Optimal Trace Scheduling Using Enumeration

GHASSAN SHOBAKI

University of California, Davis and Sun Microsystems

and

KENT WILKEN and MARK HEFFERNAN

University of California, Davis

This article presents the first optimal algorithm for trace scheduling. The trace is a global scheduling region used by compilers to exploit instruction-level parallelism across basic block boundaries. Several heuristic techniques have been proposed for trace scheduling, but the precision of these techniques has not been studied relative to optimality. This article describes a technique for finding provably optimal trace schedules, where optimality is defined in terms of a weighted sum of schedule lengths across all code paths in a trace. The optimal algorithm uses branch-and-bound enumeration to efficiently explore the entire solution space. Experimental evaluation of the algorithm shows that, with a time limit of 1 s per problem, 91% of the hard trace scheduling problems in the SPEC CPU 2006 Integer Benchmarks are solved optimally. For 58% of these hard problems, the optimal schedule is improved compared to that produced by a heuristic scheduler with a geometric mean improvement of 3.2% in weighted schedule length and 18% in compensation code size.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Code generation, compilers, optimization*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Non-numerical Algorithms and Problems—*Sequencing and scheduling*; G.2.1 [Discrete Mathematics]: Combinatorics—*Combinatorial algorithms*

General Terms: Algorithms, Performance, Design, Experimentation, Languages

Additional Key Words and Phrases: Instruction scheduling, global instruction scheduling, trace scheduling, optimal instruction scheduling, instruction-level parallelism, compiler optimizations, branch-and-bound enumeration

ACM Reference Format:

Shobaki, G., Wilken, K., and Heffernan, M. 2009. Optimal trace scheduling using enumeration. *ACM Trans. Architect. Code Optim.* 5, 4, Article 19 (March 2009), 32 pages. DOI = 10.1145/1498690.1498694 <http://doi.acm.org/10.1145/1498690.1498694>

This research was supported in part by a University of California dissertation-year fellowship for the first author during the academic year 2005–2006.

G. Shobaki is now affiliated with Advanced Micro Devices (AMD) and M. Heffernan is now affiliated with Google.

Author's addresses: G. Shobaki, Department of Computer Science, University of California, Davis and Sun Microsystems; email: gshobaki@ece.ucdavis.edu; K. Wilken and M. Heffernan, Department of Electrical and Computer Engineering, University of California, Davis; email: wilken@ece.ucdavis.edu.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2009 ACM 1544-3566/2009/03-ART19 \$5.00 DOI 10.1145/1498690.1498694 <http://doi.acm.org/10.1145/1498690.1498694>

ACM Transactions on Architecture and Code Optimization, Vol. 5, No. 4, Article 19, Publication date: March 2009.

1. INTRODUCTION

Many region shapes have been proposed for exploiting instruction-level parallelism (ILP) beyond basic block boundaries (global instruction scheduling). Common examples are traces [Fisher 1981], superblocks [Hwu et al. 1993], hyperblocks [Mahlke et al. 1992], treeregions [Havanki et al. 1998], and general acyclic regions [Bharadwaj et al. 2000]. Trace scheduling is one of the earliest and most widely used global scheduling regions.

A trace is a sequence of contiguous basic blocks in the program's control-flow graph (CFG) [Muchnick 1997]. A basic block in a trace with a CFG predecessor outside the trace is called an *entry block* and the first instruction in that block is called an *entrance* (also known as a *join*). A basic block in a trace with a CFG successor outside of the trace is called an *exit block* and the last instruction in that block is called an *exit* (also known as a *split*). A trace may have multiple entrances and multiple exits. A *code path* in a trace is a sequence of basic blocks starting at an entrance and ending at an exit below the entrance. The code path that starts at the first entrance and ends at the last exit in the trace (thus including the entire trace) is called the *main path*. All other paths are called *side paths*. The *weight* of a code path is the probability that the path is executed, given that the trace is reached. Path weights in a trace must then sum to unity. Figure 1(a) shows an example trace with three basic blocks, two entrances, and three exits. The four code paths in the trace and their weights are listed in Figure 1(c). P_3 is the main path and all other paths are side paths. The optimal formulation of this article does not assume that the main path is necessarily the most likely path.

The advantage of scheduling a whole trace at once instead of scheduling each basic block individually is that ILP may be exploited across basic block boundaries to minimize the schedule length of the main path. However, code motion across basic blocks has two undesirable side effects. The first side effect is that code may have to be duplicated in certain cases to preserve program semantics. For example, if Instruction F is moved from BB_3 to BB_2 crossing the second entrance in the trace of Figure 1(a), a duplicate of Instruction F must be added before entry to BB_3 to preserve correctness. The duplicate code is called *compensation code*. It is usually desirable and sometimes necessary to limit the amount of compensation code. The second side effect is that code motion across blocks may cause schedules of some side paths to be unnecessarily long. For example, if Instruction F is moved from BB_3 to BB_1 in the trace of Figure 1(a), side paths P_1 and P_2 will execute an extra instruction.

Since the introduction of trace scheduling, several attempts have been made to address excessive compensation code and unnecessary degradation of side paths [Freudenberger et al. 1994; Lah and Atkin 1983; Linn 1983; Smith et al. 1992; Su et al. 1984]. Most previous work on solving the compensation-code and side-path problems focuses on disabling certain global code motions, thus limiting the benefit of trace scheduling. The optimal solution proposed in this article is based on a cost function that formalizes the three conflicting objectives addressed by previous work: minimizing the main-path schedule length, avoiding side-path degradation, and limiting compensation code.

Fig. 1. Example trace and its data dependence graph (DDG).

By exhaustively searching the solution space, a schedule that minimizes the weighted sum of schedule lengths across all paths is selected. This approach offers a solution to the side-path and compensation-code problems that does not unnecessarily restrict global code motion.

The exhaustive search is performed by a branch-and-bound enumerator that uses pruning techniques from previous work [Narasimhan and Ramanujam 2001; Shobaki and Wilken 2004] as well as some new pruning techniques that have been developed specifically for trace scheduling.

Another advantage of this enumerative approach over existing heuristic approaches is the ability to decide during the construction of a schedule whether it is beneficial to leave certain issue slots empty even if instructions are available for filling them. This is explained in Section 4.3.

Trace scheduling is a two-step process. In the first step, a trace is formed by selecting a sequence of basic blocks on a path with high execution frequency. In the second step, instructions in the selected trace are scheduled as if they were in a single basic block. This article presents an optimal solution to the scheduling problem and does not address the trace formation problem.

The article is organized as follows. Section 2 defines the problem and terminology. Section 3 summarizes prior work. Section 4 describes the basics of the proposed optimal algorithm, and Section 5 explains the details of the pruning techniques. Section 6 presents the experimental results. Conclusions and future work are covered in Section 7.

2. PROBLEM DEFINITION

The input to an instruction scheduler is a directed acyclic graph (DAG) called the *data dependence graph* (DDG). Each node in a DDG represents an instruction. A directed edge from node i to node j with label l indicates that instruction j depends on instruction i with a latency l . A DDG node with no predecessors is called a *root node* and the corresponding instruction is called a *root instruction* while a node with no successors is called a *leaf node* and the corresponding instruction is called a *leaf instruction*. DDGs in this article are represented in a standard format in which there is only one root node and one leaf node. Any DDG can be converted to standard format by introducing a dummy root

Fig. 2. Path DDSGs for the trace of Figure 1.

and/or leaf node and connecting them to the original root and/or leaf nodes with unit-latency edges. Figure 1(b) shows the DDG for the trace of Figure 1(a).

Besides satisfying the latency constraints represented by the DDG, a scheduler must satisfy the resource constraints dictated by the machine model. In this article, a machine model consists of a number of functional-unit types (pipelines) and a number of instances of each type, along with a mapping of instructions to functional unit types. If multiple units of a given type are available in one cycle, an instruction of that type can execute on any of these units. It is assumed that all functional units are fully pipelined.

Given a DDG and a machine model, a feasible schedule is an assignment of an issue cycle to each instruction in the DDG that satisfies the latency and resource constraints. A schedule starts at cycle one, and the *total length* of a schedule is the number of the last cycle in which an instruction is issued. In the original trace scheduling paper, the implicit objective was to minimize the total length [Fisher 1981]. In the present article, however, side paths are not ignored and the objective is to minimize a weighted sum of schedule lengths across all code paths. As described below, the weighted sum also accounts for compensation code.

The *weighted length* f of a schedule S is defined as

$$f(S) = \sum_{i=1}^N |S_i| W_i \quad (1)$$

where N is the number of code paths, $|S_i|$ is the length of code path i in schedule S , and w_i is the weight of code path i .

An optimal trace schedule is defined as a schedule with minimum weighted length. Optimal instruction scheduling is known to be an NP-hard problem even when the objective is minimizing the total schedule length [Hennessy and Gross 1983]. In this article, however, it is shown that a branch-and-bound solution can solve the vast majority of real problems within reasonable time.

The data dependencies in a single code path are represented by a subgraph of the overall DDG. That subgraph is called the path's *data dependence subgraph* (DDSG). Figure 2 shows the DDSGs of the four paths in the trace of Figure 1.

