EPIC Instruction Scheduling Based on Optimal Approaches

Steve Haga
University of Maryland
College Park, MD
stevhaga@eng.umd.edu

Rajeev Barua
University of Maryland
College Park, MD
barua@eng.umd.edu

ABSTRACT

This paper presents a method for instruction scheduling that considers the scheduling restrictions inherent in VLIW processors, particularly EPIC. EPIC imposes restrictions on the nature and code-order of instructions that may issue in the same cycle. To express these restrictions, the instructions are grouped into instruction classes such as arithmetic, memory, floating point and branch instructions. Allowable ordered combinations of instruction types are called *templates*.

Most existing methods for instruction scheduling do not consider templates except at their last stage. For example, trace scheduling and similar methods focus on moving instructions across basic blocks. After movement, the instructions are usually put in templates using greedy algorithms such as list scheduling. There is a significant opportunity to improve performance if algorithms superior to greedy are used for template scheduling.

The scheduling method in this paper takes the following approach. It begins with a provably optimal algorithm for template scheduling. This algorithm is not feasible, however, since it results in an exponential compile time. Two classes of methods are used therefore to drastically reduce the compile time. First, aggressive branch-and-bound techniques are used to prune portions of the search space while retaining the optimality guarantee. Second, non-optimal heuristics are used to guide the search towards more promising solutions quickly. Results show that our techniques are able to reduce the number of NOPs in integer programs by 56% on average compared to the list-scheduling based greedy algorithm in the SGICC compiler, with only a 17% increase in compile time. Floating point programs see a 35% reduction in NOPs with a 22% increase in compile time.

Keywords

EPIC, templates, instruction scheduling, IA-64

EPIC 1 '01 Austin, Texas USA

1. INTRODUCTION

This paper presents a new approach for scheduling the instructions of a basic block in VLIW machines that impose constraints on instruction placement. These constraints are represented by a set of allowable *templates*. Each template specifies one allowed sequence of instruction types (integer ALU, branch, floating point, etc.). Fixed-width VLIWs employ a single template, while EPIC architectures [15, 2] allow instructions in multiple templates to execute simultaneously. Code scheduling under template constraints is an important but difficult problem because poor schedules require many extra NOP instructions. Adding NOPs wastes resources, impacting performance, power, and memory requirements.

The approach described contains several innovative optimizations that improve code density, where code density is defined as the ratio of the number of useful instructions to total instructions, where total instructions include useful instructions and NOPs. First, we present a provably optimal algorithm to perform instruction scheduling. Second, since the optimal algorithm is exponential time, *branch and bound techniques* [5] are used to drastically reduce the compile time; these techniques make use of a novel upper bound for the code density of a basic block. Third, an innovative, constant-time method for optimally selecting templates for a set of parallel instructions is presented. Fourth, non-optimal heuristics are applied to the larger basic blocks, where optimal scheduling may take too long.

The success of these techniques is shown in the results. For integer benchmarks, 56% of NOPs are removed, as compared to the SGICC compiler for the IA-64 [14, 1], at a cost of 17% more compile time. In floating point benchmarks, 35% of NOPs are removed, as compared to the existing compiler, at a cost of 22% more compile time. Our results are produced by modifying the SGICC compiler.

A new instruction scheduling method is needed for EPICs because traditional instruction scheduling algorithms are ineffective for template-constrained systems. Well known techniques based on trace-scheduling [7, 12, 11, 4] focus on ways to move instructions across basic blocks, but say nothing on how to assign and fill templates. Instruction scheduling, an NP-complete problem [13], is usually done either by *list scheduling* [13], or by some technique which considers the resource constraints of the machine. No current method considers the template restrictions of EPICs prior to the ordering of instructions; thus, extra NOPs may be produced. The cause is that template choices are correlated: choosing to schedule an instruction limits the set of potential templates available to for the next choice. Further, the choice

M	I	I
M	I	I
M	I	I
M	I	I
M	L	X
M	L	X

M	M	I
M	M	I
M	M	I
M	M	I
M	F	I
M	F	I

M	M	F
M	M	F
M	I	B
M	I	B
M	B	B
M	B	B

B	B	B
B	B	B
M	M	B
M	M	B
M	F	B
M	F	B

Figure 1: A list of all available templates for IA-64. Dark vertical bars represent stop bits.

of instructions to schedule on this cycle can easily affect the quality of the solution for scheduling on the next cycle.

