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

Abid M. Malik, Tyrel Russell, Michael Chase, Peter van Beek

David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada

Abstract

Modern processors have multiple pipelined functional units and can issue more than one instruction per clock cycle. This puts great pressure on the instruction scheduling phase in a compiler to expose maximum instruction level parallelism. Instruction level parallelism (ILP) at the local level using basic blocks is limited. Compilers increase ILP by doing instruction scheduling at the global level using larger regions, which are created by combining basic blocks. Superblocks are one of the most commonly used scheduling regions for global instruction scheduling. Superblock scheduling is NP-complete, and is done sub-optimally in production compilers using heuristic approaches. In this work, we present a constraint programming approach to the superblock instruction scheduling problem that is both optimal and fast enough to be incorporated into production compilers. We experimentally evaluated our optimal scheduler on the SPEC2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler was very robust and scaled to the largest superblocks. Depending on the architectural model, between 99.991% to 99.999% of all superblocks were solved to optimality. The scheduler was able to routinely solve the largest superblocks, including superblocks with up to 2600 instructions. This compares favorably to the recent best work by Shobaki and Wilken on optimal superblock scheduling using dynamic programming and enumeration.

1 Introduction

Modern processors are pipelined and can issue more than one instruction per clock cycle. The main challenge is to find an order of instructions that minimizes pipeline stalls with out violating dependency or resource constraints. Depending upon the scope, there are two

*Technical Report CS-2006-37, David R. Cheriton School of Computer Science, University of Waterloo, 2006.

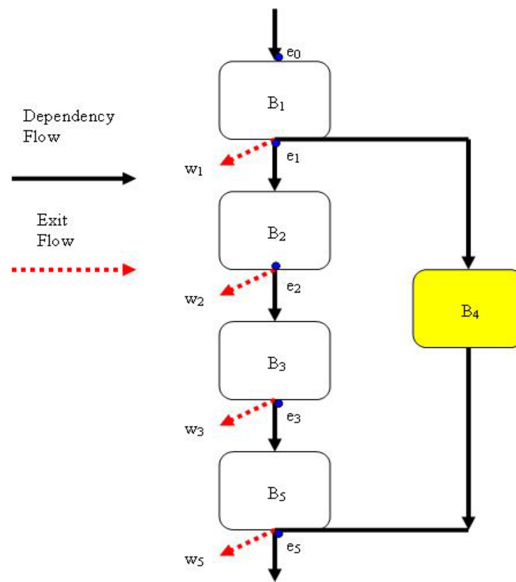


Figure 1: Section of a Control Flow Graph (CFG) with five basic blocks. Control enters through e_0 and can leave through e_1, e_2, e_3 or e_5.

types of instruction scheduling: local and global instruction scheduling. In local instruction scheduling, the reordering is done within a basic block. On wider issue machines, this approach does not detect enough parallelism among instructions to keep all the functional units busy. This bottleneck has stimulated substantial research effort in global instruction scheduling, where instructions are allowed to move across basic blocks.

Figure 1 shows a region in a Control Flow Graph (CFG)[1]consisting of five basic blocks. Instructions in the basic block B_4 are independent of the instructions in basic blocks B_2, B_3 and B_5. We can increase ILP by inserting instructions from B_4 into the free slots available in B_2, B_3 and B_5. This is only possible if we schedule instructions in all basic blocks at the same time. Many regions have been proposed for performing global instruction scheduling. The most commonly used regions are traces [13], superblocks [20] and hyperblocks [28]. The compiler community has mostly targeted superblocks for global instruction scheduling because of their simple implementation as compared to the other regions. Superblock scheduling is harder than basic block scheduling. In basic block scheduling, all resources are considered available for the basic block under consideration. In superblock scheduling, having multiple basic blocks with conflicting resource and data requirements, each basic block competes for the available resources [18]. A number of heuristics have been developed for superblock scheduling. However, even the best heuristics produce sub-optimal solutions [30].

Previous work on optimal instruction scheduling is mainly basic block based and several approaches, including: branch-and-bound enumeration [6, 16, 17, 25, 36], dynamic programming [23], integer linear programming [1, 4, 22, 24, 38], and constraint programming

[1]A CFG is an abstract data structure used in compilers to represent a program.

[12, 37, 27] have been proposed. However, little work has been done on optimal global instruction scheduling [36, 39]. 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. Winkel [39] presents an integer linear programming model for global instruction scheduling problem for Itanium processors. The model is limited to small regions with size up to 200 instructions. Shobaki and Wilken [36] were the first to develop a robust optimal scheduler for superblocks that scaled up to large superblocks. Their experimental work is limited to superblocks with size up to 1236 instructions. For their work, instruction latencies are 2 cycles for floating point (FP) adds, 3 cycles for loads and FP multiplies, 9 cycles for FP divides and 1 cycle for all other instructions. Our test suite, obtained from the IBM TOBEY compiler, contains larger superblocks with size upto 2600 instructions and more varied latencies. Our test suite also contains zero latency edges, which are used to capture anti-dependencies and output dependencies[2] between two instructions. This makes the optimal superblock scheduling problem more challenging.

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 SPEC2000 integer and floating point benchmarks, using four different architectural models. On this benchmark suite, the optimal scheduler scaled to the largest superblocks and was very robust. Depending on the architectural model, at most 15 superblocks out of 187334 superblocks used in our experiments could not be solved within a 10-minute time bound. In our experiments we also performed a detailed analysis of several state-of-the-art heuristics for superblock scheduling in comparison to the optimal scheduler.

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, [19, 29, 31]).

[2]An anti-dependency occurs when an instruction requires a value that is later updated; an output dependency occurs when the ordering of instructions will affect the final output value of a variable.

2.1 Target Machine Architecture

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.

2.2 Superblock Scheduling

A basic block is a collection of instructions with a unique entrance and a unique exit point. Larger scheduling units for global instruction scheduling such as traces, superblocks and hyperblocks are built by combining basic blocks. A trace [13] can have multiple side exit points (control flow out of the trace) and multiple side entrance points (control flow into the trace). Traces are created by marking the most frequently executed paths in a CFG using profiling. Scheduling based on traces is known as trace scheduling. In trace scheduling, a trace is scheduled independently ignoring side exit and side entrance points. This may move some instructions across side exit and side entrance points. Book keeping is done to ensure the correct execution of the program. Figure 2 explains the book keeping process for downward movement of an instruction across a side exit point. Trace-1 and Trace-2 are connected by the control flow from instruction 2 (exit point in Trace-1) to instruction 3 (side entrance point in Trace-2). Trace scheduling moves instruction 1 across instruction 2 in the downward direction. In order to ensure execution of instruction 1, even if the program jumps from instruction 2 to instruction 3, a copy of instruction 1 is placed between instruction 2 and instruction 3. Upward movement of an instruction across a side exit point is known as *speculation* and the moved instruction is called a *speculative instruction*. An instruction is allowed to be a speculative instruction if (i) the destination of the speculative instruction is not used before it is redefined when the exit point is taken and (ii) the speculative instruction will never cause an exception that may terminate program execution when the exit point is taken. Special hardware support is needed to handle speculative instructions, but no book keeping is done.