In trace scheduling, instructions can move from one basic block to another. To simplify the presentation and the implementation, the scope of

this article is limited to upward code motion. Disabling downward code motion is a typical limitation in global scheduling [Bernstein and Rodeh 1992; Freudenberger et al. 1994; Hwu et al. 1993]. An extensive experimental study of heuristic trace scheduling by Freudenberger et al. [1994] shows that disabling downward code motion does not cause a significant loss of performance. Exploring the effect of downward code motion with optimal trace scheduling remains an interesting topic for future work.

2.1 Lower Bounds

The optimal algorithm is based on computing a tight lower bound on each instruction's issue cycle and on the schedule length of each code path in a trace. In a given DDG, the *forward lower bound* (FLB) of an instruction is a lower bound on the difference between the instruction's issue cycle and the root instruction's issue cycle. Similarly, the *reverse lower bound* (RLB) of an instruction is a lower bound on the difference between the instruction's issue cycle and the leaf instruction's issue cycle. The *release time* of an instruction is the earliest cycle in which the instruction can be scheduled. Since the root instruction is always scheduled in cycle 1, the release time of an instruction is equal to 1 plus its FLB. When scheduling is done to achieve a certain target length, it is useful to define a deadline for each instruction. The *deadline* of an instruction relative to a target length is the latest cycle in which the instruction can be scheduled in order for the target length to be feasible. In a schedule of length L , the leaf instruction is scheduled in cycle L . Thus, each instruction i must be scheduled by the deadline $L - RLB(i)$ for the target length L to be feasible. For a given target length, the scheduling range of an instruction is the interval starting at the release time and ending at the deadline.

In a given DDG, the *critical-path* (CP) distance of a node from the root (leaf) is the length of a longest DDG path between the node and the root (leaf), where a DDG path length is the sum of edge labels along the path. A node's CP distance from the root (leaf) of the DDG is a valid but often loose forward (reverse) lower bound of the instruction represented by that node. Techniques for computing tighter lower bounds by accounting for resource constraints are presented in Section 3.1.

Given a set of FLBs and RLBs for all instructions, a lower bound on the total schedule length of a given DDG can be computed by adding one to the maximum of the leaf instruction's FLB and the root instruction's RLB. Similarly, a lower bound on the schedule length of any code path can be computed by applying the same technique to the path's DDSG.

When the code path lower bounds are known, it is convenient to rewrite Equation 1 as a weighted sum of differences from the lower bounds and define a cost function as follows:

$$Cost(S) = \sum_{i=1}^N D_i w_i \quad (2)$$

where $D_i = |S_i| - L_i$ is the difference between the schedule length $|S_i|$ of path i and the path's lower bound L_i .

2.2 Compensation Cost

When an instruction is moved from a source basic block B_s to a destination basic block B_d , paths that pass through B_s but not B_d are called *losing paths* because they lose an instruction. On the other hand, paths that pass through B_d but not B_s are called *gaining paths* because they gain an instruction. Paths that pass through both basic blocks and, therefore, do not lose or gain instructions are called *common paths*. For example, if Instruction F is moved from BB_3 to BB_1 in the trace of Figure 1, P_4 is a losing path, P_1 and P_2 are gaining paths, and P_3 is a common path.

When an instruction is moved above a side entrance, each path that starts at that entrance and includes the moved instruction is a losing path. To compensate that path for its loss, the moved instruction must be duplicated before the entrance. In this article, it is assumed that a new basic block, called a *compensation block*, is created between the off-trace predecessor blocks and the entry block of a losing path.

When an instruction is moved above a given entrance, not all paths starting at the entrance are losing paths. Paths starting at that entrance and ending at an exit that appears before the moved instruction in the original program order are not losing paths because they did not originally include the moved instruction. However, these paths still execute the compensation block in the trace schedule. All paths executing a compensation block are called *compensated paths*. Losing paths must be compensated paths in any correct trace schedule. However, some compensated paths may not be losing paths. For example, if there had been an entrance at BB_2 in the trace of Figure 1, moving Instruction F from BB_3 to BB_1 would have made the path consisting of BB_2 a compensated path but not a losing path.

Since during the scheduling of one trace it is not known how the compensation block will be scheduled with the neighboring off-trace basic blocks, the schedule length of the compensation block needs to be estimated according to some reasonable cost model. The cost model used here sets the estimated length of a compensation block to the lower bound of the DDSG that represents the duplicate instructions in the block. This is the minimum cost that is necessary to ensure that the loss of instructions by upward code motion is not treated as an improvement of the losing paths, which is consistent with the purpose of global code motion. Code motion between two basic blocks is intended to provide more flexibility in scheduling the common paths, not to falsely improve the losing paths by taking instructions out of them. To account for the compensation cost, the differential path length D_i in Equation 2 is modified for each compensated path to include the estimated length of the compensation block. This leads to the following definition for D_i

$$D_i = |S_i| + C_i - L_i \quad (3)$$

where $|S_i|$ and L_i are as defined above, and C_i is the estimated length of the compensation block preceding path i if path i is a compensated path in schedule S .

On a machine where compensation code avoidance is more important, a code-size component may be added to the cost function of Equation 2 to account for

any negative impact of increased code size, such as instruction-cache performance degradation:

$$Cost(S) = \sum_{i=1}^N D_i w_i + kI \quad (4)$$

where I is the total number of instructions in all compensation blocks and k is a parameter, called the *code-size factor*, that expresses the relative importance of static code size. Equation 4 provides a systematic way of controlling the tradeoff between schedule length and static code size by setting k to an appropriate value and minimizing a single cost function. Different values for the code-size factor are studied experimentally in Section 6.3, where it is shown that by setting k to a small enough value, smaller code size can be achieved without affecting the optimality of the weighted schedule length. In the rest of the article, k is equal to zero, and the cost is the weighted schedule length relative to the lower bounds.

3. PREVIOUS WORK

This article extends previous work on trace scheduling by providing an optimal algorithm. The optimal algorithm is built on top of previous work on lower bound techniques and branch-and-bound enumeration for basic-block scheduling. This section summarizes related previous work.

3.1 Lower Bound Techniques

Various lower-bound techniques have been developed for basic-block scheduling based on analyzing resource availability versus resource requirements. These techniques can be applied to any DDG whether it represents a basic block or a trace.

A fundamental lower-bound algorithm was developed by Rim and Jain [1994]. The algorithm is based on relaxing the NP-complete scheduling problem to a minimum-lateness release time and deadline (MLRD) problem that can be solved optimally in polynomial time. Given a set of initial releasetimes and deadlines for each node in a DDG, the algorithm computes a potentially tighter release time for the DDG's leaf node, which is a lower bound on the total schedule length. The initial release times and deadlines are based on CP distances from the root and leaf nodes. In this article, Rim and Jain's algorithm is also called *relaxed scheduling*.

In subsequent work, Langevin and Cerny [1996] observe that a tighter lower bound can be computed if the release times of the nodes are computed by recursively applying the Rim-Jain algorithm to the subgraph between each node and the root node. This recursive form of relaxed scheduling has the additional advantage of computing potentially tighter release times for the individual nodes in the DDG.

Lower-bound techniques can be applied to the DDG in both directions. In the reverse direction, the roles of the root and leaf nodes are interchanged and the directions of all edges are reversed. The same technique is then applied to compute tighter RLBs leading to tighter deadlines.

3.2 Heuristic Schedulers

List scheduling is probably the most widely used technique for instruction scheduling [Muchnick 1997]. It is a greedy algorithm that maintains a *ready list* of instructions and selects one ready instruction for scheduling based on certain heuristics. An instruction is *ready* if all of its predecessors in the DDG have been issued and the corresponding latencies have been satisfied. The CP distance from the leaf is a commonly used heuristic for selecting a ready instruction when the objective is minimizing total schedule length.

The original trace scheduling paper [Fisher 1981] used CP list scheduling. Most subsequent work on trace scheduling focused on improving the original algorithm by limiting the amount of compensation code [Freudenberger et al. 1994; Lah and Atkin 1983; Linn 1983; Smith et al. 1992; Su et al. 1984], a problem that was recognized in the original paper. The paper by Freudenberger et al. [1994] provides an extensive study of compensation code and identifies two different approaches to limiting compensation code: avoidance (avoiding code motions that lead to compensation code) and suppression (using global data and control flow information to detect cases where compensation code is redundant). The present article only considers compensation code avoidance because it is part of the NP-complete scheduling problem for which an optimal solution is sought. In contrast, compensation code suppression is a program-flow problem that has a polynomial-time solution [Muchnick 1997], which can be used with both heuristic and optimal trace schedulers.

Prior work on compensation code avoidance focuses on disabling certain kinds of global code motion, usually by adding extra edges to the DDG. Examples include disabling global code motion involving basic blocks with low- execution frequency [Fisher 1981], global motion of instructions that are not on the DDG's critical path [Su et al. 1984] and downward code motion [Freudenberger et al. 1994].

