optimal instruction schedules in a reasonable time, even for very large basic blocks. The new approach first uses a set of graph transformations to simplify the data-dependency graph while preserving the optimality of the final schedule. The simplified graph results in a simplified integer program which can be solved much faster. A new integer-programming formulation is then applied to the simplified graph. Various techniques are used to simplify the formulation, resulting in fewer integer-program variables, fewer integer-program constraints and fewer terms in some of the remaining constraints, thus reducing integer-program solution time. The new formulation also uses certain adaptively added constraints (cuts) to reduce solution time. The proposed optimal instruction scheduler is built within the Gnu Compiler Collection (GCC) and is evaluated experimentally using the SPEC95 floating point benchmarks. Although optimal scheduling for the target processor is considered intractable, all of the benchmarks' basic blocks are optimally scheduled, including blocks with up to 1000 instructions, while total compile time increases by only 14%.*

1 Introduction

Instruction scheduling is one of the most important compiler optimizations because of its role in increasing pipeline utilization. Conventional approaches to instruction scheduling are based on heuristics and may produce schedules that are suboptimal. Prior work has considered optimal instruction scheduling, but no approach has been proposed that can optimally schedule a large number of instructions in reasonable time. This paper presents a new approach to optimal instruction scheduling which uses a combination of graph transformations and an advanced integer-programming formulation. The new approach produces optimal schedules in reasonable time even for scheduling problems with 1000 instructions.

This research was supported by Equator Technologies, Mentor Graphics' Embedded Software Division, Microsoft Research, the National Science Foundation's CCR Division under grant #CCR-9711676, and by the University of California's MICRO program.

PLDI 2000, Vancouver, British Columbia, Canada.

1.1 Local Instruction Scheduling

The local instruction scheduling problem is to find a minimum length instruction schedule for a basic block. This instruction scheduling problem becomes complicated (interesting) for pipelined processors because of *data hazards* and *structural hazards* [11]. A data hazard occurs when an instruction i produces a result that is used by a following instruction j, and it is necessary to delay j's execution until i's result is available. A structural hazard occurs when a resource limitation causes an instruction's execution to be delayed.

The complexity of local instruction scheduling can depend on the maximum data-hazard *latency* which occurs among the target processor's instructions. In this paper, latency is defined to be the difference between the cycle in which instruction i executes and the first cycle in which data-dependent instruction j can execute. Note that other authors define latency (delay) to be the cycle difference minus one, e.g., [2, 17]. We prefer the present definition because it naturally allows write-after-read data dependencies to be represented by a latency of 0 (write-after-read dependent instructions can execute in the same cycle on a typical multiple-issue processor, because the read occurs before the write within the pipeline).

Instruction scheduling for a single-issue processor with a maximum latency of two is easy. Instructions can be optimally scheduled in polynomial time following the approach proposed by Bernstein and Gertner [2]. Instruction scheduling for more complex processors is hard. No polynomial-time algorithm is known for optimally scheduling a single-issue processor with a maximum latency of three or more [2]. Optimal scheduling is NP-complete for realistic multiple-issue processors [3]. Because optimal instruction scheduling for these more complex processors is considered intractable, production compilers use suboptimal heuristic approaches. The most common approach is *list scheduling*, where instructions are represented as nodes in a directed acyclic data-dependency graph (DAG) [15]. A graph edge represents a data dependency, an edge weight represents the corresponding latency, and each DAG node is assigned a priority. *Critical-path list scheduling* is a common variation, where an instruction's priority is based on the maximum-length path through the DAG from the node representing the instruction to any leaf node [15]. While critical-path list scheduling is usually effective, it can produce suboptimal results even for small scheduling problems. Consider the DAG in Figure 1, taken from [17], where each edge is labeled with its latency and each node is labeled with its critical-path

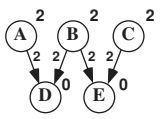

Figure 1: Simple example where critical-path list scheduling can produce a suboptimal schedule.

priority. For this DAG, nodes A, B and C all have the same priority, so the scheduler can arbitrarily select the order of these instructions. If the initial order is A, C, B or C, A, B, the next cycle will be a stall because the latency from B to D or from B to E is not satisfied. Other orders of A, B and C will produce an optimal schedule which has no stall.

1.2 Optimal Instruction Scheduling

Although optimal instruction scheduling for complex processors is hard in theory, in practice it may be possible to optimally solve important instances of instruction scheduling problems in reasonable time using methods from combinatorial optimization. Prior work has used various combinatorial optimization approaches to optimally schedule instructions for complex processors. However none of these approaches can optimally schedule large basic blocks in reasonable time.

Ertl and Krall [8] use constraint logic programming and consistency techniques to optimally schedule instructions for a single-issue processor with a maximum latency greater than two. Vegdahl [23] and Kessler [13] use dynamic programming to optimally schedule instructions. Chou and Chung [6] and Tomiyama *et al.* [22] use approaches that implicitly enumerate all possible schedules to find an optimal schedule. For efficiency, [6] and [22] propose techniques to prune the enumeration tree so that redundant or equivalent schedules are not explicitly enumerated. Experimental results for these various approaches show that they are effective only for small to medium-sized basic blocks (30 instructions or less).

Prior work has also used *integer programming* to optimally schedule instructions. An integer program is a linear program, with the added requirement that all problem variables must be assigned a solution value that is an integer. Like the other approaches to optimal instruction scheduling, prior work using integer programming has only produced approaches that are effective for small to medium-sized scheduling problems. Arya [1] proposes an integer programming formulation for vector processors, although the basic approach is general and can be applied to other processor types. The experimental results in [1] suggest the formulation can significantly improve the instruction schedule. However, the results are limited to only three small to medium-sized problems (12 to 36 instructions). Only the smallest problem is solved optimally, with the other problems timing out before an optimal solution is found. Leupers and Marwedel [14] propose an integer programming formulation for optimally scheduling a multiple-issue processor. Although their work focuses on DSP processors, again the basic approach is general and can be used for other processor types. The experimental results in [14] are also limited to a few small to medium-sized problems (8 to 37 instructions). While the solution time might be acceptable for the largest problem studied (37 instructions solves in 130 seconds), the solution time does not appear to scale well with problem size (the next smaller problem, 24 instructions, solves in 7 seconds). Thus the approach does not appear practical for large instruction scheduling problems. Chang, Chen and King [5] propose an integer programming formulation that combines local instruction scheduling with local register allocation. Experimental results are given for one simple 10-instruction example which takes 20 minutes to solve optimally. These results suggest that the approach has very limited practicality.

Although prior work using integer programming has produced limited success for optimal instruction scheduling, integer programming has been used successfully to optimally solve various other compiler optimization problems, including array dependence analysis [19], data layout for parallel programs [4] and global register allocation [9]. Integer programming is the method of choice for solving many large-scale real-world combinatorial optimization problems in other fields [16], including other scheduling problems such as airline crew scheduling. This successful use of integer programming elsewhere suggests that improved integer programming formulations may be the key to solving large-scale instruction scheduling problems.

This paper presents a new approach to optimal instruction scheduling based on integer programming, the first approach which can solve very large scheduling problems in reasonable time. The paper is organized as follows. Section 2 describes a basic integer-programming formulation for optimal instruction scheduling, which is representative of formulations proposed in prior work. The material in Section 2 provides the reader with background on how instruction scheduling can be formulated as an integer programming problem, and provides a basis for comparing the new integer-programming formulation. Experimental results in Section 2 show that the basic formulation cannot solve large instruction scheduling problems in reasonable time, which is consistent with the results from prior work. Section 3 introduces a set of DAG transformations which can significantly simplify the DAG, while preserving the optimality of the schedule. The simplified DAG leads to simplified integer programs which are shown experimentally to solve significantly faster. Section 4 introduces a new integer-programming formulation that simplifies the integer program by reducing the number of integer-program variables, reducing the number of integer-program constraints, and simplifying the terms in some of the remaining constraints. The simplified integer programs are shown experimentally to solve dramatically faster. The last section summarizes the paper's contributions and outlines future work.

2 Basic Integer-Programming Formulation

This section describes a basic integer-programming formulation for optimal instruction scheduling, which is representative of formulations proposed in prior work. The basic formulation provides background for the DAG transformations presented in Section 3 and the new integer-programming formulation presented in Section 4.

