An Application of Constraint Programming to Superblock Instruction Scheduling

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

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{ammalik,vanbeek}@cs.uwaterloo.ca

Abstract. Modern computer architectures have complex features that can only be fully taken advantage of if the compiler schedules the compiled code. A standard region of code for scheduling in an optimizing compiler is called a superblock. Scheduling superblocks optimally is known to be NP-complete, and production compilers use non-optimal heuristic algorithms. In this paper, we present an application of constraint programming to the superblock instruction scheduling problem. The resulting system is both optimal and fast enough to be incorporated into production compilers, and is the first optimal superblock scheduler for *realistic* architectures. In developing our optimal scheduler, the keys to scaling up to large, real problems were in applying and adapting several techniques from the literature including: implied and dominance constraints, impact-based variable ordering heuristics, singleton bounds consistency, portfolios, and structure-based decomposition techniques. We experimentally evaluated our optimal scheduler on the SPEC 2000 benchmarks, a standard benchmark suite. Depending on the architectural model, between 98.29% to 99.98% of all superblocks were solved to optimality. The scheduler was able to routinely solve the largest superblocks, including superblocks with up to 2,600 instructions, and gave noteworthy improvements over previous heuristic approaches.

1 The Problem

Modern computer architectures have complex features that can only be fully taken advantage of if the compiler schedules the compiled code. This instruction scheduling, as it is called, is one of the most important steps for improving the performance of object code produced by a compiler as it can lead to significant speedups [1]. As well, in VLIW (very large instruction word) architectures, instruction scheduling is necessary for correctness as the processor strictly follows the schedule given by the compiler (this is not true in so-called out-of-order processors). In the remainder of this section, we briefly review the necessary background in computer architecture before defining the superblock instruction scheduling problem, the problem that we address in this paper (for more background on these topics see, for example, [1,2,3]).

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 97–111, 2008.

We consider multiple-issue, pipelined processors. Multiple-issue and pipelining are two techniques for performing instructions in parallel and processors that use these techniques are now standard in desktop and laptop machines. In such processors, there are multiple functional units and multiple instructions can be issued (begin execution) in each clock cycle. Examples of functional units include arithmetic-logic units (ALUs), floating-point units, memory or load/store units that perform address computations and accesses to the memory hierarchy, and branch units that execute branch and call instructions. The number of instructions that can be issued in each clock cycle is called the *issue width* of the processor. On most architectures, including the PowerPC [4] and Intel Itanium [5], the issue width is less than the number of available functional units.

Pipelining is a standard hardware technique for overlapping the execution of instructions on a single functional unit. A helpful analogy is to a vehicle assembly line [2] where there are many steps to constructing the vehicle and each step operates in parallel with the other steps. An instruction is issued on a functional unit (begins execution on the pipeline) and associated with each instruction is a delay or *latency* between when the instruction is issued and when the instruction has completed (exits the pipeline) and the result is available for other instructions that use the result. Also associated with each instruction is an *execution time*, the number of cycles between when the instruction is issued on a functional unit and when any subsequent instruction can be issued on the same functional unit. An architecture is said to be *fully pipelined* if every instruction has an execution time of 1. However, most architectures are not fully pipelined and so there will be cycles in which instructions cannot be issued on a particular functional unit, since the unit will still be executing a previously-issued instruction.

Further, some processors, such as the PowerPC and Intel Itanium, contain *serializing instructions*, instructions that require exclusive access to the processor in the cycle in which they are issued. This can happen when an architecture has only one of a particular resource, such as a condition register, and needs to ensure that only one instruction is accessing that resource at a time. In the cycle in which such instructions are issued, no other instruction can be executing or can be issued—for that one cycle, the instruction has sole access to the processor and its resources.

Example 1. Consider a PowerPC 603e processor [4]. The processor has four functional units—an ALU, a floating-point unit, a load/store unit, and a branch unit—and an issue width of two. On this processor a floating point addition has an execution time of 1 cycle and a latency of 3 cycles. In contrast, a floating point division has an execution time of 18 cycles and also a latency of 18 cycles. Thus, once a floating-point division instruction is issued on the floating-point unit, no other floating point instruction can be issued (because there is only one unit) until 18 cycles have elapsed and no other instruction can use the result of that floating-point division until 18 cycles have elapsed. Finally, on the PowerPC 603e, about 15% of the instructions executed by the processor are serializing instructions.

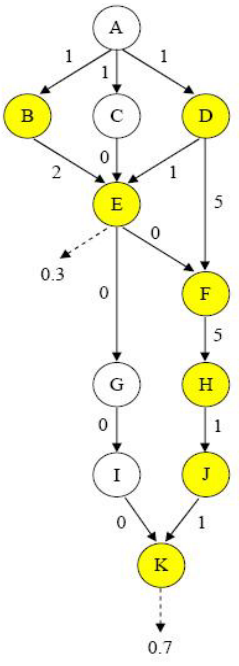
A compiler needs an accurate architectural model of the target processor that will execute the code in order to schedule the code in the best possible manner. In the rest of the paper, we refer to an architectural model as *idealized* if it assumes that (i) the issue width of the processor is equal to the number of functional units, (ii) the processor is fully pipelined, and (iii) that the processor contains no serializing instructions. An architectural model is referred to as *realistic* if it does not make any of these assumptions.