An outline of the rest of the paper is as follows. In section 2, EPIC and IA-64 are described. In section 3 is a motivational example of the dangers of not considering templates when ordering instructions is presented. In section 4, related works are examined. In section 5, optimal approaches are considered. These include schedules for an instruction group, as well as an entire basic block. Branch and bound techniques are presented, to reduce compile-time. In section 6, non-optimal heuristics to further improve run time are presented. In section 7, the performance of the approach is evaluated. In section 8, conclusions are presented.

2. EPIC TEMPLATES

EPIC [15] architectures are a class of VLIWs that introduce many new features, including variable-length parallelism. While much of our approach is applicable to any VLIW, our research focuses on EPICs, where variable-length parallelism is achieved by the mechanism of *stop bits*. Conceptually, after every instruction there is a stop bit that may be set. These stop bits form the boundaries of parallelism; a sequence of instructions without any set stop bits between them are said to belong to the same *instruction group*. EPIC templates specify the stop bit placement, as well as the allowed combinations of functional unit types. A set of instructions that have been mapped onto a template are called a *bundle*. The bundle does not represent one instruction group. Instead, the stop bits indicate the instruction groups, and there may be several stop bits inside of one bundle, or several bundles between stop bits.

Since our algorithms are tested for IA-64 [9], we now consider the IA-64 ISA, the EPIC architecture of greatest practical interest. IA-64 uses bundles of three instructions and provides 24 templates, as shown in figure 1. It defines four functional units: Integer, Memory, Floating Point, and Branch; and six instruction types. Four of these types (I, M, F and B) indicate instructions which must be executed on the corresponding functional unit. The two other instruction types are A (which may be called a *super type* because it may execute on either an I or an M) and LX (which fills 2 bundle slots and uses the I and B functional units). The current implementation of IA-64, Itanium [10], imposes additional constraints; when an instruction group violates these constraints, Itanium splits it into two cycles.

3. INSTRUCTION ORDERING WITHOUT REGARD FOR TEMPLATES

An example is now presented, which highlights the need to consider templates when scheduling instructions. Current basic block scheduling methods do not consider templates until after assigning instruction groups. Of these methods, list scheduling is the simplest to describe; and it is the approach used by SGICC– the compiler we use and compare results against. Therefore we consider list scheduling in the following example. The resulting schedule, however, with its poor fit to the templates, is typical of all current methods.

Consider the example Directed Acyclic Graph, DAG, of figure 2(a), showing insructions and their dependencies. List scheduling selects one instruction out of those that are ready to schedule *i.e.,* whose dependencies are met, based on a user-specified priority function; for SGICC, it is the height in the dependence DAG. Figure 2(b) shows that, with the wrong heuristic, list scheduling produces three NOPs for six instructions, using the IA-64 templates, where as the optimal schedule in figure 2(c) has no NOPs at all.

List scheduling results in figure 2 as follows. In the first cycle, there are two ready instructions: M1 and I2. List scheduling sees that M1 is used earlier and schedules it first. Then it must decide whether to end the instruction group or to schedule I2 also; it schedules I2 to achieve parallelism. Having no more possible instructions, a stop bit is inserted. The dependencies are updated and there are now three ready instructions: M3, M4, and M5. M3 has the earliest deadline, and so is scheduled next. It is an M-type instruction, where as the the IA-64 templates are such that a stop bit after the second slot forces the third slot to be an I type. Therefore the first bundle is given a NOP, and a new template is begun for M3. M4 is scheduled next. Then M5 is tried. since it does not fit into the template, that template's last slot is filled by a NOP. It decides to delay M5. On the next cycle M5 and F6 are both ready and both can be scheduled. The optimal solution shown in figure 2(c) uses one less template.

