Fast Optimal Instruction Scheduling for Single-issue Processors with Arbitrary Latencies

Peter van Beek[1] and Kent Wilken[2]

[1] Department of Computer Science
University of Waterloo
Waterloo, ON, Canada N2L 3G1
vanbeek@uwaterloo.ca
[2] Department of Electrical and Computer Engineering
University of California
Davis, CA, USA 95616
wilken@ece.ucdavis.edu

Abstract Instruction scheduling is one of the most important steps for improving the performance of object code produced by a compiler. The local instruction scheduling problem is to find a minimum length instruction schedule for a basic block subject to precedence, latency, and resource constraints. In this paper we consider local instruction scheduling for single-issue processors with arbitrary latencies. The problem is considered intractable, and heuristic approaches are currently used in production compilers. In contrast, we present a relatively simple approach to instruction scheduling based on constraint programming which is fast and optimal. The proposed approach uses an improved constraint model which allows it to scale up to very large, real problems. We describe powerful redundant constraints that allow a standard constraint solver to solve these scheduling problems in an almost backtrack-free manner. The redundant constraints are lower bounds on selected sub-problems which take advantage of the structure inherent in the problems. Under specified conditions, these constraints are sometimes further improved by testing the consistency of a sub-problem using a fast test. We experimentally evaluated our approach by integrating it into the Gnu Compiler Collection (GCC) and then applying it to the SPEC95 floating point benchmarks. All 7402 of the benchmarks' basic-blocks were optimally scheduled, including basic-blocks with up to 1000 instructions. Our results compare favorably to the best previous approach which is based on integer linear programming (Wilken et al., 2000): Across the same benchmarks, the total optimal scheduling time for their approach is 98 seconds while the total time for our approach is less than 5 seconds.

1 Introduction

Instruction scheduling is one of the most important steps for improving the performance of object code produced by a compiler. The local instruction scheduling problem is to find a minimum length instruction schedule for a basic block—a

straight-line sequence of code with a single entry point and a single exit point—subject to precedence, latency, and resource constraints. In this paper we consider local instruction scheduling for single-issue processors with arbitrary latencies. This is a classic problem which has received a lot of attention in the literature and remains important as single-issue RISC processors are increasingly being used in embedded systems such as automobile brake systems and air-bag controllers.

Instruction scheduling for a single-issue processor is NP-complete if there is no fixed bound on the maximum latency [8, 18]. Such negative results have led to the belief that in production compilers one must take a heuristic or approximation algorithm approach, rather than an exact approach to basic-block scheduling (e.g., see [17]). Recently, however, Wilken et al. [21] showed that through various modeling and algorithmic techniques, integer linear programming could be used to produce optimal instruction schedules for large basic blocks in a reasonable amount of time.

In this paper, we present a relatively simple constraint programming approach to instruction scheduling which is fast and optimal. The key to scaling up to very large, real problems is an improved constraint model. We describe powerful redundant constraints that allow a standard constraint solver to solve these scheduling problems in an almost backtrack-free manner. The redundant constraints are lower bounds on selected sub-problems which take advantage of the structure inherent in the problems. Under specified conditions, these constraints are sometimes further improved by testing the consistency of a sub-problem using a fast test.

We experimentally evaluated our approach by integrating it into the Gnu Compiler Collection (GCC) and then applying it to the SPEC95 floating point benchmarks. All 7402 of the benchmarks' basic-blocks were optimally scheduled, including basic-blocks with up to 1000 instructions. Our results compare favorably to the best previous approach which is based on integer linear programming (Wilken et al., 2000): Across the same benchmarks, the total optimal scheduling time for their approach is 98 seconds while the total time for our approach is less than 5 seconds.

2 Background and Definitions

We first define the instruction scheduling problem studied in this paper followed by a brief review of the needed background from constraint programming (for more background on these topics see, for example, [9, 15, 17]).

Throughout the paper, the number of elements in a set U is denoted by $|U|$, the minimum and maximum values in a finite set U of integers are denoted by $\min(U)$ and $\max(U)$, respectively, and the interval notation $[a, b]$ is used as a shorthand for the set of integers $\{a, a+1, \ldots, b\}$.