Instruction scheduling is done on certain regions of a program. All compilers schedule *basic blocks*, where a basic block is a straight-line sequence of code with a single entry point and a single exit point. However, basic blocks alone are considered insufficient for fully utilizing a processor's resources and most optimizing compilers also schedule a generalization of basic blocks called *superblocks*. A superblock is a collection of basic blocks with a unique entrance but multiple exit points [6]. We use the standard labeled directed acyclic graph (DAG) representation of a superblock. 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. Each node or instruction i has an associated execution time $d(i)$. *Exit nodes* are special nodes in a DAG representing the branch instructions. Each exit node i has an associated weight or exit probability $w(i)$ that represents the probability that the flow of control will leave the superblock through this exit point. The probabilities are calculated by running the instructions on representative data, a process known as *profiling*.

Given a labeled dependency DAG for a superblock and a target architectural model, a *schedule* for a superblock is an assignment of a clock cycle to each instruction such that the latency and resource constraints are satisfied. The resource constraints are satisfied if, at every time cycle, the resources needed by all the instructions issued or executing at that cycle do not exceed the limits of the processor.

Definition 1 (Superblock Instruction Scheduling). *The* weighted completion time *or cost of a superblock schedule is* $\sum_{i=1}^{n} w(i)e(i)$*, where* n *is the number of exit nodes,* $w(i)$ *is the weight of exit* i*, and* $e(i)$ *is the clock cycle in which exit* i *will be issued in the schedule. The* superblock instruction scheduling problem *is to construct a schedule with minimum weighted completion time.*

Example 2. Consider the superblock shown in Figure 1. Nodes E and K are branch instructions, with exit probability 0.3 and 0.7, respectively. Consider an idealized processor with two functional units. One functional unit can execute clear instructions and the other can execute shaded instructions. Figure 1(b) shows two possible schedules, S_1 and S_2. The weighted completion time for schedule S_1 is $0.3 \times 4 + 0.7 \times 15 = 11.7$ cycles and for schedule S_2 is $0.3 \times 5 + 0.7 \times 14 = 11.3$ cycles. Schedule S_2 is an *optimal* solution.



(a)

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)

Fig. 1. (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 2

2 Why CP?

Superblock instruction scheduling for realistic multiple-issue processors is NP-complete [7] and currently is done using heuristic approaches in all commercial and open-source research compilers. The most common heuristic approach is a greedy list scheduling algorithm coupled with a priority heuristic. Many sophisticated heuristics have been proposed including critical path [3], dependence height and speculative yield [8], G^* [9], speculative hedge [10], balance scheduling [11], and successive retirement [9]. However, even the best heuristic approaches can produce sub-optimal solutions.

Optimal approaches for instruction scheduling have also been proposed. We first review previous work on basic block scheduling, the easier special case of superblock scheduling where there is only one exit and all of the instructions in the block are always executed. Previous work on optimal approaches to basic block instruction scheduling can be categorized by those approaches that are targeted only towards idealized—i.e., unrealistic—architectural models [12,13,14,15] and those approaches that have been developed for more realistic architectural models [16,17,18]. Broadly speaking, previous work has shown that (i) for an *idealized* multi-issue processor, optimal approaches can scale up to the largest basic blocks

that arise in practice, and (ii) for more realistic architectures, optimal approaches can be used but do not yet scale up beyond 10-40 instructions (the largest blocks that arise in practice have 2,600 instructions). In our work, we present a constraint programming approach that applies to *realistic* architectures and scales up to the largest blocks. Our work builds on a previously developed constraint programming approach for basic block scheduling, which assumed an idealized architecture [15].

In contrast to optimal basic block scheduling, there has been relatively little work on optimal superblock scheduling. Winkel [19] presents an integer linear programming model for instruction scheduling for Itanium processors. However, the approach has two limitations. First, the model is limited to small regions with size up to 200 instructions. Second, and more importantly, the approach minimizes the *length* of the schedule. This measure is appropriate for basic blocks, which consist of straight line code. But it is not appropriate for regions that contain multiple exits and whose paths of execution may rarely fall through to the last instruction. Shobaki and Wilken [20,21] were the first to develop a robust optimal scheduler for superblocks that scaled up to large superblocks. Their approach is based on enumeration. However, their work is targeted to idealized architectures and assumes that the functional units are fully pipelined, the issue width of the processor is equal to the number of functional units, and there are no serializing instructions. It is not at all clear how to successfully extend these previously proposed enumeration and integer programming approaches to realistic architectures and cost functions. In our constraint programming approach, we remove these assumptions and present the first optimal superblock scheduling approach for realistic architectures. Further, even though our target architectures are realistic, our approach scales up to more difficult and larger superblocks than in previous work.

3 How CP?

In this section, we present our constraint programming approach for superblock instruction scheduling. We first present the basic model—a model that is correct but inefficient—followed by the techniques we used to improve our model and solving approach. Our description is at a high-level; see [22] for more details.

