Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors using Constraint Programming*

Abid M. Malik[1], Jim McInnes[2], and Peter van Beek[1]
[1] University of Waterloo, Waterloo, Canada
[2] IBM Canada Toronto Lab

Abstract

Instruction scheduling is one of the most important steps for improving the performance of object code produced by a compiler. A fundamental problem that arises in instruction scheduling is to find a minimum length 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. Solving the problem exactly is NP-complete, and heuristic approaches are currently used in most compilers. In contrast, we present a scheduler that finds provably *optimal* schedules for basic blocks using techniques from constraint programming. In developing our optimal scheduler, the keys to scaling up to large, real problems were improvements to the constraint model and to the constraint propagation phases. We experimentally evaluated our optimal scheduler on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler was very robust and scaled to the largest basic blocks—all but a handful of the hundreds of thousands of basic blocks in our benchmark suite could not be solved optimally within a reasonable time limit, and the scheduler was able to routinely solve the largest basic blocks, including basic blocks with up to 2600 instructions. This compares favorably to the best previous exact approaches.

1 Introduction

Modern architectures are pipelined and can issue multiple instructions per time cycle. On such processors, the order in which the instructions are scheduled can significantly impact performance. A fundamental problem that arises in instruction scheduling is to find a minimum length 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. Basic

*Technical Report CS-2005-19, School of Computer Science, University of Waterloo, 2005.

block scheduling is important in its own right and also as a building block for scheduling larger groups of instructions such as superblocks [6, 29].

Basic block instruction scheduling for realistic multiple-issue processors is NP-complete [11, 15], and most compilers use a heuristic approach—a fast method that sometimes gives sub-optimal solutions—rather than an exact approach to basic-block scheduling (e.g., see [12, 24]). Although heuristic approaches have the advantage that they are fast, a basic block scheduler which finds provably optimal schedules may be useful where longer compile times are tolerable, such as when compiling for software libraries, digital signal processing, or embedded applications [12, 21]. In addition to offering improved schedules when incorporated into the compiler, an optimal scheduler can also be used to improve existing heuristic approaches. First, an optimal scheduler can be used as a gold standard against which to evaluate existing heuristics, and to determine when and why these heuristics fail to find improvements. Second, an optimal scheduler can be used to generate machine learning data from which improved heuristics can be automatically learned [28].

Previous work on optimal basic block schedulers has taken several approaches, including: branch-and-bound enumeration [5, 13, 14, 21, 29], dynamic programming [19], integer linear programming [1, 4, 18, 20, 31], and constraint programming [10, 30]. With the exception of [14, 30, 31] (to which we do detailed comparisons 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 50 instructions. Further, their experimental results suggest that none of them would scale up beyond problems of this size. A major challenge when developing an exact approach to an NP-complete problem is to develop a solver that scales and is robust in that it rarely fails to find a solution in a timely manner on a wide selection of real problems. Wilken, Liu, and Heffernan [31] were the first to develop a robust optimal scheduler that scaled up to large basic blocks. However, only experimental results for a single-issue processor were presented and thus it is unclear how well their approach works on multiple-issue processors (the general form of the problem, as addressed in this paper). Van Beek and Wilken [30] improve on these results for single-issue processors, using a constraint programming approach. This work on single-issue processors forms the starting point of the present work. Heffernan and Wilken [14] were the first to present experimental results on solving large basic blocks targeted towards a multiple-issue processor.

In this paper, we present a constraint programming approach to instruction scheduling for multiple-issue processors that is robust and optimal. In a constraint programming approach, one models a problem by stating constraints on acceptable solutions, where a constraint is simply a relation among several unknowns or variables, each taking a value in a given domain. The problem is then usually solved by interleaving a backtracking search with a series of constraint propagation phases. In the constraint propagation phase, the constraints are used to prune the domains of the variables by ensuring that the values in their domains are locally consistent with the constraints. In developing our optimal scheduler, the keys to scaling up to large, real problems were improvements to the constraint model and to the constraint propagation phases.

We experimentally evaluated our optimal scheduler on the SPEC 2000 integer and floating

point benchmarks, using four different architectural models. On this benchmark suite, the optimal scheduler scaled to the largest basic blocks and was very robust. Depending on the architectural model, at most 2–25 basic blocks out of the hundreds of thousands of basic blocks used in our experiments could not be solved within a 10-minute time bound. This represents approximately a 50-fold improvement over previous work. As well, the scheduler was able to routinely solve the largest basic blocks that we found in practice, including basic blocks with up to 2600 instructions.

2 Background

In this section, 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, [16, 23, 24]).

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 multiple-issue pipelined processors. On such processors, there are multiple functional units, and multiple instructions can be issued (begin execution) each clock cycle. Associated with each instruction is a delay or *latency* between when the instruction is issued and when the result is available for other instructions that use the result. In this paper, we assume that all functional units are fully pipelined, that instructions and functional units are typed, and that instructions of a given type only execute on one type of functional unit. Examples of types of instructions are load/store, integer, floating point, and branch instructions.

We use the standard labeled directed acyclic graph (DAG) representation of a basic-block (see [24]). Each node corresponds to an instruction and there is an edge from i to j labeled with a non-negative integer $l(i, j)$ if j must not be issued until i has executed for $l(i, j)$ cycles. In particular, if $l(i, j) = 0$, j can be issued in the same cycle as i; 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, denoted $cp(i, j)$, is the maximum sum of the latencies along any path from i to j, if there exists a path from i to j; $-\infty$ otherwise. A node i is a *predecessor* of a node j if there is a directed path from i to j; if the path consists of a single edge, i is also called an *immediate predecessor* of j. A node j is a *successor* of a node i if there is a directed path from i to j; if the path consists of a single edge, j is also called an *immediate successor* of i. A *sink* node is a node with no successors. For convenience, we assume that a fictitious sink node, hereafter called *the* sink node, is added to each DAG and that an edge is added from each node i in the DAG to the sink node, where the label on the edge is the latency of instruction i.