Another problem addressed by previous work is side-path degradation when the scheduling objective is limited to minimizing the total schedule length. Smith et al. [1992] observe that in nonnumerical code it is often hard to identify the most likely path, and thus, ignoring side paths can lead to poor overall performance. They propose an approach that they call *conscientious trace scheduling*, which is also known as *successive retirement* (SR) [Chekuri et al. 1996]. In SR, basic blocks in a trace are scheduled in control-flow order, and ready instructions belonging to the basic block currently being scheduled are given priority over ready instructions from other basic blocks. As reported by Smith et al. [1992], SR does well at minimizing the lengths of some side paths, but often produces main-path schedules that are not as short as those produced by the CP heuristic.

The optimal algorithm proposed in this article is based on an explicit cost model that encompasses the three conflicting objectives addressed by previous heuristics: minimizing total schedule length, avoiding side-path degradation, and minimizing code size. The resulting schedules are optimal with respect to the cost model. If a schedule that satisfies all three objectives exists, it is found by the enumerative search. If no such schedule exists, a schedule with the least expensive compromise is selected.

3.3 Optimal Instruction Scheduling

Previous research on optimal instruction scheduling used two different combinatorial optimization techniques [Wolsey 1998]: integer linear programming [Wilken et al. 2000; Winkel 2007] and enumeration [Chou and Chung 1995; Narasimhan and Ramanujam 2001; Shobaki and Wilken 2004]. The authors are not aware of any study that directly compares the two optimization techniques on the same set of scheduling problems. The optimal superblock scheduling algorithm by Shobaki and Wilken [2004] and the optimal global scheduling algorithm by Winkel [2007] are the most relevant pieces of work.

Winkel [2007] describes an optimal scheduling algorithm that can schedule a whole routine using integer programming. He reports significant performance improvements on some CPU2006 benchmarks versus a state-of-the-art heuristic global scheduler.

Shobaki and Wilken [2004] describe an optimal algorithm for scheduling a superblock using enumeration. The superblock is a special case of the trace with only one entrance [Hwu et al. 1993]. The presence of side entrances and the need for compensation code make trace scheduling a fundamentally more complex problem than superblock scheduling. More specifically, the number of paths in a superblock is a linear function of the number of basic blocks, while it is a quadratic function in traces. This makes it harder to compute tight lower bounds and search for an optimal solution using combinatorial optimization techniques. Furthermore, the need to account for compensation code during optimal scheduling adds significant complexity to the algorithm.

A particularly important idea used in previous enumerative approaches to optimal instruction scheduling is applying relaxed scheduling at the enumeration tree nodes to compute the desired lower bounds [Narasimhan and Ramanujam 2001]. Another important idea is using history information to prune certain tree nodes if they cannot lead to better schedules than previously visited nodes [Shobaki and Wilken 2004]. In addition to these two techniques, a technique called *node superiority* can further speed up the enumeration process [Chou and Chung 1995].

4. BASIC ALGORITHM

The proposed algorithm uses branch-and-bound enumeration. The solution space is explored in a manner that can be represented by a decision tree called the *enumeration tree*. Relaxed scheduling is used as the lower bound technique at each tree node for pruning the solution space. The details of the pruning techniques are described in Section 5.

As shown in Figure 3, the first step in the algorithm is applying a heuristic scheduling technique to the trace to find an initial feasible schedule. In the next step, lower bounds are computed as detailed below and the cost of the heuristic schedule is evaluated. If the cost is zero (all paths are scheduled at their lower bounds), the heuristic schedule is optimal and the solution is complete. Otherwise, the heuristic schedule is saved as the best known schedule and enumeration is used to search for a lower-cost feasible schedule. The search iteratively explores one total schedule length at a time instead of exploring all

Fig. 3. Block diagram for the optimal trace scheduling algorithm.

possible schedule lengths at once. The advantage of this iterative approach is that it allows the computation of tighter scheduling ranges relative to the target schedule length of each iteration. In the first iteration, feasible schedules with length equal to the total schedule length lower bound are explored and the process continues until the maximum interesting schedule length has been explored. A formula for computing the maximum interesting schedule length is developed in Section 4.1.

During each iteration, the enumerator acts like a list scheduler but with the additional feature of backtracking when the lower bounds indicate that a lower-cost schedule is not feasible. The enumerator tries to construct a feasible schedule incrementally, starting with an empty schedule and adding one instruction or stall at a time. At each tree node, the scheduled instructions form a *partial schedule*. In each step, the enumerator either makes forward progress by augmenting the current partial schedule or determines that a lower-cost schedule cannot be found at the current target length. In the latter case, the enumerator backtracks by removing the last instruction added. The partial schedule is augmented by choosing an instruction from the ready list. In the case of backtracking, an alternate ready instruction is attempted and so on.

When the enumerator finds a feasible schedule, the schedule's cost is computed and compared to the best known cost. If the new cost is lower, the new schedule is saved as the best known schedule and the best cost is updated. The process continues until the solution space at each interesting schedule length is explored or a zero-cost schedule is found.

In this article, scheduling is always performed in the forward direction. During schedule construction, the enumerator keeps track of the current basic block. A ready instruction that belongs to the current basic block is called a *local ready instruction*, while a ready instruction belonging to another basic block is called an *external ready instruction*.

4.1 Schedule-Length Upper Bound

Given a feasible schedule of total length L and cost U_C , U_C is an upper bound on the cost, but L is not necessarily an upper bound on the total schedule length. A schedule of total length $L + 1$ or longer may have a cost less than U_C if some of

its high-weight side paths have shorter schedules. A formula can be derived for computing a total-schedule-length upper bound U_S given a cost upper bound U_C .

First, Equation 2 is rearranged so that the main path cost appears as a separate term:

$$Cost(S) = D_m w_m + \sum_{i=1}^{N-1} D_i w_i = (|S| - L_S) w_m + \sum_{i=1}^{N-1} D_i w_i \quad (5)$$

where $|S|$ is the total schedule length, L_S is a lower bound on the total schedule length and D_m and w_m are the differential length and weight of the main path.

The minimum cost for a schedule of a given total length occurs when each side path is scheduled at its lower bound. In this case, the summation in Equation 5 vanishes, and the minimum cost for length L is

$$Cost_{Min}(L) = (L - L_S) w_m \quad (6)$$

A total schedule length whose minimum cost is equal to or greater than the cost upper bound U_C does not need to be considered. Hence, to find the maximum interesting schedule length U_S , the right-hand side in Equation 6 is equated to the cost upper bound U_C and U_S is substituted for L :

$$U_C = (U_S - L_S) w_m \quad (7)$$

Solving this equation for U_S gives

$$U_S = \frac{U_C}{w_m} + L_S \quad (8)$$

This equation gives the desired exclusive upper bound on the total schedule length. In searching for schedules with a lower cost than a known cost U_C , total schedule lengths greater than or equal to U_S need not be examined.

4.2 Cost Computation

This section describes how the lower bounds in the second stage of the algorithm (see Figure 3) are computed and used to evaluate the cost of a trace schedule.

4.2.1 Static Lower Bounds. The lower bounds that are computed before enumeration to evaluate the cost of the heuristic schedule are called *static lower bounds* to distinguish them from the *dynamic lower bounds* that are computed during enumeration and used to prune the enumeration tree as described in Section 5.2.

Static lower bounds are computed for instructions and paths. First, the Langevin-Cerny technique is applied once in each direction to the entire DDG to compute the FLB and RLB for each instruction. These lower bounds are used to compute the scheduling range for each instruction at each target schedule length. Figure 4(a) shows the static scheduling ranges of the instructions in the trace of Figure 1 for a target schedule length of nine cycles on a single-issue machine. These ranges are based on the FLBS and RLBs computed by the Langevin-Cerny technique as shown in Figure 4(b).

Fig. 4. Instruction lower bounds and scheduling ranges for the trace of Figure 1. Ranges are for a target length of 9 cycles on a single-issue machine.

Fig. 5. Path lower bounds for the trace of Figure 1.

The same technique is then applied separately to each code path's DDSG to compute a static lower bound on the path's schedule length. These path lower bounds are then used throughout the algorithm to evaluate the cost function of Equation 2. Path lower bounds for the trace of Figure 1 are shown in Figure 5.

4.2.2 Compensation-Trace Interface. If a compensation block includes instructions with long latencies (greater than one), a correct trace schedule must include the right number of stalls between the compensation block and the on-trace code to ensure that all latencies are satisfied before the execution of on-trace code. The number of stalls is a function of both the on-trace schedule and off-trace (compensation) schedule. However, the compensation schedule is not known when a trace is scheduled and the compensation cost is a lower bound on the compensation block's schedule length. Hence, stalls are accounted for in the compensation cost by adding the latencies between compensation code and on-trace code to the compensation block's DDSG before the lower bound is

Fig. 6. A heuristic-based schedule for the trace of Figure 1.

computed. The details of computing the lower bound of a DDSG with unsatisfied latencies are described in Shobaki [2006].

For example, Figure 6 shows a CP list schedule for the trace of Figure 1. The compensation block for the side entrance includes instruction G with latency 3. The first on-trace instruction at the entrance is Instruction H, which is dependent on Instruction G with latency 3. Since Instruction H is scheduled in the first cycle following the compensation block, an edge with latency 3 is added to the compensation DDSG. Had Instruction H been scheduled in the second cycle following the compensation block, the latency of the added edge would have been 2, and so on. The section of a trace schedule that may have unsatisfied latencies with instructions in a compensation block is called a *compensation-critical section* for that compensation block. In the example of Figure 6, the compensation-critical section consists of Cycles 8 and 9.