3.1 Basic Model

Given a labeled dependency DAG $G = (N, E)$ for a superblock and a target architectural model, we model each instruction or node i by a variable x_i. The domain of each variable $dom(x_i)$ is a subset of $\{1, \ldots, m\}$, which are the available time cycles. Assigning a value $t \in dom(x_i)$ to a variable x_i has the intended meaning that instruction i will be issued at time cycle t. The domain $dom(x_i) = \{a, \ldots, b\}$ of a variable x_i is represented by the endpoints of the interval $[a, b]$.

To model the latencies of the instructions, for each pair of variables x_i and x_j such that $(i, j) \in E$, a latency constraint of the form $x_i + l(i, j) \leq x_j$ is

added to the constraint model, where $l(i,j)$ is the latency on the edge (i,j). Global cardinality constraints (GCC) [23] are used to model the resources of the processor. A GCC over a set of variables and values states that the number of variables instantiating to a value must be between a given upper and lower bound. For each type t of functional unit, a GCC over all variables of type t is added to the constraint model, where the lower bound is zero and the upper bound is the number of functional units of type t. As well, a GCC over all variables is added, where the lower bound is zero and the upper bound is the issue width of the processor.

So far, the model assumes an idealized architecture where each unit is fully pipelined and there are no serializing instructions. To model a non-fully pipelined processor, we add auxiliary variables to the constraint model. Recall that in a non-fully pipelined processor, some instructions have execution times greater than 1. Let i be an instruction with execution time $e(i) > 1$ and let x_i be the corresponding variable. The auxiliary variables $p_{i,j}$, $1 \leq j \leq e(i) - 1$, are added into the model, where each variable $p_{i,j}$ is of the same functional unit type as x_i. The constraints $x_i + j = p_{i,j}$, $1 \leq j \leq e(i) - 1$, are also added into the model. Finally, we also add the variables $p_{i,j}$, all of which are of type t, to the GCC functional unit constraint for type t.

Serializing instructions can be modeled in a manner similar. Let i be a serializing instruction and let x_i be the corresponding variable. Let F be the total number of functional units in the processor. The auxiliary variables $s_{i,j}$, $1 \leq j \leq F - 1$, are added into the constraint model. There is one auxiliary variable for every functional unit except for the one on which instruction i is issued; the functional unit type of each auxiliary variable is assigned accordingly. The constraints $x_i = s_{i,j}$, $1 \leq j \leq F - 1$, are also added into the model. Finally, for each type t, we add all auxiliary variables of type t to the corresponding GCC functional unit constraint for type t.

Example 3. Consider again the superblock shown in Figure 1 and assume initially the same idealized processor as in Example 2. The constraint model would have variables $\mathrm{A}, \ldots, \mathrm{K}$, and the constraints,

$$\begin{array}{lll} \mathrm{B} \geq \mathrm{A} + 1, & \ldots & \mathrm{GCC(B,\ D,\ E,\ F,\ H,\ J,\ K)}, \\ \mathrm{C} \geq \mathrm{A} + 1, & \mathrm{K} \geq \mathrm{I}, & \mathrm{GCC(A,\ C,\ G,\ I)}, \\ \mathrm{D} \geq \mathrm{A} + 1, & \mathrm{K} \geq \mathrm{J} + 1, & \end{array}$$

where the lower and upper bounds of each GCC constraint are 0 and 1 (the number of functional units of each type), respectively, and the cost function is $0.3 \times \mathrm{E} + 0.7 \times \mathrm{K}$. Somewhat more realistically, suppose instead that instruction D is not fully pipelined and has an execution time $e(\mathrm{D}) = 3$ and that instruction G is a serializing instruction. The auxiliary variables $p_{\mathrm{D},1}$, $p_{\mathrm{D},2}$, and $s_{\mathrm{G},1}$ would be added to the model along with the constraints $\mathrm{D}+1 = p_{\mathrm{D},1}$, $\mathrm{D}+2 = p_{\mathrm{D},2}$, and $\mathrm{G} = s_{\mathrm{G},1}$. Finally, one of the GCC constraints would incorporate the auxiliary variables and would become GCC(B, D, E, F, H, J, K, $p_{\mathrm{D},1}$, $p_{\mathrm{D},2}$, $s_{\mathrm{G},1}$).

We have described a correct, but minimal, model for the superblock scheduling problem targeted towards realistic architectures. As is usual in constraint

programming, the minimal model cannot solve all but the smallest instances as it does not scale beyond 40 instructions. We next describe the improvements we made to scale up our constraint programming approach to instances with 2,600 instructions (the largest that we have found in practice).

3.2 Improving the Model and Solving Approach

In developing our optimal scheduler, the keys to scaling up to large, real problems were in applying and adapting several techniques from the literature including: implied and dominance constraints, impact-based variable ordering heuristics, singleton bounds consistency, portfolios, and structure-based decomposition techniques.

Implied constraints do not change the set of solutions while dominance constraints may but preserve an optimal solution. Both types of constraints can increase the amount of constraint propagation and so greatly improve the efficiency of the search for a solution (see, e.g., [24] and references therein). In our work, many instances of each of these constraints are added to the constraint model in an extensive preprocessing stage that occurs once. The extensive preprocessing effort pays off as the model is solved many times.