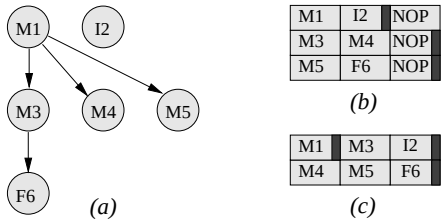

Figure 2: Scheduling example. (a) the DAG (b) the result of scheduling without considering templates: (MI;I MMI; MFI;) (c) an optimal schedule: (M;MI; MMF;)

4. RELATED WORK

The term instruction scheduling is often applied to methods which assign basic blocks to instructions; these methods are not to be confused with instruction scheduling within a basic block, such as in our research. The purpose of these basic-block assigning techniques is usually to reduce the run time by moving code, when possible, into less frequently used blocks, or to perform if-conversion, so as to increase the available parallelism. One technique, *trace scheduling* [7], constructs *traces*, which are identify as a sequence of basic blocks where the next block in the sequence is a successor of the current block, and where there is a very high probability that if the first block is entered, the entire trace is executed. Since the probability of this entire path occurring is high, the trace may be optimized, even if it causes less-frequent blocks

outside of the trace to run more slowly. Computation that is not used in the trace is attempted to be moved outside of the trace, a form of instruction scheduling, across basic blocks. *Superblocking* [12] and *hyperblocking* [11] are techniques based on trace scheduling. An alternative approach, *wavefront scheduling* [4], is suited for machines that allow speculation and predication, such as EPIC. It must be understood that all of these methods are performed prior to instruction scheduling within basic blocks. Our technique is not meant to replace these existing methods, but to work with them. We later describe how our method may, in fact, be integrated with trace-scheduling based methods.

Concerning scheduling instructions within a basic block: list scheduling, described in section 3, has proven sufficient for superscalars– they have a reorder buffer to correct bad schedules and no template constraints. VLIWs have motivated more advanced techniques that consider the resource constraints of the system. One such technique, [8], which targets embedded processors, creates a DAG of pairwise edges that represent instructions which are mutually schedulable. Instruction scheduling is then performed by finding a minimum-cost clique covering for the DAG. Another possible approach, [3], schedules instructions based on Finite State Automata methods. The dependencies of each instruction are represented as a bit-vector. If the AND-ing of two bit vectors yields an empty vector, then the two corresponding instructions do not share any resources, and may be scheduled together. Given that a particular set of instructions has been scheduled in a certain cycle, the OR-ing of their bit vectors represents the resources currently used. To construct the finite state automata, a bit vector of resources used by a set of instructions represents a state, and choosing to schedule an additional instruction on this cycle represents a transition to a new state.

Neither of these methods address templates, nor may they be easily modified to this end, for the following three reasons. First templates impose variable resource restrictions, where as the methods of [8],[3] rely upon the constant resources available in fixed-width VLIWs. Instruction groups may begin in different slot positions, resulting in different resources constraints. For instance, the issue window of Itanium is six instructions if the instruction group begins in slot the first slot, but is only four instructions, if the group begins in the third slot. Second, the methods of [8],[3] only consider whether individual pairs of instructions are schedulable, where as template restrictions apply to the entire instruction group. For example, in an Itanium instruction group, not more than $\frac{1}{3}$ of the instructions may be of F type and not more than $\frac{2}{3}$ of the instructions may be of I type (from figure 1). Because the template constraints are relative to the entire instruction group, comparisons of instruction pairs are insufficient. Third, these template constraints are loose. NOP instructions may be used to meet the constraints. And satisfying all of the relative constraints does not guarantee a match to the templates. for instance, an instruction group with one F and two I types meets the IA-64 constraints, but there is no IIF template.

In [6] a template-aware method of instruction scheduling for the Itanium implementation of IA-64 is presented, based on Integer Linear Programming (ILP) and dynamic programming. The resource constraints of the Itanium processor are formulated into an ILP problem, which is solvable by commercial software. The solution of the ILP problem identifies the instruction groups of the final code, but does not consider templates. Fitting these instruction groups into templates is not simple, because the template selection depends on the starting slot position for the instruction group. To overcome this difficulty, the solutions for each possible starting and ending slot position are found. Dynamic programming then determines the globally best schedule. This approach often uses the provably fewest number of execution cycles for an Itanium. This claim is not strongly supported, however, since it depends on the validity of the conditions of the ILP model; one such condition is that the Itanium's issue width is 6 instructions, when in fact, the issue width is less for some starting slot positions. While they found that resource violations are uncommon, their benchmarks do not include floating point or highly parallel code, which are more likely to cause such a violation– an instruction group of just two floating types has a violation if not begun in slot 0, and an instruction group of 5 or 6 instructions may not fit into Itanium's variable-width issue window.