2.1 Optimal Instruction Scheduling, Basic Formulation

In the basic formulation, the basic block is initially scheduled using critical-path list scheduling. The length U of the resulting schedule is an upper bound on the length of an optimal schedule. Next, a lower bound L on the schedule length is computed. Given the DAG's critical path c and the processor's issue rate r, a lower bound on the schedule for a basic block with n instructions is:

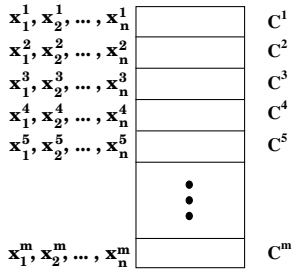

Figure 2: The array of 0-1 scheduling variables for a basic block with n instructions for a schedule of m clock cycles.

$$L = 1 + \max\{c, \lceil n/r \rceil - 1\}$$

If $U = L$, the schedule is optimal, and an integer program is unnecessary. If $U > L$, an integer program is produced (as described below) to find a length $U-1$ schedule. If the integer program is infeasible, the length U schedule is optimal. Otherwise a length $U-1$ schedule was found and a second integer program is produced to find a length $U-2$ schedule. This cycle repeats until a minimum-length schedule is found.

To produce an m clock-cycle schedule for an n-instruction basic block, a 0-1 integer-program *scheduling variable* is created for each instruction, for each clock cycle in the schedule. The scheduling variable x_i^j represents the decision to schedule (1) or not schedule (0) instruction i in clock cycle j. The scheduling variables for the corresponding clock cycles are illustrated in Figure 2.

A solution for the scheduling variables must be constrained so that a valid schedule is produced. A constraint is used for each instruction i to ensure that i is scheduled at exactly one of the m cycles, a *must-schedule constraint* with the following form:

$$\sum_{j=1}^{m} x_i^j = 1$$

Additional constraints must ensure the schedule meets the processor's issue requirements. Consider a statically scheduled r-issue processor that allows any r instructions to issue in a given clock cycle, independent of the instruction type. For this r-issue processor, an *issue constraint* of the following form is used for each clock cycle j :

$$\sum_{i=1}^{n} x_i^j \leq r$$

If a multiple-issue processor has issue restrictions for various types of instructions, a separate issue constraint can be used for each instruction type [14].

A set of *dependency constraints* is used to ensure that the data dependencies are satisfied. For each instruction i, the following expression resolves the clock cycle in which i is scheduled:

$$\sum_{j=1}^{m} j * x_i^j$$

Because only one x_i variable is set to 1 and the rest are set to 0 in an integer program solution, the expression produces the corresponding coefficient j, which is the clock cycle in which instruction i is scheduled. Using this expression, a dependency constraint of the following form is produced for each edge in the DAG to enforce the dependency of instruction i on instruction k, where the latency from k to i is the constant L_{ki}:

$$\sum_{j=1}^{m} j * x_k^j + L_{ki} \leq \sum_{j=1}^{m} j * x_i^j$$

Prior work has proposed a basic method for simplifying the integer program and hence reducing integer-program solution time. Upper and lower bounds can be determined for the cycle in which an instruction i can be scheduled, thereby reducing an instruction's scheduling range. All of i's scheduling variables for cycles outside the scheduling range set by i's upper and lower bounds are unnecessary and can be eliminated. After scheduling range reduction, if any instruction's scheduling range is empty (its upper bound is less than its lower bound), no length m schedule exists and an integer program is not needed.

Chang, Chen and King propose using the critical path distance from any leaf node (any root node) or the number of successors (predecessors) to determine an upper bound (lower bound) for each instruction [5]. For the r-issue processor defined above, a lower bound L_i on i's scheduling range is set by:

$$L_i = 1 + \max\{cr_i, \lceil (1 + p_i)/r \rceil - 1\} \quad (1)$$

where cr_i is the critical path distance from any root node to i, and p_i is the number of i's predecessors. Similarly, an upper bound U_i on i's scheduling range is set by:

$$U_i = m - \max\{cl_i, \lceil (1 + s_i)/r \rceil - 1\} \quad (2)$$

where cl_i is the critical path distance from i to any leaf node, and s_i is the number of i's successors.

Collectively, the reduced set of scheduling variables, the must-schedule constraints, the issue constraints, and the dependency constraints constitute a basic 0-1 integer programming formulation for finding a schedule of length m. Applied iteratively as described above, this formulation produces an optimal instruction schedule.

2.2 Basic Formulation, Experimental Results

The basic formulation is built inside the Gnu Compiler Collection (GCC), and is compared experimentally with critical-path list scheduling. As shown in [2], optimal instruction-scheduling for a single-issue processor with a two-cycle maximum latency can be done in polynomial time. However, optimal scheduling for a single-issue processor with a three-cycle maximum latency, the next harder scheduling problem, is considered intractable [2]. If this easiest hard scheduling problem cannot be solved optimally in reasonable time, there is little hope for optimally scheduling more complex processors. Thus, this paper focuses on optimal scheduling for a single-issue processor with a three-cycle maximum latency.

The SPEC95 floating point benchmarks were compiled using GCC 2.8.0 with GCC's instruction scheduler replaced by an optimal instruction scheduler using the basic formulation. The benchmarks were compiled using GCC's highest level of optimization (-O3) and were targeted to a single-issue processor with a maximum latency of three cycles. The target processor has a latency of 3 cycles for loads, 2 cycles for all floating point operations and 1 cycle for all integer

operations. The SPEC95 integer benchmarks are not included in this experiment because for this processor model there would be no instructions with a 2-cycle latency, which makes the scheduling problems easier to solve.

The optimal instruction scheduler is given a 1000 second time limit to find an optimal schedule. If an optimal schedule is not found within the time limit, the best improved schedule produced using integer programming (if any) is selected, otherwise the schedule produced by list scheduling is selected. The integer programs are solved using CPLEX 6.5, a commercial integer-programming solver [12], running on an HP C3000 workstation with a 400MHz PA-8500 processor and 512MB of main memory. The experimental results for the basic formulation are shown in Table 1.

	Basic Formulation
Total Basic Blocks (BB)	7,402
BB from List Scheduling shown to be Optimal w/o IP	6,885
BB Passed to IP Formulation	517
BB IP Solved Optimally	482
BB IP Timed Out	35
BB IP Improved and Optimal	15
BB IP Improved but Non-Optimal	0
Total Cycles IP Improved	15
Total Scheduling Time (sec.)	35,879

Table 1: Experimental results using the basic integer programming formulation.

Various observations are made about these data. First, using only list scheduling most of the schedules, 6,885 (93%), are shown to be optimal because the upper bound U equals the lower bound L or because after scheduling range reduction for a schedule of length $U-1$, an instruction's scheduling range is empty. This is not surprising because this group of basic blocks includes such trivial problems as basic blocks with one instruction. For the 517 non-trivial problems that require an integer program, 482 (93%) are solved optimally and 35 (7%) are not solved using a total of 35,879 CPU seconds (10.0 hours). As a point of reference, the entire benchmark suite compiles in 708 seconds (11.8 minutes) when only list scheduling is used. Thus, the basic formulation fails in two important respects: not all basic blocks are scheduled optimally, and the scheduling time is long. Only 15 of the basic blocks (3% of the non-trivial basic blocks) have an improved schedule, and the total static cycle improvement is only 15 cycles, a modest speedup. Speedup will be much higher for a more complex processor (longer latency and wider issue). For example, results in [21] for the multiple-issue Alpha 21164 processor show that list scheduling is sub-optimal for more than 50% of the basic blocks studied. The proper conclusion to draw from the results in Table 1 is not that optimal instruction scheduling does not provide significant speedup, but that the basic integer-programming formulation cannot produce optimal schedules in reasonable time, even for the easiest of the hard scheduling problems. A much better approach to optimal instruction scheduling is needed.

3 DAG Transformations

A set of graph transformations is proposed which can simplify the DAG before the integer program is produced. These transformations are fast (low-order polynomial time in the size of the DAG) and are shown to preserve the optimality of the final schedule. The integer program produced from a transformed DAG is significantly simplified and solves much faster.