4.2.3 Cost Computation Example. Cost computation is illustrated in Figure 6 on a CP list schedule for the trace of Figure 1. Note that the entrance of the third basic block (BB_3) has changed from Instruction F to Instruction H. To estimate the cost of the compensation block, a lower bound is computed for the 2-node DDSG with an unsatisfied latency of 3. This lower bound is four cycles on a single-issue machine. That includes two cycles for the instructions and two cycles for the stalls needed to satisfy the compensation block's latencies before the dependent on-trace code is executed. The estimated length of the compensation block is then added to the schedule length of each compensated path. In this case, the only compensated path is P_4 . To compute the cost, the path lower bound computed in Figure 5 is subtracted from the schedule length of each path. The cost of 1.28 indicates that, considering all paths, the schedule is on average 1.28 cycles longer than the lower bound.

4.3 Scheduling Stalls

When a trace schedule is constructed, it is sometimes beneficial to leave an issue slot empty even if a ready instruction can be scheduled in that slot. Identifying cases where scheduling a stall results in a better schedule is one advantage of the proposed enumerative approach over greedy heuristic approaches. Scheduling a stall instead of a ready instruction can be beneficial in two cases:

Fig. 7. Optimal schedule with a stall to avoid costly upward code motion.

Case 1. Avoiding costly upward code motion. Moving an instruction from its original basic block to the current basic block may improve the schedules of the paths that are common to both basic blocks. However, the upward code motion results in gaining and/or compensated paths. Sometimes, the added cost of the gain and/or compensation outweighs the benefit. When the ready list includes an external instruction and the added cost of moving the instruction up is greater than the cost reduction resulting from scheduling the instruction early, scheduling a stall instead of the external instruction results in a lower-cost schedule.

An example is shown in Figure 7. The trace in the figure consists of two basic blocks with two entrances and one exit. Each entrance defines a path. After Instruction A is scheduled in Cycle 1, Instruction D, which is external to the first basic block, is ready for scheduling in Cycle 2. However, moving Instruction D above the second entrance would require compensation code and would not shorten the schedule of either path. This costly upward code motion can be avoided by not filling the empty slot in Cycle 2 and leaving Instruction D in its original basic block as shown in Figure 7(c).

Case 2. Avoiding early side entrances. The schedule length of a path starting at a side entrance increases when the entrance is scheduled early. However, scheduling an entrance early may be beneficial to other paths. Optimal scheduling resolves this trade off in a systematic manner. If the paths starting at a side entrance have high enough weights, the schedule-length increase caused by scheduling their entrance early outweighs the benefit to other paths (if any). In this case, delaying the entrance and scheduling a stall results in a lower-cost schedule. An example is shown in Figure 8. The trace consists of two paths: P_1 and P_2 . Instruction B, which is the entrance to P_2 , is ready in Cycle 2. However, because Instruction C cannot be scheduled earlier than Cycle 4, scheduling B in Cycle 2 would lengthen P_2 without shortening P_1 . This is avoided by scheduling a stall in Cycle 2 and delaying the entrance of P_2 to Cycle 3, as shown in Figure 8(c).

To the authors' knowledge, no trace scheduling heuristic has been proposed to address the two cases described above. All existing heuristics greedily schedule an instruction when the ready list is not empty. The enumerator's ability to decide whether scheduling a stall is beneficial can result in a significant reduction in compensation-code size, especially on wider-issue machines (see the experimental results in Section 6.4).

Fig. 8. Optimal schedule with a delayed entrance.

5. PRUNING TECHNIQUES

Efficient enumeration requires effective pruning techniques. A pruning technique is a feasibility test at a tree node that determines whether finding a schedule of the desired property is possible below that node. The desired properties in the proposed algorithm are a total schedule length equal to the target schedule length and a cost less than the best known cost. The following three feasibility tests are applied at each tree node:

5.1 First Test: Range Tightening

When an instruction is scheduled, the release times of some other instructions may be tightened. Specifically, when the enumerator steps from cycle C to cycle $C+1$, the release times of all unscheduled ready instructions are tightened to $C+1$. Since increasing an instruction's release time may increase the release times of its DDG successors due to latency constraints, the tightened release times are propagated to the successors and the scheduling ranges are checked. If an instruction's release time is larger than its deadline, the current tree node is declared infeasible and the enumerator backtracks.

5.2 Second Test: Dynamic Path Lower Bounds

Using the tightened release times obtained in the first feasibility test, a dynamic lower bound on each path's schedule length is computed. The dynamic path lower bounds, which are potentially tighter than the static lower bounds, are used to compute a tighter dynamic cost lower bound. The dynamic cost lower bound at each tree node is compared to the best known cost. If the dynamic cost lower bound is not less than the best known cost, no better schedules exist below the current tree node. Therefore, infeasibility is detected and backtracking occurs.

Initially, at the root tree node, the lower bound of each code path is equal to its static lower bound. Then, as instructions are scheduled, path lower bounds potentially increase for one or two of the following three reasons:

1. The range of at least one instruction in the path has been tightened. The range of an instruction is tightened if its release time is tightened by the first feasibility test, or if the instruction is scheduled, in which case its scheduling range becomes exactly one cycle. In this case, the path is called a *tightened path*.

2. An upward code motion occurred and the path is a gaining path.
3. An upward code motion occurred and the path is a compensated path.

The lower bound of a gaining, tightened or compensated path is recomputed by applying relaxed scheduling to the updated DDSG and the cost lower bound is updated. At a tree node, a code path can be fully scheduled, partially scheduled, or completely unscheduled. In the first case, the path's final schedule length is known and relaxed scheduling, is not needed. In the second and third cases, a relaxed schedule for the mix of scheduled (if any) and unscheduled instructions is computed using the Rim-Jain algorithm [Rim and Jain 1994]. Unlike static lower-bounds that are computed by applying the lower-bound technique to an isolated DDSG, dynamic path lower-bounds are computed in-context using each instruction's scheduling range in the total DDG. This captures the interactions among paths as instructions' scheduling ranges are tightened. The details of modifying the original Rim-Jain algorithm to compute an in-context path lower bound are omitted in this article, and the interested reader is referred to Shobaki [2006]. However, the process is intuitively explained in the complete example of Section 5.4.

5.2.1 Updating Lower Bounds of Paths with Side Entrances. Schedules in this article are constructed in the forward direction, that is, by considering issue cycles in increasing time order. During schedule construction, the entrance to a side path is not known until the basic block above the side path has been fully scheduled. The next instruction scheduled will be the entrance to the side path. When the first instruction of a side path is scheduled, the entrance is *resolved*. After scheduling an entrance, no compensation code can be added to the entrance's compensation block, so the compensation block is *closed*.

Subtlety. Even though a basic block can be empty in a trace schedule, a path's on-trace schedule cannot be empty because a path must have an exit, which is a branch or the DDG's leaf node. Since branches cannot move up and the leaf node is always scheduled last, the on-trace schedule of each path will include at least the path's exit.

A path starting at a side entrance consists of two components: the on-trace code and the off-trace (compensation) code. Computing a lower bound for each component and then adding the lower bounds does not necessarily give a correct lower bound for the path. This is because if an instruction moves from an on-trace component to a compensation block with empty slots, it may make the on-trace component shorter without making the off-trace component longer. For example, consider scheduling the trace of Figure 9(a) on a dual-issue machine. Let Side Path CDE be the side path containing the three instructions C, D, and E. The DDSG for Side Path CDE is shown in Figure 9(b). Figure 9(c) shows a partial schedule in which Side Path CDE is a compensated path due to the upward code motion of Instruction C. The boundary between the first basic block and the second basic block is drawn in a dotted line because the actual boundary will not be determined until Instruction B is scheduled. Side path CDE then has an unresolved entrance.

Fig. 9. Dynamic lower bounds for compensated paths.

Consider the problem of computing a dynamic lower bound on the schedule length of Side Path CDE in the partial schedule shown in the figure. The lower bound on the on-trace schedule length of the path is two cycles and the lower bound on the compensation block cost is one cycle. Summing these two lower bounds gives an incorrect path lower bound of three cycles. This lower bound is incorrect because if Instruction D is scheduled next as shown in the complete schedule of Figure 9(d), it will be moved to the compensation block. Since Instructions C and D are independent, the compensation block can be scheduled in one cycle on a dual-issue machine. The on-trace schedule length will be one cycle because Instruction E is the only on-trace instruction left in Path CDE. This gives a total schedule length of two cycles for Path CDE.

A dynamic lower bound for any path starting at a side entrance can be computed by summing the on-trace lower bound and the off-trace lower bound when instructions can no longer move into the compensation block. This is the case when one of the following two conditions is satisfied:

- The compensation block at the path’s entrance has been closed due to entrance resolution.
- The path no longer includes any instructions that can move above the entrance.

Both conditions are checked dynamically at each tree node to determine whether adding the on-trace lower bound and the off-trace lower bound is legal.