We consider single-issue pipelined processors (see [9]). On such processors a single instruction can be issued (begin execution) each clock cycle, but for some

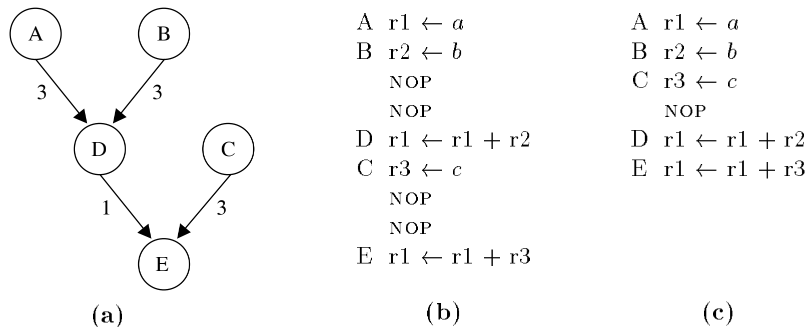

Figure 1. (a) Dependency DAG associated with the instructions to evaluate $(a+b)+c$ on a processor where loads from memory have a latency of 3 cycles and integer operations have a latency of 1 cycle; (b) non-optimal schedule; (c) optimal schedule.

instructions there is a delay or *latency* between when the instruction is issued and when the result is available for other instructions which use the result.

We use the standard labeled directed acyclic graph (DAG) representation of a basic-block, where each node corresponds to an instruction (see [17]). There is an edge from i to j labeled with a positive integer $l(i,j)$ if j must not be issued until i has executed for $l(i,j)$ cycles. In particular, if $l(i,j) = 1$, j can be issued in the next cycle after i has been issued and if $l(i,j) > 1$, there must be some intervening cycles between when i is issued and when j is subsequently issued. These cycles can possibly be filled by other instructions. The *critical path distance* from a node i to a node j in a DAG is the length of the longest path from i to j, if there exists a path from i to j; $-\infty$ otherwise.

Definition 1. (Local Instruction Scheduling Problem) *Given a labeled dependency DAG $G = (N, E)$ for a basic-block, a* schedule *S for a single-issue processor specifies an issue or start time $S(i)$ for each instruction or node such that $S(i) \neq S(j)$, $i, j \in N, i \neq j$ (no two instructions are issued simultaneously), and $S(j) \geq S(i) + l(i,j), (i,j) \in E$ (the issue or start time of an instruction depends upon the issue times and latencies of its predecessors) The* local instruction scheduling problem *is to construct a schedule with minimum length; i.e., $\max\{S(i) \mid i \in N\}$ is minimized.*

Example 1. Figure 1 shows a simple dependency DAG and two possible schedules for the DAG. The non-optimal schedule requires four NOP instructions (null operations) because the values loaded are used by the following instructions. The optimal schedule requires one NOP and completes in three fewer cycles.

Constraint programming is a methodology for solving combinatorial problems. A problem is modeled by specifying constraints on an acceptable solution, where a constraint is simply a relation among several unknowns or variables, each taking a value in a given domain. Such a model is often referred to as a constraint satisfaction problem or CSP model.

Definition 2 (Constraint Satisfaction Problem (CSP)). *A* constraint satisfaction problem *consists of a set of* n variables, $\{x_1, \ldots, x_n\}$*; a finite domain* $dom(x_i)$ *of possible* values *for each variable* x_i, $1 \leq i \leq n$*; and a collection of* r constraints, $\{C_1, \ldots, C_r\}$. *Each constraint* C_i, $1 \leq i \leq r$, *is a constraint over some set of variables, denoted by* $vars(C)$, *that specifies the allowed combinations of values for the variables in* $vars(C)$. *A* solution *to a CSP is an assignment of a value to each variable that satisfies all of the constraints.*

CSPs are often solved using a backtracking algorithm. At every stage of the backtracking search, there is some current partial solution which the algorithm attempts to extend to a full solution by assigning a value to an uninstantiated variable. One of the keys behind the success of constraint programming is the idea of constraint propagation. During the backtracking search when a variable is assigned a value, the constraints are used to reduce the domains of the uninstantiated variables by ensuring that the values in their domains are "consistent" with the constraints. The form of consistency we use in our approach to the instruction scheduling problem is bounds consistency.

Definition 3 (Bounds Consistency). *Given a constraint* C*, a value* $d \in dom(x)$ *for a variable* $x \in vars(C)$ *is said to have a* support *in* C *if there exist values for each of the other variables in* $vars(C) - \{x\}$ *such that* C *is satisfied. A constraint* C *is* bounds consistent *if for each* $x \in vars(C)$*, the value* $\min(dom(x))$ *has a support in* C *and the value* $\max(dom(x))$ *has a support in* C.

A CSP can be made bounds consistent by repeatedly removing unsupported values from the domains of its variables.

