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

Abid M. Malik
University of Waterloo, Canada
ammalik@cs.uwaterloo.ca

Jim McInnes
IBM Canada Toronto Lab
jimm@ca.ibm.com

Peter van Beek
University of Waterloo, Canada
vanbeek@cs.uwaterloo.ca

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 key to scaling up to large, real problems was in the development of preprocessing techniques for improving the constraint model. 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—all but a handful of the hundreds of thousands of basic blocks in our benchmark suite were solved optimally within a reasonable time limit—and scaled to 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 clock 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 block instruction scheduling for realistic multiple-issue processors is NP-complete [9], 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 [6, 12]). 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 [6].

Previous work on optimal basic block schedulers has taken several approaches, including: branch-and-bound enumeration [3, 7, 8], dynamic programming [10], integer linear programming [1, 2, 17], and constraint programming [5, 16]. With the exception of [8, 16, 17], 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.

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, a problem is modeled 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 constraint model is then usually solved using a backtracking search. The novelty of our approach is in the extensive computational effort put into a preprocessing stage in order to improve the constraint model and thus reduce the effort needed in backtracking search.

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 two or three 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

Proceedings of the 18th IEEE International
Conference on Tools with Artificial Intelligence (ICTAI'06)
0-7695-2728-0/06 $20.00 © 2006 IEEE

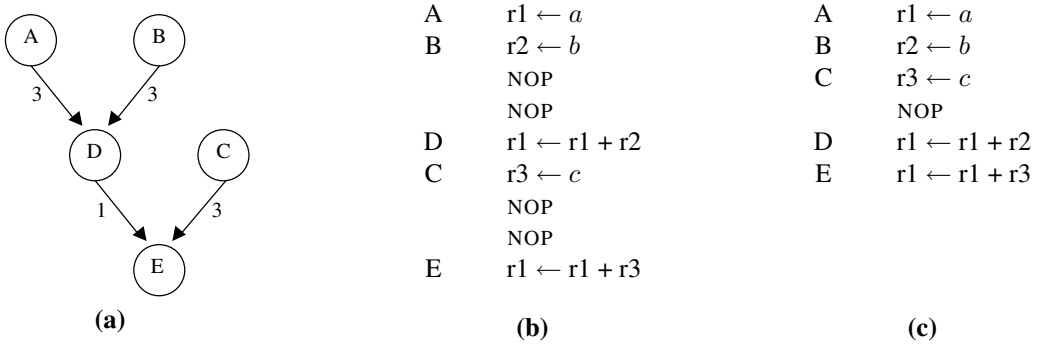

Figure 1. Dependency DAG and two possible schedules.

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.

An extended version of this paper containing proofs, additional descriptions of the constraints, and additional experimentation is available [11].

2. Background

In this section, we define the instruction scheduling problem studied in this paper followed by a brief review of the needed background from constraint programming.

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. Each instruction has a type and can only execute on a functional unit of that type. Examples of instruction types are load/store, integer, floating point, and branch.

We use the standard labeled directed acyclic graph (DAG) representation of a basic-block. 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; 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. 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 processor specifies an issue or start time for each instruction or node such that the latency and resource constraints are satisfied. 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.

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, where a problem is modeled in terms of variables, values, and constraints.

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

Proceedings of the 18th IEEE International
Conference on Tools with Artificial Intelligence (ICTAI'06)
0-7695-2728-0/06 $20.00 © 2006 IEEE

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. A constraint C is *bounds consistent* if for each $x \in vars(C)$, the value $\min(dom(x))$ has a support in C—there exists values for each of the other variables in C such that C is satisfied— 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_2: $\mathrm{D} \geq \mathrm{B} + 3$, C_4: $\mathrm{E} \geq \mathrm{D} + 1$,
C_5: all-different(A, B, C, D, E),

where constraint C_5 enforces that its arguments are pairwise different. The constraints are not bounds consistent. For example, the minimum value 1 in the domain of D does not have a support in 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, with a focus on the preprocessing techniques we used for improving the constraint model.

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 clock cycles. Assigning a value $d \in dom(i)$ to a variable i has the intended meaning that instruction i will be issued at clock cycle d. The domain $dom(i) = \{a, \ldots, b\}$ of a variable i is represented by the endpoints of the interval $[a, b]$.

Table 1. Notation for specifying constraints.