There are four fundamental differences in the problem being solved by our work and that solved in [6]. First, we improve the code density and the execution time, while [6] primarily improves the execution time, and secondarily, the code density. Second, we accomplish this by considering the templates at every step of instruction scheduling; in [6], templates are not considered until after instruction groups are assigned. Thus the situation illustrated in figure 2(b) is still possible with [6], since the schedule of figure 2(b) does use the minimum number of cycles, despite extra NOPs. Indeed, [6] does not claim to minimize the number of NOPs, but only the number of execution cycles. Third, our approach currently considers the general IA-64 instruction scheduling problem (*i.e.,* no issue width limits), but [6] considers the highly-constrained Itanium implementation. Indeed, if [6] is modified to solve our general problem, it simply chooses some random partitioning of instruction groups which uses the fewest number of cycles; in contrast, our technique also chooses a partitioning of instruction groups which executes in the fewest number of cycles– but not randomly, rather, so as to minimize the number of NOPs. Fourth, the extension of our approach to consider Itanium's limitations, as discussed in section 5.3, is straight-forward and would guarantee a minimum cycle solution, as well as a minimum number of NOPs; however, it is not clear how template considerations could be formulated as an ILP problem such as in [6]. Thus our methodology, when modified to consider real-machine constraints will also guarantee a minimum number of cycles– and a minimum number of templates as well, which [6] cannot be modified to do. The fact that our approach, in its current form, is not bound to implementation constraints, has the positive benefit that we can study the general IA-64 problem, such as what portion of NOPs is imposed by the ISA, regardless of the implementation constraints, allowing us to uncover what properties of the IA-64 templates account for this behavior.

Our work and [6] are not in conflict. If it is found that their approach is quicker (a decision which requires more performance data on their technique), it could be employed first, and our technique only employed for those basic blocks for which [6] yields low code density.

5. OPTIMAL BASE ALGORITHM

In this section, we consider optimal methods to schedule

a basic block. In section 5.1, we examine what properties an EPIC must have in order to be applicable to our methods. In section 5.2, we develop an optimal algorithm to select the templates for one instruction group. In section 5.3, we present a recursive search algorithm to find the best schedule of a basic block, by means of trying all possibilities. In section 5.4, we improve the compile-time of the search algorithm, through branch and bound techniques. In order to further reduce compile times, section 6 considers non-optimal heuristics.

5.1 Assumptions

We first discuss our optimal base algorithm, which begins with certain assumptions. It is assumed that whatever EPIC architecture is being considered has the property of *paired templates*. Templates are paired if, for each stop-bit contained within any template, there is another template which is identical, except that it does not have that stop bit. The IA-64 is an EPIC which has paired templates, as can be seen by examination of table 1. Assuming paired templates, we arrive at an obvious but important optimization which is used in our algorithm:

THEOREM 1 (OPTIMALITY WITHOUT SPLITTING). *If all templates are paired, then the code density of an optimally scheduled instruction group never improves by dividing that instruction group into two instruction groups, separated by a stop bit, and optimally scheduling each.*

PROOF. Suppose an instruction group, Z is partitioned into two instruction groups, A and B. A is optimally scheduled into the available templates, and a stop bit is inserted after the last instruction of A. B is now optimally scheduled, starting at the first slot after the stop bit, if it occurs in the middle of a template. The resulting schedule may be written as "A;B;", where stop bits are indicated by a ";".

Regardless of how many instructions are contained in A or B, there is only one template containing the ";" that lies between A and B. Replacing this template by its template pair yields the schedule "AB;", which contains only valid templates, since "A;B;" contained valid templates and only template has been changed, and that in a valid way. Since "AB;" has no internal stop bits, it represent a single instruction group, and since A and B partitioned Z, "AB;" represents a valid schedule of Z. There may exist even better schedules for Z, but at least we have identified one schedule which is as good as the best possible solution with partitioning. Thus, partitioning an instruction group never reduces the number of bundles needed to optimally schedule it. □

It is desirable to not sacrifice execution time in the interest of code density, which is facilitated by our assumption of a machine without constraints. Since we currently target a general IA-64 machine under the assumption that instruction groups of arbitrary size issue within a single cycle, a solution which requires the minimum number of cycles may be guaranteed if all instructions are scheduled by their deadlines. Since only minimum cycle solutions are considered, the best measure of the quality of a schedule is its code density. Extension of the algorithm to real machines is discussed in section 5.3.