Complex book keeping is done when an instruction is moved across a side entrance. In Figure 3(a), instruction 5 is moved upward across the side entrance point. To ensure the execution of instruction 5, if the control is entering the trace through the side entrance, a copy of instruction 5 is placed at the entrance point. In Figure 3(b), instruction 1 is moved downward across the entrance point, and placed after instruction 4. For correct execution of the program, if the control enters the trace through the side entrance, instruction 1 should not be executed. The entrance point is moved down after instruction 1 and copies of instruction 3 and instruction 4 are placed at the side entrance. Book keeping for side entrance points makes other compiler optimization phases more difficult. This can be avoided by removing side entrance points in traces. Hwu et al. [20] give a solution by introducing superblocks

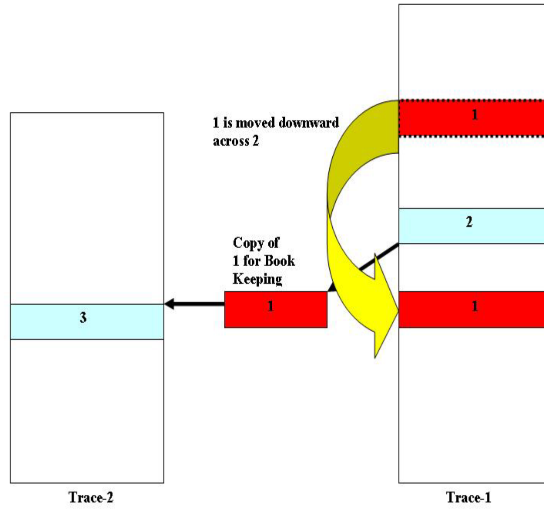


Figure 2: Book keeping for the downward movement across a side exit point: instruction 2 is a side exit point for Trace-1 and instruction 3 is a side entrance point in Trace-2. Instruction 1 is moved downward across instruction 2. A copy of instruction 1 is placed between instruction 2 and instruction 3.

which have unique entrance and multiple exit points. Superblocks are built from traces, and tail duplication is performed to remove the side entrances into a trace. All blocks from the side entrance to the end of the trace are duplicated, and all side entrances are redirected into the copy. Therefore, a superblock can have a single entry point but might have more than one exit point. Figure 4 shows the formation of a superblock. Basic blocks B_1, B_2 and B_4 form superblock S_1. Basic blocks B_3 and B_4' form superblock S_2.

We use the standard labeled directed acyclic graph (DAG) representation of a superblock (see [31]). 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. A *root* node is a node with no predecessor. For convenience, we assume that a fictitious sink node, hereafter called *the* sink node, is added to each DAG and that an

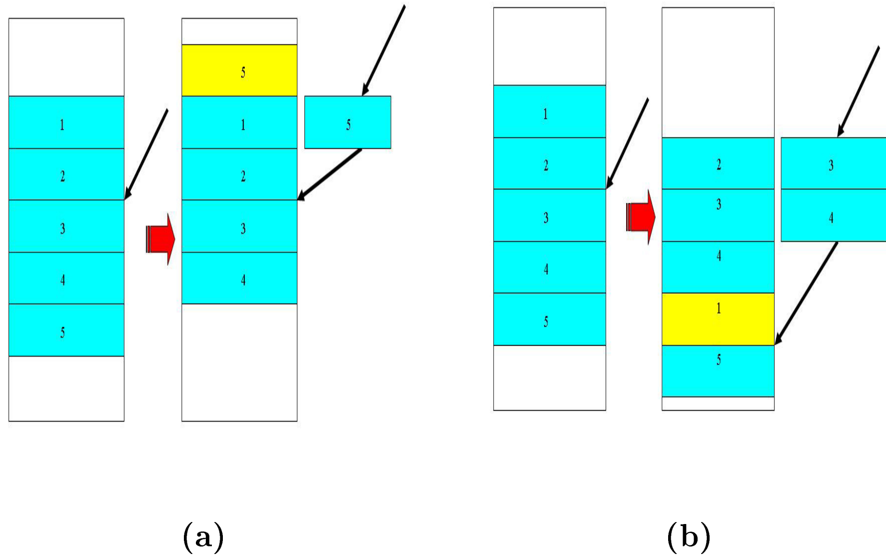


Figure 3: Book keeping (a) upward movement across side entrance instruction; (b) downward movement across side entrance instruction.

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. *Exit nodes* are special nodes in a DAG, representing branch instructions, which have some associated weightage. The weightage represents the chance that the flow of control will leave the superblock through this exit point and are calculated using profiling. The weightage for an exit node e_i, denoted by w_i, is also known as the *exit probability*. Figure 5 shows a DAG for a superblock.

When scheduling a basic block in local instruction scheduling, the objective is to minimize the schedule length of the basic block. In the case of global scheduling with superblocks, the objective is to minimize the *Weighted Completion Time* (WCT); i.e., the number of cycles from the entry point to each exit point, weighted by the exit probability. The weighted completion time is referred to as the cost function for the *optimal superblock scheduling problem*.

Definition 1 (Weighted Completion Time) Let $G(V, E)$ be a DAG that represents a superblock, where V is the set of nodes and E is the set of edges in G. Let there be n branch instructions, i.e., exit points. Let w_i be the weight of branch e_i, where $1 \leq i \leq n$ and $e_i \in V$. The weight of a branch instruction is equal to the exit probability associated with that branch instruction. Hence, $\sum_{i=1}^{n} w_i = 1$. The weighted completion time, WCT, for G is $\sum_{i=1}^{n} w_i e_i$.

A schedule for a superblock is an assignment of a clock cycle to each instruction such that the precedence, latency and resource constraints are satisfied.

Definition 2 (Optimal Superblock Scheduling) Let S be a schedule for a superblock's

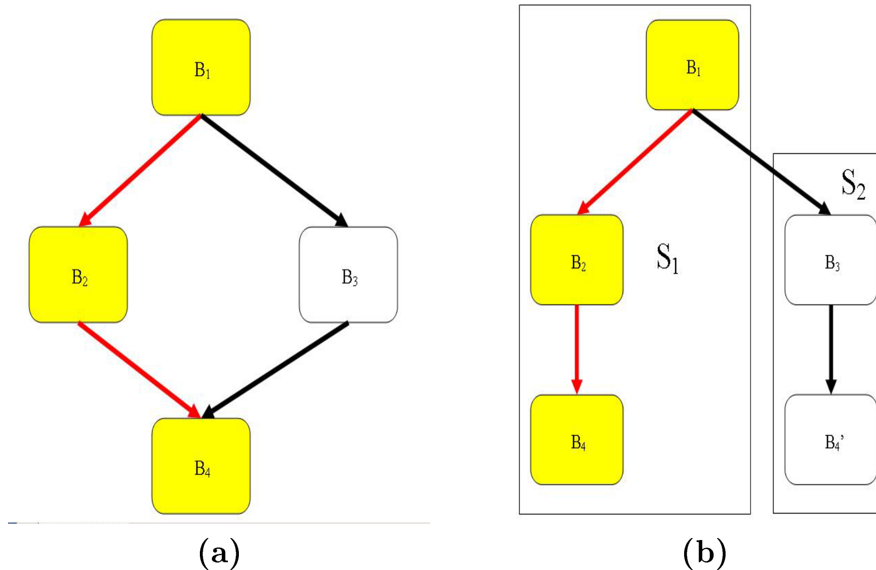
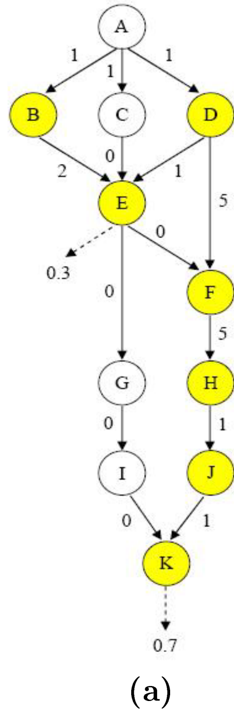


Figure 4: Superblock formation: B_i is a basic block in a CFG (a) Path $B_1 \to B_2 \to B_4$ has the highest probability of execution; (b) In order to remove the entrance from B_3 to path $B_1 \to B_2 \to B_4$, a copy of B_4 is created, called tail duplication, and the flow from B_3 is directed towards B_4'.

DAG G with weighted completion time W. Schedule S is an *optimal schedule* for G, iff W is the minimum for all schedules for G.

Example 1 Consider the superblock in Figure 5. Nodes E and K are branch instructions, with exit probability 0.3 and 0.7 respectively. Consider a fully pipelined processor with two functional units. One functional unit can execute clear instructions and the other can execute shaded instructions. Figure 5(b) shows two possible schedules, S_1 and S_2. For S_1, $WCT = 0.3 \times 4 + 0.7 \times 15 = 11.7$ cycles. And for S_2, $WCT = 0.3 \times 5 + 0.7 \times 14 = 11.3$ cycles. Schedule S_2 is an *optimal solution*.