The first condition can be checked by keeping track of the current basic block number (basic blocks are numbered sequentially in control-flow order within the trace). When all the original instructions of the current basic block are scheduled, the current basic block number is incremented. A side-path’s entrance is resolved when the current basic block number is equal to or greater than the path’s first block number.

Checking for the second condition requires keeping track of the number of instructions in a path that may move upward. Once the number of upward movable instructions in a path has been computed before enumeration, the enumerator can keep track of upward mobility by decrementing the count of movable instructions each time an instruction is moved above an entrance. When the dynamic count of movable instructions in a path drops to zero, the on-trace and off-trace lower bounds can be added to compute a tighter dynamic lower bound for the path.

When neither condition is satisfied at a given tree node, the lower bound of a path with a side entrance is not updated at that node unless the path is a gaining path with respect to the last scheduling decision. In the latter case, a static lower bound is computed for the updated DDSG with the gained instruction.

5.3 Third Test: History-Based Domination

Information about previously explored tree nodes can be used to prune related tree nodes. When the entire subtree below a tree node is explored, the minimum cost of any schedule below that tree node is known. Information about the explored tree node can be kept in a *history table*. When a tree node with the same set of scheduled instructions is later visited, the current tree node is compared with the history node. If it can be shown that the current node cannot have a feasible schedule with a better cost than the best cost under the history node, the current node is *dominated* by the history node and does not need to be explored any further. This section develops a sufficient condition for history-based domination.

Consider a tree node x . Let the partial schedule at x be P_x , let $S(x)$ be the set of instructions scheduled in P_x , and let $U(x)$ be the set of instructions in the DDG that are not scheduled in P_x . A feasible schedule below x is denoted by $P_x Q_x$ to indicate that it is formed by concatenating two partial schedules: the prefix P_x and a suffix Q_x in which the instructions of $U(x)$ are scheduled. The data dependencies among unscheduled instructions in $U(x)$ are represented by a subgraph of the total DDG that is formed by removing the nodes representing instructions in $S(x)$ and the edges incident to them. The resulting subgraph is called the unscheduled data dependence graph (UDDG).

A sufficient domination condition is developed for tree nodes with similar partial schedules. Two partial schedules P_x and P_y are *similar* if $S(x) = S(y)$. Two tree nodes x and y are similar if the corresponding partial schedules P_x and P_y are similar and the nodes are at the same depth in the enumeration tree.

Definition. Tree node x dominates a similar tree node y if for any feasible schedule $P_y Q_y$ below y there is a feasible schedule $P_x Q_y$ below x such that $\text{cost}(P_x Q_y) \leq \text{cost}(P_y Q_y)$.

At tree node x , each unscheduled instruction i in $U(x)$ has a dynamic release time $r_x(i)$ that is potentially tighter than the static release time $r(i)$. $r_x(i)$ may be tighter than $r(i)$ due to the resource and latency constraints imposed by the scheduled instructions in P_x .

Given two similar tree nodes x and y , it can be proven that x dominates y if the cost of the partial schedule P_x is less than or equal to the cost of the partial schedule P_y and the remaining scheduling subproblem below x is not more constrained than the remaining subproblem below y . The constraints on the remaining subproblems are the resource constraints, the latency constraints and pending compensation code effects. These three constraints are discussed next.

5.3.1 Resource Constraints. To show that every suffix schedule Q_y that is feasible below y is also feasible below x , the number of remaining issue slots

below x must be at least the same as the number of remaining issue slots below y for each issue type.

5.3.2 Latency Constraints. To show that every suffix schedule Q_y that is feasible below y is also feasible below x , *unsatisfied latencies* from scheduled instructions to unscheduled instructions should be checked. If an instruction with latency L is scheduled in cycle C in P_x and the cycle immediately following P_x is less than $C + L$, the latency L is unsatisfied. Unsatisfied latencies establish initial conditions on the remaining scheduling subproblems. An efficient way of checking unsatisfied latencies is to examine the dynamic release times of unscheduled instructions. If every dynamic release time below x is less than or equal to the corresponding dynamic release time below y , the subproblem below x is not more latency constrained than the subproblem below y .

5.3.3 Pending Compensation Code Effects. Comparison between the partial schedules at two tree nodes should include both compensation code and on-trace code. However, compensation code and on-trace code are treated differently in cost computation: the cost for on-trace code is an actual schedule length, while the cost of the compensation code is an estimate based on the cost model. This makes the comparison difficult unless compensation blocks are either identical or closed in both partial schedules. For example, consider a trace with one side entrance on a dual-issue machine. Let P_x and P_y be two partial schedules with equal cost, but in P_x the compensation block includes one instruction i , while the compensation block is empty in P_y . If an instruction j that is parallel to i and has the same latency is later scheduled above the side entrance, the compensation cost in P_x will not change while the compensation cost in P_y will increase. To rule out such situations, each compensation block is required to be either closed in both partial schedules or to include the same instructions in both partial schedules. This is called the compensation closure/match condition.

The closure or match of compensation blocks must take into account the compensation-trace interface. Recall that the compensation-block cost accounts for the stalls that are appended to satisfy any latencies between compensation code and on-trace code. The number of stalls needed is a function of the compensation-critical section of the schedule. One straightforward solution to this problem is to require all compensation-critical sections in both partial schedules to be either identical (in the case of match) or fully scheduled (in the case of closure). So, whenever the compensation closure/match condition is referenced below, the compensation-critical sections are also included.

When two partial schedules P_x and P_y are compared, a path entrance may be resolved in P_x but not in P_y or vice versa. Similarly, a path exit may be scheduled in one partial schedule and unscheduled in the other. However, if the compensation closure/match condition is satisfied for two similar partial schedules, every path entrance or exit either appears in both partial schedules or does not appear in either partial schedule. This is shown in the next lemma.

LEMMA. *Given two partial trace schedules P_x and P_y at two similar tree nodes, such that the compensation closure/match condition is satisfied, then for each*

side path in the trace:

1. *The path entrance is resolved in P_x if and only if it is resolved in P_y .*
2. *The path exit is scheduled in P_x if and only if it is scheduled in P_y .*

PROOF. Since P_x and P_y in this lemma are indistinguishable, only one direction of the implication (“if” or “only if”) needs to be proved.

1. Assume that a side path is resolved in P_x but not resolved in P_y . Let B be the first basic block in that path. Since the same instructions are scheduled in both P_x and P_y , there must be an instruction i that is scheduled in Block B in P_x but is scheduled in some earlier basic block in P_y . Instruction i will then appear in the path’s compensation block in P_y but will not appear in the path’s compensation block in P_x . It follows that compensation blocks are not identical in both partial schedules and that the path’s compensation block in P_y is not closed, which contradicts the compensation closure/match condition.
2. The path exit is necessarily a branch instruction. It immediately follows from the similarity condition that if a branch instruction is scheduled in P_x it must be scheduled in P_y . $\square$

THEOREM. *A tree node x dominates a similar tree node y if all the following conditions are satisfied:*

1. *Partial-Cost Condition: $cost(P_x) \leq cost(P_y)$*
2. *Resource Condition: For each issue type, the number of available issue slots below x is greater than or equal to the number of available slots below y .*
3. *Latency Condition: For each unscheduled instruction i , $r_x(i) \leq r_y(i)$.*
4. *Compensation Condition: Every open compensation block and its compensation-critical section are identical in both P_x and P_y .*

PROOF. First, it is shown that a feasible schedule below y is also a feasible schedule below x , by comparing the residual scheduling problems below x and y . A scheduling problem is defined by the instruction set to schedule, the resource constraints and the latency constraints. The comparison between the two residual problems must then consider each of these three elements:

1. The set of instructions to schedule: Due to the similarity relationship between x and y , the two sets of unscheduled instructions $U(x)$ and $U(y)$ are identical.
2. Resource Constraints: It follows from the resource condition that any schedule Q_y satisfying the resource constraints below y will necessarily satisfy the resource constraints below x .
3. Latency Constraints: The UDDG representing the latency constraints among unscheduled instructions is the same below x and y . The difference between the two nodes is in the dynamic release times imposed by the scheduled instructions on the unscheduled instructions. Assume that Q_y is a feasible schedule below y . Q_y then satisfies the dynamic release times at y . But, according to the latency condition, each dynamic release time at x is less

Fig. 10. History domination example for the trace of Figure 8.

than or equal to the corresponding release time at y . It follows that Q_y also satisfies the dynamic release times at x , and therefore Q_y is feasible below x .

It remains to show that the cost of $P_x Q_y$ is less than or equal to the cost of $P_y Q_y$. The cost of a schedule consists of an on-trace component and a compensation component.

First consider the on-trace component. For any code path in the trace, it follows from the above lemma that there are only three cases to consider:

Case 1. The path is fully scheduled in both P_x and P_y . In this case, $\text{cost}(P_x Q_y) = \text{cost}(P_x)$ and $\text{cost}(P_y Q_y) = \text{cost}(P_y)$ for this path, and the initial inequality still holds.

Case 2. The path is totally unscheduled in both P_x and P_y . In this case, $\text{cost}(P_x Q_y) = \text{cost}(P_y Q_y) = \text{cost}(Q_y)$ for this path, and both costs increase by the same amount.