Two forms of implied constraints are added to the model: $x_i + d(i,j) \leq x_j$ and $x_j \leq x_i + d(i,j)$. Roughly, the first form is added if a pair of nodes i and j in the DAG for a superblock form a region; i.e., there is more than one path from i to j [12]. If the region is small enough, it is solved exactly using a backtracking algorithm; if it is large, the distance $d(i,j)$ is estimated, making sure that the estimate is a lower bound. Again roughly, the second form is added if i and j define a region and are articulation nodes—an articulation node is a node which disconnects the graph once removed—and the region defined by i and j is small enough to be solved quickly and exactly in isolation. It can be shown that the solution to the isolated subproblem can be used to form a tight upper bound on the distance between i and j in any optimal schedule.

Heffernan and Wilken [14] present a set of graph transformations for dependency DAGs for basic blocks and show that optimally scheduling the transformed DAGs using branch-and-bound enumeration is faster and more robust. We adapted these transformations to superblock scheduling and proved under what conditions they preserve optimality. In our context, the transformations add simple dominance constraints to the model of the form $x_i \geq x_j$. Adding dominance constraints requires identifying pairs of disjoint, isomorphic subgraphs A and B in a dependency DAG for a superblock. 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). We use a fast heuristic approach to find pairs of disjoint, isomorphic subgraphs adapted from our work on basic block scheduling [15].

Example 4. Consider the DAG shown in Figure 2(a). Nodes H and I are called speculative nodes in the compiler literature as they can be moved across exit node G. The subgraphs with nodes {C, E} and {H, I} are isomorphic and satisfy

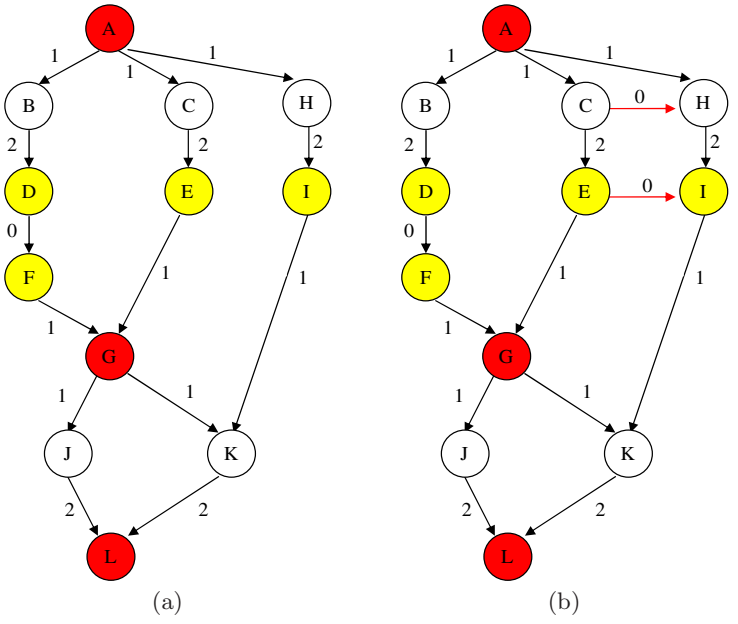


Fig. 2. Example of adding dominance constraints in a superblock: (a) actual DAG; (b) the constraints C ≤ H and E ≤ I (zero latency edges) would be added to the constraint model. Nodes A, G and L are exit nodes.

the conditions for adding dominance constraints. Hence, the constraints C ≤ H and E ≤ I can be added to the model. Figure 2(b) shows the DAG with the added constraints. Note that the added constraints do not change the speculative characteristic of exit node G, as nodes H and I still can be moved across—i.e., can be scheduled either before or after—node G. Similarly, the constraint J ≤ K can be added.

Once the constraint model has been extensively preprocessed by adding implied and dominance constraints, it is ready to be solved. Recall that the superblock scheduling problem is an optimization problem. To turn it into a satisfaction problem, we first establish an upper bound on the cost function using a fast heuristic scheduling method (a list scheduling algorithm, as discussed in the Experimental evaluation section). Given an upper bound on the cost function, we then prune the cost variables using singleton bounds consistency and enumerate the possible solutions to the cost function using techniques adapted from [25]. The solutions to the cost function are then stepped through in increasing order of cost until one is found that can be extended to a solution to the entire constraint model. Testing whether a solution to the cost function can be extended is done using a backtracking search algorithm. Of course, once a solution to the entire constraint model is found it is a provably optimal solution.

To reduce the brittleness or variability in performance of our backtracking search algorithm, we use a portfolio approach. Portfolios of multiple algorithms

have been proposed and shown to dramatically improve performance on some instances (see, e.g., [26]). In instruction scheduling, thousands of superblocks arise each time a compiler is invoked on some software project and a limit needs to be placed on the time given for solving each instance in order to keep the total compile time to an acceptable level. Given a set of possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ and a time deadline d, a *portfolio* P for a single processor is a sequence of pairs, $P = [(\mathcal{A}_{k_1}, t_1), (\mathcal{A}_{k_2}, t_2), \dots, (\mathcal{A}_{k_m}, t_m)]$, where each $\mathcal{A}_{k_i}$ is a backtracking algorithm, each t_i is a positive integer, and $\Sigma_{i=1}^{m} t_i = d$. To apply a portfolio to an instance, algorithm $\mathcal{A}_{k_1}$ is run for t_1 steps. If no solution is found within t_1 steps, algorithm $\mathcal{A}_{k_1}$ is terminated and algorithm $\mathcal{A}_{k_2}$ is run for t_2 steps, and so on until either a solution is found or the sequence is exhausted as the time deadline d has been reached.