5.2 Instruction Group Template Selection

For the first time, we show that certain EPICs have properties that allow for constant-time, *i.e.,* O(1), selection of optimal templates for an instruction group. This is a noteworthy, in that one might expect that the list of instructions must be traversed in order to be scheduled, in order to place them into templates. O(1) time is achievable, none-the-less, because it is found that optimal template selection may be performed without checking each instruction. All that is required is to know the *number* of instructions of each instruction type. If these numbers are not known, the additional cost of counting these is linear; however, for the bounding methods based on constant-time template selection, which are presented later, there is no need to count the instructions, and the algorithm is truly constant time.

Three properties are identified, which an EPIC must have in order to be guaranteed to have optimal template selection in constant time. First, a deterministic template choice for bundles with an internal stop bit. When an instruction group finishes in the middle of a bundle, the subsequent instruction group begins in the next slot of that bundle, which is to say that a template with an internal stop bit is used. A deterministic template choice means that, given the instructions already scheduled into the bundle prior to the stop bit, there is only one choice for the remaining instruction types of the bundle; otherwise, different algorithms would be needed for each starting slot. From table 1, it is seen that IA-64 has only one template pair for each internal stop bit location. Second, super types must not partially overlap. For instance IA-64 has one super type, 'A', which represents those instructions that may be scheduled into either an M or an I functional unit. An overlapping super type (which would prevent quick selection if it existed) could be for instructions that schedule in either the I or F functional units; it overlaps because it also maps to I. When super types do not partially overlap, there is never ambiguity about which type to choose. Third, ideally the architecture should not allow intra-dependencies within an instruction group; otherwise the instructions may not be schedulable to the chosen templates. IA-64 and most architectures break this condition. It is only a technical condition, however, since no instance is found in our testing, where the instructions could not be mapped to the chosen templates. For the remaining, we loosely speak of IA-64 as meeting the above criterion.

The template selection algorithm, as it pertains to IA-64, is shown in table 1 and is now briefly described. It takes as input: *slot*, the starting slot position for this instruction group, which depends on where the previous instruction group finishes, and $TypeCnt$, an array which contains the counts of every instruction type within the instruction group. If the previous instruction group did not end on a bundle boundary, the selection begins by filling in the remaining slots of the first bundle according to the predetermined template. Slots are filled by choosing and removing instruction types from $TypeCnt$, which is accomplished by the DEC function, to be described momentarily. Once this first bundle is complete, the basic approach is to first remove the types with the fewest choices. Glancing at table 1, we see that all LX instructions have only choice, MLX. So we choose as many MLX templates as their are LX instructions, remove as many M instructions, and then solve for the remaining types. Next, the F types are considered. The B types are considered last, since B instructions only appear at the end portion of bundles.

Table 2 describes DEC, the decrementing function used by the algorithm in table 1. When there are instructions of

more than one type which may fill the same template slot, *i.e.,* when there is a choice between a type and a super type, always choose the instruction from the lowest available type that matches the required functional unit). When there is no appropriate instruction to fill the slot, a NOP is used.

5.3 An Optimal, Recursive Algorithm

The previous section described how to schedule a single instruction group optimally; in this section, we build on this and show how to optimally schedule the entire basic block. Realistically, finding the optimal schedule is impractical, but it is studied because it serves as a good basis for an algorithm that is modified to be feasible. To find the optimal solution, all possible schedules are examined and the best solution found is chosen. We accomplish this exhaustive search by a recursive algorithm, shown in table 3. At each step of the recursion, all possible instruction groups are chosen from among the set of ready instructions. For each choice, a recursive call finds the best schedule of the remaining code.

This algorithm introduces a number of variables to describe various sets of instructions, which we now describe. At any point in the scheduling algorithm, there are some instructions that remain to be scheduled, denoted by the set U. A subset of U, those instructions that are ready to be scheduled (*i.e.,* whose operands are available), is denoted by R. Finally, among R there are again two types: those instructions which must be scheduled this cycle, because their deadlines have been reached, denoted by the set M, and those instructions which may be delayed because they are not yet on the critical path, denoted by the set S. When choosing instructions to schedule in this cycle, only combinations of S are examined, as the members of M must be chosen. Table 3 also introduces the variables C: all instructions chosen from S, P: all instructions chosen for the current instruction group ($P = C \cup M$), and $LeftTypes$: an array that tracks the types of the remaining instructions.