Case 3. The path starts in P_x or P_y and ends in Q_y . Since nodes x and y are at the same depth, the number of cycles added to the path's length by Q_y is the same below both x and y , thus preserving the initial inequality.

Next, consider the compensation component. By the compensation condition, each compensation block is either closed or the same in both partial schedules. If a compensation block is closed in both partial schedules, its cost is totally accounted for in the costs of P_x and P_y . If, on the other hand, a compensation block is open, then by the compensation match condition, the compensation cost of P_x is equal to that of P_y . Appending Q_y to both partial schedules will add the same instructions to each compensation block and its compensation-critical section, thus preserving the initial inequality. $\square$

5.3.4 History Domination Example. To illustrate history-based domination, consider the enumeration tree of Figure 10 for the trace of Figure 8. The target schedule length is four cycles on a single-issue machine. Assume for illustration that the lower bounds are three cycles for P_1 (which is clearly a loose bound) and 2 cycles for P_2 and that the two paths have equal weights.

The enumerator first explores nodes 1 through 4 and finds the feasible schedule: $\langle A, \text{stall}, B, C \rangle$. Relative to the lower bounds, the cost of this schedule is

0.5 cycles because the schedule length for P_1 is one cycle longer than its lower bound. Since the cost is not zero, the enumerator continues the search and backtracks to Node 1, storing Nodes 4, 3, and 2 in the history table. Starting from Node 1 again, the enumerator constructs the partial schedule $\langle A, B, \text{stall} \rangle$ at Node 6. At this point, Node 6 is similar to Node 3 in the history table and history domination checking is performed between the partial schedules $P_x = \langle A, \text{stall}, B \rangle$ and $P_y = \langle A, B, \text{stall} \rangle$ as follows:

Partial-Cost Condition: The cost of P_x is less than the cost of P_y because the side-path schedule is one cycle shorter.

Resource Condition: There is one empty slot available below each node.

Latency Condition: Instruction C is the only unscheduled instruction. The dynamic release time of Instruction C is 4 under both nodes.

Compensation Condition: No instructions are moved above the side entrance. Therefore, the compensation block is empty in both partial schedules, and this condition is satisfied.

Since all the domination conditions are satisfied, Node 3 dominates Node 6 and the enumerator does not need to explore Node 6 any further.

5.4 Complete Enumeration Example

The proposed optimal algorithm is illustrated by applying it to the trace in Figure 1. Scheduling is done for a single-issue machine. The analysis of the example focuses on the second pruning technique, dynamic path lower bounds.

A feasible schedule is first produced using a heuristic technique. The CP list schedule of Figure 6 is used in this example. The heuristic schedule's cost of 1.28 cycles becomes the cost upper bound U_C . This cost upper bound and the total-schedule-length lower bound of 9 are plugged into the upper-bound formula (Equation 8) to compute an exclusive upper bound on the total schedule length:

$$U_S = 9 + 1.28/0.28 = 13.6$$

Thus, the maximum schedule length that needs to be examined is 13. Any schedule of total length of 14 or greater will have a cost greater than 1.28.

The first iteration explores all schedules whose total length is nine cycles. Figure 11(a) shows the first five nodes in the enumeration tree during this iteration. In Figures 11(b,c,d), the DDSG of Path 1 is used as an example to illustrate the computation of dynamic path lower bounds. Initially, the ranges of all instructions in Path 1 are equal to their static scheduling ranges.

In the first enumeration step, Instruction A is scheduled in Cycle 1 and the enumerator steps forward to Node 1. This step does not tighten any lower bounds, and the cost at Node 1 maintains its initial value of zero (see Figure 11(a)). In the second step, Instruction F is scheduled in Cycle 2. Since Cycle 2 is now occupied, the release time of Instruction B is tightened to 3 as shown in Figure 11(c). This, in turn, tightens the release time of Instruction C to 4. With the scheduling of Instruction F, Path 1 is a tightened path and a gaining path. Due to this gain and tightening, the dynamic lower bound of Path 1 is now 4. A similar lower bound tightening of one cycle occurs in Path 2 (not shown in the figure). Tightening the lower bounds of these two paths by one

Fig. 11. Computation of dynamic path lower bounds for the trace of Figure 1.

Fig. 12. Optimal schedule for the trace of Figure 1.

cycle per path tightens the cost lower bound by 0.52 (the weight of Path 1 plus the weight of Path 2). This dynamic cost appears next to Node 2 in Figure 11(a). A similar one-cycle tightening of Instruction C's release time (Figure 11(d)) occurs when the enumerator steps from Node 3 to Node 4, scheduling Instruction G in Cycle 4 (Figure 11(a)). The resulting dynamic cost of 1.04 is shown next to Node 4 in Figure 11(a). Enumeration continues down that path in the tree, and a feasible schedule of cost 1.04 is found (not shown in the figure).

The enumerator then back tracks to the deepest tree node whose cost is less than 1.04, namely Node 3. The search continues with lower bound and cost tightening as described above until a feasible schedule of cost 0.8 (not shown) then a feasible schedule of cost 0.56 are found before the search at length 9 terminates. The schedule of cost 0.56, which is the best schedule at length 9, is shown in Figure 12. All paths are scheduled at their lower bounds except for Path 2, which has gained two instructions.

With a best cost of 0.56, the schedule-length upper bound becomes:

$$U_S = 9 + 0.56/0.28 = 11$$

This indicates that a schedule of total length 11 will at best have a cost equal to the current best cost of 0.56. Hence, to complete the search only length 10 needs to be examined. An iteration at length 10 (not shown) is performed and no feasible schedule with cost less than 0.56 is found. This concludes the search with the optimal schedule of cost 0.56 shown in Figure 12.

In this search, with the aid of dynamic path lower bounds, only three feasible schedules are examined. If the search is performed without lower-bound based pruning, 30 feasible schedules will be examined (18 schedules at length 9 and 12 schedules at length 10), mainly due to the many possible placements of the upwardly movable instructions F and G.

6. EXPERIMENTAL RESULTS

The optimal trace scheduling algorithm described in this article was implemented and applied to a set of traces generated by the Sun Studio Compiler. Sun Studio 12.0 was modified to print trace DDGs to a text file. The SPEC CPU2006 Integer Benchmarks (or CPU2006 INT for short) were compiled with the modified compiler using the highest level of optimization (O5). Profile feedback was enabled for accurate trace formation. The DDGs of the formed traces were then input to the optimal scheduler. Optimal scheduling was performed for an UltraSPARC IV processor, which has five types of functional units [Sun Microsystems 2004]:

- Two integer arithmetic units. An integer arithmetic instruction can execute on either unit.
- One memory unit
- One branch unit
- One floating point addition unit
- One floating point multiplication unit

In each cycle, the processor can issue up to four instructions.

The machine model in this article assumes that all functional units are pipelined and that each instruction uses only one type of functional unit in each cycle. This assumption is valid for all but a few UltraSPARC IV instructions [Sun Microsystems 2004]. The experiments reported here exclude traces with nonpipelined instructions (such as FP div, FP sqrt, INT mul and INT div) or with instructions that use multiple functional units in one cycle. Traces including these instructions constitute only 4.3% of the CPU2006 INT traces.

The optimal scheduler uses the latencies assigned by the Sun Compiler. Latencies range from one cycle for integer arithmetic instructions to seven cycles for some floating point instructions. The scheduling experiments were performed on a Sun Fire v490 system (an UltraSPARC IV machine) running at 1.8 GHz.

To ensure that, for the traces included in the experiment, the optimal scheduler's machine model has comparable precision to the Sun Compiler's machine model, the schedule lengths computed by the two machine models were first compared for the same instruction order. When scheduling a trace, the Sun compiler computes an estimate of the schedule length based on the compiler's scheduling machine model. This schedule length is printed to the text file exported by the compiler. Before a trace is processed by the optimal scheduler, the total schedule length of the input instruction stream (the heuristic order) is computed according to the optimal scheduler's machine model and the resulting total schedule length is compared to that computed by the Sun Compiler. If the

Table I. Trace Distribution in the CPU2006 INT Benchmarks

	Max.	Avg.
Instructions per trace	424	24.0
Basic blocks per trace	60	3.2
Paths per trace	226	3.8

Table II. Overall Optimal Trace Scheduling Results for CPU2006 INT

1	TOTAL TRACES PROCESSED	42,743
2	Hard traces	20,236 (47%)
3	Hard traces solved optimally	18,459 (91%)
4	Avg. solution time per trace (ms)	45
5	Hard traces improved and optimal	11,184 (55%)
6	Hard traces improved and non-optimal	675 (3%)
7	Overall weighted schedule length improvement	2.7%
8	Overall compensation code size reduction	15%
9	Overall code size reduction	1.3%

two schedule lengths match, the trace is processed by the optimal scheduler; otherwise, it is excluded. The matching rate was 90.3%.

6.1 Trace Distribution

Table I shows some statistics for the traces used in the experiments. There were 47,320 traces in the CPU2006 Integer Benchmarks, including large traces with up to 424 instructions, 60 basic blocks and 226 paths. On average, a trace has 24 instructions, 3.2 basic blocks and 3.8 paths.