3.1 DAG Standard Form

The transformations described in the following sections are for DAGs in *standard form*. A DAG in standard form has a single root node and a single leaf node. A DAG with multiple leaf and root nodes is transformed into a DAG in standard form by adding an artificial leaf node and an artificial root node. The artificial root node is the immediate predecessor of all root nodes. A latency-one edge extends from the artificial root node to each root node in the DAG. Similarly, an artificial leaf node is the immediate successor of all DAG leaf nodes. A latency-one edge extends from each leaf node of the DAG to the artificial leaf node. These artificial nodes do not affect the optimal schedule length of the original DAG nodes and are removed after scheduling is complete.

3.2 DAG Partitioning

Some DAGs can be partitioned into smaller subDAGs which can be optimally scheduled individually, and the subDAG schedules can be recombined to form a schedule that is optimal for the entire DAG. Partitioning is advantageous because the integer-program solution time is super-linear in the size of the DAG, and thus the total time to solve the partitions is less than the time to solve the original problem. Partitioning is also advantageous because even though the original DAG may require an integer program to find its optimal schedule, one or more partitions may be optimal from list scheduling and an integer program is not required for those partitions.

A DAG can be partitioned at a *partition node*. A partition node is a node that dominates the DAG's leaf node and post-dominates its root node. A partition node forms a barrier in the schedule. No nodes above a partition node may be scheduled after the partition node, and no nodes below the partition node may be scheduled before the partition node. An efficient algorithm for finding partition nodes is described in Section 3.5.1. The algorithm's worst-case execution time is $O(e)$, where e is the number of edges in the DAG.

Figure 3a shows a DAG which can be divided into two partitions at partition node D. Nodes A, B, C, and D form one partition, and nodes D, E, F, and G form the other partition. As illustrated, the two partitions are each optimally scheduled, and the schedules are then combined to form an optimal schedule for the entire DAG.

3.3 Redundant Edge Elimination

A DAG may include *redundant edges*. An edge between nodes A and B ($edge_{AB}$) is redundant if there is another path P_{AB} from A to B, and the *distance* along P_{AB} is greater than or equal to the distance across $edge_{AB}$. The distance along a path is the sum of the latencies of the path's edges. The distance across an edge is the edge's latency. In a schedule of the DAG nodes, $edge_{AB}$ enforces the partial order of A and B and the minimum latency between A and B. However, because P_{AB} enforces both of these conditions, $edge_{AB}$ is unnecessary and can be removed. Each DAG edge requires a dependency constraint in the integer program. Therefore,

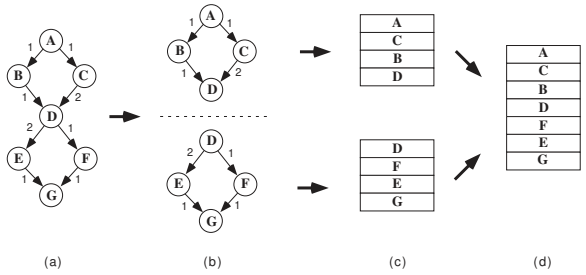

Figure 3: Example DAG that can be partitioned.

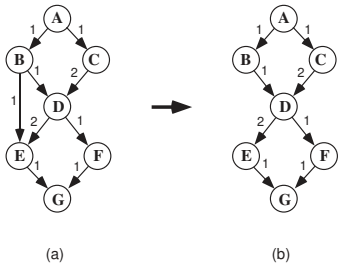

Figure 4: Example redundant edge, $edge_{BE}$.

removing redundant edges reduces the integer program's size and hence reduces its solution time. Removing redundant edges can also create new opportunities for partitioning, further simplifying the integer program. An efficient algorithm for finding redundant edges in a DAG partition is described in Section 3.5.2. The algorithm's worst-case execution time is $O(n_P e_P)$ where n_P is the number of nodes and e_P is the number of edges in the DAG partition.

In Figure 4, $edge_{BE}$ is a redundant edge which can be removed. When $edge_{BE}$ is removed, the DAG is reduced to the DAG shown in Figure 3, which can then be partitioned.

3.4 Region Linearization

Region linearization is a DAG transformation which orders the set of nodes contained in a *region* of the DAG. A region R is defined by a pair of nodes, the entry node A and the exit node B. R is the subDAG induced by node A, node B, and all nodes that are successors of A and predecessors of B, nodes which are called the region's *interior nodes*. R must contain two disjoint paths from A to B and each path must include a node other than A or B. Figure 5 shows three example regions inside the rectangles. For region EI in Figure 5b, node E is the entry node, node I is the exit node, nodes F, G, and H are the interior nodes, and node X is external to the region.

In region linearization, list scheduling is used to produce a schedule for the instructions in each DAG region. Under certain conditions (described below), the original region subDAG can be replaced by a linear chain of the region's nodes in the order determined by list scheduling, while preserving the optimality of the final overall schedule. This can significantly simplify the DAG, and hence the integer program. This paper considers region linearization for a single-issue processor. Region linearization for multiple-issue processors is also possible and will be considered in a future paper.

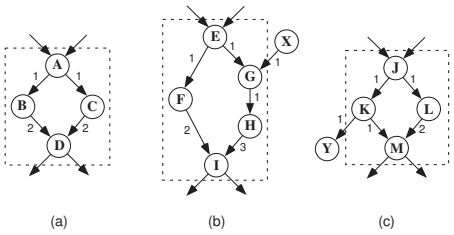

Figure 5: Example regions in a DAG.

3.4.1 Single-Entry Single-Exit Regions

An order O of the nodes in a region can be enforced while preserving optimality if for every optimal schedule of the DAG, the region's nodes can be reordered to the order O, producing a valid schedule with the same length. An order O that satisfies this condition is a *dominant order* for the region. The simplest type of region for which a dominant order can be found is a *single-entry, single-exit* (SESE) region. An SESE region is a region where the entry node dominates the interior nodes, and the exit node post-dominates the interior nodes. Figure 5a illustrates an SESE region. SESE regions can be found in $O(n)$ time [18].

Theorem 1: *If the schedule of an SESE region meets the following conditions, then the schedule's node order O is a dominant order of the region's nodes:*

- *The schedule of the interior nodes is* dense. *A schedule is dense if the schedule contains no empty cycles.*
- *The first interior node in order O has an incoming edge in the DAG that is one of the minimum-latency edges outgoing from the entry node to the region's interior.*
- *The last interior node in order O has an outgoing edge in the DAG that is one of the minimum-latency edges incoming to the exit node from the region's interior.*

Proof: Assume an optimal schedule where the SESE region's nodes are in order O'. Remove the region's nodes from the schedule. The nodes in order O can always be placed in the vacancies left by the removed nodes. The entry node in order O will remain the first region node in the schedule and the exit node will remain the last region node in the schedule. Thus, order O satisfies the latencies between the region's nodes and the external nodes. A vacancy cannot occur within the minimum latency from the entry node to the interior nodes, nor within the minimum latency to the exit node from the interior nodes. Therefore, the latencies to the first interior node and from the last interior node in order O are satisfied. If the schedule of the vacancies left by the interior nodes in order O' is dense, then the interior node schedule for order O can be placed directly in the vacancies. Or, if the schedule of the vacancies left by the interior nodes of order O' is not dense, the dense schedule of order O's interior nodes can be stretched to match the schedule of the vacancies. The latencies in the stretched schedule between the interior nodes are satisfied, because the latencies are greater than or equal to the latencies in the dense schedule of the interior nodes of order O. □

A dominant order for an SESE region can be enforced in a DAG by replacing the region subDAG by a subDAG with the region's nodes in the dominant order. The interior

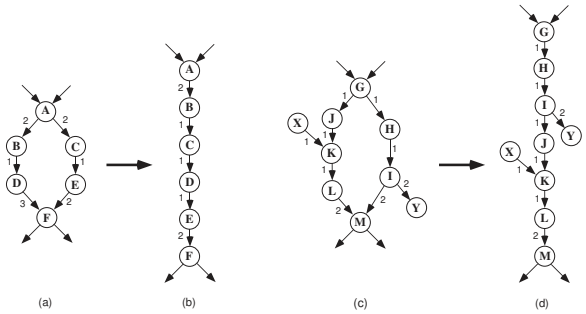

Figure 6: Examples of region linearization transformation.

nodes in the transformed region are separated by latency one edges. The entry node is separated from the first interior node and the last interior node is separated from the exit node by the same edges that occur between these nodes in the original DAG. For the DAG in Figure 6a, the node order A, B, C, D, E, F is a dominant order. Figure 6b shows the region after the linearization transformation.