In contrast to previous work, where the differences in the possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ often involves the variable ordering heuristic, we created variability in solving performance by increasing levels of constraint propagation from light-weight to heavy-weight. For our approach, we used a deterministic backtracking algorithm capable of performing three levels of constraint propagation,

Level = 1	bounds consistency,
Level = 2	singleton bounds consistency, and
Level = 3	singleton bounds consistency to a depth of two.

and the portfolio involved three phases in increasing order. We chose bounds consistency—instead of the more usual arc consistency—as in our problem it is equivalent but more efficient. In bounds consistency, one ensures that each upper and lower bound of the domain of a variable is consistent with each constraint (see, e.g., [27] and references therein). In singleton bounds consistency, one temporarily assigns a value to a variable and then performs bounds consistency. In singleton bounds consistency to a depth of two, one temporarily assigns a value to a variable and then performs singleton consistency. In each, if the value is found to be inconsistent it is not part of any solution and can be removed from the domain of the variable.

During phase one, a standard dynamic variable ordering heuristic based on minimum domain size is used. However, in the next two phases which involve singleton consistency a variation of an impact-based variable ordering heuristic [28] is used. The idea in impact-based heuristics is to measure the importance of a variable for reducing the search space. Here, we record the number of changes that are made due to each variable during the singleton consistency propagation. This information is then used to select the next variable to branch on with the goal being to branch on a variable that causes the most reductions in the domains of the other variables. The impact-based heuristic is very effective and essentially comes for free as a side-effect of enforcing singleton consistency.

As a final technique for scaling up our constraint programming approach to the largest instances, we also adapted a structure-based decomposition technique [29]. For some of the largest superblock instances, all of the exit nodes were articulation nodes. We showed that such instances could be solved optimally by

solving them progressively. Let $e_1, \ldots, e_n$ be the exit nodes. We first solve the subproblem consisting of e_1 and all of its predecessor nodes. Variable e_1 is then fixed using the optimal solution to the subproblem and we then in turn solve the subproblem consisting of e_2 and all of its predecessor nodes, and so on. The proof that this procedure preserves optimality requires careful attention to the resource contention at each exit node.

3.3 Experimental Evaluation

The constraint programming model was implemented and evaluated on all of the 154,651 superblocks from the SPEC 2000 integer and floating point benchmarks (www.spec.org). This benchmark suite consists of source code for software packages that are chosen to be representative of a variety of programming languages and types of applications. The benchmarks were compiled using IBM's Tobey compiler [30] targeted towards the IBM PowerPC processor [4], and the superblocks were captured as they were passed to Tobey's instruction scheduler. The Tobey compiler performs instruction scheduling before register allocation and once again afterward, 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 realistic architectural models we used in our evaluation. In these architectures, the functional units are not fully pipelined, the issue width of the processor is not equal to the number of functional units, and there are serializing instructions.

architecture	issue width	simple int. units	complex int. units	memory units	branch units	floating pt. units
1r-issue	1	1				
PowerPC 603e (ppc603e)	2	1		1	1	1
PowerPC 604 (ppc604)	4	2	1	1	1	1
6r-issue	6	2		2	3	2

The following table shows the total time (hh:mm:ss) to schedule all super blocks in the SPEC 2000 benchmark suite and the percentage of superblocks that were solved to optimality, for various realistic architectural models and time limits for solving each superblock.

	1 sec.		10 sec.		1 min.		10 min.	
	time	%	time	%	time	%	time	%
1r-issue	1:30:20	97.34	7:15:46	99.38	10:22:36	99.96	15:08:44	99.98
ppc603e	3:57:13	91.83	30:53:56	93.90	108:50:01	97.18	665:31:00	97.70
ppc604	2:17:44	95.47	17:09:48	96.60	61:29:31	98.43	343:04:46	98.87
6r-issue	3:04:18	93.59	25:03:44	94.76	87:04:34	97.78	511:19:14	98.29

Table 1. *Superblock scheduling after register allocation.* For the SPEC 2000 benchmark suite, number of cycles saved ($\times 10^9$) by the optimal scheduler over a list scheduler using the dependence height and speculative yield heuristic and using the critical path heuristic, and the percentage reduction (%), for various realistic architectural models. The time limit for solving each superblock was 10 minutes.