6.2 Enumeration Results

In the proposed optimal algorithm, a heuristic technique is used to find an initial feasible schedule. In the experiments of this section, the heuristic schedule produced by the Sun compiler is used. The cost function of the heuristic schedule is first computed. If the cost is zero, the heuristic schedule is optimal. Otherwise, it may be suboptimal and the trace is passed to the enumerator to search for an optimal schedule. Trace scheduling problems that are passed to the enumerator are considered *hard* problems. The enumeration time limit was set to 1 second per trace. Optimal trace scheduling results for CPU2006 INT are summarized in Table II.

Out of the 42,743 traces that were processed, 47% were hard. Hard traces constituted 63% of the static trace code size and 47% of the trace weighted schedule length based on profile feedback information. The enumerator optimally solved 91% of the hard traces within 1 second per trace. The average solution time across the traces that were optimally scheduled was 45 ms. Optimal scheduling time is studied in more detail in Section 6.6. 55% of the hard traces were scheduled optimally with an improved schedule relative to the heuristic schedule. The enumerator found improved schedules for another 3% of the hard traces even though an optimal schedule was not found within the time limit. Recall that the optimal scheduler may find improved schedules before reaching optimality.

Table III. Optimal Scheduling Results for the Hard Traces in the CPU2006 INT Benchmarks

Benchmark	Weighted Sched-Length Improvement	Comp. Code Size Reduction	Code Size Reduction
astar	2.3%	21%	1.9%
bzip2	3.1%	31%	1.8%
gcc	3.2%	13%	1.3%
gobmk	2.5%	10%	0.7%
h264ref	2.0%	11%	0.6%
hmmer	7.6%	9%	0.2%
libquantum	7.1%	20%	0.8%
mcf	6.1%	39%	5.6%
omnetpp	2.0%	35%	3.4%
perlbench	2.9%	17%	1.5%
sjeng	3.4%	13%	1.1%
xalancbmk	1.7%	18%	1.1%
Geom. Mean	3.2%	18%	1.2%

Rows 7, 8, and 9 show the overall improvements in schedule length and code size made by the optimal scheduler relative to the heuristic scheduler for the hard traces. Row 7 shows the overall weighted-schedule-length improvement, which takes into account trace frequencies of execution based on profile feedback information. The contribution of a trace to the overall weighted schedule length (optimal or heuristic) is weighted by the trace's frequency of execution. Row 8 shows the overall reduction in compensation code size. The reduction is very significant, because, as explained in this article, it is very hard for a heuristic scheduler to minimize the schedule lengths of multiple paths and, at the same time, minimize compensation code size. Row 9 shows the corresponding reduction in trace code size. Note that the reduction in static code size is independent of the precision of profile feedback information. Reduced code size is known to lead to better I-cache performance, and hence better overall performance.

Table III details the improvement in schedule length and code size for the hard traces in each benchmark in CPU2006 INT. The results in this table show that the optimal trace scheduler improves the weighted schedule length for every benchmark and significantly reduces the compensation cost for the vast majority of the benchmarks. The maximum weighted schedule length improvement is 7.6% for hmmer, while the maximum compensation code-size reduction is for mcf, where the optimal scheduler reduces compensation code size by 39%. The geometric mean of the improvement is 3.2% for the weighted schedule length, 18% for the compensation code size, and 1.2% for the total hard trace code size.

6.3 Code Size

The code-size factor of Equation 4 can be used to control the tradeoff between schedule length and code size in optimal scheduling. To study this tradeoff, optimal trace scheduling was performed using different code-size factor

Table IV. Schedule Length and Code Size Improvements as Functions of Code-Size Factor

Code-Size Factor (Cycles per Instr)	Weighted Sched. Length Imp.	Comp. Code Size Reduction	Tot. Comp. Code Reduction
0	1.97%	25%	1.8%
0.001	1.97%	28%	2.0%
0.01	1.97%	29%	2.1%
0.1	1.92%	35%	2.5%
0.2	1.50%	46%	3.4%
0.3	1.04%	58%	4.2%
0.4	0.48%	68%	4.9%
0.5	-0.95%	75%	5.4%
0.6	-1.01%	76%	5.5%

settings and the improvements in both schedule length and code size were measured for each setting. Larger code-size factors lead to looser lower bounds and consequently slower enumeration and more timeouts. Hence, to achieve a fair comparison of the different code-size factors, optimal scheduling was applied only to the hard traces that were solved optimally within 1 s per trace with all code-size factor settings. Table IV shows the weighted schedule length improvement, compensation-code reduction and total code-size reduction relative to the Sun Compiler heuristic for code-size factors ranging from 0 to 0.6 cycles per instruction.

With a code-size factor of zero, the optimal scheduler produces schedules with minimum weighted schedule length. Increasing the code-size factor biases the optimal scheduler towards producing less compensation code at the cost of increasing the weighted schedule length. For example, using a code-size factor of 0.5 cycles per instruction results in schedules with 75% less compensation code but with about 1% increase in weighted schedule length relative to the heuristic. However, examining the improvement numbers at smaller code-size factors shows that using a small enough code-size factor can further reduce the compensation code size without significantly increasing the weighted schedule length. In fact, the third column in the table shows that using a code-size factor of 0.01 cycles per instruction reduces the compensation code size by an additional 4% while still producing the minimum weighted schedule length.

6.4 Heuristics and Issue Rates

In this section, the optimal scheduler is used to evaluate the performance of two different heuristics on machines with various issue rates. The same UltraSPARC IV model is used here, but the issue rate is hypothetically varied from one instruction per cycle to six instructions per cycle to study the effect of issue rate on heuristic performance relative to optimality. The heuristics used in the experiments were critical path (CP) and successive retirement (SR). The cost function defined in this article aggregates three components: main-path schedule length, side-path schedule lengths, and compensation cost. The experiments in this section try to isolate the improvements in each of these three

Table V. Improvements of Optimal vs. Heuristic Scheduling for Different Heuristics and Issue Rates

		Critical Path				Successive Retirement			
		1-issue	2-issue	3-issue	6-issue	1-issue	2-issue	3-issue	6-issue
1	Total length imp.	0.0%	0.2%	0.6%	0.4%	0.2%	0.4%	0.9%	0.8%
2	Comp. code reduction	55%	16%	24%	27%	-6%	11%	13%	15%
3	Weighted length imp.	1.4%	0.8%	1.6%	1.5%	1.2%	0.9%	1.8%	2.1%

components for two different heuristics. Table V shows the improvements relative to each heuristic on machines with four different issue rates.

Row 3 shows the overall improvement measured by the weighted schedule length, while each of Rows 1 and 2 shows the improvement of one component of the weighted length: total schedule length (or main-path length) in Row 1 and compensation code size in Row 2. The results in this table indicate that both heuristics do well at minimizing the total schedule length but at the cost of generating excessive compensation code.

Comparing Rows 1 and 2 for the two heuristics shows that the optimal scheduler provides more improvement in total length over SR (Row 1), while it provides more improvement in compensation cost over CP (Row 2). These results are consistent with the objective of each heuristic. CP tends to minimize the total schedule length at the expense of generating more compensation code, while SR tries to reduce the amount of compensation code at the expense of producing longer schedules.

Examining the overall improvement in Row 3 shows that SR outperforms CP on a single-issue machine, while CP performs better on all other machines. The CP heuristic tends to make more upward code motions that cause side-path degradation. On wider-issue machines, CP performs better than SR because the abundance of issue slots leaves enough slots for side paths, thus limiting their degradation. The SR heuristic, on the other hand, tries to minimize side-path degradation by limiting the amount of upward code motion. Hence, SR outperforms CP on narrow-issue machines where there is a stronger competition among paths for issue slots.

Row 2 shows that the optimal schedules have significantly less compensation code than the heuristic schedules in all cases except for SR on a single-issue machine. Significant reduction in compensation-code size of the optimal scheduler relative to CP is expected because CP is not compensation-code conscious. Interestingly, the optimal scheduler also reduces compensation-code size compared to SR on all multiple-issue machines. Although SR tries to minimize compensation code size by avoiding upward code motion, it greedily schedules an instruction when the ready list is not empty. As explained in Section 4.3, this greedy strategy often results in costly upward code motions when the ready list contains only external instructions. The optimal scheduler, on the other hand, leaves an issue slot empty if the upward code motion's cost outweighs its benefit. Encountering a ready list with only external instructions is more likely to happen on wider issue machines, and that explains why the gap in compensation-code size between SR and the optimal scheduler increases as the issue rate increases. When the issue rate is one, SR produces less compensation

Table VI. Timeouts at Different Levels of Pruning

	Level 1	Level 2	Level 3
Hard traces	20,326	20,236	20,236
Timeouts	6,630 (33%)	4,726 (23%)	1,777 (9%)

Table VII. Enumerator Performance at Different Levels of Pruning

	Level 1	Level 2	Level 3
Number of hard traces	13,607	13,607	13,607
Avg. soln. time per trace (ms)	69	37	6
Avg. tree nodes per trace	11,345	4,510	327
Avg. feasible schedules per length	76.5	3.8	0.7

code than the optimal scheduler, because scheduling is resource constrained at that issue rate and that makes it unlikely to encounter a ready list with only external instructions.