List scheduling is the commonly used algorithm for superblock scheduling. It is a greedy algorithm and gives near optimal solutions. It maintains a priority queue of ready instructions which are instructions with no predecessors. The priority of an instruction is calculated using a heuristic. For a given clock cycle, the list scheduler picks the top instructions in the priority queue. The number of picked instructions depend upon the number of functional units and the available free slots in the given cycle. If it could not find any instruction, it inserts a NOP (No OPeration). It continues this process until all instructions are scheduled. Many heuristics have been crafted to find a good schedule including critical path [20], successive retirement [5], dependence height and speculative yield [13], G^* [5], speculative hedge [10] and balance scheduling [30]. In our work we did a detail analysis of critical path, G^*, dependence height and speculative yield and speculative hedge heuristics with respect to their success for finding optimal solutions. We dropped the balance scheduling and successive retirement heuristics because of their high computational cost.



cycle	S_1		S_2	
1	A		A	
2	C	B	C	D
3		D		B
4	G	E		
5	I		G	E
6			I	
7				F
8		F		
9				
10				
11				
12				H
13		H		J
14		J		K
15		K		

(b)

Figure 5: (a) Superblock representation: nodes E and K are exit nodes with exit probabilities 0.3 and 0.7 respectively; (b) two possible schedules for Example 1.

Superblocks, when introduced [20], were scheduled using the critical path heuristic. The critical path heuristic is good when the aim is to minimize the distance between the *root* and the *sink* node. In superblock scheduling the objective is to minimize the WCT. *Exit nodes*, which define the WCT, may not be on the critical path of a DAG representing a superblock. Hence, this heuristic may not be a good choice for finding a good schedule for superblock scheduling.

The Dependence Height and Speculative Yield (DHASY) [13] heuristic is a modified version of the critical path heuristic for superblock scheduling. Instead of a plain critical path, a weighted critical path to all exit points is used to prioritize the instruction nodes in a superblock. The priority of an instruction x is calculated as,

$$priority(x) = \sum_{e \in B} (w_e(cp(1,n) + 1 - ((cp(1,e) - cp(x,e)))$$

where B is the set of exit nodes that are descendants of x, $cp(1,n)$ is the critical path distance between the root and the sink node, $cp(1,e)$ is the critical path distance between the root node and exit node e and $cp(x,e)$ is the critical path distance between instruction x and exit node e.

In the G* heuristic [5], a superblock is scheduled using the critical path heuristic. The rank for each exit point is then calculated by dividing the cycle in which the exit point is scheduled by the sum of the exit probabilities for the exit point under consideration and its preceding exit points. The exit points are sorted in ascending order. The final schedule for the superblock is obtained by taking an exit point from the sorted list one by one and scheduling it as early as possible with its predecessors.

The Speculative Hedge [10] heuristic calculates the priority of an instruction by the sum of the weights of the branches that it helps schedule early. Speculative Hedge investigates each operation to determine whether it helps still unscheduled exit points or not. An operation can help an exit point in two ways: (i) the operation is on the critical path to the exit point and delaying the operation will delay the exit point, and (ii) the operation uses a critical resource that is critical to the exit point, and preferring some other operation will delay the exit point. An operation's priority is the sum of the exit probabilities helped by the operation.

2.3 Constraint Programming

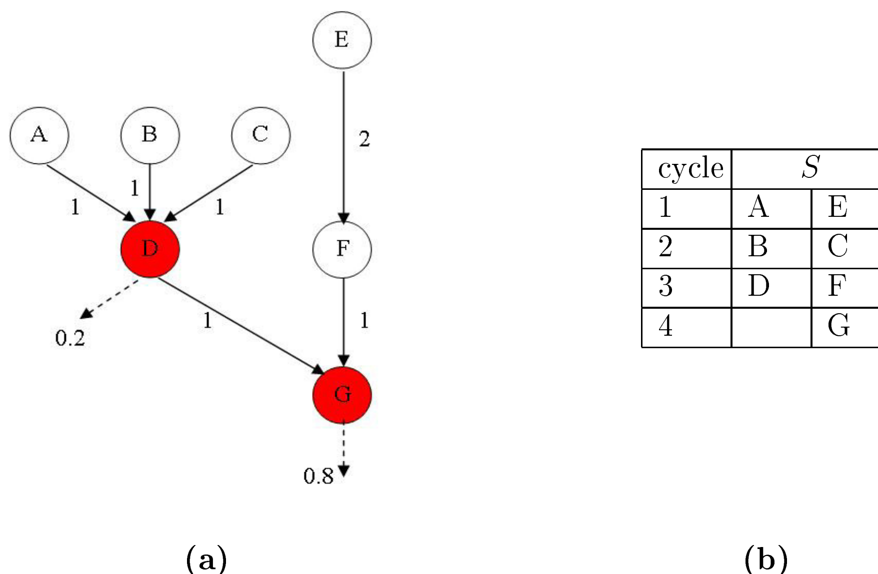
Constraint programming is a methodology for solving combinatorial problems. A problem is modeled by specifying constraints on an acceptable solution, where a constraint is simply a relation among several unknowns or variables, each taking a value in a given domain.

Definition 3 (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 are bounds consistency.

Definition 4 (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 explains the concept of bounds consistency constraint propagation.



cycle	S	
1	A	E
2	B	C
3	D	F
4		G

Figure 6: (a) Superblock (taken from [30]) for Example 2. Nodes D and G are branch instructions; (b) a possible schedule for Example 2.

Example 2 Consider the constraint model of the small instruction scheduling problem in Figure 6(a) with variables A, ..., G, each with domain $\{1, 2, 3, 4\}$, and the constraints,

C_1: $\mathrm{D} \geq \mathrm{A} + 1$, C_3: $\mathrm{D} \geq \mathrm{C} + 1$, C_5: $\mathrm{G} \geq \mathrm{F} + 1$,
C_2: $\mathrm{D} \geq \mathrm{B} + 1$, C_4: $\mathrm{F} \geq \mathrm{E} + 2$, C_6: $\mathrm{G} \geq \mathrm{D} + 1$,
C_7: $gcc(A, B, C, D, E, F, G, width = 2)$,

where constraint C_7, a global cardinality constraint (gcc), enforces that atmost two instructions can be issued in any cycle. 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, C_2 and C_3 as there is no corresponding values for A, B and C that satisfies the constraints. Enforcing bounds consistency using constraints C_1 through C_6 reduces the domains of the variables as follows: $dom(\mathrm{A}) = \{1, 2\}$, $dom(\mathrm{B}) = \{1, 2\}$, $dom(\mathrm{C}) = \{1, 2\}$, $dom(\mathrm{D}) = \{2, 3\}$, $dom(\mathrm{E}) = \{1, 2\}$, $dom(\mathrm{F}) = \{3\}$, and $dom(\mathrm{G}) = \{4\}$. We are considering dual-issue fully pipelined processors. Enforcing bounds consistency using C_7 reduces the domain of D to $dom(\mathrm{D}) = \{3\}$. Now, arbitrarily picking[3] A and E for the first cycle and enforcing bounds consistency using C_7 again will reduce the domains of variables as

[3]In constraint programming, special heuristics are adopted to select variables.

follows: $dom(\text{A}) = \{1\}$, $dom(\text{B}) = \{2\}$, $dom(\text{C}) = \{2\}$, $dom(\text{D}) = \{3\}$, $dom(\text{E}) = \{1\}$, $dom(\text{F}) = \{3\}$ and $dom(\text{G}) = \{4\}$, which is a schedule S given in Figure 6(b).

3 Our Solution

In this section, we present our constraint model of the superblock 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 $d \in dom(i)$ 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 $[a, b]$.