3.4.2 Non-SESE regions

Regions which are not SESE regions contain *side-exit nodes* and *side-entry nodes*. A side-exit node is an interior node with an immediate successor that is not a region node. A side-entry node is an interior node with an immediate predecessor that is not a region node. Figures 5b and 5c are examples of non-SESE regions. In Figure 5b, node G is a side-entry node. In Figure 5c, node K is a side-exit node. The conditions for linearizing non-SESE regions are more restrictive than those for SESE regions because of the additional dependencies to and from nodes outside the region.

Theorem 2: *If the schedule of a non-SESE region meets the following conditions then the schedule's node order O is a dominant order of the region's nodes:*

- *The order O satisfies the conditions for a dominant order of an SESE region.*
- *All nodes that precede a side-exit node in order O are predecessors of the side-exit node in the DAG.*
- *All nodes that follow a side-entry node in order O are successors of the side-entry node in the DAG.*

Proof: Assume an optimal schedule with the region's nodes in order O'. Remove the region's nodes from the schedule. The region's nodes in order O can always be placed in the vacancies left by the removed nodes. The order O satisfies the conditions for a dominant schedule for an SESE region. Therefore, following the proof of Theorem 1, the dependencies between interior nodes are satisfied. Similarly, the dependencies between nodes outside the region and the region entry and exit nodes are satisfied. Only the predecessors of a side-exit node in the DAG precede the side-exit node in order O. Therefore, the minimum number of nodes precede each side-exit node in order O. If the region nodes are removed from the schedule and reordered in order O, a side-exit node cannot be placed in a vacancy later than the original location of the side-exit node in the schedule. Therefore, the dependencies from the side-exit node to nodes outside the region are satisfied. A symmetric argument proves that the dependencies to side-entry nodes from nodes outside the region are also satisfied.□

An algorithm for finding all of the regions in a DAG partition is described in Section 3.5.3. The algorithm's worst-case execution time is $O(n_P e_P)$ where n_P is the number of nodes and e_P is the number of edges in the DAG partition. In Section 3.5.4, an algorithm for linearizing a region is described which has a worst-case execution time of $O(e_P+n_R^2)$, where n_R is the number of nodes in the region.

Figure 6c shows a non-SESE region with a side-entry node K and a side exit node I. The node order G, H, I, J, K, L, M is a dominant order for the region. Figure 6d shows the region after the linearization transformation.

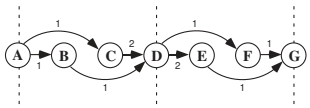

Figure 7: Example DAG with nodes in a linear sequence.

3.5 Efficient Algorithms for DAG Transformations

This section describes a set of efficient algorithms for performing the DAG transformations described in Section 3.2 through Section 3.4.

3.5.1 DAG Partitioning Algorithm

An algorithm is proposed for finding the partition nodes of a DAG. If a DAG is drawn as a sequence of nodes in a *topological sort* [7], then all edges from nodes preceding a partition node terminate at or before the partition node. Figure 7 shows the DAG from Figure 3a redrawn as a linear sequence in the topological sort A, B, C, D, E, F, G. No edge extends across the dashed lines at the partition nodes[1].

An efficient algorithm for finding partition nodes iterates through the nodes of the DAG in a topological sort $O = (N_1, ..., N_n)$, where n is the number of nodes in the DAG. A variable `latest` is maintained across the iterations of the algorithm. `latest` equals the latest immediate successor of any node before the node of the current iteration. Initially, `latest` is set equal to the root node. Partition nodes are determined and `latest` is updated as follows:

```
for i ← 1 to n
    if N_i = latest
        N_i is a partition node
    for each N_k ∈ S(N_i)
        if N_k is later in O than latest
            latest ← N_k
```

where $S(N_i)$ is the set of immediate successors of N_i.

The original instruction order of the DAG nodes before instruction scheduling can be used as the topological sort for the algorithm. Each DAG edge and node is considered only once in the execution of the algorithm. Therefore, the algorithm runs in $O(n + e)$ time, where n is the number of nodes and e is the number of edges.

Table 2 illustrates the execution of the algorithm on the example DAG in Figure 7. The column 'current `latest`' indicates the value of `latest` at the start of the iteration. The column 'new `latest`' indicates the value of `latest` at the end of the iteration.

[1]The root and leaf nodes of a DAG are, by definition, partition nodes.

iteration i	N_i	current `latest`	new `latest`	partition node?
1	A	A	C	yes
2	B	C	D	no
3	C	D	D	no
4	D	D	F	yes
5	E	F	G	no
6	F	G	G	no
7	G	G	G	yes

Table 2: Execution of the partitioning algorithm on the DAG in Figure 7.

3.5.2 An Algorithm for Finding Redundant Edges

An efficient algorithm is proposed for finding the redundant edges of a partition. The algorithm iterates through all nodes in the partition except the last node. At each iteration, each edge extending from the current node is compared with parallel paths in the DAG. Let $O = (N_1, ..., N_{n_P})$ be a topological sort of the partition nodes, where n_P is the number of nodes in the partition. Redundant edges are determined as follows:

for $i \leftarrow 1$ to $n_P - 1$
  for each $N_j \in S(N_i)$
    for each $N_k \in S(N_i)$, where $N_k \neq N_j$
      if $l(edge_{N_iN_k}) + D(N_k, N_j) \geq l(edge_{N_iN_j})$
        $edge_{N_iN_j}$ is redundant

where $S(N_i)$ is the set of immediate successors of node N_i in the DAG, and $l(edge_{N_iN_k})$ is the latency of $edge_{N_iN_k}$. $D(N_k, N_j)$ is the critical-path distance in the DAG from node N_k to node N_j. $D(N_k, N_j) = -\infty$ if no path exists from N_k to N_i.

The elimination of redundant edges requires the critical-path distance between all nodes in the partition. An algorithm described in [7] calculates the critical-path distance from one node of a DAG to every other node in the DAG. The worst-case execution time of this algorithm is $O(e_P)$, where e_P is the number of edges in the DAG partition. Therefore, the worst-case execution time for calculating the critical-path distances between all nodes of the partition is $O(n_P e_P)$, where n_P is the number of nodes in the partition. Once critical-path distances are calculated, the algorithm for finding redundant edges iterates through each edge $edge_{N_iN_j}$ of the partition once. At each iteration, $edge_{N_iN_j}$ is compared with all other edges extending from the same node N_i. The number of edges which may extend from a single node is $O(n_P)$. Therefore, the worst-case execution time of the redundant-edge finding algorithm is $O(n_P e_P)$. Although the number of edges in a partition can be $O(n_P^2)$, the DAGs from the experiments described in 2.2 generally have $O(n_P)$ edges.

3.5.3 An Algorithm for Finding Regions

An efficient algorithm is proposed for finding the regions in a partition. Section 3.4 defines a region by conditions on the paths from the region's entry node to its exit node. A region may be equivalently defined by *relative dominance*. Node C dominates node B relative to node A if every path in the DAG from A to B includes C, and $C \neq A$. Under this definition, node B dominates itself relative to A. Let $R(A, B)$ be defined as the dominator of B relative to A which is earliest in a topological sort. If A is a predecessor of B, $R(A, B)$ is uniquely defined because each dominator of B relative to A is necessarily a predecessor or successor of any other dominator of B relative to A. If A is not a predecessor of B or $A = B$, $R(A, B)$ is undefined.

Theorem 3: *Nodes A and B define a region if and only if there exist two immediate predecessors of B, nodes I_1 and I_2, for which $R(A, I_1)$ and $R(A, I_2)$ are defined and $R(A, I_1) \neq R(A, I_2)$.*

Proof: Let I_1 and I_2 be immediate predecessors of B, and $R(A, I_1)$ and $R(A, I_2)$ are defined and $R(A, I_1) \neq R(A, I_2)$. Because $R(A, I_1) \neq R(A, I_2)$, a common dominator of I_1 and I_2 relative to A does not exist (excluding A). Accordingly, there must exist disjoint paths P_{AI_1} from A to I_1 and P_{AI_2} from A to I_2. P_{AI_1} concatonated with $edge_{I_1B}$ and P_{AI_2} concatonated with $edge_{I_2B}$ define two disjoint paths from A to B. Therefore, A and B define a region.