Example 2. Consider the CSP model of the small instruction scheduling problem in Example 1 with variables A, ..., E, each with domain $\{1, \ldots, 6\}$, and the following constraints,

C_1: $\mathrm{D} \geq \mathrm{A} + 3$, C_3: $\mathrm{E} \geq \mathrm{C} + 3$, C_5: all-different(A, B, C, D, E),
C_2: $\mathrm{D} \geq \mathrm{B} + 3$, C_4: $\mathrm{E} \geq \mathrm{D} + 1$,

where constraint C_5 enforces that its arguments are pair-wise different. The constraints are not bounds consistent. For example, the minimum value 1 in the domain of D does not have a support in constraint C_1 as there is no corresponding value for A that satisfies the constraint. Enforcing bounds consistency using constraints C_1 through C_4 reduces the domains of the variables as follows: $dom(\mathrm{A}) = \{1, 2\}$, $dom(\mathrm{B}) = \{1, 2\}$, $dom(\mathrm{C}) = \{1, 2, 3\}$, $dom(\mathrm{D}) = \{4, 5\}$, and $dom(\mathrm{E}) = \{5, 6\}$. Subsequently enforcing bounds consistency using constraint C_5 further reduces the domain of C to be $dom(\mathrm{C}) = \{3\}$. Now constraint C_3 is no longer bounds consistent. Re-establishing bounds consistency causes $dom(\mathrm{E}) = \{6\}$.

3 Previous Work

Instruction scheduling for a single-issue processor is NP-complete if there is no fixed bound on the maximum latency d [8, 18].

Previous work has identified polynomial algorithms for the special case when $d \leq 2$. These algorithms can also be used as approximation algorithms for the general problem. Bernstein and Gertner [2] give a polynomial time algorithm based on list scheduling when $d \leq 2$. The algorithm can be used as an approximation algorithm when $d > 2$ and is guaranteed to give a schedule whose length is no more than a factor of $2 - 2/d$ times that of an optimal schedule [3]. Palem and Simons [18] extend this work by allowing timing constraints in the form of release times (the earliest time at which an instruction can start) and deadlines (the latest time by which an instruction must complete). Such constraints can be important in embedded systems[1]. Recently, Wu et al. [22] gave an improved algorithm for the case when $d \leq 2$ and timing constraints are allowed. It is a long-standing open problem whether there exists a polynomial time algorithm for any fixed $d > 2$.

Previous work has also developed optimal algorithms for the general problem when $d > 2$. The approaches taken include dynamic programming [10], integer linear programming [1, 5, 14, 21], and constraint programming [7]. However, with the exception of [21] (to which we do a detailed comparison later in the paper), these previous approaches have only been evaluated on a few problems with the sizes of the problems ranging between 10 and 40 instructions. Further, their experimental results suggest that none of them would scale up beyond problems of this size. For example, Ertl and Krall [7] present a constraint programming approach which solves the problem optimally. Their CSP model has latency constraints and an all-different constraint. As our experiments confirm (see Table 3 and the discussion at the end of Section 5), such a model does not scale beyond 50 instructions. However, real problems can contain a thousand or more instructions.

4 CSP Model

In this section, we present our CSP model of the local instruction scheduling problem. In the constraint programming methodology we cast the problem as a CSP in terms of variables, values, and constraints. The choice of variables defines the search space and the choice of constraints defines how the search space can be reduced so that it can be effectively searched using backtracking search. Each constraint can be classified as to whether it is redundant or non-redundant. A constraint is redundant if its removal from the CSP does not change the set of solutions.

[1] Note that timing constraints can be viewed as just a special case of latency constraints. Thus, any approach that solves the general problem by allowing arbitrary latencies (such as the one in this paper), can also handle timing constraints.

Table 1. Notation used in specifying the constraints.

lower(i)	lower bound of domain of variable i
upper(i)	upper bound of domain of variable i
pred(i)	set of immediate predecessors of node i in DAG
succ(i)	set of immediate successors of node i in DAG
between(i, j)	set of nodes between nodes i and j
$l(i, j)$	latency on edge between nodes i and j
$cp(i, j)$	critical path distance between nodes i and j
$d(i, j)$	lower bound on distance between nodes i and j

We model each instruction by a variable with names $1, \ldots, n$ (we use i to refer interchangeably to variable i, instruction i, and node i in the DAG). Each variable takes a value from the domain $\{1, \ldots, m\}$ which are the available time cycles. Assigning a value $t \in dom(i)$ to a variable i has the intended meaning that instruction i will be issued at time cycle t.

We now specify the five types of constraints in the model: latency, all-different, distance, predecessor and successor constraints. The notation we use is summarized in Table 1. As is clear, for a minimal correct model of the instruction scheduling problem all that is needed are the latency and all-different constraints. The distance, predecessor, and successor constraints are therefore redundant. However, they were found to be essential in improving the efficiency of the search for a schedule.