The six main types of constraints in the model are latency, resource, distance, predecessor and successor, safe pruning, and dominance constraints. For details on these constraints, consult the work by Malik et al. [27]. We discuss here only the dominance constraints and distance constraints for subgraphs in a DAG for a superblock.

3.1 Dominance constraints

Heffernan and Wilken [17] 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 and hence are safe. Malik et al. [27] found that these transformations also worked well in a constraint programming approach for basic block scheduling. The transformations add simple constraints to the model of the form $i \geq j$, which they call dominance constraints.

For the transformation, 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). That adding dominance constraints between A and B is safe is based on the following theorem.

Theorem 1 (Heffernan and Wilken [17]) Let A and B be isomorphic subgraphs in a DAG G_b of a basic block, 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 $k \notin V(B)$,

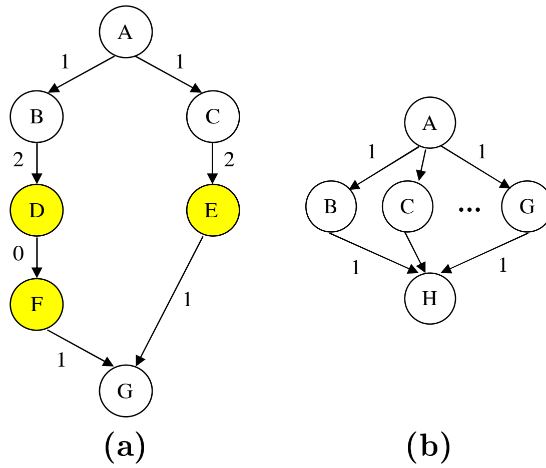


Figure 7: Examples of adding dominance constraints: (a) (adapted from [17]) the constraints B $\leq$ C, D $\leq$ E, and F $\leq$ E would be added to the constraint model; (b) the constraints B $\leq$ C, C $\leq$ D, $\ldots$, F $\leq$ G would be added to the constraint model.

$l(b_i, k) \leq cp(a_i, k)$, $1 \leq i \leq r$, and (iv) for any edge (b_i, a_j), $l(b_i, a_j) \leq cp(a_i, b_j)$, then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$ in G_b is safe.

Example 3 Consider the DAG shown in Figure 7a. Dominance constraints can be added iteratively as follows. First, the subgraphs with nodes $V(A) = \{\text{B, D}\}$ and $V(B) = \{\text{C, E}\}$ 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, $cp(\text{D}, \text{E})$ was $-\infty$ and is now 0. Second, the subgraphs with nodes $V(A) = \{\text{F}\}$ and $V(B) = \{\text{E}\}$ are isomorphic and now satisfy the conditions of the theorem. Hence, the constraint F $\leq$ E can be added to the model.

Adding a dominance constraint in a dependency DAG for a superblock is safe, if it does not change the optimal cost function value; i.e, WCT of the DAG. The number of speculative instructions across an exit node define the speculative characteristic of the exit node. The speculative characteristic of exit nodes and their schedule length affect the WCT value. Let G be a DAG of a superblock S. Let l_i^* be the minimum schedule length of exit node e_i from the root node of G for all schedules of S. If there are n exit nodes in S, then a lower bound, C^*, on the value of WCT can be calculated as:

$$C^* = \sum_{i=1}^{n} w_i l_i^*.$$

Let l_i be the schedule length of exit node e_i from the root node of G in any schedule of S. The following relationship holds between C^* and the value of WCT, C, for any schedule:

$$C = \sum_{i=1}^{n} w_i l_i \geq C^* = \sum_{i=1}^{n} w_i l_i^*.$$

Let C' be the difference between C and C^* i.e.,

$$\begin{aligned} C' &= C - C^*, \\ C' &= \sum_{i=1}^{n} w_i(l_i - l_i^*), \\ C' &= \sum_{i=1}^{n} w_i \delta_i, \end{aligned}$$

where $\forall i, \delta_i = l_i - l_i^*$. δ_i gives the distance of e_i from its minimum schedule length in any schedule for the superblock. In order to ensure an optimal solution for G after a transformation, we have to ensure that δ_i does not change after the transformation. The value of δ_i depends upon the number of instructions that can be moved across e_i; i.e., the speculative characteristic of e_i and the minimum schedule length of e_i. If we preserve the speculative characteristic and the minimum schedule length of e_i, we preserve the value of δ_i and hence the optimal cost function value for G after transformation.

Definition 5 (Immediate Predecessor Exit Node) If all paths from an exit node e_i to a node j do not contain any other exit node, then e_i is an immediate predecessor exit node of j.

Definition 6 (Immediate Successor Exit Node) If all paths from a node j to an exit node e_i do not contain any other exit node, then e_i is an immediate successor exit node of j.

Example 4 Consider Figure 5. Nodes E and K are exit nodes. Node E is the immediate predecessor exit node for nodes F, G, H, I and J. Node K is the immediate successor exit node for nodes F, G, H, I and J.

We restate the theorem by Heffernan and Wilken [17] for dependency DAGs for superblocks.

Theorem 2 Let A and B be isomorphic subgraphs in a DAG G_s of a superblock S, 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), $l(b_i, a_j) \leq \mathit{cp}(a_i, b_j)$, and (v) neither a_i nor b_i, $1 \leq i \leq r$, are exit nodes. Then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$ in G_s is safe.

Proof. Figure 8 shows the two possibilities of adding a dominance constraint in a superblock. Subgraph A and B are same as in Theorem 2; i.e., A dominates B. To show that the transformations are safe, we must show that the transformations preserve the speculative characteristic and the minimum schedule length of the exit nodes.

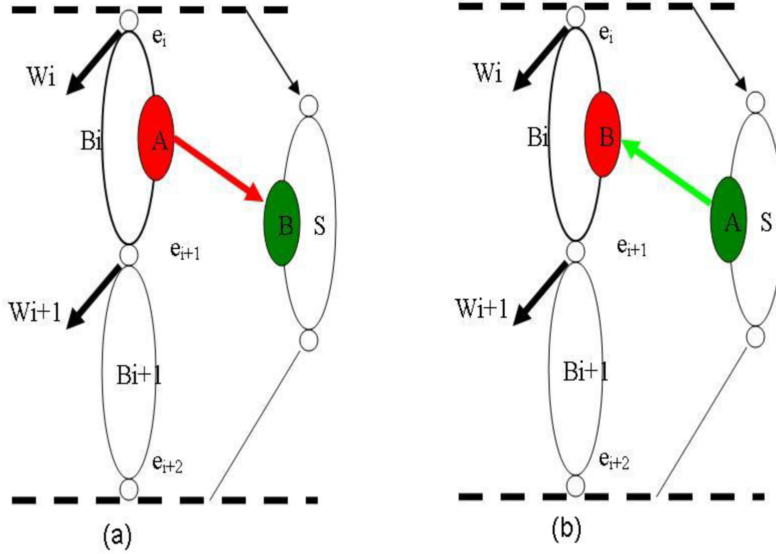


Figure 8: Adding dominance constraints in a superblock. A and B are isomorphic graphs (a) Case-1: $V(B)$ consists of speculative nodes (b) Case-2: $V(A)$ consists of speculative nodes.

Part-1 : Preserving the speculative characteristic of exit nodes.

Case-1 : *V(B) consists of speculative nodes that can be moved across basic blocks.* By inserting zero-latency edges from V(A) to V(B) (adding dominance constraints $V(A) \leq V(B)$), we are restricting the movement of $b_i \in V(B)$ to be below e_i; i.e., e_i, which is an immediate predecessor exit node for $a_i \in V(A)$, now is also the immediate predecessor exit node for $b_i \in V(B)$. According to condition (ii) of Theorem 2, there is a path from each predecessor of $a_i \in V(A)$ to $b_i \in V(B)$. As there is a path from e_i to each predecessor of $a_i \in V(A)$, then there is also a path from e_i to $b_i \in V(B)$, which also makes e_i the immediate predecessor exit for $b_i \in V(B)$. Thus the transformations do not change the speculative characteristic of the exit nodes in the superblock.