Let nodes A' and B' define a region. By definition, there exist two disjoint paths $P_{A'B'}$ and $P'_{A'B'}$ from A' and B'. $P_{A'B'}$ and $P'_{A'B'}$ must each include a different immediate predecessor of B', nodes I'_1 and I'_2 respectively. A is a predecessor of I'_1 and I'_2 so $R(A, I'_1)$ and $R(A, I'_2)$ are defined, and $R(A, I'_1) \neq R(A, I'_2)$ because there exist disjoint paths from A to I'_1 and from A to I'_2. □

Let $O = (N_1, .., N_n)$ be a topological sort of the nodes in a partition. The value $R(A, B)$ for each pair of nodes A and B in the partition can be determined as follows:

for $i \leftarrow 2$ to n
  for $j \leftarrow 1$ to i
    if $R(N_j, N_k)$ is undefined $\forall\ N_k \in P(N_i)$
      if $N_j \in P(N_i)$
        $R(N_j, N_i) \leftarrow N_i$
      else
        $R(N_j, N_i) \leftarrow$ undefined
    else if $R(N_j, N_{k_1}) \neq R(N_j, N_{k_2})$,
        for some $N_{k_1} \in P(N_i), N_{k_2} \in P(N_i)$ where
        $R(N_j, N_{k_1})$ and $R(N_j, N_{k_2})$ are defined
      $R(N_j, N_i) \leftarrow N_i$
      N_j and N_i define a region
    else
      for some $N_k \in P(N_i)$ where $R(N_j, N_k)$ is defined
        $R(N_j, N_i) \leftarrow R(N_j, N_k)$

where $P(N_i)$ is the set of immediate predecessors of node N_i in the DAG.

For each iteration i of the outer loop, every edge terminating at a node preceding N_i in order O is considered $O(1)$ times. Therefore, the worst case execution time of the algorithm is $O(n_P e_P)$.

3.5.4 An Algorithm for Region Linearization

An algorithm for region linearization is proposed which uses critical-path list scheduling to find a schedule S for a region R. If node order O of S is a dominant order for the region R as defined in Section 3.4 then O is enforced by linearizing the region nodes in the DAG.

The region finding algorithm described in Section 3.5.3 finds the entry node and exit node for each region of a partition. The interior nodes of the region can be determined by performing a forward depth-first search from the entry node and a reverse depth-first search from the exit node. Nodes which are touched in both searches are interior nodes of the region. The searches can also identify side-entry nodes and side-exit nodes.

By Theorem 2, in scheduling a non-SESE region R, no interior node may be scheduled before a side-exit node which is not a predecessor of the side-exit node in the DAG. Symmetrically, no interior node may be scheduled after a side-entry node which is not a successor of the side-entry node in the DAG. To enforce these constraints during region scheduling, temporary edges are added to the region subDAG. Latency one edges are added from each side-exit node to each interior node which is not a predecessor of the side-exit node. Similarly, latency one edges are added to each side-entry node from every interior node which is not a successor of the side-entry node. If there exists a cycle in the dependency graph after this transformation, no dominant order for the region is found. The temporary dependencies are removed after scheduling the region.

After adding the temporary dependencies, the region nodes are scheduled using critical-path list scheduling. If the node order O of the resulting region schedule is a dominant order of the region, then the region is linearized as described in Section 3.4.

Finding the interior nodes of a region requires two depth-first searches of the partition. The worst-case execution time of the searches is $O(e_P)$. A non-SESE region may have $O(n_R)$ side-exit and side-entry nodes, where n_R is the number of nodes in the region. Each side-exit and side-entry node may require the addition of $O(n_R)$ temporary dependences to and from other region nodes. Therefore, adding and removing temporary dependencies to a region has a worst-case execution time of $O(n_R^2)$. Critical-path list scheduling of the region has a worst-case execution time of $O(n_R^2)$ [15]. If a dominant order is produced by list scheduling, replacing the region subDAG with a linearized subDAG requires the deletion of $O(n_R)$ edges and the insertion of $O(n_R)$ edges. Collectively, therefore, the worst-case execution time of the region linearization algorithm is $O(e_P + n_R^2)$.

3.6 DAG Transformations, Experimental Results

The experiment described in Section 2.2 was repeated with GCC's instruction scheduler replaced by an optimal instruction scheduler that includes the DAG transformations and the basic integer-program formulation. As before, each basic block is first scheduled using list scheduling and the basic blocks with schedules that are shown to be optimal are removed from further consideration. For this experiment, the DAG transformations are then applied to the remaining basic blocks, and the basic blocks are scheduled again using list scheduling. The resulting schedules are checked for optimality and those basic blocks without optimal schedules are then solved using integer programming. The results from this experiment are shown in Table 1.

Various observations can be made by comparing the data in Table 3 with the data obtained using only the basic formulation in Table 1. The DAG transformations reduce the total number of integer programs by 28%, to 374 from 517. The DAG transformations reduce the number of integer programs that time out by 83%, to 6 from 35. Total optimal scheduling time using the DAG transformations is reduced by about five-fold, to 7743 seconds from 35,879 seconds. The number of basic blocks that have an improved schedule using the DAG transformations increases by 60%, to 24 from 15. There is a 1 cycle average schedule improvement for the 15 blocks which improved using the basic formulation alone. The additional 9 basic blocks which improved using the DAG transformations have an average improvement of 1.8 cycles. This suggests that the DAG transformations help solve the more valuable problems with larger cycle improvements. Overall these data show that graph transformation is an important technology for producing optimal instruction schedules in reasonable time. However additional new technology is clearly needed.

Total Basic Blocks (BB)	7,402
BB Optimal from List Scheduling	6885
BB Optimal from DAG Transformations	143
BB Passed to IP Formulation	374
BB IP Solved Optimally	368
BB IP Timed Out	6
BB IP Improved and Optimal	23
BB IP Improved but Not Optimal	1
Total Cycles IP Improved	31
Total Scheduling Time (sec.)	7,743

Table 3: Experimental results using DAG transformations and the basic integer-programming formulation.

4 Advanced Integer-Programming Formulation

This section describes an advanced formulation for optimal instruction scheduling which dramatically decreases integer-program solution time compared with the basic formulation.

4.1 Advanced Scheduling-Range Reduction

As described in Section 2.1, the basic formulation uses a technique that can reduce an instruction's scheduling range, and hence the number of scheduling variables. Significant additional scheduling-range reductions are possible.

The basic formulation uses *static range reduction* based on critical-path distance or the number of successors and predecessors. This technique is termed static range reduction because it is based on static DAG properties. An additional criterion for static range reduction is proposed which uses the number of predecessors (or successors) and the minimum latency from (or to) any immediate predecessor (or successor). For the r-issue processor defined earlier, if instruction i has p_i predecessors, the predecessors will occupy at least $\lfloor p_i/r \rfloor$ cycles. If the minimum latency from a predecessor of i to i is $pred\_minl_i$, i can be scheduled no sooner than cycle $1 + \lfloor p_i/r \rfloor + pred\_minl_i$. Given this additional criterion, Equation 1 can be extended to create a tighter lower bound L_i on i's scheduling range:

$$L_i = 1 + \max\{cr_i, \lceil (1+p_i)/r \rceil - 1, \lfloor p_i/r \rfloor + pred\_minl_i\}$$

Symmetrically, Equation 2 can be extended to create a tighter upper bound U_i on i's scheduling range:

$$U_i = m - \max\{cl_i, \lceil (1+s_i)/r \rceil - 1, \lfloor s_i/r \rfloor + succ\_minl_i\}$$

where $succ\_minl_i$ is the minimum latency from i to a successor of i.

A new range reduction technique, *iterative range reduction*, is proposed. Iterative range reduction uses initial logical implications (described below) to reduce the scheduling range of one or more instructions. This range reduction may in turn allow the ranges of predecessors or successors to be tightened. An instruction i's lower bound can be tightened by selecting the maximum of:

- The static lower bound L_i, or

- For each immediate predecessor j, j's lower bound plus the latency of $edge_{ji}$.