Latency constraints. Given a labeled dependency DAG $G = (N, E)$, for each pair of variables i and j such that $(i, j) \in E$, a latency constraint of the form $j \geq i + l(i, j)$ is considered for addition to the constraint model. A latency constraint is added if it is not redundant. A latency constraint between i and j is redundant if there exists a $k < j$ such that, $l(i, j) \leq l(i, k) + cp(k, j)$. In other words, the constraint is redundant if there is a path from i to j that goes through k that is equal to or longer than the direct path $l(i, j)$. (If the constraint is redundant, adding it will have no effect as the remaining latency constraints will derive a stronger result.) Since we are enforcing bounds consistency, the actual form of the constraints added to the constraint model are,

$$\text{lower}(j) \geq \text{lower}(i) + l(i, j)$$

and its symmetric version,

$$\text{upper}(i) \leq \text{upper}(j) - l(i, j).$$

The latency constraints are easy to propagate when establishing lower and upper bounds for the variables, and easy to propagate incrementally during the backtracking search.

All-different constraints. A single all-different constraint over all n of the variables is needed to ensure that at most one instruction is issued each cycle. Fast

algorithms for enforcing bounds consistency on an all-different constraint have been proposed. In our implementation, we used the $O(n^2)$ propagator described in [20] and included the optimization suggested by Puget [20] of first removing any fixed values (time cycles that have already been assigned to variables) from the lower and upper bounds of the uninstantiated variables, and the techniques suggested by Leconte [12] for taking advantage of the fact that, when propagating the all-different constraint during the backtracking search, we are *re*-establishing bounds consistency; i.e., the constraint was previously bounds consistent[2].

Distance constraints. Dependency DAGs that arise from real instruction scheduling problems appear to contain much structure, no doubt because they arise from high-level programming languages. In what follows, we are interested in sub-DAGS called regions [21] which are induced from a given dependency DAG. Real problems typically contain many such regions embedded within them, with larger problems containing many thousands.

Definition 4. (Region [21]) *Given a labeled dependency DAG $G = (N, E)$, a pair of nodes $i, j \in N$ define a* region *in G if there is more than one path between i and j and there does not exist a node k distinct from i and j such that every path between i and j goes through k. A node h distinct from i and j that lies on a path from i to j is said to be between i and j and the set of all such nodes is denoted by* $\text{between}(i, j)$.

For each pair of nodes i and j which define a region, a distance constraint of the form $j \geq i + d(i, j)$ is considered for addition to the constraint model. A distance constraint is added if it is an improvement over the critical path distance; i.e., $d(i, j) > cp(i, j)$. (If the distance is not greater than the critical path distance, adding the constraint will have no effect as the latency constraints will derive a stronger result.) The distance constraints are lower bounds on the number of cycles that must elapse between when i is scheduled and j is scheduled. Although syntactically identical to latency constraints and hence propagated in the same manner, they are conceptually distinct and are key to effectively reducing the size of the search space.

An initial estimate of $d(i, j)$ is given by the following.

$$\begin{aligned} d(i,j) = \min\{l(i,k) \mid k \in (\text{succ}(i) \cap \text{between}(i,j))\} - 1 \\ + \mid \text{between}(i,j) \mid \\ + \min\{l(h,j) \mid h \in (\text{pred}(j) \cap \text{between}(i,j))\} - 1 \\ + 1 \end{aligned}$$

To explain, the nodes in $\text{between}(i, j)$ must all be scheduled after node i and before node j. We do not know which node in $\text{between}(i, j)$ will be or must be

[2] Propagators with better worst case complexity are known: $O(n \log n)$ [20] and $O(n)$ [16]. Since the all-different propagator is a bottle-neck in our current implementation, it would be interesting to investigate whether these algorithms would work better in practice on instruction scheduling problems.

scheduled first. However, it can be seen that any successor of node i that is in between(i, j) can only be scheduled once the minimum latency among those successors has been satisfied. As well, once all of the nodes in between(i, j) have been scheduled, node j can only be scheduled once the minimum latency of its predecessors in between(i, j) has been satisfied. The number of nodes that are between node i and node j can quickly be determined given the critical path distances between all pairs of nodes, since a node k is on a path from i to j if $cp(i, k) > 0$ and $cp(k, j) > 0$. The initial estimate of $d(i, j)$ can be viewed as a generalization of rules which were proposed for bounding the range of values for variables in integer programming formulations of instruction scheduling [5, 21], although in that work it was not applied to regions.