Case-2 : *V(A) consists of speculative nodes that can be moved across basic blocks.* By inserting zero-latency edges from V(A) to V(B), we are restricting the movement of $a_i \in V(A)$ to be above e_{i+1}; i.e., e_{i+1}, which is the immediate successor exit node for $b_i \in V(B)$, now is also the immediate successor exit node for $a_i \in V(A)$. According to condition (iii) of Theorem 2, there is a path from $a_i \in V(A)$ to successors of $b_i \in V(B)$. As there is a path from each successor of $b_i \in V(B)$ to e_{i+1}, then there is also a path from each $a_i \in V(A)$ to e_{i+1}, which also makes e_{i+1} the immediate successor exit for $a_i \in V(A)$. Thus the transformations do not change the speculative characteristic of the exit nodes in the superblock.

Part-2 : Preserving the minimum schedule length of exit nodes.

The minimum schedule length of e_{i+1} can be determined by scheduling a subgraph G' containing e_{i+1} and all its predecessors using an optimal scheduler. According to Theorem 1, adding dominance constraint within G' preserves the minimum schedule length of e_{i+1}.

Case-1 : *V(B) consists of speculative nodes that can be moved across basic blocks.* This case can be further divided into following three sub-cases:

- When for every $b_i \in V(B)$, there is a path from b_i to e_{i+1}. This makes each $b_i \in V(B)$ a predecessor of e_{i+1}, i.e., $b_i \in V(G')$. Then the transformations are within subgraph G'. The transformations preserve the minimum schedule length of e_{i+1}.
- When for every $b_i \in V(B)$, there is no path from b_i to e_{i+1}. It means $b_i \ni V(G')$. Then the transformations are outside of subgraph G'. The transformations do not change G'. The transformations preserve the minimum schedule length of e_{i+1}.
- When for some $b_i \in V(B)$, there is a path from b_i to e_{i+1} and for some $b_i \in V(B)$ there is no path from b_i to e_{i+1}. Let B_1 be a subgraph of B consisting of $b_i \in V(B)$ which has a path to e_{i+1}. Let A_1 be a subgraph of A which is isomorphic to B_1. Then adding dominance constraints from $a_i \in V(A_1)$ to $b_i \in V(B_1)$ are within G' and dominance constraints from $a_i \in V(A - A_1)$ to $b_i \in V(B - B_1)$ are outside of G'. The transformations preserve the minimum schedule length of e_{i+1}.

Case-2 : *V(A) consists of speculative nodes that can be moved across basic blocks.* In this case all immediate successors of $b_i \in V(B)$ are predecessors of e_{i+1}. According to condition (iii) of Theorem 2, there is a path from $a_i \in V(A)$ to successors of $b_i \in V(B)$. As there is a path from each successor of $b_i \in V(B)$ to e_{i+1}, then there is also a path from each $a_i \in V(A)$ to e_{i+1}, which also makes e_{i+1} the immediate successor exit for $a_i \in V(A)$. The transformations are within subgraph G'.The transformations preserve the minimum schedule length of e_{i+1}.

Thus, the transformations are safe. □

Example 5 Consider the DAG shown in Figure 9(a). Nodes H and I are speculative nodes as they can be moved across exit node G. Hence, the number of speculative instructions across exit node G is 2. The subgraphs with nodes $V(A) = \{\text{C, E}\}$ and $V(B) = \{\text{H, I}\}$ are isomorphic and satisfy the conditions of Theorem 1. Hence, the constraints C $\leq$ H and E $\leq$ I can be added to the model. Figure 9(b) shows the DAG with the added constraints. The added constraints do not change the speculative characteristic of exit node G, as node H and node I still can be moved across node G.

Testing isomorphism is NP-complete in general. Malik et al. [27] give a fast heuristic to determine whether two components are isomorphic. In this work, we adopted the same strategy for finding isomorphic graphs to add dominance constraint.

3.2 Upper bound distance constraints

Wilken et al. [38] introduced the concept of *region* in their work for optimal basic block scheduling using integer programming. Using the concept of region, Van Beek and Wilken

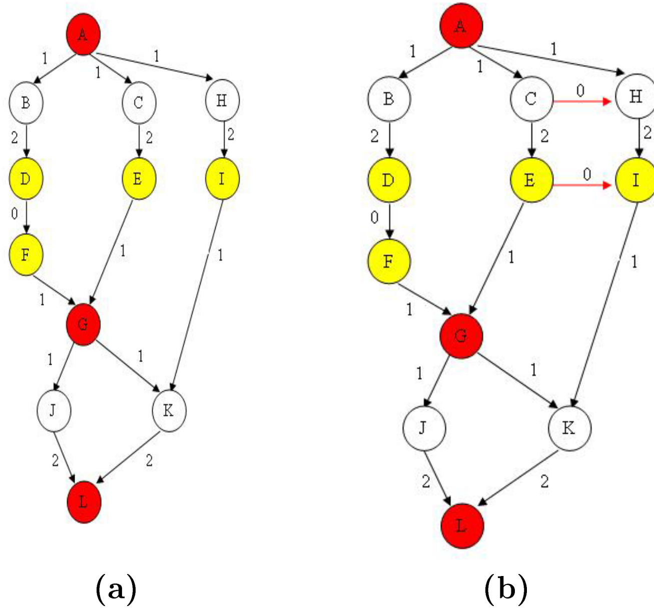


Figure 9: Example of adding dominance constraints in a superblock; (a) actual DAG; (b) the constraints C $\leq$ H, E $\leq$ I (zero latency edges) would be added to the constraint model. Nodes A, G and L are exit nodes.

[37] introduced the distance constraint in their work for optimal basic block scheduling using constraint programming. The distance constraint improves the lower bound for the distance that must exist between a pair of instructions, defining the region, in any schedule. In our work, we give an upper bound for the distance constraint for a region between a pair of articulation nodes. Consider articulation nodes x_i and x_j in a superblock, with no exit node in between them, a distance constraint of the form $x_i + d_{ij} \geq x_j$ is added to the constraint model. If there is no resource contention at x_i, then d_{ij} is the minimum schedule length, l^*_{ij}, between x_i and x_j in any legal schedule for the superblock. If there is resource contention at x_i, then $d_{ij} = l^*_{ij} + 1$. Adding upper bound distance constraints for such regions is based on Theorem 3.

Definition 7 (Region) A pair of nodes x_i, x_j in a DAG define a *region* if there is more than one path between x_i and x_j and there does not exist a node x_k distinct from x_i and x_j such that every path between x_i and x_j goes through x_k.

Definition 8 (Articulation Node) Let G be a graph. Node $V_i \in V(G)$ is an *articulation node* for G, if the subgraph of G induced by $V(G)/\{V_i\}$ is unconnected.

Definition 9 (Resource Contention) Let G be a DAG for superblock. Let B_{i-1} and B_i be two basic blocks in G connected by exit node e_i. If instructions from B_{i-1} and B_i compete for slots available at the clock cycle in which e_i can be issued, then there is said to be *resource contention* at the exit node e_i.

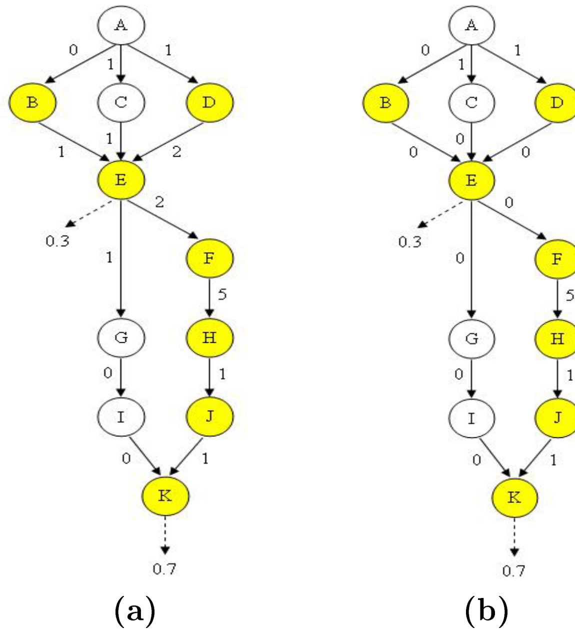


Figure 10: Articulation node and resource contention: (a) no resource contention at the articulation node E; (b) resource contention at the articulation node E.