Given a labeled dependency DAG for a basic block, a *schedule* for a multiple-issue pro-

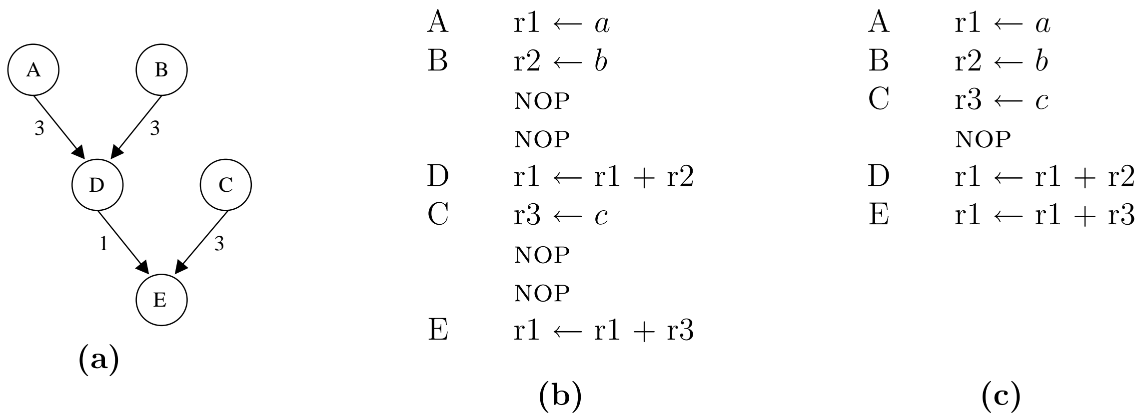

Figure 1: (a) Dependency DAG associated with the instructions to evaluate $(a + b) + c$ on a processor where loads from memory have a latency of 3 cycles and integer operations have a latency of 1 cycle; (b) non-optimal schedule for a single-issue processor; (c) optimal schedule.

cessor specifies an issue or start time for each instruction or node such that the latency constraints are satisfied and the resource constraints are satisfied. The latter are satisfied if, at every time cycle, the number of functional units that can execute a set of instruction types is greater than or equal to the number of instructions of those types issued at that cycle. The *length* of a schedule is the number of the cycle in which the sink node is issued. The *basic block instruction scheduling problem* is to construct a schedule with minimum length.

Basic block instruction scheduling under the assumption that all functional units are fully pipelined is the special case of resource-constrained project scheduling [9, 25] where all of the activities have unit execution times and we seek a schedule which minimizes the makespan.

Example 1 Figure 1 shows a simple dependency DAG and two possible schedules for the DAG, assuming a single-issue processor that can execute all types of instructions. The schedule (b) requires four NOP instructions (null operations) because the values loaded are used by the following instructions. The better schedule (c), the optimal or minimum length schedule, requires only 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.

Definition 1 (Constraint Model) A *constraint model* 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 $vars(C_i)$, that specifies the allowed combinations of values for the variables in $vars(C_i)$. A *solution* to a constraint model is an assignment of a value to each variable that satisfies all of the constraints.

Constraint models are often solved using a backtracking algorithm. At every stage of the backtracking search, there is some current partial solution that 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 2 (Bounds Consistency Constraint Propagation) 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, where for each variable y its value is taken from $[\min(dom(y)), \max(dom(y))]$. 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 constraint model can be made bounds consistent by repeatedly removing unsupported values from the domains of its variables.

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

In this section, we present our constraint model of the basic block instruction scheduling problem. In the constraint programming methodology a problem is modeled 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.

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). The domain of each variable $dom(i)$ is a subset of $\{1, \ldots, m\}$ which are the available time cycles. Assigning a value

Table 1: Notation used in specifying the constraints.

Notation	Meaning
$lower(i)$	lower bound of domain of variable i
$upper(i)$	upper bound of domain of variable i
$type(i)$	type of node/instruction i
k_t	number of functional units of type t
$l(i,j)$	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
$onpath(i,j,t)$	set of all nodes of type t that are on some path from node i to node j. Note that $i \in onpath(i,j,t)$ if $type(i) = t$ and $j \in onpath(i,j,t)$ if $type(j) = t$. These are all of the instructions of type t that must be issued with or after node i is issued and must all be issued with or before node j is issued.
$pred(i)$	set of all immediate predecessors of node i
$succ(i)$	set of all immediate successors of node i
$pred(i,t)$	set of all immediate predecessors of node i that are of type t
$succ(i,t)$	set of all immediate successors of node i that are of type t
$I([a,b],t)$	set of all variables of type t whose domains intersect the interval $[a,b]$. These are all of the instructions of type t that may need these time cycles to execute on functional units of type t.

$d \in dom(i)$ to a variable i has the intended meaning that instruction i will be issued at time cycle d. The domain $dom(i) = \{a, \ldots, b\}$ of a variable i is represented by the endpoints of the interval $[a,b]$. We use the notation $lower(i)$ and $upper(i)$ to refer to these endpoints.

We now specify the six types of constraints in the model: latency, resource, distance, predecessor and successor, safe pruning, and dominance constraints. Some of 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 resource constraints. However, the distance, predecessor and successor, safe pruning constraints, and dominance were found to be essential in improving the efficiency of the search for a schedule.

3.1 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,

$$lower(j) \geq lower(i) + l(i, j)$$

and its symmetric version,

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

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.

3.2 Resource constraints

For each type t of instruction/functional unit a resource constraint is needed to ensure that the number of instructions of type t issued at each time cycle does not exceed the number of functional units of type t. Such resource constraints are a special case of a well-studied constraint called the global cardinality constraint [27]. A global cardinality constraint over a set of variables and values states that the number of variables instantiating to a value must be between a given upper and lower bound, where the bounds can be different for each value. Here, for each type t a global cardinality constraint over all variables of type t is added to the constraint model, where all of the lower bounds are set equal to zero and all of the upper bounds are set equal to the number of functional units of type t. Note that when all of the upper bounds are set equal to one—in our case, when there is a single functional unit for some type t—the global cardinality constraint is equivalent to the well-known all-different constraint, which enforces that its arguments are pair-wise different.

Fast algorithms for enforcing bounds consistency on a global cardinality constraint have been proposed. In our implementation, we used the efficient algorithm presented in [22, 26]. The algorithm runs in time $O(t + n)$, where t is the time to sort the bounds of the domains of the variables and n is the number of variables. We note that for scheduling basic blocks, it has been shown that bounds consistency is dramatically better than other, more expensive, forms of consistency [22, 26].

3.3 Distance constraints

For each pair of nodes i and j, 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 factors in effectively reducing the size of the search space.

Table 2: Additional notation used in specifying the distance constraints.