6.5 Pruning Techniques

To study the effectiveness of the different pruning techniques presented in this article, enumeration experiments were performed at three different levels of pruning:

- (1) Level One: Only range tightening was applied.
- (2) Level Two: Range tightening and dynamic path lower bounds were applied.
- (3) Level Three: All pruning techniques, including history-based domination, were applied.

The heuristic in these experiments was the Sun compiler's heuristic and optimal scheduling was done for the real UltraSPARC IV machine with an issue rate of 4.

First, the number of timeouts with a 1-s limit was measured at all three levels of pruning and the results are shown in Table VI. The percentage of timeouts dropped from 33% with the first level of pruning to 9% with the highest level of pruning.

To fairly compare enumeration performance at the three levels, an enumeration experiment was performed on a set of traces that were solved optimally with all levels of pruning. The enumerator with level-one pruning was first applied to the hard problems in CPU2006 INT. Then the hard traces that were solved optimally within 1 s with level-one pruning were enumerated using the second and third levels of pruning. The results are shown in Table VII.

The number of hard traces that were solved optimally at all three levels of pruning was 13,607. Three metrics are used to evaluate the enumerator's performance: the average solution time per trace (Row 2), the average number of enumeration tree nodes that were needed to schedule a trace optimally (Row 3), and the average number of feasible schedules enumerated at each

Table VIII. Optimal Scheduling Time Relative to Total Compile Time

TIME LIMIT (S)	0.01	0.1	1.0	10
Optimal scheduling time (s)	196	586	2,629	12,979
Percentage of total compile time	1.4%	4.1%	18%	90%
Hard traces solved optimally	65%	82%	91%	95%
Weighted schedule length improvement	2.20%	2.57%	2.65%	2.74%
Compensation code reduction	11%	13%	15%	16%
Code size reduction	0.9%	1.1%	1.3%	1.4%

schedule length (Row 4). The experiment shows that each pruning technique significantly reduces all three metrics. When all pruning techniques are used, pruning of infeasible or uninteresting subtrees of the enumeration tree is done so early that, on average, only 0.7 leaves per tree are reached.

6.6 Compile Time

Optimal scheduling is more computationally expensive than heuristic scheduling. This section studies optimal scheduling time compared to the total compile time of a production compiler. The total compile time of CPU2006 INT using the Sun Compiler with profile feedback is 14,347 s (approximately 4 hours). Optimal scheduling of the hard traces in CPU2006 INT was done using four different time limits. Table VIII shows the optimal scheduling time and the corresponding overall schedule-length and code-size improvements in each case.

The results in the table show how the optimal scheduling timeout setting can be selected to keep the total compile time within the acceptable limits in a given environment. With a time limit of 10 ms per trace, the optimal trace scheduling time is only 1.4% of the total compile time, and the resulting schedule-length and code size improvements are still significant. With this setting, the optimal scheduler solves 65% of the hard traces optimally and the overall weighted schedule length improvement is 80% of the improvement obtained with a time limit of 10 s. The overall compensation code reduction with a time limit of 10 ms per trace is 69% of that obtained with a time limit of 10 s. per trace. These results show that the compile-time cost of the proposed algorithm is quite reasonable and can be controlled by setting a time limit on the optimal scheduling time of each trace.

7. CONCLUSIONS AND FUTURE WORK

This article describes an optimal algorithm for trace scheduling. The algorithm optimizes a trace's total schedule length without degrading side paths or generating excessive compensation code. The algorithm was implemented and applied to traces generated by the Sun Compiler using the CPU2006 Integer Benchmarks. Ninety-one percent of the hard traces are optimally scheduled within 1 s per trace. The optimal schedules are improved relative to the heuristic schedules in both weighted schedule length and compensation code size. The geometric mean improvement for the hard traces in CPU2006 INT is 3.2% in weighted schedule length and 18% in compensation code size. The significant difference in compensation code size suggests that existing greedy heuristics cannot simultaneously minimize trace schedule length and compensation

code size. Developing better heuristics that precisely approximate optimality in both time and space will likely involve some form of backtracking or look-ahead.

Optimal trace scheduling time can be controlled by setting a per-trace time limit. With a time limit of 10 ms per trace, optimal scheduling adds only 1.4% to the total compile time of CPU2006 INT and results in 80% of the weighted schedule length improvement that is obtained with a limit of 10 s per trace. So, applying the optimal trace scheduling algorithm of this article to a production compiler seems to be fairly practical for the highest optimization level.

The cost function introduced in this article includes a code-size factor that makes it possible to control the tradeoff between schedule length and static code size. By using a small enough code-size factor, the optimal scheduler achieves more reduction in static code size while still producing the optimal weighted schedule length.

The enumerative technique presented in this article may be extended in the future to optimally schedule more general scheduling regions, such as the nonlinear acyclic regions used in wavefront scheduling [Bharadwaj 2000]. The results of this article show that branch-and-bound enumeration is an effective technique for solving compiler optimization problems with multiple conflicting objectives. By defining the appropriate cost functions and developing the appropriate pruning techniques, the same approach may be used in the future to solve combined compiler optimization problems, such as instruction scheduling with register allocation and instruction scheduling with optimal memory-system performance.

ACKNOWLEDGMENTS

The authors would like to thank the compiler group at Sun Microsystems for permission to use the equipment and software. This work has benefited greatly from discussions with Peter Damron, the project lead of the SPARC code generation team at Sun. The experimental results presented in this article would not have been possible without Peter's valuable insight on the machine model. Great thanks go to May Alqudsi-Shobaki for her help in preparing the manuscript. Finally, the authors thank the anonymous reviewers for their constructive comments that helped improve the final article, especially the experimental section.

REFERENCES

- BERNSTEIN, D. AND RODEH, M. 1991. Global scheduling for super-scalar machines. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York.
- BHARADWAJ, J., MENEZES, K., AND MCKINSEY, C. 2000. Wavefront scheduling: path based data representation and scheduling of subgraphs. *J. Instruc.-Level Paral.*, 2, 7.
- CHEKURI, C., JOHNSON, R., MOTWANI, R., NATARAJAN, B., RAU, B., AND SCHLANSKER, M. 1996. Profile-driven instruction-level parallel scheduling with applications to superblocks. In *Proceedings of the 29th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA.
- CHOU, H. AND CHUNG, C. 1995. An optimal instruction scheduler for superscalar processor. *IEEE Trans. Parall. Distrib. Syst.* 6, 303–313.
- FISHER, J. 1981. Trace Scheduling: a technique for global micro-code compaction. *IEEE Trans. Comput.* 30, 478–490.

- FREUDENBERGER, S., GROSS, T., AND LOWNY, P. 1994. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Trans. Program. Lang. Syst.* 16, 1156–1214.
- HAVANKI, W., BANERJIA, S., AND CONTE, T., 1998. Treeregion scheduling for wide-issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, IEEE, Los Alamitos, CA.
- HENNESSY, J. AND GROSS, T. 1983. Post-pass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.* 5, 422–448.
- HWU, W., MAHLKE, S., CHEN, W., CHANG, P., WARTER, N., BRINGMANN, R., OUELLETTE, R., HANK, R., KUYOHARA, T., HAAB, G., HOLM J., AND LAVERY, D. 1993. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 229–248.
- LAH, J. AND ATKIN, D. 1983. Tree compaction of micro-programs. In *Proceedings of the 16th Annual Workshop on Micro-Programming*. ACM, New York.
- LANGEVIN, M. AND CERNY, E. 1996. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Des. Autom. Electron. Syst.* 1, 443–456.
- LINN, J. 1983. SRDAG Compaction—a generalization of trace scheduling to increase the use of global context information. In *Proceedings of the 16th Annual Workshop on Micro-Programming*. ACM, New York.
- MAHLKE, S., LIN, D., CHEN, W., HANK, R., AND BRINGMANN, R. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA.
- MUCHNICK, S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.
- NARASIMHAN, M. AND RAMANUJAM, J. 2001. A fast approach to computing exact solutions to the resource-constrained scheduling problem. *ACM Trans. Des. Autom. Electron. Syst.* 6, 490–500.
- RIM, M. AND JAIN, R. 1994. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst.* 13, 451–458.
- SHOBAKI, G. AND WILKEN, K. 2004. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA.
- SHOBAKI, G. 2006. Optimal global instruction scheduling using enumeration. PhD dissertation, Dept. of Computer Science, University of California, Davis.
- SMITH, M., HOROWITZ, M., AND LAM, M. 1992. Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York.
- SU, B., DING, S., AND JIN, L. 1984. An improvement of trace scheduling for global microcode compaction. In *Proceedings of the 17th Annual Workshop on Microprogramming*. ACM, New York.
- SUN MICROSYSTEMS, 2004. UltraSPARC Processors Documentation, www.sun.com/processors/documentation.html.
- WILKEN, K., LIU, J., AND HEFFERNAN, M. 2000. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.
- WINKEL S. 2007. Optimal versus heuristic global code scheduling. In *Proceedings of the 40th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA.
- WOLSEY, L. 1998. *Integer Programming*. John Wiley and Sons, New York.

Received March 2006; revised March 2008; accepted July 2008