Example 6 Consider Figure 10. Assume a fully pipelined processor with issue-width equal to four. Basic block B_1 consists of nodes A, B, C, D and E. Basic block B_2 includes of nodes E, F, G and H. Node E is an articulation node. There is resource contention at E in Figure 10(b), as nodes B, C, D from B_1 and nodes F, G from B_2 compete for the slots in the cycle in which E can be issued. There is no resource contention at E in Figure 10(a).

Theorem 3 The schedule length of a region between two consecutive articulation nodes, with no exit nodes in-between them, can not be more than $l^* + 1$, where l^* is the optimal schedule length of the region when scheduled independently.

Proof. Consider a region r_{ij} between exit nodes x_i and x_j in a superblock S. Where x_i is a predecessor of x_j, and there are no exit nodes between x_i and x_j as per the statement of Theorem 3. When r_{ij} is scheduled independently, all resources are considered at the disposal of region r_{ij} at clock cycle 1. An optimal scheduler will give optimal schedule length l^* for r_{ij}. In any schedule for S, r_{ij} can not have schedule length less than l^*. When there is no resource contention at x_i, the situation is the same as if the region was being scheduled independently. If there is resource contention, then some resources might still be occupied by the instructions which are predecessors of x_i. Let t_i be the clock cycle of x_i. If we insert a free slot at $t_i + 1$, then all resources will be available for the region r_{ij}, and all the nodes in r_{ij} can be scheduled within $t_i + l^* + 1$ schedule length from x_i. Thus, the distance between x_i and x_j can not be more than $l^* + 1$ in any schedule for S. $\Box$

Example 7 Figure 11(a) is a region in a superblock bounded by articulation nodes A and E. Nodes X and Y are predecessor nodes of A. With latency of more than zero between A and

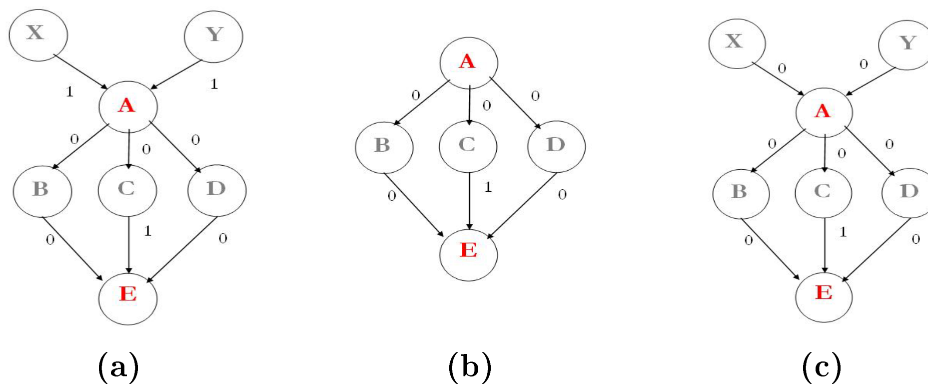


Figure 11: Region for Example 7: (a) no resource contention at the articulation node A; (b) region between node A and E in isolation; (c) resource contention at the articulation node A.

cycle	*Schedule*	
1	A	B
2	C	D
3	E	

Table 1: An optimal schedule for the region in Figure 11(b).

its predecessor nodes, there is no resource contention at node A. The distance between A and E can not be less than the minimum schedule length of the region scheduled independently by an optimal scheduler. Figure 11(b) shows the region in isolation. Considering a dual-issue processor, Table 1 gives a minimum length schedule for the region. In Figure 11(c), the latency between A and its predecessor nodes is not zero. This gives rise to resource contention at A. The worst case is when one of the predecessor node takes the slot parallel to node A. But, we can still schedule all the nodes in the region (excluding A) within the minimum length after the node A issue slot.

3.3 Improved lower and upper bounds for cost variables

The cost function, *cost*, is given by,

$$cost = \sum_{i=1}^{n} w_i x_i,$$

where w_i is the exit probability of exit node e_i with schedule length x_i. The cost function value from any efficient heuristic approach can be used as an upper bound. Given an upper bound, c, on *cost* and bounds on the variables in the cost function, i.e. exit nodes; it is straightforward to improve upper bounds for each cost function variable by considering each exit node at their minimum domain value except the exit node under consideration. An upper bound improvement for exit node e_j can be calculated as:

$$
\begin{aligned}
c &\geq \sum_{i=1}^{n} w_i x_i \\
c &\geq c_j x_j + \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i x_i \\
\frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i x_i}{c_j} &\geq x_j \\
\frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i \cdot \min(x_i)}{w_j} &\geq x_j
\end{aligned}
$$

$$
\left\lfloor \frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i \cdot \min(x_i)}{w_j} \right\rfloor \geq x_j \tag{1}
$$

We also use singleton consistency to prune the upper bounds of the cost variables. 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 on the upper bounds of the domains of the variables. Let x_j be a cost variable and $dom(x_j) = [a, b]$. We temporarily instantiate $x_j \leftarrow b$ and test whether the CSP is consistent by propagating the constraints and also by testing, once the constraints have been propagated and the lower bounds have potentially been updated, whether Equation 1 is satisfied. If the CSP is not consistent or the equation is not satisfied, the domain of x_j is set to $[a, b-1]$ and the process repeats.

Given a lower bound c on *cost* and bounds on the variables in the cost function, the lower bound for each cost function variable can be improved by considering each exit node at their maximum domain except the node under consideration. A lower bound improvement for exit node e_j can be determined by Equation 2.

$$
\left\lceil \frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{e+1} c_i \cdot \max(x_i)}{w_j} \right\rceil \leq x_j \tag{2}
$$

We also use singleton consistency to prune the lower bounds of the cost variables. Let x_j be a cost variable and $dom(x_j) = [a, b]$. We temporarily instantiate $x_j \leftarrow a$ and test whether the CSP is consistent by propagating the constraints and also by testing, once the constraints have been propagated and the lower bounds have potentially been updated, whether Equation 2 is satisfied. If the CSP is not consistent or the equation is not satisfied, the domain of x_j is set to $[a+1, b]$ and the process repeats.

3.4 Solving an instance

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. A lower bound on the value of the cost function of an optimal schedule is given by:

$$
\begin{aligned}
cost &= \sum_{i=1}^{n} w_i x_i \\
&\geq \sum_{i=1}^{n} w_i \cdot \min(x_i)
\end{aligned}
$$

An upper bound on the value of the cost function is found by a heuristic approach. If $lower(cost) = upper(cost)$, then the schedule given by the heuristic approach is optimal. If $lower(cost) \neq upper(cost)$, an optimal schedule can be determined depending upon the characteristic of the superblock. For a superblock with all exit points as articulation points, Algorithm 1 is adopted. In Algorithm 1, *ConstraintModel* constructs a constraint model of DAG G. *SubConstraintModel* gives a sub-constraint model of G with exit node e_i as final exit point. *OptimalSchedule* gives the optimal schedule length for exit node e_i using sub-constraint model CM'. The optimal scheduler for basic blocks by Malik et al. [27] has been used as *OptimalSchedule*. *UpdateDomain* makes the domain of exit node e_i the singleton domain equal to L_i.