$r_1(i, j, t)$	The minimum number of cycles that must elapse before the first instruction in $onpath(i, j, t)$ can be issued; i.e., $\min\{cp(i, k) \mid k \in onpath(i, j, t)\}$, the minimum critical-path distance from node i to any node in $onpath(i, j, t)$.
$r_2(i, j, t)$	The minimum number of cycles to issue all of the instructions in $onpath(i, j, t)$; i.e., $\lceil \lvert onpath(i, j, t) \rvert / k_t \rceil$, the size of the set of instructions divided by the number of functional units that can execute instructions of type t, rounded up to the next highest integer value.
$r_3(i, j, t)$	The minimum number of cycles that must elapse between when the last instruction in $onpath(i, j, t)$ is issued and node j can be issued; i.e., $\min\{cp(k, j) \mid k \in onpath(i, j, t)\}$, the minimum critical-path distance from any node in $onpath(i, j, t)$ to node j.

In what follows, we are interested in subgraphs called regions [31], which are induced from a given dependency DAG. Basic blocks typically contain many such regions embedded within them, with larger blocks containing many thousands.

Definition 3 (Region [31]) A pair of nodes i, j in a DAG define a *region* 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.

Given a region defined by nodes i and j, we wish to add a distance constraint $j \geq i + d(i, j)$, for some integer value $d(i, j)$. Following [31], if the region is small enough, we solve the region exactly (in isolation) and determine the optimal value for $d(i, j)$. To solve a region in isolation, we use the same constraint solver as for an entire basic block, but the constraint model is restricted to just the latency and resource constraints, plus any distance constraints that have been found so far. The regions in the DAG are examined in an "inside-out" manner so that distance constraints for inner regions can be used when solving larger outer regions.

For larger regions, we estimate the value, ensuring that our estimate is always less than or equal to the optimal value. We found that a threshold of 25 nodes worked well in practice; for regions larger than this the distance was estimated. Consider the notation shown in Table 2. For larger regions, initially we estimate $d(i, j)$ using,

$$d(i, j) = \max_t \{r_1(i, j, t) + r_2(i, j, t) + r_3(i, j, t) - 1\},$$

where we are finding the maximum over all instruction types t. Note that the nodes that are on a path from node i to node j can be determined quickly given the critical-path distances between all pairs of nodes, since a node k is on a path from i to j iff $cp(i, k) \geq 0$ and $cp(k, j) \geq 0$. The estimate of the distance can sometimes be improved by "removing" a

small number of nodes (between one and three nodes) from $onpath(i, j, t)$. This was done whenever removing these nodes led to an increase in the value of $d(i, j)$; i.e., the decrease in $r_2(i, j, t)$ was more than offset by the increase in $r_1(i, j, t) + r_3(i, j, t)$. The estimate is a generalization and improvement over the distance constraints presented in [30], to handle multiple-issue, multiple types of instructions, and zero latency edges.

Example 3 Consider the dependency DAG shown in Figure 2 where the clear nodes are of one instruction type and the shaded (yellow) nodes are of a different instruction type. Assume there is a single functional unit for each type of instruction. For the region defined by A and F, the initial estimate of the distance is $d(\mathrm{A}, \mathrm{F}) = 4$. Similarly, for the region defined by A and G, the initial estimate of the distance is $d(\mathrm{A}, \mathrm{G}) = 5$. The estimate of the distance $d(\mathrm{A}, \mathrm{G})$ can be improved to $d(\mathrm{A}, \mathrm{G}) = 6$ by "removing" node G from $onpath(\mathrm{A}, \mathrm{G}, \mathrm{shaded})$. The distance constraints $\mathrm{F} \geq \mathrm{A} + 4$ and $\mathrm{G} \geq \mathrm{A} + 6$ would be added to the constraint model, as both $d(\mathrm{A}, \mathrm{F})$ and $d(\mathrm{A}, \mathrm{G})$ are improvements over the critical-path distances between those nodes.

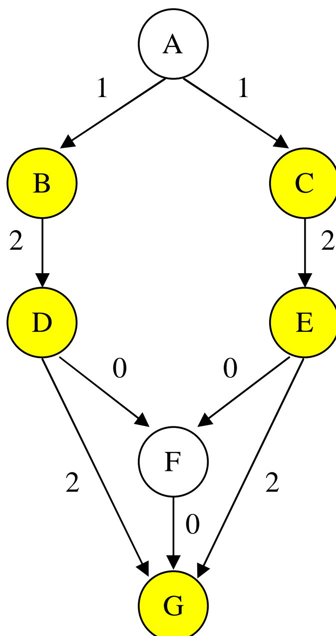

Figure 2: Example of adding distance constraints between nodes that define regions. The constraints $\mathrm{F} \geq \mathrm{A} + 4$ and $\mathrm{G} \geq \mathrm{A} + 6$ would be added to the constraint model.

3.4 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,

$$lower(i) \geq \min\{lower(k) \mid k \in P\}$$

$$
\begin{aligned}
&+ \lceil |P|/k_t \rceil - 1 \\
&+ \min\{l(k,i) \mid k \in P\}
\end{aligned}
$$

for every type t and every subset P of $\mathit{pred}(i,t)$ where $|P| > k_t$,

where the operator $\lceil x \rceil$ returns the smallest integral value not less than x. It can be seen that a predecessor constraint can be propagated in $O(|\mathit{pred}(i)|^2)$ 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,

$$
\begin{aligned}
\mathit{upper}(i) \quad \leq \quad & \max\{\mathit{upper}(k) \mid k \in P\} \\
& - \lceil |P|/k_t \rceil + 1 \\
& - \min\{l(i,k) \mid k \in P\},
\end{aligned}
$$

for every type t and every subset P of $\mathit{succ}(i,t)$ where $|P| > k_t$.

The predecessor and successor constraints are propagated in a preprocessing stage and also during search. They can be viewed as an adaptation of edge-finding rules (see [2]) and are an easy generalization of the similarly named constraints presented in [30] to handle multiple-issue and multiple types of instructions.

Example 4 Consider the partial DAG shown in Figure 3, where the domains of the variables are as shown. Assume there is a single functional unit for each type of instruction. Propagating the predecessor constraint associated with node E improves the lower bound of the variable. The earliest that the set $P = \{\text{C, D}\}$ of immediate predecessors of node E can be scheduled is cycle 8, and, therefore, cycle 9 is the earliest that the last of its predecessors could be scheduled. Therefore, the earliest that E can be scheduled is cycle 11.