	DHASY heuristic								critical path heuristic							
	1r-issue		ppc603e		ppc604		6r-issue		1r-issue		ppc603e		ppc604			
benchmark	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%		
ammp	47.4	0.2	669.3	3.2	225.9	1.1	221.2	1.3	457.5	1.8	949.9	4.5	243.9	1.2		
applu	5.2	0.4	0.3	0.0	0.6	0.1	0.1	0.0	23.7	1.9	4.1	0.4	0.5	0.0		
apsi	52.4	1.1	43.6	1.0	45.8	1.1	31.8	1.0	341.5	7.1	89.0	2.0	84.9	2.0		
art	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.7	0.1	1.1	0.0	1.1	0.0		
bzip2	51.8	0.3	282.1	1.8	281.6	1.9	137.5	0.9	300.1	1.5	405.0	2.5	356.7	2.3		
crafty	50.9	0.7	67.8	1.2	30.1	0.6	32.0	0.6	162.1	2.3	119.6	2.1	53.9	1.0		
eon	303.1	2.7	65.7	0.7	47.1	0.5	124.7	1.5	610.6	5.5	127.0	1.3	150.1	1.6		
equake	22.4	0.5	12.1	0.3	11.8	0.3	0.1	0.0	20.6	0.5	1.5	0.0	1.1	0.0		
facerec	19.4	0.3	27.0	0.5	3.6	0.1	1.6	0.0	28.7	0.5	32.9	0.7	3.6	0.1		
fma3d	36.4	0.4	48.4	0.6	86.2	1.1	18.6	0.3	51.8	0.5	49.2	0.6	73.4	1.0		
galgel	1.6	0.1	0.7	0.1	0.5	0.1	0.0	0.0	4.9	0.5	2.2	0.2	1.3	0.1		
gap	18.9	0.0	67.6	0.0	43.2	0.0	31.0	0.0	99.9	0.0	69.3	0.0	38.8	0.0		
gcc	28.7	0.6	33.4	0.8	18.8	0.5	16.6	0.4	65.9	1.3	51.2	1.2	26.3	0.6		
gzip	11.2	0.1	36.8	0.3	22.1	0.2	29.9	0.2	158.1	1.0	50.6	0.4	22.2	0.2		
lucas	0.0	0.0	0.2	0.1	0.0	0.0	0.0	0.0	4.5	1.2	0.5	0.2	0.0	0.0		
mcf	54.1	1.5	43.9	1.4	38.9	1.2	0.0	0.0	89.0	2.5	93.7	2.9	94.9	3.0		
mesa	34.0	0.2	53.3	0.4	31.0	0.3	306.5	3.3	85.3	0.6	32.2	0.3	31.0	0.3		
mgrid	1.7	0.5	0.3	0.1	0.0	0.0	0.0	0.0	3.1	0.9	0.5	0.2	0.0	0.0		
parser	483.1	1.9	530.8	2.6	507.9	2.6	277.5	1.5	956.8	3.8	808.8	3.9	526.0	2.7		
perlbmk	67.8	0.2	386.4	1.5	117.6	0.5	76.6	0.3	181.7	0.6	439.6	1.7	141.3	0.6		
sixtrack	122.6	3.5	6.3	0.2	4.0	0.1	1.5	0.0	655.4	18.6	19.9	0.6	5.0	0.1		
swim	0.0	0.2	0.1	1.7	0.1	1.9	0.0	0.0	8.5	99.8	6.6	102.3	3.2	58.1		
twolf	288.5	1.5	43.9	0.3	88.6	0.6	69.7	0.5	689.0	3.5	378.3	2.3	198.1	1.3		
vortex	41.7	0.4	252.9	3.1	274.6	3.7	220.3	3.3	212.3	2.0	310.6	3.9	306.5	4.1		
vpr	57.6	0.5	26.8	0.3	41.1	0.5	6.1	0.1	227.1	2.1	64.4	0.7	38.2	0.4		
wupwise	83.5	1.0	30.8	0.4	20.8	0.3	22.6	0.4	523.6	6.3	211.5	2.9	69.3	1.0		

We also evaluated our optimal scheduler with respect to how much it improves on previous heuristic approaches. Most production compilers use a greedy list scheduling algorithm coupled with a heuristic priority function for scheduling. Here we compare against a list scheduler with a realistic resource model using the dependence height and speculative yield (DHASY) heuristic [8] and a critical path heuristic [3]. We chose the former heuristic as it is considered one of the best available (it is the default heuristic used in the Trimaran compiler [31], for example) and the latter heuristic because it is a standard reference point.

Table 1 gives the number of cycles saved ($\times 10^9$) by the optimal scheduler over the list scheduler using the dependence height and speculative yield heuristic and using the critical path heuristic, and the percentage reduction (%), for various