The second criterion allows i's lower bound to be tightened iteratively as the lower bounds of i's immediate predecessors are tightened iteratively. Similarly, instruction i's upper bound can be tightened by selecting the minimum of:

- The static upper bound U_i, or
- For each immediate successor j, j's upper bound minus the latency of $edge_{ij}$.

Following an initial logical implication, the predecessor and successor range reductions may iteratively propagate through the DAG and may lead to additional logical implications that can reduce scheduling ranges. These new logical implications may in turn allow additional predecessor and successor range reductions. This process can iterate until no further reductions are possible.

For the r-issue processor defined earlier, a logical implication can be made for instructions that have a one-cycle scheduling range. If an instruction i has a one-cycle scheduling range that spans cycle C^k, i must be scheduled at cycle C^k. If r instructions with one-cycle ranges must be scheduled at cycle C^k, no other instruction can be scheduled at cycle C^k. Thus, cycle C^k can be removed from the scheduling range of any other instruction j which includes cycle C^k. Based on j's range reduction, the ranges of j's predecessors and successors may be reduced. This may lead to additional instructions with one-cycle ranges, and the process may iterate for further range reductions.

Iterative range reduction can lead to scheduling ranges that are infeasible, which implies that no length m schedule exists. Two infeasibility tests are:

- The scheduling range of any node is empty because its upper bound is less than its lower bound.
- For any k-cycle range, more than rk instructions have scheduling ranges that are completely contained within the k cycles, i.e., the scheduling ranges violate the *pigeon hole principle* [10].

Figure 8 illustrates iterative range reduction using the one-cycle logical implication for a single-issue processor. Figure 8a shows each node labeled with the lower and upper bounds that are computed using static range reduction for a schedule of length six[2]. Nodes A, C, E and F have one-cycle ranges, so the corresponding cycles (1,2,5 and 6) are removed from the scheduling ranges of all other nodes, resulting in the scheduling ranges shown in Figure 8b. Predecessor and successor ranges can then be tightened using the iterative range reduction criterion, producing the scheduling ranges shown in 8c. The resulting scheduling ranges are infeasible because node B's scheduling range is empty.

A second logical implication based on *probing* is used to reduce scheduling range. Probing is a general approach that can be used for preprocessing any 0-1 integer program to improve its solution time [20]. Probing selects a 0-1 integer program variable and attempts to show that the variable cannot be 1 by assuming the variable's value is 1 and then showing that the resulting problem is infeasible. If the problem is infeasible, by contradiction, the variable's value must

[2]List scheduling produces a length 7 schedule for this DAG, so an integer program is produced to find the next shorter schedule, as described in Section 2.1.

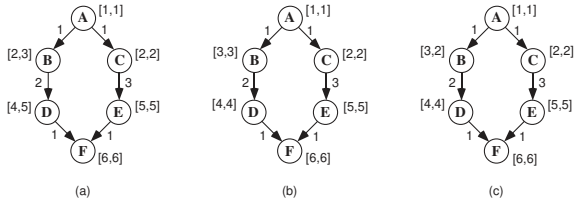

Figure 8: Example scheduling range reduction based on one-cycle scheduling range.

be 0 and the variable can be eliminated from the problem. General-purpose probing is used in commercial solvers, but has a very high computation cost [12].

A specific probing technique called *instruction probing* is proposed for reducing instruction scheduling ranges. Instruction probing is computationally efficient because it exploits knowledge of the instruction scheduling problem and exploits the DAG's structure. Instruction probing is done for each instruction i, starting with i's lower bound. A lower-bound probe consists of temporarily setting i's upper bound equal to the current lower bound. This has the effect of temporarily scheduling i at the current lower bound. Based on i's reduced scheduling range, the ranges of i's predecessors are temporarily tightened throughout the DAG. If the resulting scheduling ranges are infeasible, the probe is successful and i's lower bound is permanently increased by 1. Based on i's new lower bound, the ranges of i's successors are permanently tightened throughout the DAG. If the resulting scheduling ranges are infeasible, the overall scheduling problem is feasible. Otherwise, the new lower bound is probed and the process repeats. If a lower-bound probe is unsuccessful, i's lower-bound probing is complete. i's upper bound is then probed in a symmetric manner.

Figure 9 illustrates the use of instruction probing for a single-issue processor. Figure 9a shows the scheduling ranges that are produced using static range reduction for a schedule of length 8. Figure 9b shows the temporary scheduling ranges that result from probing node B's lower bound. Based on the one-cycle logical implication, cycle 2 is removed from node C's range. Node C's increased lower bound in turn causes the lower bounds for nodes E, G and H to be temporarily tightened. Because node E has a one-cycle range (cycle 5), cycle 5 is removed from node D's range, which in turn causes node F's lower bound to be tightened. Nodes F and G must be scheduled at the same cycle, which is infeasible. Thus, node B cannot be scheduled at cycle 2 and cycle 2 is permanently removed from B's scheduling range. The consequence of B's permanent range reduction is shown in Figure 9c. Based on B's tightened lower bound, the lower bounds of nodes D, F and H are permanently tightened. Based on node B's one-cycle range, cycle 3 is removed from node C's range. Due to node D's one-cycle range, cycle 6 is removed from node G's range. The resulting ranges are infeasible because nodes F and G must be scheduled at the same cycle, thus no 8-cycle schedule exists for the DAG.

4.2 Optimal Region Scheduling

For a region as defined in Section 3, the critical-path distance between the region's entry and exit nodes may not be sufficiently tight. This can lead to excessive integer-program solution time because the integer program produced from the

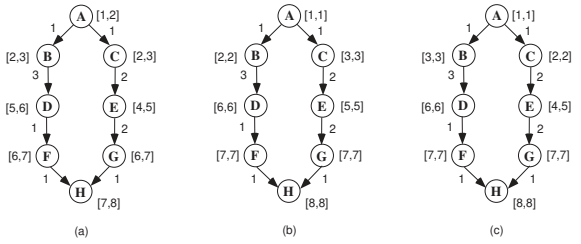

Figure 9: Example scheduling range reduction using instruction probing.

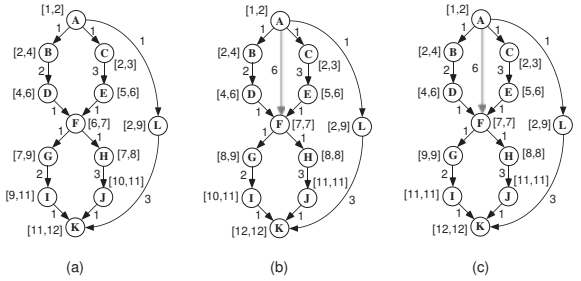

Figure 10: Example pseudo edge insertion.

DAG is under-constrained. A region is *loose* if the critical-path distance from the region's entry node to its exit node is less than the distance from the entry node to the exit node in an optimal schedule of the region. Clearly in any valid overall schedule, a region's entry and exit nodes can be no closer than the distance between these nodes in an optimal schedule of the region. A loose region can be tightened by computing an optimal schedule for the region to determine the minimum distance between the region's entry and exit nodes. A *pseudo edge* can then be added to the DAG from the entry node to the exit node, with a latency that equals the distance between these nodes in the optimal region schedule. Pseudo edges may allow the scheduling ranges of the entry and exits nodes to be reduced. These range reductions may then iteratively propagate through the DAG as described in the previous subsection and may result in scheduling ranges that are infeasible for scheduling the basic block in m cycles. Even if the overall scheduling problem is not shown to be infeasible, the problem will solve faster because of the reduced number of scheduling variables.

The application of region scheduling is illustrated in Figure 10. Figure 10a shows the scheduling ranges for a schedule of length 12. The region AF is an instance of the region in Figure 8, which has an optimal schedule of length 7. Thus, after region scheduling is applied to region AF a latency 6 pseudo edge is added from node A to node F, as shown in Figure 10b. The pseudo edge causes the lower bounds for nodes F, G, H, I, J and K to be tightened to the ranges shown in Figure 10b. Based on node H's one-cycle range, cycle 8 is removed from node G's range. The increase in node G's lower bound causes node I's lower bound to increase. The resulting ranges are shown in Figure 10c. These ranges are infeasible because nodes I and J must be scheduled at the same cycle. Thus, the overall schedule is shown to be infeasible based on the optimal schedule of only one inner region.