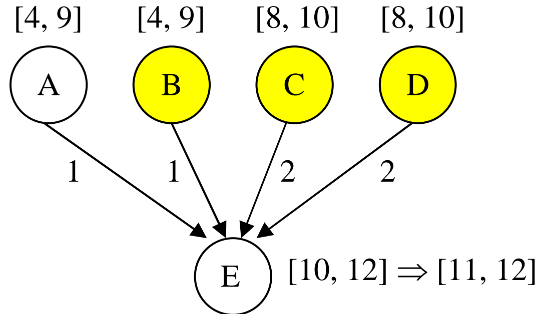

Figure 3: Example of improving the lower bound of a variable using a predecessor constraint.

3.5 Safe pruning constraint

Given a constraint model, we say that it is *safe* to prune a value from the domain of a variable whenever it is the case that, if there was a solution to the constraint model before pruning the value, there is still a solution to the model after pruning. For each instruction type t, a safe pruning constraint over all variables of type t is added to the constraint model. The safe pruning constraint is based on the following theorem.

Theorem 1 Suppose that all of the latency and resource constraints have been propagated. If there exists an interval $[a, b]$ such that,

(i) for all $i \in I([a, b], t)$, $lower(i) = a$,

(ii) for all $i \in I([a, b], t)$, for all $k \in pred(i)$, $upper(k) + l(k, i) \leq lower(i)$,

(iii) $|\ I([a, b], t)\ | \ \leq \ (b - a + 1) \times k_t$,

then it is safe to prune the upper bounds of the variables $i \in I([a, b], t)$ as follows,

$$upper(i) = \min(upper(i), b).$$

Proof. Suppose there was a solution to the constraint model before pruning. Call this the original solution. There are two cases.

1. Suppose that in the original solution each variable in $I([a, b], t)$ is assigned a value from its domain that is less than or equal to b. Clearly this is still a solution after pruning.

2. Suppose that in the original solution there exist variables in $I([a, b], t)$ that have been assigned values from their domains that are greater than b. We will show that each of these variables can be given a consistent value from $[a, b]$.

   a. *Latency constraints*: We will show that any value in $[a, b]$ satisfies the latency constraints. Let i be any variable that has been reassigned a value. Let k be an immediate predecessor of i and consider the latency constraint $k + l(k, i) \leq i$. Lowering the value of i cannot violate the constraint since $upper(k) + l(k, i) \leq lower(i)$ (by condition (ii)) and we assumed that the latency constraints have been propagated. Thus, any value in the domain of i will satisfy this constraint. Let k be an immediate successor of i and consider the latency constraint $i + l(i, k) \leq k$. Lowering the value of i cannot violate this constraint.

   b. *Resource constraints*: We will show that it is possible to reassign values to these variables from $[a, b]$ and satisfy the relevant resource constraint. Condition (i) implies that before pruning there is no variable i of type t such that $lower(i) < a$ and $a \leq upper(i)$; i.e., before pruning there is no variable whose domain intersects both $[c, a - 1]$ and $[a, d]$ where $c < a \leq d \leq b$. We also know that after pruning there is no variable whose domain intersects both $[c, b]$ and $[b+1, d]$ where $a \leq c \leq b < d$. This means that we can look at the resource constraint over the variables in

$I([a,b])$ in isolation; whatever values are assigned to the variables in this set cannot impact the values that variables outside of this set can be assigned. Condition (iii) ensures there are enough values so that all of the variables in $I([a,b],t)$ can be assigned a value such that the resource constraint is satisfied.

□

Corollary 1 Suppose that all of the latency and resource constraints have been propagated. If there exists an interval $[a, b]$ such that,

(i) for all $i \in I([a,b],t)$, $upper(i) = b$,

(ii) for all $i \in I([a,b],t)$, for all $k \in succ(i)$, $upper(i) + l(i,k) \leq lower(k)$,

(iii) $|\ I([a,b],t)\ | \ \leq \ (b-a+1) \times k_t$,

then it is safe to prune the lower bounds of the variables $i \in I([a,b],t)$ as follows,

$$lower(i) = \max(lower(i), a).$$

Example 5 Consider the partial DAG shown in Figure 4, where the domains of the variables are as shown. Assume there is a single functional unit for each type of instruction. The safe pruning constraint can be applied iteratively as follows. First, the interval [2,2], where $I([2,2], \text{clear}) = \{\text{B}\}$, satisfies the theorem. Hence, node B can have its domain pruned to [2,2]. Second, the interval [3,3], where $I([3,4], \text{clear}) = \{\text{C}\}$, now satisfies the theorem. Hence, node C can have its domain pruned to [3,3]. Third, the interval [3,4], where $I([3,4], \text{shaded})$ $= \{\text{D, E}\}$, also now satisfies the theorem. Hence, nodes D and E can have their domains pruned to [3,4].

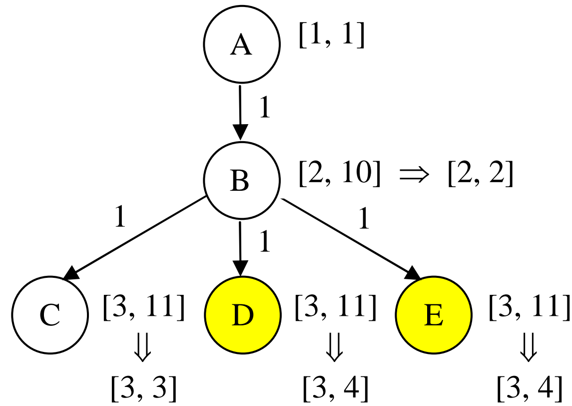

Figure 4: Example of improving the bounds of variables using a safe pruning constraint.

3.6 Dominance constraints

Heffernan and Wilken [14] present a set of graph transformations for dependency DAGs for basic blocks and show that optimally scheduling the transformed DAGs using branch-and-bound enumeration is faster and more robust. The DAG transformations reduce the search space while preserving optimality. We found that these transformations also worked well in our constraint programming approach. In our context, the transformations add simple constraints to the model of the form $i \geq j$, which we call dominance constraints.

In what follows, we are interested in pairs of disjoint, isomorphic subgraphs A and B induced from a given dependency DAG. Subgraphs A and B are isomorphic if there is a mapping from the node set of A to the node set of B such that A and B are identical (identical instruction types, edges, and latencies on the edges).