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 i and j
$op(i, j, t)$	set of all nodes of type t that are on some path from node i to node j. Note that $i \in op(i, j, t)$ if i is of type t. Similarly for j. 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
$I([a, b], t)$	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 clock cycles to execute on functional units of type t.

We now specify the five types of constraints in the model: latency, resource, distance, safe pruning, and dominance constraints. Some of the notation we use is summarized in Table 1. Given a labeled dependency DAG $G = (N, E)$, for each pair of variables i and j such that $(i, j) \in E$, latency constraints of the form $j \geq i + l(i, j)$ are added to the constraint model. 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 clock 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 [14]. Note that 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.

As is clear, for a minimal correct model of the instruction scheduling problem all that is needed are the latency and resource constraints. However, it is now well-established that adding implied (or redundant) constraints and dominance

Proceedings of the 18th IEEE International
Conference on Tools with Artificial Intelligence (ICTAI'06)
0-7695-2728-0/06 $20.00 © 2006 IEEE

Table 2. Notation for distance constraints.

$r_1(i,j,t)$	The minimum number of cycles that must elapse before the first instruction in $op(i,j,t)$ can be issued; i.e., $\min\{cp(i,k) \mid k \in op(i,j,t)\}$, the minimum critical-path distance from node i to any node in $op(i,j,t)$.
$r_2(i,j,t)$	The minimum number of cycles to issue all of the instructions in $op(i,j,t)$; i.e., $\lceil \lvert op(i,j,t) \rvert / k_t \rceil$, 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 $op(i,j,t)$ is issued and node j can be issued; i.e., $\min\{cp(k,j) \mid k \in op(i,j,t)\}$, the minimum critical-path distance from any node in $op(i,j,t)$ to node j.

constraints to a constraint model can greatly improve the efficiency of the search for a solution (see, e.g., [15]). Implied constraints are constraints which do not change the set of solutions to the constraint model. Dominance constraints do not necessarily preserve the set of solutions but do preserve at least one of the solutions. Both types of constraints can increase the amount of constraint propagation and so cause the domains of the variables to be further restricted. In our context, adding the distance, safe pruning, and dominance constraints was found to be essential in improving the efficiency of the backtracking search for a schedule—without them, only small problems could be consistently solved. Many instances of each of these three types of constraints are added in an extensive preprocessing stage.

3.1. 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.

In what follows, we are interested in subgraphs called regions [17], 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 2 (Region [17]) 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 [17], 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.

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 $op(i,j,t)$. 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 [16], 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 $op(\mathrm{A},\mathrm{G},\mathrm{shaded})$. The distance constraints $\mathrm{F} \geq \mathrm{A} + 4$ and $\mathrm{G} \geq \mathrm{A} + 6$ would be added to the constraint model,

Proceedings of the 18th IEEE International
Conference on Tools with Artificial Intelligence (ICTAI'06)
0-7695-2728-0/06 $20.00 © 2006 IEEE

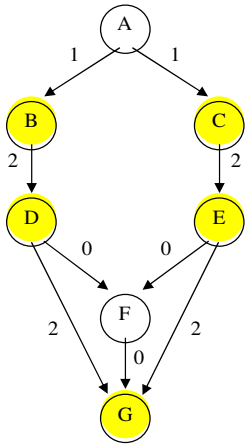

Figure 2. Adding distance constraints.

as both $d(\mathrm{A},\mathrm{F})$ and $d(\mathrm{A},\mathrm{G})$ are improvements over the critical-path distances between those nodes.

3.2. Safe pruning constraint

Given a constraint model, 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 safe pruning constraints 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 $[a,b]$ and a type t such that, (i) for all $i \in I([a,b],t)$, $\min(dom(i)) = a$, (ii) for all $i \in I([a,b],t)$, for all $k \in pred(i)$, $k + l(k,i) \leq i$, and (iii) $|I([a,b],t)| \leq (b - a+1) \times k_t$, then adding the constraints $i \leq b, i \in I([a,b],t)$, is safe.

A symmetric version of the theorem can be formulated for safely pruning the lower bounds of variables.

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. The safe pruning constraint can be applied iteratively as follows. First, the interval [2,2], where $I([2,2],\mathrm{clear}) = \{\mathrm{B}\}$, satisfies the theorem. Hence, node B can have its domain pruned to [2,2]. Second, the interval [3,3], where $I([3,4],\mathrm{clear}) = \{\mathrm{C}\}$, now satisfies the theorem. Hence, node C can have its domain pruned to [3,3]. Third, the interval [3,4], where $I([3,4],\mathrm{shaded}) = \{\mathrm{D}, \mathrm{E}\}$, also now satisfies the theorem. Hence, nodes D and E can have their domains pruned to [3,4].