Algorithm 1: Algorithm for finding an optimal schedule for a superblock with exit points as articulation points

---

input : Dependency DAG G for a Superblock S
output: Optimal Schedule for S
CM = ConstraintModel (G);
for i=0 *to* n **do**
{

$$CM' = SubConstraintModel(CM, i);$$

$$L_i = OptimalSchedule(CM', Schedule);$$

$$UpdateDomain(i, L_i, CM);$$

}
return Schedule;

Example 8 Consider Figure 12. Using Algorithm 1, an optimal schedule for the superblock can be obtained by first finding the minimum schedule length of exit node e_1, then fixing this node at its minimum schedule length slot and finding the minimum schedule length for

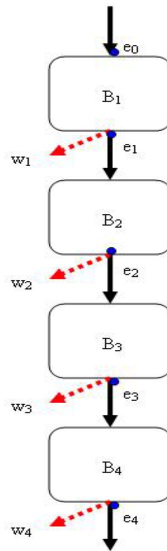


Figure 12: Superblock for Example 8; all exit nodes are articulation points

next exit node e_2 and so on. This methodology ensures an optimal solution by the following theorems.

Lemma 1 When scheduling a superblock using Algorithm 1, the first basic block of the super block will have schedule length equal to l^*, where l^* is the optimal schedule length of the basic block when scheduled independently.

Proof. When a superblock is scheduled using Algorithm 1, the resource condition for the first basic block is same as when it is scheduled independently using an optimal scheduler. $\square$

Theorem 4 Scheduling a superblock with all exit nodes as articulation nodes using Algorithm 1, each exit node in the superblock is at the minimum schedule length from the root node of the superblock.

Proof. Let S be a schedule for a superblock obtained using Algorithm 1. Let there be n exit nodes in the superblock with e_0 as the root node. Let L_i, $0 \leq i \leq n$, be the schedule length of exit node e_i from e_0 in S. Let S' be any other feasible schedule of the superblock. Let L'_i, $0 \leq i \leq n$, be the schedule length of exit node e_i from e_0 in S'. Our claim is:

$$\forall i, L_i \leq L'_i$$

We will use a proof by contradiction approach. Suppose there exist values of i for which this claim is not true. Let j be the smallest such value; i.e. for $\forall i < j$, the claim is true, and e_j is the first exit node which contradicts the claim. For this condition to be true, we have to examine two cases.

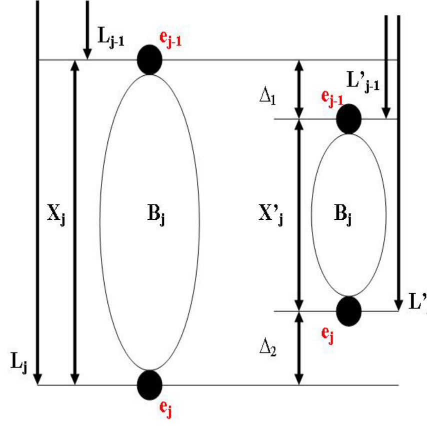


Figure 13: Case-2: $L_j > L'_j$ exists when $L_{j-1} < L'_{j-1}$

Case-1: $L_j > L'_j$ exists when $L_{j-1} = L'_{j-1}$.

This is not possible. As we are using a true optimal scheduler. Had there been a schedule which gives $L'_j < L_j$, when $L_{j-1} = L'_{j-1}$, the optimal scheduler would have found it.

Case-2: $L_j > L'_j$ exists when $L_{j-1} < L'_{j-1}$.

Figure 13 explains *case-2* graphically. Let X_j be the schedule length of the basic block B_j, that exists between exit nodes e_j and e_{j-1}, in the schedule S, that is found using Algorithm 1. Let X'_j be the schedule length of the same basic block in the schedule S', having $L_j > L'_j$, found by any other heuristic. The relationship between X_j and X'_j can be expressed as:

$$X_j = X'_j + \triangle_1 + \triangle_2, \tag{3}$$

where $\triangle_1 = L'_{j-1} - L_{j-1}$ and $\triangle_2 = L_j - L'_j$. We know that $L_j > L'_j$ and $L_{j-1} < L'_{j-1}$. Therefore, $\triangle_1, \triangle_2 \geq 1$. Let $\triangle_{total} = \triangle_1 + \triangle_2$. Then we can safely say that $\triangle_{total} \geq 2$. According to Theorem 3, $X_{op_j} + 1 \geq X_j \geq X_{op_j}$, where X_{op_j} is the optimal schedule length of basic block B_j when scheduled independently. When there is resource contention at e_{j-1}, then $X_j = X_{op_j} + 1$ and Equation 3 can be written as:

$$X_{op_j} = X'_j + \triangle_{total} - 1 \tag{4}$$

In Equation 4, $\triangle_{total} - 1 \geq 1$. Thus, $X_{op_j} > X'_j$. This can not be true. Now, if there is no resource contention at e_j, then $X_j = X_{op_j}$. Equation 3 can be written as,

$$X_{op_j} = X'_j + \triangle_{total} \tag{5}$$

In Equation 5, $\triangle_{total} \geq 2$. Thus, $X_{op_j} > X'_j$. This is not possible. Hence, *case-2* is not possible.

Thus, the contradiction, $L_i > L'_i$, is not true. □

Theorem 5 Scheduling a superblock with all exit nodes as articulation nodes using Algorithm 1 will give an optimal schedule for the superblock.

Proof. By using Theorem 4. □

If $lower(cost) \neq upper(cost)$, and each exit point in a superblock is not an articulation node, we used backtracking along with constraint propagation, as used by Malik et al. [27] for their CSP model for basic block instruction scheduling, to find an optimal solution.

4 Experimentation

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

The constraint programming model was implemented and evaluated on all of the superblocks from the SPEC2000 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 [21], and the superblocks were captured as they were passed to Tobey's instruction scheduler. The superblocks 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 superblocks. 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 popular superblock scheduling heuristics: critical path heuristic [20], dependence height and speculative yield heuristic [13], the G^* heuristic [5] and the speculative hedge heuristic [10]. Table 2 shows the number of superblocks in the SPEC 2000 benchmark suite where the

optimal scheduler failed to complete within the given time limit of 10 minutes[4]. Table 2 summarizes the performance of the optimal scheduler.

Shobaki and Wilken [36] were the first to present experimental results on solving large superblocks targeted towards a multiple-issue processor. Their test suite contains the superblocks from the SPEC2000 integer and floating point benchmarks. They reported that on average 98.7% of the superblocks were scheduled optimally within one second. Also, on average they were not able to solve about 1.3% of superblocks. They also claimed that they were able to improve 80% of the hard problems (the problems that were passed to the enumerators) . Comparing with Shobaki and Wilken's work, we speculate that our test suite contains more difficult problems for the following five reasons. First, our test suite contains longer and more varied latencies. Second, our 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, our test suite contains many larger basic blocks (work [36] used the GCC compiler and the largest DAG was 1236 instructions). Fourth, our test suite contains more speculation (more instructions that can move up to higher basic blocks) as there is little speculation after register allocation.

Tables 3 and 4 systematically studies the scaling behavior of the optimal scheduler, as we report the results broken down by increasing size ranges of the superblocks. For reference, the number of superblocks in each size range is also given. It can be seen that the optimal scheduler scales well, finding improved solutions for large superblocks. Not surprisingly, as the superblock size increases, the heuristic method has more opportunities to make a mistake and the fraction of superblocks improved by the optimal scheduler increases. For the largest superblocks, up to 40.9% of the schedules are improved by the optimal scheduler (see the 4-issue architecture in Table 4). Tables 3 summarizes the percentage improvements in schedule length of the optimal schedule over the schedule found by a list scheduling algorithm using the critical-path heuristic, the G^* heuristic, the dependence height and speculative yield heuristic and the speculative hedge heuristic. Somewhat surprising is that on all size ranges the optimal scheduler can find substantial improvements. In other words, commonly used heuristic methods, sometimes find schedules that are very sub-optimal.

5 Conclusions

We presented a constraint programming approach to superblock 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 was able to routinely solve the largest superblocks with up to 2600 instructions.

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

Acknowledgements

We thank Jim McInnes from the IBM Toronto Lab for helpful discussions and assistance with gathering the superblocks used in our experiments.