Using terminology similar to that for the safe pruning constraint, we say that it is *safe* to add a constraint to a constraint model whenever it is the case that, if there was a solution to the constraint model before adding the constraint, there is still a solution after adding the constraint. Adding dominance constraints, when it is safe to do so, is based on the following theorem.

Theorem 2 (Heffernan and Wilken [14]) Let A and B be isomorphic subgraphs with node sets $V(A) = \{a_1, \ldots, a_r\}$ and $V(B) = \{b_1, \ldots, b_r\}$. If,

(i) a_i is neither a predecessor or a successor of b_i, $1 \leq i \leq r$,

(ii) for all $k \in \mathit{pred}(a_i)$ such that $k \notin V(A)$, $l(k, a_i) \leq \mathit{cp}(k, b_i)$, $1 \leq i \leq r$,

(iii) for all $k \in \mathit{succ}(b_i)$ such that $k \notin V(B)$, $l(b_i, k) \leq \mathit{cp}(a_i, k)$, $1 \leq i \leq r$,

(iv) for any edge $(b_i,\ a_j)$, $l(b_i, a_j) \leq \mathit{cp}(a_i, b_j)$,

then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$ is safe.

Example 6 Consider the DAG shown in Figure 5a. Dominance constraints can be added iteratively as follows. First, the subgraphs with nodes $V(A)$ = {B, D} and $V(B)$ = {C, E} are isomorphic and satisfy the conditions of the theorem. Hence, the constraints B $\leq$ C and D $\leq$ E can be added to the model. Adding these constraints updates the critical path distances. In particular, $\mathit{cp}(\mathrm{D}, \mathrm{E})$ was $-\infty$ and is now 0. Second, the subgraphs with nodes $V(A)$ = {F} and $V(B)$ = {E} are isomorphic and now satisfy the conditions of the theorem. Hence, the constraint F $\leq$ E can be added to the model.

Heffernan and Wilken [14] find isomorphic subgraphs that satisfy the theorem using backtracking search with a time cutoff. The search starts with isomorphic subgraphs that consist of single nodes (i.e., they have the same instruction type) that satisfy condition (i) of the theorem and either condition (ii) or condition (iii). These nodes are called seed nodes. The backtracking search expands these subgraphs to adjacent nodes, maintaining isomorphism, until either (a) all of the conditions of the theorem are satisfied (in which case,

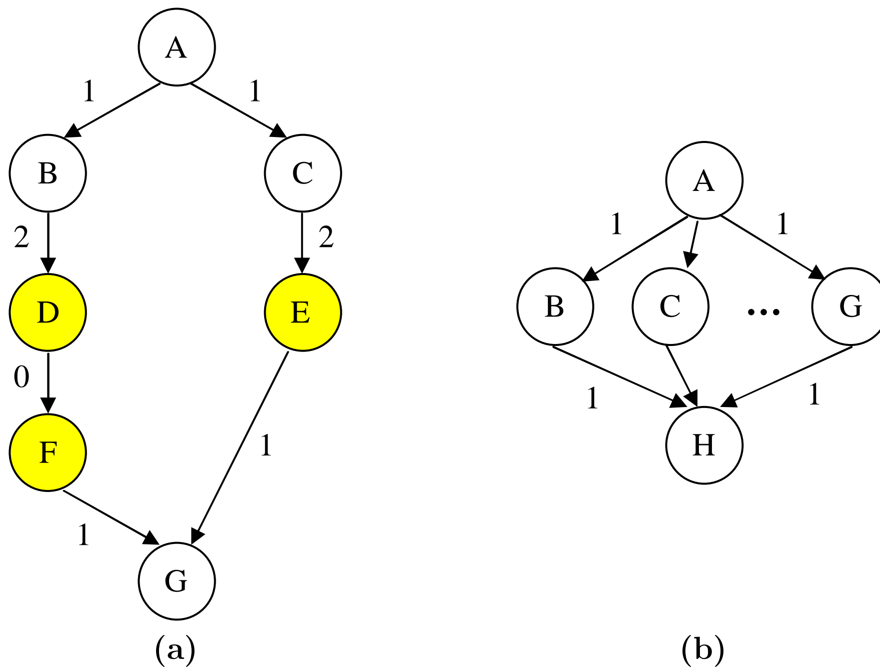

Figure 5: Examples of adding dominance constraints: (a) (adapted from [14]) the constraints B $\leq$ C, D $\leq$ E, and F $\leq$ E would be added to the constraint model; (b) the constraints B $\leq$ C, C $\leq$ D, . . . , F $\leq$ G would be added to the constraint model.

dominance constraints can be added), or (b) the subgraphs cannot be expanded any further or the time cutoff is reached (in which case, this pair of seed nodes leads to failure and we try another pair of seed nodes).

In our work, we find isomorphic subgraphs by focusing on regions (see Definition 3). Given a region defined by nodes i and j, we conceptually remove the source node i and the sink node j of the region and perform a depth-first search to find the separate components or subgraphs of the region. We then check whether pairs of components are isomorphic and satisfy the conditions of the theorem (or can be made to do so by dropping a few nodes). We focus on separate components of regions as during the backtracking search for a solution, often both orderings of these components must be tried to verify that there is no solution. Thus, the dominance constraints, by establishing an ordering on the variables between these components, can greatly reduce the search space.

Testing isomorphism is NP-complete in general. Here, a fast heuristic test is used to determine whether two components are isomorphic. The nodes in each component are independently sorted based on features of the nodes, and the order of the nodes constitutes a potential isomorphism mapping, which is then verified. Observe that whenever the heuristic (sort) test returns true, the pair of subgraphs is isomorphic, and that sometimes the heuristic returns false even though there exists a true mapping. However, experimental evidence suggests that the heuristic works well. Consider the following sets S_1 and S_2, where $S_1 \subseteq S_2$.

Construct the first set S_1 as follows. For all pairs of components, add only those pairs to S_1 that pass the heuristic test. This gives some of the pairs of components that are isomorphic (although it may miss some); i.e., S_1 is a subset of the set of all isomorphic pairs of components. Construct the second set S_2 as follows. For all pairs of components, add only those pairs to S_2 that have the same numbers of instructions of each instruction type. This gives the pairs of components that are *potentially* isomorphic (although some may not be); i.e., S_2 is a superset of the set of all isomorphic pairs of components. We found that the difference $S_2 - S_1$ was most often empty and always small, thus providing evidence that the heuristic test catches almost all isomorphic pairs of components.