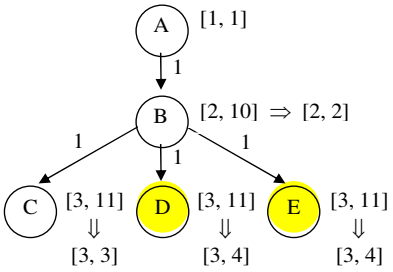

Figure 3. Improving bounds of variables using safe pruning constraints.

3.3. Dominance constraints

Heffernan and Wilken [8] 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 adaptations of 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). Adding dominance constraints, when it is safe to do so, is based on the following theorem.

Theorem 2 (Heffernan and Wilken [8]) 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 $k \in pred(a_i)$ such that $k \notin V(A)$, $l(k,a_i) \leq cp(k,b_i)$, $1 \leq i \leq r$, (iii) for all $k \in succ(b_i)$ such that $k \notin V(B)$, $l(b_i,k) \leq cp(a_i,k)$, $1 \leq i \leq r$, and (iv) for any edge (b_i, a_j), $l(b_i,a_j) \leq cp(a_i,b_j)$, then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$, is safe.

Example 5 Consider the DAG shown in Figure 4a. Dominance constraints can be added iteratively as follows. First, the subgraphs with nodes $V(A) = \{\mathrm{B}, \mathrm{D}\}$ and $V(B) = \{\mathrm{C}, \mathrm{E}\}$ are isomorphic and satisfy the conditions of the theorem. Hence, the constraints $\mathrm{B} \leq \mathrm{C}$ and $\mathrm{D} \leq \mathrm{E}$ can be added to the model. Adding these constraints updates the critical path distances. In particular, $cp(\mathrm{D},\mathrm{E})$ was $-\infty$ and is now 0. Second, the subgraphs with nodes $V(A) = \{\mathrm{F}\}$ and

Proceedings of the 18th IEEE International
Conference on Tools with Artificial Intelligence (ICTAI'06)
0-7695-2728-0/06 $20.00 © 2006 IEEE

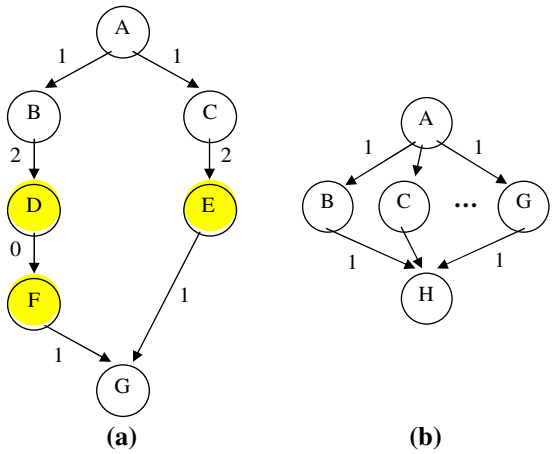

Figure 4. Adding dominance constraints.

$V(B) = \{\text{E}\}$ 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 [8] find isomorphic subgraphs that satisfy the theorem using a backtracking search with a time cutoff. In our work, we find isomorphic subgraphs by focusing on regions (see Definition 2) and using a heuristic test. Focusing on regions appears to find the "right" constraints to add to the constraint model, and using the heuristic greatly speeds up the computation.

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 subgraph 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 4b 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 [4]. 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 backtracking search for a schedule.

3.4. Solving an instance

Solving an instance of an instruction scheduling problem proceeds as follows.

We first 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 (and sometimes other forms of consistency) are enforced on the constraints until no further changes result. To enforce bounds consistency on the global cardinality constraints, we used the efficient algorithm presented in [13]. A dynamic variable ordering is used to select the next variable to instantiate. Given a selected variable x, the backtracking search first branches on x assigned to $\min(dom(x))$, then on x assigned to $\min(dom(x)) + 1$, 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

Table 3. Number of basic blocks where the optimal scheduler found an improved schedule (imp.), percentage of basic blocks with improved schedules (%), and number of basic blocks where the optimal scheduler failed to complete within a time limit of 10 minutes (t.o.), for ranges of basic block sizes and various issue widths.