Optimal region schedules are computed starting with the smallest inner regions, progressing outward to the largest outer region, the entire DAG partition. Optimal schedules for the inner regions can usually be found quickly using the infeasibility tests. If an optimal region schedule requires an integer program and the inner regions have pseudo edges, a dependency constraint is produced for each pseudo edge, in the manner described in Section 2.1. As the optimal region scheduling process moves outward, the additional pseudo edges progressively constrain the scheduling of the large outer regions, allowing the optimal scheduling of the outer regions to be solved quickly, usually using only the infeasibility tests.

4.3 Branch-and-Cut

Typically 0-1 integer programs are initially solved as a linear program (LP) in which the solution values can be non-integers [24]. If a variable in the initial solution is a non-integer, *branch-and-bound* can be used to find an all integer solution [24]. Branch-and-bound selects a *branch variable*, which is one of the variables that was not an integer in the LP solution. Two subproblems are created in which the branch variable is fixed to 0 and to 1, and the subproblems are solved as LPs. The collection of subproblems form a *branch-and-bound tree* [24]. The root of the tree is the initial LP of the integer program and each node is an LP subproblem with some variables fixed to integer values. By solving the LP subproblems, the branch-and-bound tree is traversed until an all integer solution is found, or the problem is determined to be infeasible [24].

For some integer programming applications, the number of nodes in the branch-and-bound tree that must be solved to produce an optimal integer solution can be dramatically reduced by adaptively adding application-specific constraints, or *cuts*, to the LP subproblem at each node in the branch-and-bound tree. The cuts are designed to eliminate areas of the solution space in the LP subproblem which contain no integer solutions. This enhancement to the branch-and-bound method is called *branch-and-cut* [24]. Two types of cuts are proposed for solving instruction scheduling integer programs: *dependency cuts* and *spreading cuts*.

4.3.1 Dependency Cuts

In an LP solution, an instruction may be fractionally scheduled over multiple cycles. For an instruction k that is data dependent on instruction j, fractions of j can be scheduled after all cycles in which k is scheduled, without violating the corresponding dependency constraint. This is illustrated in Table 4, which shows a partial LP solution for a scheduling problem that includes instructions j and k, where k is dependent on j with latency 1. The solution satisfies the dependency constraint between j and k because j is scheduled at cycle $1 * 0.5 + 5 * 0.5 = 3$, while k is scheduled at cycle 4. However a fraction of j is scheduled after all fractions of k. This invalid solution can be eliminated in the subsequent LP subproblems by adding the following *dependency cut* for cycle c, the last cycle in which j can be scheduled given the position of the last fraction of k in the current LP solution:

$$\sum_{i=LB(j)}^{c} x_j^i \geq \sum_{i=LB(k)}^{c+l_{jk}} x_k^i$$

where $LB(j)$ and $LB(k)$ are the scheduling range lower bounds for j and k, respectively, and l_{jk} is the latency of the dependency between j and k.

For example, the following dependency cut can be added for cycle 3 for the solution in Table 4 to prevent j from being fractionally scheduled after a fraction k in cycle 4 in subsequent LP subproblems:

$$x_j^1 + x_j^2 + x_j^3 \geq x_k^2 + x_k^3 + x_k^4$$

clock cycle	*variables*
C_1	$x_j^1 = 0.5$
C_2	
C_3	
C_4	$x_k^4 = 1.0$
C_5	$x_j^5 = 0.5$

Table 4: Example LP solution to illustrate a dependency cut.

4.3.2 Spreading Cuts

In an LP solution, an instruction k may be fractionally scheduled closer to k's immediate predecessors than the latencies allow in an integer solution, but still satisfy the corresponding dependency constraints. This is illustrated in Table 5, which shows a partial LP solution for a scheduling problem which includes instructions i, j, and k. Instruction k is dependent on both i and j with latency 2. The solution satisfies the dependency constraints of the LP because k is scheduled at cycle $3 * 0.5 + 4 * 0.5 = 3.5$, and i and j are scheduled at cycle $1 * 0.5 + 2 * 0.5 = 1.5$. However, in cycle 3, a fraction of k is scheduled closer to i and j than the latency allows in an all integer solution. This invalid solution can be eliminated in the subsequent LP subproblems by adding the following *spreading cut* for cycle c, the lowest cycle in which a fraction of k is scheduled in the current solution:

$$\sum_{i=c-l+1}^{c} x_k^i + \sum_{I \in P(k)} x_I^{c-l+1} \leq 1$$

where $P(k)$ is the set of immediate predecessors of instruction k.

For example, the following spreading cut can be added for cycle 3 for the solution in Table 5 to force the fractions of i and j to be spread further apart from a fraction of k in cycle 3 in subsequent LP subproblems:

$$x_k^2 + x_k^3 + x_i^2 + x_j^2 \leq 1$$

clock cycle	*variables*	
C_1	$x_i^1 = 0.5$	$x_j^1 = 0.5$
C_2	$x_i^2 = 0.5$	$x_j^2 = 0.5$
C_3	$x_k^3 = 0.5$	
C_4	$x_k^4 = 0.5$	

Table 5: Example LP solution to illustrate a spreading cut.

Symmetric spreading cuts can be used to prevent instructions from being fractionally scheduled closer to their immediate successors than the latencies allow in an integer solution.

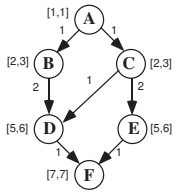

Figure 11: Example DAG with redundant dependency $edge_{CD}$.

4.4 Redundant Constraints

Some constraints in the integer program may be redundant and can be removed. This simplifies the integer program and reduces its solution time.

If an instruction i has a one cycle scheduling range at cycle C, the must-schedule constraint for instruction i can be eliminated, and the issue constraint for cycle C can be eliminated for a single-issue processor.

The integer program includes a dependency constraint for each DAG edge. The dependency constraint ensures that each instruction is scheduled sufficiently earlier than any of its dependent successors. However, if the scheduling ranges of dependent instructions are spaced far enough apart, the data dependency between the two instructions will necessarily be satisfied in the schedule produced by the integer program. In this case, the dependency constraint for the corresponding edge can be removed from the integer program. More precisely, if an instruction k is dependent on instruction j with latency L, then the j to k dependency constraint is redundant if:

$$L + \text{Upper bound of } j \leq \text{Lower bound of } k$$

For the example DAG in Figure 11, the dependency constraint for $edge_{CD}$ is redundant because the upper bound for node C is cycle 3, the lower bound for node D is cycle 5 and the latency of $edge_{CD}$ is 1.

4.5 Algebraic Simplification

An algebraic simplification reduces the number of dependency constraint terms and reduces the size of the coefficients of the remaining terms. The basic dependency constraint for $edge_{jk}$ has one term for each cycle in j's range and one term for each cycle in k's range. The dependency constraint is simplified to include terms only for cycles in which j's and k's scheduling ranges overlap. The simplified constraints allow the integer program to solve faster.