A special case of the theorem was found to occur often in practice. Consider the DAG shown in Figure 5b where the region defined by A and H contains many nodes all of the same type and all at the same latencies. All of these nodes are symmetric and the dominance constraints that would be added are equivalent to so-called symmetry-breaking constraints [7]. We recognize this special case as follows. For each instruction type t, we sort the variables by their lower bounds, and then step through all instructions with the same lower bound and check if the pairs of nodes satisfy the theorem. If so, dominance constraints are added.

Overall, we found that our techniques often discovered many pairs of components within a basic block that satisfied the theorem, sometimes with several hundred nodes each. We also found that the dominance constraints that were added greatly improved the efficiency of the search for a schedule, thus providing additional evidence for the effectiveness of the graph transformations proposed by Heffernan and Wilken [14].

3.7 Solving an instance

Solving an instance of an instruction scheduling problem is divided into several phases.

In phase one, we construct the constraint model and use 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 constraint model is passed to the backtracking algorithm. The backtracking search interleaves constraint propagation with branching on variables. During constraint propagation, bounds consistency is enforced on the constraints until no further changes result. A dynamic variable ordering is used that 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 $lower(x)$, then on x assigned to $lower(x) + 1$, and so on, until either a solution is found or the domain of x is exhausted. 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 constraint model is passed to the backtracking algorithm. This is repeated, each time incrementing m until a solution is found, an upper bound on the length of a schedule is reached, or a time limit is exceeded. An upper bound on the length of a schedule is established by running a list-scheduling algorithm using a critical-path heuristic (see Section 4). If a solution is found or the upper bound on the

length of a schedule is reached, a provably optimal solution has been found. If, instead, the time limit is exceeded, we proceed to phase two of the solution process.

In phase two, the level of constraint propagation during backtracking search is increased to a variation of singleton consistency [8]. In singleton consistency, a variable is temporarily instantiated to a single value and the constraint model is tested for consistency. If the consistency test fails, the value can be removed from the domain of the variable. In our work, we iteratively instantiated and tested the consistency of the lower and upper bounds of the domains of the variables. The consistency test consisted of enforcing bounds consistency on the constraints. We found that singleton consistency sometimes dramatically reduced the domains of the variables during search. As well, when testing the consistency of the bounds, we record the number of changes that are made during the bounds consistency propagation. This information is used in phase two to select the next variable to branch on. The goal is to branch on a variable that causes the most reductions in the domains of the other variables. As for phase one, if a solution is found or the upper bound on the length of a schedule is reached, a provably optimal solution has been found.

In phase three, the level of constraint propagation during backtracking search is increased once again to perform singleton consistency to a depth of two. Each variable is temporarily instantiated to a single value and we test whether the constraint model is singleton consistent. This level of propagation is expensive and is viable only for smaller but difficult basic blocks.

In our experiments, we found that the following scheme worked best for stepping through the phases. First, if the basic block contains 300 or fewer instructions, phase one is allocated 5 seconds, phase two is allocated 15 seconds, and the remaining time is allocated to phase three. Second, if the basic blocks contains more than 300 instructions, phase one is allocated 5 seconds and the remaining time is allocated to phase two.

4 Experimental Evaluation

In this section, we describe the experimental evaluation of our optimal basic block scheduler.

The constraint programming model was implemented and evaluated on all of the basic blocks from the SPEC 2000 integer and floating point benchmarks [http://www.spec.org]. The benchmarks were compiled using IBM's Tobey compiler [3] targeted towards the IBM® PowerPC® processor [17], and the basic blocks were captured as they were passed to Tobey's instruction scheduler. The basic blocks contain four types of instructions: branch, load/store, integer, and floating point. The range of the latencies is: all 1 for branch instructions, 1–12 for load/store instructions (the largest value is for a store-multiple instruction, which stores to memory the values in a sequence of registers), 1–37 for integer instructions (the largest value is for division), and 1–38 for floating point instructions (the largest value is for square root). The Tobey compiler performs instruction scheduling before global register allocation and once again afterwards, and our test suite contains both versions of the basic blocks. The compilations were done using Tobey's highest level of optimization, which includes aggressive optimization techniques such as software pipelining and loop unrolling.

The following table shows the four architectural models we used in our evaluation. We

assumed that all functional units were fully pipelined and that the issue width of the processor was equal to the number of functional units.

- 1-issue processor executes all types of instructions.
- 2-issue processor with one floating point functional unit and one functional unit that can execute integer, load/store, and branch instructions.
- 4-issue processor with one functional unit for each type of instruction.
- 6-issue processor with the following functional units: two integer, one floating point, two load/store, and one branch.

The optimal constraint programming scheduler was compared experimentally with list scheduling, the most popular heuristic method for scheduling basic blocks in compilers [12, 24]. List scheduling is a greedy algorithm which uses a heuristic for which instruction to schedule next. Following Muchnick [24], our list scheduling heuristic used critical-path distance (max delay) as the primary feature, earliest start time as a tie-breaker if the critical-path distances were equal, and order within the instruction stream as a tie-breaker if both the critical-path and earliest start times were equal. We refer to this heuristic as the critical-path heuristic. Although it is a popular heuristic, the primary reason for adopting this heuristic is that critical-path heuristics were also used in previous work [14, 30, 31], thus allowing a fairly direct comparison of previous experimental results with our experimental results.

Tables 3 & 4 show the number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found a shorter schedule than the heuristic scheduler and also the number of basic blocks where the optimal scheduler failed to complete within the given time limit of 10 minutes[1]. The results are broken down into whether the instruction scheduling is being done before or after global register allocation. It can be seen that the optimal scheduler is robust in that it almost always completed within the given time limit. Although not shown in the tables, this remains true even if the time limit is decreased from 10 minutes to 100 seconds. (At most 5 additional failures resulted for each issue width when scheduling before register allocation and at most 10 when scheduling after.) It is interesting to note that the basic blocks that arise from before register allocation are harder for the optimal scheduler (as measured by the number of times the scheduler failed to complete within a given time bound) than those that arise from after register allocation.