		1-issue			2-issue			4-issue			6-issue		
range	#blocks	imp.	%	t.o.	imp.	%	t.o.	imp.	%	t.o.	imp.	%	t.o.
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	1	379	22.9	3	263	15.9	
251–2600	203	63	31.0	2	78	38.4	2	83	40.9		61	30.0	3
Total	183,112	2,867	1.6	2	3,593	2.0	3	4,135	2.3	3	2,147	1.2	3

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 an optimal schedule is reached, or a time limit is exceeded. An upper bound on the length of an optimal schedule is established by running a list-scheduling algorithm using a critical-path heuristic (see the next section). If a solution is found or the upper bound on the length of an optimal schedule is reached, a provably optimal solution has been found.

Further details on the solving process can be found in the extended version of the paper [11].

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[1]. The benchmarks were compiled using IBM's Tobey compiler targeted towards the IBM® PowerPC® processor, 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, 1–37 for integer instructions, and 1–38 for floating point instructions. The compilations were done using Tobey's highest level of optimization, which includes aggressive optimization techniques such as software pipelining and loop unrolling. The Tobey compiler performs instruction scheduling before global register allocation and once again afterward. Because of space limitations, we report only on the experiments on the basic blocks from after register allocation. A report on the full set of experiments can be found in the extended version of the paper [11].

We used the following four architectural models in our evaluation: a 1-issue processor executes all types of instructions; a 2-issue processor with one floating point functional unit and one functional unit that can execute integer, load/store, and branch instructions; a 4-issue processor with one functional unit for each type of instruction; and a 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 [6]. List scheduling is a greedy algorithm which uses a heuristic for which instruction to schedule next. Following Muchnick [12], our heuristic used critical-path distance as the primary feature and earliest start time as a tie-breaker. Although a popular heuristic, the primary reason for adopting this heuristic is that critical-path heuristics were also used in previous work [8, 16, 17], thus allowing a fairly direct comparison of previous experimental results with our experimental results.

Table 3 shows 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

[1] http://www.spec.org

Table 4. 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 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

to complete within the given time limit of 10 minutes[2]. 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 8 additional failures resulted for each issue width when scheduling after register allocation.) 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. 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.

Depending on the architectural model, the optimal scheduler took between 1:48:49 (hh:mm:ss) and 2:05:55 to schedule all of the basic blocks in the entire SPEC benchmark. The SPEC benchmark consists of 26 different software projects. The maximum amount of time taken scheduling the basic blocks in any individual project was 49:31 (mm:ss). While such long compile times would not be tolerable in everyday use, these times are well within acceptable limits when compiling for software libraries, embedded applications, or final release builds. We note that adding the implied distance constraints and the safe pruning and dominance constraints were critical to achieving this performance. Without these constraints, many individual basic blocks could *not* be solved within the amount of time that we can now solve the entire ensemble of basic blocks.

Wilken, Liu, and Heffernan [17] and van Beek and Wilken [16] 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 [16] 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 three reasons. First, the current test suite contains longer and more varied latencies (in [17], 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 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).

Heffernan and Wilken [8] 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.

Table 4 summarizes the percentage improvements in schedule length of the optimal schedule over the sched-

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

ule found by a list scheduling algorithm using the critical-path heuristic. Somewhat surprising is that on some 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.

5. Conclusion

We presented a constraint programming approach to basic block instruction scheduling for multiple-issue processors that is optimal yet robust on large, real problems. The key to scaling up to large, real problems was in the development of preprocessing techniques for improving the constraint model. 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.

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] 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.

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

[4] J. M. Crawford, M. L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of the 5th International Conf. on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

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

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

[7] S. Haga and R. Barua. EPIC instruction scheduling based on optimal approaches. In *Workshop on Explicitly Parallel Instruction Computing Arch. and Compiler Tech.*, 2001.

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

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

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

[11] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. Technical Report CS-2005-19, School of Computer Science, University of Waterloo, 2005.

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

[13] 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.

[14] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. of the 13th National Conf. on Artificial Intelligence*, pages 209–215, 1996.

[15] B. M. Smith. Modelling. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Chapter 11. Elsevier, 2006.

[16] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proc. of the 7th International Conf. on Principles and Practice of Constraint Programming*, pages 625–639, 2001.

[17] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 121–133, 2000.