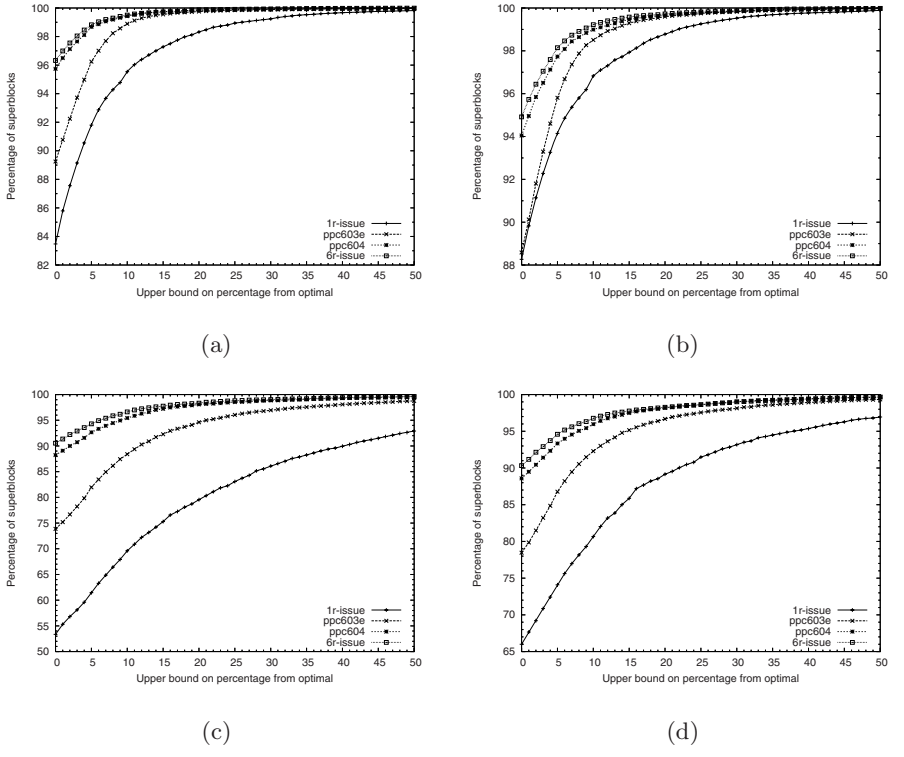


Fig. 3. Performance of optimal scheduler versus list scheduler, for various realistic architectures: (a) before register allocation using DHASY heuristic; (b) after register allocation using DHASY heuristic; (c) before register allocation using critical path heuristic; and (d) after register allocation using critical path heuristic

realistic architectural models after register allocation. We compiled the SPEC 2000 benchmark with the training data set associated with the benchmark using the Tobey compiler. The compiler uses the training data to construct a profile for each branch instruction. The profile is used to calculate the information regarding the number of times each instruction is executed.

Figures 3(a–d) summarizes the performance of the optimal scheduler versus the list scheduler, for various architectures. For example, consider the 1r-issue architecture and the DHASY heuristic. The list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for approximately 84% of all superblocks before register allocation and approximately 88% of all superblocks after register allocation. In other words, the optimal scheduler improves on 16% and 12% of superblocks, respectively. Further, the list scheduler is within 10% of optimal for approximately 95% of all superblocks before register allocation and approximately 97% of all superblocks after register allocation, for this architecture. The graph also shows that, although quite rare, there exists superblocks for which the optimal scheduler finds improvements of up to 50% over the DHASY heuristic. As a second example, consider the 1r-issue architecture and the critical path

heuristic. The list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for approximately 54% of all superblocks before register allocation and approximately 65% of all superblocks after register allocation. Further, the list scheduler is within 10% of optimal for approximately 70% of all superblocks before register allocation and approximately 80% of all superblocks after register allocation, for this architecture.

4 Added Value of CP?

Using constraint programming brought added value in two ways.

The first value added by constraint programming is that it allowed us to achieve the primary goal of our project, which was to develop a superblock instruction scheduler that was realistic yet fast enough to be incorporated into a production compiler. Using constraint programming, it was relatively easy to add additional constraints to model realistic architectures and it is not clear how to similarly extend previously proposed enumeration and integer programming approaches. As well, we had previously shown that constraint programming could be much faster than integer programming on a restricted form of these types of problems [13]. But perhaps the most important reason we were able to achieve our goal is that constraint programming allows and facilitates *programming* in the computer science sense of the word. This was crucial to scaling up to large instances, as it allowed us to design and implement domain-specific structure-based decomposition techniques and to incorporate and fine-tune ideas such as portfolios and impact-based variable ordering heuristics into our solver.

The second value added by constraint programming is that it allowed us to find optimal solutions. Although heuristic approaches have the advantage that they are very fast, a scheduler that finds optimal schedules can be useful in practice when longer compiling times are tolerable such as when compiling for software libraries, digital signal processing or embedded applications [1]. As well, an optimal scheduler can be used to evaluate the performance of heuristic approaches. Such an evaluation can tell whether there is a room for improvement in a heuristic or not.

5 Conclusions

We presented a constraint programming approach to superblock instruction scheduling for realistic architectural models. Our approach is optimal and robust on large, real instances. The keys to scaling up to large, real problems were in applying and adapting several techniques from the literature including: implied and dominance constraints, impact-based variable ordering heuristics, singleton bounds consistency, portfolios, and structure-based decomposition techniques. We experimentally evaluated our optimal scheduler on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler scaled to the largest superblocks. Depending on the architectural model, between 98.23% to 99.98% of all superblocks were solved to optimality. The scheduler

was able to routinely solve the largest superblocks, including blocks with up to 2,600 instructions. The schedules produced by the optimal schedule showed an improvement of 0%-3.8% on average over a list scheduler using the dependence height and speculative yield heuristic, considered one of the best heuristics available, and an improvement of 0%-102% on average over a critical path heuristic. One final conclusion we draw from our work is that constraint programming can be a fruitful approach for solving NP-hard compiler optimization problems.