Wilken, Liu, and Heffernan [31] and van Beek and Wilken [30] present experimental results for a 1-issue processor. Note that, although both of these solvers could solve all of the basic blocks in the SPEC95 floating point benchmarks in seconds, when the solver in [30] was applied to the current test suite of basic blocks, hundreds of problems could not be solved. We speculate that the current test suite contains more difficult problems for the following four reasons. First, the current test suite contains longer and more varied latencies (in [31], the latencies were uniformly 1 for integer instructions, 2 for floating point instructions, and 3 for memory instructions). Second, the current test suite contains shorter latencies (our

[1] All of the experiments were run on a 2.40 GHz Intel® Pentium® 4 processor with 1 GB of main memory.

DAGs contain many latency 0 edges, which are used to capture anti-dependencies and output dependencies between two instructions). Third, the current test suite contains many larger basic blocks (previous work used the GCC compiler and the largest DAG was approximately 1000 instructions). Fourth, the current test suite contains blocks from both before and after register allocation (previous work only used blocks from after register allocation).

Heffernan and Wilken [14] were the first to present experimental results on solving large basic blocks targeted towards a multiple-issue processor. Their test suite contains the basic blocks from the SPEC 2000 floating point benchmarks (with the Fortran90 benchmarks omitted) and are from after register allocation. They report the number of basic blocks where their optimal scheduler failed to complete within a time limit of 100 seconds. In their worst case, a 2-issue processor model, their optimal solver failed on over 200 basic blocks. If we restrict our experimental results to the same benchmarks and the same time limit, our optimal solver failed on only 4 basic blocks, a 50-fold improvement.

To systematically study the scaling behavior of the optimal scheduler, we report the results broken down by increasing size ranges of the basic blocks. For reference, the number of basic blocks in each size range is also given (see Tables 5 & 6). It can be seen that the optimal scheduler scales well, finding improved solutions for large basic blocks. Not surprisingly, as the basic block size increases, the heuristic method has more opportunities to make a mistake and the fraction of basic blocks improved by the optimal scheduler increases. For the largest basic blocks, up to 40.9% of the schedules are improved by the optimal scheduler (see the 4-issue architecture after register allocation). It is interesting to note that the heuristic method does relatively much better on the basic blocks that arise from before register allocation, and the optimal scheduler does relatively much better on the basic blocks that arise from after register allocation. The blocks after register allocation are more constrained, which make them harder for the heuristic method and easier for the optimal method.

Tables 7 & 8 summarize the percentage improvements in schedule length of the optimal schedule over the schedule found by a list scheduling algorithm using the critical-path heuristic. Somewhat surprising is that on all size ranges the optimal scheduler can find substantial improvements, as measured by the maximum improvement. In other words, critical-path list scheduling, a commonly used heuristic method, sometimes finds schedules that are very sub-optimal.

Table 3: *Basic block instruction scheduling before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule, and (b) the optimal scheduler failed to complete within the given time limit of 10 minutes, for various issue widths.

		1-issue		2-issue		4-issue		6-issue	
	#blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3,128	37		45		34		32	
applu	653	22		25		27		13	
apsi	2,210	25		77		84		27	
art	355	2		3		5		0	
bzip2	972	6		6		5		1	
crafty	4,969	57		57		41		17	
eon	4,509	25		27	1	43	1	21	
equake	486	2		4		6		3	
facerec	1,221	15		52		58		19	
fma3d	10,034	163		170	13	241	8	122	10
galgel	5,369	88		125	2	110	3	32	1
gap	19,729	130		130		104		19	
gcc	42,686	182		181		151		48	
gzip	1,610	26		26		34		1	
lucas	915	40		51		50		33	
mcf	364	10		10		8		0	
mesa	14,903	130		182	2	236	2	98	
mgrid	207	2		9		6		3	
parser	3,561	23		23		9		1	
perlbmk	16,450	89		89		79		10	
sixtrack	10,950	260		418	2	370	2	74	
swim	345	2		7		6		1	
twolf	7,468	71		72		56		16	
vortex	11,945	67		67		96		14	
vpr	3,369	28		28	2	23		7	
wupwise	591	14		21		14		0	
Total	168,999	1,516	0	1,905	22	1,896	16	612	11

Table 4: *Basic block instruction scheduling after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule, and (b) the optimal scheduler failed to complete within the given time limit of 10 minutes, for various issue widths.

		1-issue		2-issue		4-issue		6-issue	
	#blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3,459	84		115		88		76	
applu	734	46		76		65		47	
apsi	2,650	77		141		137		102	
art	486	1		2		8		0	
bzip2	1,060	9		9		25		9	
crafty	5,135	78	1	78	1	134	1	40	
eon	4,972	106		133		168		112	
equake	503	6		8		10		6	
facerec	1,436	26		72		89		54	
fma3d	11,280	429	1	485	1	579	1	275	1
galgel	6,120	178		269		255		165	
gap	20,625	213		213		193		86	
gcc	45,565	266		265		415		166	
gzip	1,723	14		14		35		4	
lucas	1,014	47		52		55		35	
mcf	407	11		11		12		1	
mesa	16,478	233		262	1	321	1	150	
mgrid	221	10		22		21		15	
parser	3,935	27		27		41		15	
perlbmk	17,542	187		188		194		66	
sixtrack	12,568	543		850		893		564	2
swim	388	8		19		15		6	
twolf	7,695	92		95		94		23	
vortex	12,808	92		90		173		102	
vpr	3,654	40		40		61		16	
wupwise	654	44		57		54		12	
Total	183,112	2,867	2	3,593	3	4,135	3	2,147	3

Table 5: *Basic block instruction scheduling before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for ranges of basic block sizes and various issue widths.

		1-issue		2-issue		4-issue		6-issue	
range	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
3–5	90,169	202	0.2	210	0.2	109	0.1	0	0.0
6–10	45,366	376	0.8	440	1.0	313	0.7	17	0.0
11–20	20,477	331	1.6	384	1.9	477	2.3	156	0.8
21–30	5,381	200	3.7	233	4.3	271	5.0	107	2.0
31–50	3,930	176	4.5	261	6.6	315	8.0	134	3.4
51–100	2,390	164	6.9	251	10.5	273	11.4	103	4.3
101–250	1,131	60	5.3	109	9.6	112	9.9	69	6.1
251–2600	155	7	4.5	17	11.0	26	16.8	26	16.8
Total	168,999	1,516	0.9	1,905	1.1	1,896	1.1	612	0.4

Table 6: *Basic block instruction scheduling after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for ranges of basic block sizes and various issue widths.