Example 3. Consider the dependency DAG shown in Figure 2 (ignore for now the lower and upper bounds associated with the nodes). For the region defined by A and H, $d(\mathrm{A}, \mathrm{H}) = (\min\{l(\mathrm{A}, \mathrm{B}), l(\mathrm{A}, \mathrm{C})\} - 1) + |\mathrm{between}(\mathrm{A}, \mathrm{H})| + (\min\{l(\mathrm{F}, \mathrm{H}), l(\mathrm{G}, \mathrm{H})\} - 1) + 1 = 0 + 6 + 2 + 1 = 9$. Similarly, $d(\mathrm{A}, \mathrm{F}) = 5$ and $d(\mathrm{E}, \mathrm{H}) = 5$. The distance constraint H $\geq$ E + 5 would be added to the constraint model and the distance constraint F $\geq$ A + 5 would not be added (as it is not an improvement over the critical path distance between A and F). The distance constraint between A and H is taken up again in Example 4.

An attempt is made to improve the initial estimate of $d(i, j)$ for the distance constraints if the number of nodes between i and j is sufficiently small. We found that a value of 50 was robust on real problems. This was determined empirically using a set of five instruction scheduling examples of varying size and then verified on an additional set of ten examples.

The method for improvement works as follows. Given an initial estimate of $d(i, j)$, the region defined by i and j is (conceptually) extracted from the DAG and considered in isolation. We test whether scheduling node i at time 1 and node j at time $d(i, j) + 1$ is consistent. The test for consistency is done by propagating the relevant latency constraints and any previously added distance constraints, and an all-different constraint over the variables in the region. If the constraints are inconsistent, the value of $d(i, j)$ is incremented and the test is repeated, stopping when a value is found for $d(i, j)$ such that the constraints over the region are found to be consistent. Note that we are determining lower bounds, not solving the region exactly, as the idea is to test the consistency of the constraints quickly and approximately. The regions in the DAG are examined in an "inside-out" manner so that distance constraints for inner regions can be used in the consistency tests of the larger outer regions.

Example 4. Consider again the dependency DAG shown in Figure 2 and previously discussed in Example 3. The initial estimate of $d(\mathrm{A}, \mathrm{H}) = 9$ can be improved. Figure 2a shows the bounds on the variables after propagating the latency and previously added distance constraints. Propagating the all-different constraint determines that the constraints are inconsistent, because four instructions (D, E, F and G) must be issued in the three-cycle interval [5,7]. Figure 2b shows the bounds on the variables after propagating all the constraints for the

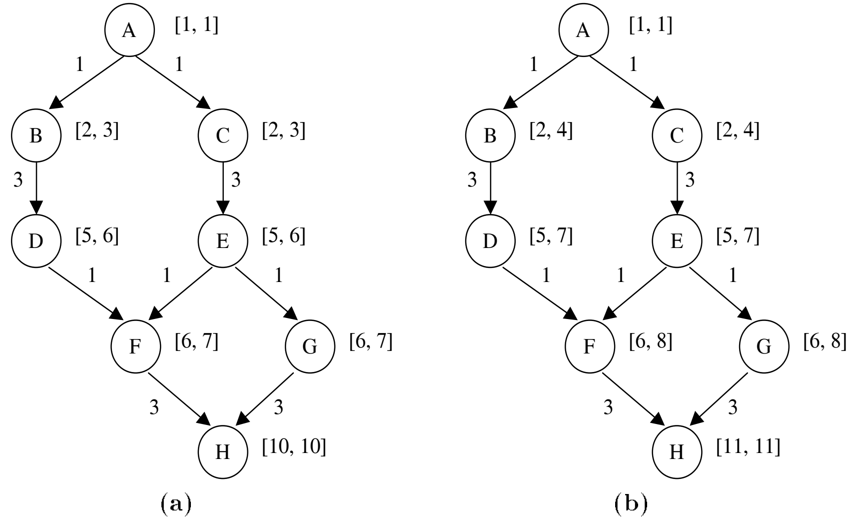

Figure 2. Example of improving the lower bound estimate for the distance constraint for the region defined by A and H.

improved estimate $d(\mathrm{A},\mathrm{H}) = 10$. The constraints are consistent, so the constraint $\mathrm{H} \geq \mathrm{A} + 10$ is added to the constraint model.

Predecessor and successor constraints. For each node i which has more than one immediate predecessor, a single predecessor constraint of the following form is added.

$$\begin{aligned}\mathrm{lower}(i) \geq\ & \min\{\mathrm{lower}(k) \mid k \in P\} \\ & + |P| - 1 \\ & + \min\{l(k,i) \mid k \in P\}, \\ & \quad \text{for every subset } P \text{ of } \mathrm{pred}(i) \text{ where } |P| > 1.\end{aligned}$$

The predecessor constraints can be viewed as both a generalization of the latency constraints and as an adaptation of edge finding rules [4, 11]. It can be seen that a predecessor constraint can be propagated in $O(|\mathrm{pred}(i)|^2)$ time by first sorting the predecessors of i by increasing lower bounds and then stepping through the lower bounds, each time finding the minimum latency among the remaining predecessors. A symmetric version, called successor constraints, for the immediate successors of a node is given by the following.