Suppose instruction i is dependent on k with latency L. The scheduling ranges for k and i are $[a,b]$ and $[c,d]$, respectively, and the dependency constraint is not redundant. That is, $c + 1 \leq b + L$. The basic dependency constraint is:

$$\sum_{j=a}^{b} j * x_k^j + L \leq \sum_{j=c}^{d} j * x_i^j \qquad (3)$$

Subtracting c from both sides of (3) yields:

$$\sum_{j=a}^{b} j * x_k^j + L - c \leq \sum_{j=c}^{d} j * x_i^j - c$$

Because the x_k and x_i variables are nonzero for only one cycle j:

$$\sum_{j=a}^{b}(j+L-c)*x_k^j \leq \sum_{j=c}^{d}(j-c)*x_i^j \quad (4)$$

After schedule range tightening, $a \leq c-L$, and because the dependency constraint is not redundant, $c-L+1 \leq b$. Accordingly, (4) may be expanded to:

$$\sum_{j=a}^{c-L}(j+L-c)*x_k^j + \sum_{j=c-L+1}^{b}(j+L-c)*x_k^j \leq \sum_{j=c}^{d}(j-c)*x_i^j \quad (5)$$

The right side of (5) is nonnegative. If x_k^j is nonzero for a cycle $j \leq c-L$, the left side of (5) is nonpositive. Then, the inequality is necessarily satisfied independent of the values of the x_i variables. Therefore, the left summation on the left hand side of (5) may be eliminated, and the dependency constraint becomes:

$$\sum_{j=c-L+1}^{b}(j+L-c)*x_k^j \leq \sum_{j=c}^{d}(j-c)*x_i^j \quad (6)$$

Let $M = b+L-c$. The right hand side of (6) can be transformed as follows:

$$\sum_{j=c}^{d}(j-c)*x_i^j = M - \sum_{j=c}^{d}(M-j+c)*x_i^j$$

Because the left hand side of (6) is at most M, all right hand side cycles after $b+L-1$ don't affect the constraint because they have negative coefficients and will produce a right hand side result that is greater than M. Thus the right hand side of (6) can be simplified to:

$$M - \sum_{j=c}^{b+L-1}(M-j+c)*x_i^j \quad (7)$$

Substituting (7) for the right hand side in (6), yields the simplified constraint:

$$\sum_{j=c-L+1}^{b}(j+L-c)*x_k^j \leq M - \sum_{j=c}^{b+L-1}(M-j+c)*x_i^j$$

To illustrate the simplification, assume k has scheduling range [8,15], i has scheduling range [14,78], and i is dependent on k with latency 1. The original data dependency constraint:

$$\sum_{j=8}^{15} j*x_k^j + 1 \leq \sum_{j=14}^{78} j*x_i^j$$

is simplified to:

$$\sum_{j=14-1+1}^{15}(j+1-14)*x_k^j \leq (15+1-14) - \sum_{j=14}^{15+1-1}(15+1-j)*x_i^j$$

which is:

$$x_k^{14} + 2*x_k^{15} \leq 2 - 2*x_i^{14} - x_i^{15}$$

Before simplification the dependency constraint has 74 terms. After simplification there are only 4 variable terms. The maximum sized coefficient has been reduced to 2 from 74.

4.6 Advanced IP Formulation, Experimental Results

The experiment described in 2.2 is repeated with GCC's instruction scheduler replaced by an optimal instruction scheduler that includes the DAG transformations and the advanced integer-program formulation. The experimental results in Table 6 show the dramatic improvement provided by the new optimal scheduler. All basic blocks are scheduled optimally and the scheduling time is very reasonable. The graph transformations, advanced range reduction and region scheduling techniques reduce to 22 the number of basic blocks that require an integer program, down from 517 using the basic formulation. These 22 most difficult problems require a total solver time of only 45 seconds, an average of only 2 seconds each. The total increase in compilation time is only 98 seconds, a 14% increase in total compilation time, which includes the time for DAG transformations, advanced range reduction and region scheduling. Total scheduling time is reduced by more than 300 fold compared with the basic formulation. The improvement in code quality is more than 4 times that of the basic formulation, 66 static cycles compared with 15 cycles. The additional 6 basic blocks that are solved optimally using the advanced formulation all have improved schedules, with the average improvement of 5.8 cycles. This suggests that the hardest problems to solve are those which provide the most performance improvement.

Total Basic Blocks (BB)	7,402
BB Optimal from List Scheduling	6,885
BB Optimal from Graphs Transformation	143
BB Optimal from Advanced Range Reduction and from Region Scheduling	353
BB Passed to IP Formulation	22
BB IP Solved Optimally	22
BB IP Timed Out	0
BB Improved and Optimal	29
BB Improved but Not Optimal	0
Total Cycles Improved	66
Total Scheduling Time (sec.)	98

Table 6: Experimental results using DAG transformations and the advanced integer programming formulation.

The new approach can optimally schedule very large basic blocks. The scattergram in Figure 12 shows a dot for each of the 517 basic blocks that are processed by the graph transformations and the advanced integer programming formulation. The axes indicate the block's size and the time to optimally schedule that block. This figure shows that many very large blocks, as large as 1000 instructions, are optimally scheduled in a short time.

5 Summary

This paper presents a new approach to optimal instruction scheduling which is fast. The approach quickly identifies most basic blocks for which list scheduling produces an optimal schedule, without using integer programming. For the remaining blocks, a simplified DAG and an advanced integer-programming formulation lead to optimal scheduling times which average a few seconds per block, even for a benchmark set with some very large basic blocks. These results show that the easiest of the hard instruction scheduling

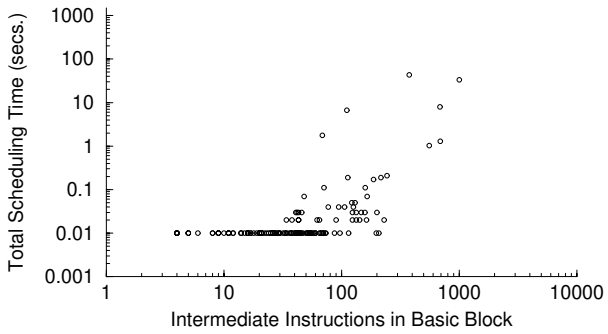

Figure 12: Scatter-gram of basic-block size versus optimal scheduling time.

problems can be solved in reasonable time. This is an important first step toward solving harder instruction scheduling problems in reasonable time, including scheduling very large basic blocks for long-latency multiple-issue processors. The proposed approach will also serve as a solid base for future work on optimal formulations of combined local instruction scheduling and local register allocation that can be solved in reasonable time for large basic blocks.

References

[1] S. Arya. An Optimal Instruction-Scheduling Model for a Class of Vector Processors. *IEEE Transactions on Computers*, C-34(11):981–995, November 1985.

[2] D. Bernstein and I. Gertner. Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[3] D. Bernstein, M. Rodeh, and I. Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Transactions on Computers*, 38(9):1308–1313, September 1989.

[4] R. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. In *Proceedings of Conference on Parallel Architectures and Compilation Techniques*, August 1994.

[5] C-M Chang, C-M Chen, and C-T King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-Issue Processors. *Computers and Mathematics with Applications*, 34(9):1–14, November 1997.

[6] H. Chou and C. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):303–313, March 1995.

[7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1989.

[8] A. Ertl and A. Krall. Optimal Instruction Scheduling Using Constraint Logic Programming. In *Programming Language Implementation and Logic Programming (PLILP)*. Springer-Verlag, 1991.

[9] D. Goodwin and K. Wilken. Optimal and Near-Optimal Global Register Allocation Using 0-1 Integer Programming. *Software—Practice and Experience*, 26(8):929–965, August 1996.

[10] J. Grossman. *Discrete Mathematics*. Macmillan, 1990.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996. Second Edition.

[12] ILOG. *ILOG CPLEX 6.5 User's Manual*. ILOG, 1999.

[13] C. Kessler. Scheduling Expression DAGs for Miniml Register Need. *Computer Languages*, 24(1):33–53, April 1998.

[14] R. Leupers and P. Marwedel. Time-Constrained Code Compaction for DSPs. *IEEE Transactions VLSI Systems*, 5(1):112–122, March 1997.

[15] S. Munchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] G. Nemhauser. The Age of Optimization: Solving Large-Scale Real-World Problems. *Operations Research*, 42(1):5–13, Jan.–Feb. 1994.

[17] K. Palem and B. Simons. Scheduling Time-Critical Instructions on RISC Machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, September 1993.

[18] K. Pingali and G. Bilardi. Optimal Control Dependence and the Roman Chariots Problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[19] W. Pugh. The Omega Test: A Fast and Practical Integer programming Algorithm for Dependence Analysis. In *Proceedings Supercomputing '91*, pages 18–22, Nov 1991.

[20] M. Savelsbergh. Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA Journal of Computing*, 6(4):445–454, Fall 1994.

[21] P. Sweany and S. Beaty. Instruction Scheduling Using Simulated Annealing. In *Proc. 3rd International Conference on Massively Parallel Computing Systems (MPCS '98)*, 1998.

[22] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction Scheduling to Reduce Switching Activity of Off-Chip Buses for Low-Power Systems with Caches. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E81-A(12):2621–2629, December 1998.

[23] S. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode*. PhD thesis, Carnegie Mellon University, December 1982.

[24] Laurence A. Wolesey. *Integer Programming*. John Wiley & Sons, Inc., 1998.