Acknowledgments

This research was supported by an IBM Center for Advanced Studies (CAS) Fellowship, an NSERC Postgraduate Scholarship, and an NSERC CRD Grant.

References

1. Govindarajan, R.: Instruction scheduling. In: Srikant, Y.N., Shankar, P. (eds.) The Compiler Design Handbook, pp. 631–687. CRC Press, Boca Raton (2003)
2. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann, San Francisco (2003)
3. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
4. Hoxey, S., Karim, F., Hay, B., Warren, H.: The PowerPC Compiler Writer's Guide. Warthman Associates (1996)
5. Intel: Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture (2002)
6. Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: An effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7(1), 229–248 (1993)
7. Hennessy, J., Gross, T.: Postpass code optimization of pipeline constraints. ACM Transactions on Programming Languages and Systems 5(3), 422–448 (1983)
8. Bringmann, R.A.: Enhancing Instruction Level Parallelism through Compiler-Controlled Speculation. PhD thesis, U. of Illinois at Urbana-Champaign (1995)
9. Chekuri, C., Johnson, R., Motwani, R., Natarajan, B., Rau, B.R., Schlansker, M.: Profile-driven instruction level parallel scheduling with application to superblocks. In: Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-29), Paris, pp. 58–67 (1996)
10. Deitrich, B., Hwu, W.: Speculative hedge: Regulating compile-time speculation against profile variations. In: Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-29), Paris (1996)
11. Eichenberger, A.E., Meleis, W.M.: Balance scheduling: Weighting branch tradeoffs in superblocks. In: Proc. of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-32), Haifa, Israel (1999)
12. Wilken, K., Liu, J., Heffernan, M.: Optimal instruction scheduling using integer programming. In: Proc. of the SIGPLAN 2000 Conference on Programming Language Design and Implementation, Vancouver, pp. 121–133 (2000)
13. van Beek, P., Wilken, K.: Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In: Proc. of the 7th Int'l Conf. on Principles and Practice of Constraint Programming, Paphos, Cyprus, pp. 625–639 (2001)

14. Heffernan, M., Wilken, K.: Data-dependency graph transformations for instruction scheduling. Journal of Scheduling 8, 427–451 (2005)
15. Malik, A.M., McInnes, J., van Beek, P.: Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. In: Proc. of the 18th IEEE Int'l Conf. on Tools with AI, Washington, DC, pp. 279–287 (2006)
16. Ertl, M.A., Krall, A.: Optimal instruction scheduling using constraint logic programming. In: Proc. of 3rd International Symposium on Programming Language Implementation and Logic Programming, Passau, Germany, pp. 75–86 (1991)
17. Kästner, D., Winkel, S.: ILP-based instruction scheduling for IA-64. In: Proc. of the SIGPLAN 2001 Workshop on Languages Compilers, and Tools for Embedded Systems, Snowbird, Utah, pp. 145–154 (2001)
18. Liu, J., Chow, F.: A near-optimal instruction scheduler for a tightly constrained, variable instruction set embedded processor. In: Proc. of the Int'l Conf. on Compilers, Architectures, and Synthesis for Embedded Systems, Grenoble, pp. 9–18 (2002)
19. Winkel, S.: Exploring the performance potential of Itanium processors with ILP-based scheduling. In: 2nd IEEE/ACM International Symposium on Code Generation and Optimization, pp. 189–200 (2004)
20. Shobaki, G., Wilken, K.: Optimal superblock scheduling using enumeration. In: Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-37), Portland, Oregon, pp. 283–293 (2004)
21. Shobaki, G.: Optimal Global Instruction Scheduling Using Enumeration. PhD thesis, University of California, Davis (2006)
22. Malik, A.M.: Constraint Programming Techniques for Optimal Instruction Scheduling. PhD thesis, University of Waterloo (2008)
23. Régin, J.C.: Generalized arc consistency for global cardinality constraint. In: Proc. of the 13th National Conference on AI, Portland, Oregon, pp. 209–215 (1996)
24. Smith, B.M.: Modelling. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
25. Trick, M.: A dynamic programming approach for consistency and propagation of knapsack constraints. In: Proc. of Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (2001)
26. Gomes, C., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Proc. of the 3rd Int'l Conf. on Principles and Practice of Constraint Programming, Linz, Austria, pp. 121–135 (1997)
27. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
28. Refalo, P.: Impact-based search strategies for constraint programming. In: Proc. of the 10th Int'l Conf. on Principles and Practice of Constraint Programming, Toronto, pp. 557–571 (2004)
29. Freuder, E.C.: Exploiting structure in constraint satisfaction problems. In: Mayoh, B., Tyugo, E., Penjam, J. (eds.) Constraint Programming. Springer, Heidelberg (1994)
30. Blainey, R.J.: Instruction scheduling in the TOBEY compiler. IBM J. Res. Develop. 38(5), 577–593 (1994)
31. Chakrapani, L.N., Gyllenhaal, J., Hwu, W.W., Mahlke, S.A., Palem, K.V., Rabbah, R.M.: Trimaran: An infrastructure for research in instruction-level parallelism. In: Proc. of the 17th International Workshop on Languages and Compilers for High Performance Computing, West Lafayette, Indiana, USA, pp. 32–41 (2005)