$$\begin{aligned}\mathrm{upper}(i) \leq\ & \max\{\mathrm{upper}(k) \mid k \in P\} \\ & - |P| + 1 \\ & - \min\{l(i,k) \mid k \in P\}, \\ & \quad \text{for every subset } P \text{ of } \mathrm{succ}(i) \text{ where } |P| > 1.\end{aligned}$$

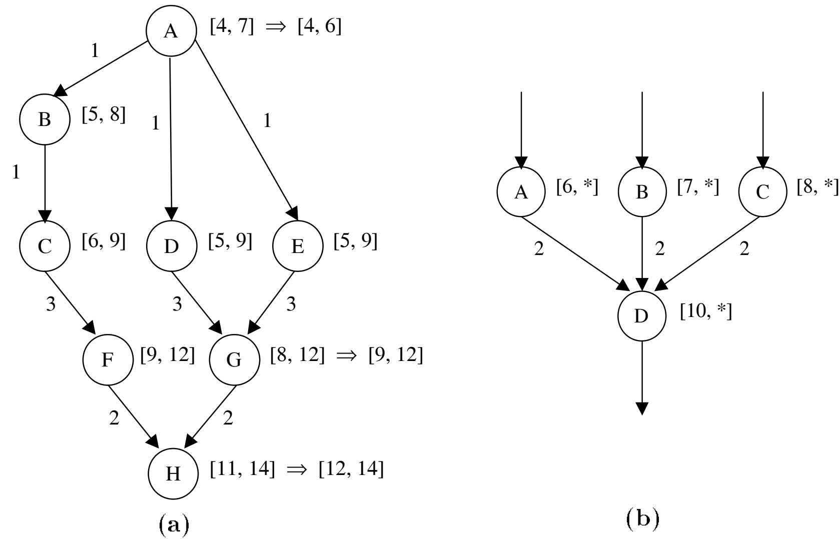

Figure 3. Examples of improving the lower and upper bounds of variables using the predecessor and successor constraints.

Example 5. Suppose that the sub-DAG shown in Figure 3a is embedded in a larger dependency DAG and that it has been determined that lower(A) = 4 and upper(H) = 14. Propagating the latency constraints results in the domains shown closest to the associated node. Propagating the predecessor and successor constraints improves the bounds of A, G, and H. The earliest that one of the predecessors of node G can be scheduled is cycle 5 and therefore cycle 6 is the earliest that the last of its predecessors could be scheduled. Therefore, because the minimum latency between G and its predecessors is 3, the earliest that G can be scheduled is cycle 9. Once the lower bound of G has been raised, in a similar manner the lower bound of H can be raised. As well, the latest that one of the successors of node A can be scheduled is cycle 9 and therefore cycle 7 is the latest that the last of its successors could be scheduled. Therefore, the latest that A can be scheduled is cycle 6. Figure 3b shows an example of a predecessor constraint that initially has no effect but could become effective during the backtracking search as, if either lower(A) or lower(B) are raised during the search, lower(D) can also be raised.

To solve an instance of an instruction scheduling problem, we start by using the constraints to establish the lower bounds of the variables and a lower bound on the length m of an optimal schedule. Given m, the upper bounds of the variables are similarly established and the CSP is passed to the backtracking algorithm. If no solution is found, a length m schedule does not exist and the value of m is incremented, the upper bounds of the variables are re-established using the new value of m, and the new CSP is passed to the backtracking algo-

rithm. This is repeated, each time incrementing m until a solution is found. The backtracking search interleaves constraint propagation with branching on variables. A dynamic variable ordering is used which selects as the next variable to instantiate the variable with the least number of values remaining in its domain, breaking ties by choosing the variable that participates in the most constraints. Given a selected variable x, the backtracking search first branches on x assigned to $\text{lower}(x)$, then on x assigned to $\text{lower}(x) + 1$, and so on, until either a solution is found or the domain of x is exhausted.

Before turning to our experimental results, it is worthwhile summarizing three of the ideas that did not make it into the final version with which we did our full scale experimentation. Our goal was to design an approach that was as simple as possible while maintaining robustness and while the following ideas proved promising when evaluated in isolation on a set of test examples, they appeared to become unnecessary when combined with the improved constraint model we described above. The first technique was identifying cycle cutsets [6] and thereby decomposing a problem into independent subproblems. We found that most of the larger problems in our test suite (not the full benchmark set, but a small subset consisting of some of the harder problems) had small cutsets ranging from two to 20 nodes that approximately evenly decomposed the problem. The second technique was a variation on singleton consistency (see, e.g., [19]) where one temporarily instantiates a variable to a single value and tests the consistency of a subproblem that includes that variable. If the consistency test fails, the value can be removed from the domain of the variable. Wilken et al. [21] showed that a related technique called probing in the context of integer linear programming worked well on the instruction scheduling problem. We found that singleton consistency could sometimes dramatically reduce the domains of the variables prior to search. The third technique was the inclusion of symmetry-breaking constraints which rule out symmetric (non) schedules. Although each of these techniques was not included in our final prototype, it is possible of course that they may still prove important should we encounter harder problems in practice than we have yet seen.