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] C. Chekuri, R. Johnson, R. Motwani, B.K. Natarajan, B.R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with applications to superblocks. In *Proceeding of the 29th Annual International Symposium on Microarchitechture (MICRO-29)*, Paris, France, December, 1996.

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

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

[8] 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.

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

[10] B. L. Deitrich and W. W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 70–79, Paris, France, December, 1996.

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

[12] 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.

[13] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. In *IEEE Transactions on Computers*, pages 478-490, July 1981.

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

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

[16] 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.

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

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

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

[20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T.Kiyohara, G. E. Haab, J. G. Holm, , and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. J. Supercomputing, vol. 7, no. 1, pp. 229-248, May 1993.

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

[22] 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.

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

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

[25] 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.

[26] 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.

[27] Abid M. Malik, Jim McInnes, and Peter 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.*

[28] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of Micro-25*, Portland, Oregon, 1992.

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

[30] W. M. Meleis and A. E. Eichenberger. Balance Scheduling: Weighting Branch Tradeoffs in Superblocks. *In Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 272–283, November 1999.

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

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

[33] 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 *In Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 600–614, Kinsale, Ireland, 2003.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] S. Winkel. Optimal Global Scheduling for Itanium Processor Family. In *Proceedings of the 2nd EPIC Compiler and Architecture Workshop (EPIC-2)*, pages 59–70,November 2002.

Table 2: For the SPEC 2000 benchmark suite, (a) total time (hh:mm:ss) to schedule all superblocks in the benchmark, and (b) number of superblocks in the benchmark where the optimal scheduler did not complete within the given time limit of 10 minutes, for various issue widths. Depending on the architecture, between 99.992% – 99.999% of all superblocks are solved to optimality.

		1-issue		2-issue		4-issue		6-issue	
	#blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3,785	2:17		3:58		3:33		2:37	
applu	529	58		2:22		2:24		58	
apsi	2,321	4:55		6:40		7:05		4:15	
art	744	55		48		10		10	
bzip2	976	29		29		21		7	
crafty	5,100	10:03		10:02		6:13		6:37	
eon	5,513	5:09		8:00		4:54		4:13	
equake	386	16		11:07	1	1:07		1:51	
facerec	1,224	5:29		6:55		5:56		1:48	
fma3d	10,898	54:44	1	2:33:13	9	1:59:32	7	2:21:27	9
galgel	5,704	5:33		7:23		6:10		2:23	
gap	23,043	6:11		6:10		4:24		2:17	
gcc	49,752	18:38		18:36		10:45		7:18	
gzip	1,605	53		53		52		19	
lucas	1,098	1:04		1:20		1:41		20	
mcf	439	33		33		12		3	
mesa	14,779	10:17		8:15		6:35		5:35	
mgrid	151	20		40		16		10:09	1
parser	4,093	27		27		21		17	
perlbmk	18,607	5:11		5:11		3:20		1:59	
sixtrack	10,436	54:15		54:20		41:30		1:27:18	5
swim	383	19		21		37		4	
twolf	8,600	4:21		4:21		4:20		1:48	
vortex	13,215	14:29		14:46		6:02		5:48	
vpr	3,430	2:04		1:39		1:07		1:21	
wupwise	549	29		19		10		9	
Total	187,334	3:30:18	1	5:28:46	10	3:59:41	7	4:51:10	15

Table 3: Maximum percentage from optimal for the critical-path heuristic (h_{cp}), the dependence height and speculative yield heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the speculative hedge heuristic (h_{spec}) for ranges of super block sizes and various issue widths.

	1-issue				2-issue			
Range (#blocks)	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5 (30,371)	26.9	26.9	26.9	26.9	26.9	26.9	26.9	26.9
6–10 (46,615)	65.8	37.5	37.5	44.7	65.8	37.5	37.5	44.7
11–15 (33,687)	82.0	25.6	25.6	65.7	82.0	26.7	25.6	65.7
16–20 (23,512)	98.2	24.7	26.9	98.2	98.2	24.7	26.9	98.2
21–30 (22,858)	159.3	28.7	27.9	155.6	159.3	28.7	27.9	155.6
31–50 (17,765)	192.2	35.2	31.1	143.7	192.2	35.2	31.1	143.7
51–100 (9,481)	246.0	40.9	37.7	246.0	246.0	40.9	37.7	246.0
101–250 (2,683)	170.6	13.8	20.5	133.5	170.6	13.8	20.5	133.5
251–2,600 (362)	86.4	5.8	6.8	24.4	86.4	7.3	11.6	15.2
Total (187,334)	246.0	40.9	37.7	246.0	246.0	40.9	37.7	246.0

	4-issue				6-issue			
Range (#blocks)	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5 (30,371)	22.2	20.0	22.2	22.2	0.0	0.0	0.0	0.0
6–10 (46,615)	41.1	40.0	41.1	40.0	28.6	22.2	28.6	25.8
11–15 (33,687)	47.4	33.3	41.1	47.4	31.4	22.2	22.2	31.4
16–20 (23,512)	73.7	22.2	34.6	73.7	52.7	21.9	23.8	52.7
21–30 (22,858)	152.9	37.8	42.7	152.9	80.4	21.1	22.1	80.4
31–50 (17,765)	136.0	28.6	35.0	129.7	61.5	29.4	29.4	61.5
51–100 (9,481)	136.5	34.2	39.1	133.0	106.0	29.3	18.4	48.9
101–250 (2,683)	556.1	14.5	16.8	556.0	285.3	9.6	9.6	285.3
251–2,600 (362)	962.1	7.6	19.8	962.1	478.1	3.3	6.4	478.1
Total (187,334)	962.1	40.0	42.7	962.1	478.1	29.4	29.4	478.1

Table 4: Number of superblocks in the SPEC2000 benchmark not scheduled optimally by the critical-path heuristic (h_{cp}), the dependence height and speculative yield heuristic (h_{dhasy}), the G* heuristic (h_{g*}) and the speculative hedge heuristic (h_{spec}) for ranges of super block sizes and various issue widths.

	1-issue				2-issue			
Range (#blocks)	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5 (30,371)	143	124	129	143	126	123	128	126
6–10 (46,615)	3,095	720	2,551	1,935	3,097	717	2,510	1,938
11–15 (33,687)	4,058	1,555	3,902	2,939	4,027	1,557	3,804	2,918
16–20 (23,512)	3,363	1,469	3,835	2,639	3,353	1,479	3,786	2,626
21–30 (22,858)	4,333	1,802	5,224	3,273	4,350	1,829	5,169	3,272
31–50 (17,765)	4,168	1,954	4,966	3,299	4,214	2,015	4,985	3,322
51–100 (9,481)	2,482	1,321	2,960	2,012	2,680	1,502	3,102	2,149
101–250 (2,683)	795	491	861	676	909	599	931	756
251–2,600 (362)	131	80	131	101	152	103	148	114
Total (187,334)	22,568	9,516	24,559	17,017	22,908	9,924	24,563	17,221

	4-issue				6-issue			
Range (#blocks)	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}	h_{cp}	h_{dhasy}	h_{g*}	h_{spec}
3–5 (30,371)	6	1	6	6	0	0	0	0
6–10 (46,615)	1,009	228	1,075	978	159	115	204	120
11–15 (33,687)	1,894	788	2,045	1,662	470	255	634	398
16–20 (23,512)	1,694	759	1,905	1,504	497	306	641	394
21–30 (22,858)	2,774	1,250	3,166	2,310	912	566	1,202	747
31–50 (17,765)	2,737	1,444	3,219	2,233	1,092	773	1,470	908
51–100 (9,481)	2,051	1,385	2,404	1,740	984	787	1,228	866
101–250 (2,683)	726	603	770	680	444	400	510	419
251–2,600 (362)	129	119	142	118	79	74	80	77
Total (187,334)	13,020	6,577	14,732	11,231	4,637	3,276	5,969	3,929