		1-issue		2-issue		4-issue		6-issue	
range	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
3–5	88,887	136	0.2	140	0.2	73	0.1	0	0.0
6–10	48,700	428	0.9	467	1.0	423	0.9	52	0.1
11–20	26,025	787	3.0	842	3.2	1,146	4.4	378	1.5
21–30	8,530	419	4.9	548	6.4	691	8.1	477	5.6
31–50	5,830	452	7.8	592	10.2	698	12.0	481	8.3
51–100	3,279	372	11.3	539	16.4	642	19.6	435	13.3
101–250	1,658	210	12.7	387	23.3	379	22.9	263	15.9
251–2600	203	63	31.0	78	38.4	83	40.9	61	30.0
Total	183,112	2,867	1.6	3,593	2.0	4,135	2.3	2,147	1.2

Table 7: *Basic block instruction scheduling before register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by critical-path heuristic, for ranges of block sizes and various issue widths. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1-issue		2-issue		4-issue		6-issue	
range	ave.	max.	ave.	max.	ave.	max.	ave.	max.
3–5	14.3	16.7	14.5	25.0	13.9	20.0		0.0
6–10	8.3	16.7	8.8	23.1	10.3	25.0	15.0	25.0
11–20	5.6	17.6	6.2	17.6	7.6	22.2	8.4	14.3
21–30	3.8	11.5	4.4	15.8	5.9	22.2	5.5	10.0
31–50	2.6	13.5	3.3	20.0	4.2	21.4	3.7	17.6
51–100	2.0	8.2	2.7	16.7	2.8	28.0	3.8	22.2
101–250	1.6	8.8	3.6	21.4	3.9	21.6	3.2	24.4
251–2600	0.3	0.7	3.4	14.9	2.9	14.0	1.2	2.9
Overall	6.3	17.6	6.4	25.0	6.6	28.0	5.4	25.0

Table 8: *Basic block instruction scheduling after register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by critical-path heuristic, for ranges of block sizes and various issue widths. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1-issue		2-issue		4-issue		6-issue	
range	ave.	max.	ave.	max.	ave.	max.	ave.	max.
3–5	13.7	16.7	13.9	25.0	12.6	20.0		0.0
6–10	7.9	15.4	8.2	16.7	9.3	25.0	12.0	25.0
11–20	5.0	14.3	5.2	21.4	7.0	27.3	8.0	20.0
21–30	3.3	12.0	4.0	16.0	5.0	17.6	5.7	15.0
31–50	2.5	11.1	3.5	20.0	4.0	24.2	4.2	17.6
51–100	1.8	9.4	2.6	12.5	2.7	14.6	2.7	17.1
101–250	1.2	7.9	1.7	10.8	1.5	13.1	1.7	10.6
251–2600	0.2	0.9	0.7	4.1	0.5	3.4	0.6	4.7
Overall	4.4	16.7	4.6	25.0	5.2	27.3	4.7	25.0

5 Conclusion

We presented a constraint programming approach to basic block instruction scheduling for multiple-issue processors. The problem is considered intractable, yet our approach is optimal and robust on large, real problems. The key to scaling up to large, real problems was in the development of an improved constraint model and the application of more powerful constraint propagation techniques. We performed an extensive experimental evaluation and demonstrated that our approach compares favorably to the best previous exact approaches. The scheduler rarely failed to find a solution within relatively short time bounds, and was able to routinely solve the largest basic blocks that we found in practice, including basic blocks with up to 2600 instructions.

Acknowledgments

This research was supported by an IBM Center for Advanced Studies (CAS) Fellowship, an NSERC Postgraduate Scholarship, and an NSERC CRD Grant. We thank Mike Chase, Claude-Guy Quimper, Tyrel Russell, John Tromp, Kent Wilken, and Huayue Wu for helpful discussions and contributions to the implementation of the constraint programming model.

Trademarks

IBM and PowerPC are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

©Copyright IBM Corp., Abid M. Malik, and Peter van Beek 2005. All rights reserved.

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] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer, 2001.

[3] R. J. Blainey. Instruction scheduling in the TOBEY compiler. *IBM J. Res. Develop.*, 38(5):577–593, 1994.

[4] 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.

[5] H. Chou and C. Chung. An optimal instruction scheduler for superscalar processors. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):303–313, 1995.

[6] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[7] J. M. Crawford, M. L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

[8] R. Debruyne and C. Bessière. Domain filtering consistencies. *J. Artificial Intelligence Research*, 14:205–230, 2001.

[9] U. Dorndorf. *Project Scheduling with Time Windows*. Physica-Verlag, 2002.

[10] M. A. Ertl and A. Krall. Optimal instruction scheduling using constraint logic programming. In *Proceedings of 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 75–86, Passau, Germany, 1991.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] R. Govindarajan. Instruction scheduling. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pages 631–687. CRC Press, 2003.

[13] S. Haga and R. Barua. EPIC instruction scheduling based on optimal approaches. In *1st Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC)*, Austin, Texas, 2001.

[14] M. Heffernan and K. Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8:427–451, 2005.

[15] J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, 1983.

[16] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[17] S. Hoxey, F. Karim, B. Hay, and H. Warren. *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.

[18] D. Kästner and S. Winkel. ILP-based instruction scheduling for IA-64. In *Proceedings of the SIGPLAN 2001 Workshop on Languages Compilers, and Tools for Embedded Systems (LCTES)*, pages 145–154, Snowbird, Utah, 2001.

[19] C. W. Kessler. Scheduling expression DAGs for minimal register need. *Computer Languages*, 24(1):33–53, 1998.

[20] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *IEEE Trans. VLSI Systems*, 5(1):112–122, 1997.

[21] J. Liu and F. Chow. A near-optimal instruction scheduler for a tightly constrained, variable instruction set embedded processor. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 9–18, Grenoble, France, 2002.

[22] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 245–250, Acapulco, Mexico, 2003.

[23] K. Marriott and P. J. Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[24] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[25] K. Neumann, C. Schwindt, and J. Zimmermann. *Project Scheduling with Time Windows and Scarce Resources*. Springer, second edition, 2003.

[26] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 600–614, Kinsale, Ireland, 2003.

[27] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 209–215, Portland, Oregon, 1996.

[28] T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning basic block scheduling heuristics from optimal data. In *Proceedings of the 15th CASCON*, Toronto, 2005.

[29] G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-37)*, pages 283–293, Portland, Oregon, 2004.

[30] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 625–639, Paphos, Cyprus, 2001.

[31] 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, 2000.