5 Experimental Results

The CSP model was implemented and was embedded inside the Gnu Compiler Collection (GCC) [http://gnu.gcc.org], version 2.8.0. The CSP model was compared experimentally with critical-path list scheduling and with the integer linear programming (ILP) formulation proposed in [21]. The SPEC95 floating point benchmarks [http://www.specbench.org] were compiled by GCC using GCC's native instruction scheduler, which uses critical-path list scheduling, the most popular heuristic scheduling method [17]. The same benchmarks were also compiled using the CSP scheduler. The compilations were done 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 op-

Table 2. Experimental results for the CSP instruction scheduler.

Total Basic Blocks (BB)	7,402
BB Passed to CSP Scheduler	517
BB Solved Optimally by CSP Scheduler	517
BB with Improved Schedule	29
Cycles Improved	66
Total Benchmark Cycles	107,245
CSP Scheduling Time (sec.)	4.5
Baseline Compile Time (sec.)	708

erations. 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 SPEC95 floating point benchmarks were chosen rather than the more recent SPEC2000 benchmarks to allow a direct comparison with the ILP optimal scheduling results in [21]. The optimal schedule length produced by the CSP scheduler was compared with that from the ILP scheduler from [21] for each basic block to verify the correctness of both formulations. The experiments were run on an HP C3000 workstation with a 400MHz PA-8500 processor and 512MB of main memory, the same processor that produced the results in [21].

The filter used in [21] was applied prior to the CSP scheduler to eliminate the trivial scheduling problems, and so that the CSP scheduler solved the same set of problems solved by the ILP scheduler in [21]. The GCC list scheduler is first run to produce an initial feasible schedule of length u, which is an upper bound on the length of an optimal schedule. A lower bound on the schedule length m is determined by the maximum of the critical path from a root node to a leaf node and the node count. If $u = m$ the list schedule is optimal and the CSP scheduler is not called. Also, for each node i the initial domain is tightened using latency constraints for a schedule length of $u - 1$. If the domain of any i is empty, the length u list schedule is optimal and the CSP scheduler is not called. A summary of the results for the CSP scheduler is shown in Table 2 and more detailed results for the CSP scheduler and the ILP scheduler are shown in Figure 4.

The results in Table 2 are identical with the results in [21] with the notable exception that the ILP scheduler uses 98.3 seconds to optimally schedule these benchmarks (a noticeable 14% compile-time increase), whereas the CSP scheduler is 22 times faster, using only 4.5 seconds (a negligible 0.6% compile-time increase). As a point of reference, the GCC list scheduler takes 0.5 seconds to schedule these benchmarks. The cycles measured in Table 2 are static cycles, one cycle for each clock cycle in each schedule. On average static cycles are reduced by 0.06% using the CSP scheduler versus the list schedule. The dynamic cycle savings will tend to be higher because the more complex basic blocks tend to appear in loops where the execution counts are higher (the improvement can be as high as several percent should an improved basic block appear within an

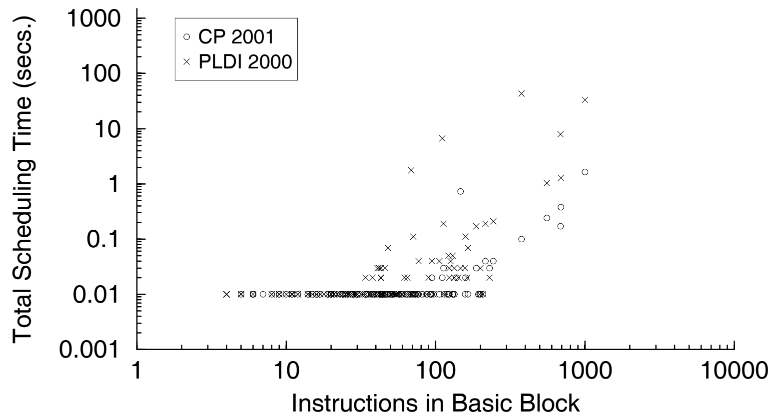

Figure 4. Scattergram of basic block size versus optimal scheduling time for the CSP and ILP schedulers.

application's critical inner loop). Also performance improvement is expected to be much higher for processors that issue multiple instructions per clock cycle, a harder scheduling problem which will be considered in future work.

Figure 4 shows a scattergram of scheduling time versus basic block size which includes a point for each of the 517 basic blocks scheduled by the CSP scheduler in the present experiment. The scattergram also shows corresponding points for the ILP scheduler from the experiment in [21]. The system timer used in both experiments has a resolution of 0.01 seconds and rounds up to the nearest 0.01 second increment. Most of these basic blocks are scheduled within the minimum timer resolution for both schedulers. The CSP scheduler takes more than 0.01 second for only 15 basic blocks while the ILP scheduler takes more than 0.01 second for 42 basic blocks. The maximum time the CSP scheduler takes to schedule an individual basic block is 1.6 seconds (for a 1006-instruction block) and the maximum time for the ILP scheduler is 43.5 seconds (for a 377-instruction block). For the basic blocks which take more than 0.01 second for either scheduler (44 basic blocks), the CSP scheduler is faster in 40 cases and the ILP scheduler is faster in only 4 cases.

Besides being faster and more robust than the ILP scheduler, the code for the CSP scheduler is significantly smaller, which implies it would be easier to implement and maintain in a production compiler. The CSP solver is also self contained, whereas the ILP scheduler uses an external (potentially expensive) commercial ILP solver.

Table 3 shows the results from a set of experiments which were run to quantify the contributions of the three CSP model improvements. The experiments used various of levels of model improvement run at various time limits, applied to fifteen representative hard problems ranging in size from 69 to 1006 instructions that were taken from the SPEC95 floating point, SPEC2000 floating point and

Table 3. Hard problems out of 15 *not* solved within specified time limit (seconds) using: (a) minimal constraint model (only latency and all-different constraints); (b) minimal model plus predecessor and successor constraints; (c) minimal model plus distance constraints based only on initial estimate, no consistency testing; (d) minimal model plus complete distance constraints; and (e) full constraint model (minimal model plus complete distance constraints and predecessor and successor constraints).

Time Limit	(a)	(b)	(c)	(d)	(e)
10	14	14	4	3	0
100	14	13	4	2	0
1000	14	13	4	2	0

MediaBench [13] benchmarks. The results show that the minimal constraint model proposed by Ertl and Krall [7] has poor scaling behavior (see column (a) in Table 3) and that together the three improvements dramatically improve the scaling behavior of a constraint programming approach.

6 Conclusions

We presented a constraint programming approach to instruction scheduling for single-issue processors with arbitrary latencies. The problem is considered intractable, yet our approach is optimal and fast on very large, real problems. The key to scaling up to very large, real problems was in the development of an improved constraint model by identifying techniques for generating powerful redundant constraints. These techniques allow a standard constraint solver to solve these scheduling problems in an almost backtrack-free manner. We performed an extensive experimental evaluation and demonstrated that our approach has the advantage over other previous approaches in terms of the robustness and speed with which optimal schedules can be found.

References

1. S. Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Transactions on Computers*, C-34(11):981–995, 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, 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, 1989.
4. J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.
5. C.-M. Chang, C.-M. Chen, and C.-T. King. Using integer programming for instruction scheduling and register allocation in multi-issue processors. *Computers and Mathematics with Applications*, 34(9):1–14, 1997.

6. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
7. M. A. Ertl and A. Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming (PLILP)*, 1991.
8. J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, 1983.
9. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.
10. C. W. Kessler. Scheduling expression DAGs for minimal register need. *Computer Languages*, 24(1):33–53, 1998.
11. C. Le Pape and P. Baptiste. Constraint-based scheduling: A theoretical comparison of resource constraint propagation rules. In *Proceedings of the ECAI Workshop on Non-Binary Constraints*, Brighton, UK, August 1998.
12. M. Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning*, pages 19–28, Key West, Florida, May 1996.
13. C. Lee, M. Potkonjak, and W. Manginoe-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications. In *Proceedings of International Symposium on Microarchitecture*, pages 330–335, December 1997.
14. R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *IEEE Trans. VLSI Systems*, 5(1):112–122, 1997.
15. K. Marriott and P. J. Stuckey. *Programming with Constraints*. The MIT Press, 1998.
16. K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and alldifferent constraint. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 306–319, Singapore, September 2000.
17. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
18. K. Palem and B. Simons. Scheduling time-critical instructions on RISC machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, 1993.
19. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 353–368, Singapore, September 2000.
20. J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 359–366, Madison, WI, July 1998.
21. K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, Vancouver, BC, June 2000.
22. H. Wu, J. Jaffar, and R. Yap. Instruction scheduling with timing constraints on a single RISC processor with 0/1 latencies. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 457–469, Singapore, September 2000.