Since table 3 employs an exhaustive search of all possibilities, our methodology is easily extended to provide minimum execution time guarantees, even for real machines. Referring to table 3, when choosing sets of C, we might restrict ourselves to only choices of C which do not produce hazards. In fact, these additional constraints actually serve to reduce the search space, thus allowing our algorithm to run faster. This is an area of future research.

While our current algorithm is only guaranteed to produce minimum cycle code for a non-constrained machine, It is still likely to have fair performance even on an Itanium. This is because code density is proportional to the real performance, since better code density means a smaller I-cache demand, and since better code density effectively increases the machine issue window (by not filling it with NOPs).

5.4 Branch and Bound to Reduce Compile-Time

The above algorithm is too slow, so we present algorithms to speed it up. Some retain optimality, but others do not; this section describes branch and bound techniques that retain optimality. Branch and bound techniques reduce the execution time of an exhaustive algorithm, while retaining the optimality guarantee, by skipping, or *pruning*, those parts of the search space that are known to not contain the best solution. For example, if the current, partial solution has a cost at least as large as the best, complete solution found so far, then regardless of how the remaining instructions are scheduled, they cannot produce a better schedule than the one already found.

From this discussion it is clear that a search order which considers better solutions earlier enables more bounding, thus reducing the search space. Thus, the searching order of the various combinations of S, those instructions that are ready but delayable, greatly affects the compile time. Our philosophy is that a search examining far different solutions early is preferable to one that considers similar solutions first. To, accomplish this we create a bit-vector with each of its bit positions representing one element of S. This vector may have any value in the range from 0 to $2^{size(S)} - 1$. Each value corresponds to a unique choice of S. We use a pseudo-random shift register to change the values of this variable, thereby trying combinations in a pseudo-random order, but also guaranteeing that no solution is tried twice (a property of the pseudo random shift register).

More bounding is possible with a good lower bound on the cost of a basic block or of that portion of a basic block which remains unscheduled. A key insight provides this lower bound:

THEOREM 2 (O(1) COMPUTATION OF LOWER BOUND). *Let U be the set of all remaining-to-be-scheduled instructions. Create an instruction group, U', containing all elements of U, but ignoring the dependencies between the elements. The cost* i.e., *minimum number of NOPs of scheduling U', Cost (U'), serves as a lower bound on the cost of scheduling U, Cost(U).*

PROOF. Consider all possible schedules of U, and choose the best, B. Define another schedule, B' that is identical to B, except that all stop bits are removed. $Cost(B') = Cost(B)$, because each has the same number of templates and the same number of NOPs in those templates. By the property of paired templates, all of the templates of B' are legal. And by theorem 1, B' is a valid schedule of U'. Since the optimal schedule of a group, such as U', is at least as good as any particular schedule, such as U, we arrive at: $Cost(U') \leq Cost(B') = Cost(B) = Cost(U)$. □

Three pruning strategies are presented in table 4. First, $COST\_PRUNING$ uses theorem 2 to find a lower bound on th cost of the current solution, even before it has scheduled all instructions. If the cost of scheduling up to the current point plus the lower bound cost to schedule the remaining instructions is greater than or equal to the best solution found so far, then this path is known to be hopeless and is pruned. The second strategy, $FILLABLE\_NOP$, looks for a template containing a NOP that could have been filled with a delayed instruction, i. It can be shown that the solution where i is included is guaranteed to be at least as good as the solution without it. The third strategy, $SPARSE\_TEMPLATE$, identifies if any selected templates contains a single instruction, where that instruction is delayable. It can be shown that the solution where this instruction is delayed is guaranteed to be at least as good as the current solution, allowing additional pruning.

Where as the methods described in table 4 identify *bad* choices and thus skip them, it is also possible to identifies *good* solutions and quit early (without having to consider the remaining combinations of C. If a given solution is provably optimal, *i.e.,* it has a cost equal to the lower bound,

```
FIND_TEMPLATES(TypeCnt, slot)          // Inputs: # of instructions of each type & starting slot
  enumerate (A, I, M, F, L, B)         // Let A=0, I=1, M=2, F=3, L=4, B=5

  if (slot = 1)                        // To start in slot 1 is necessarily to use template M;MI
    DEC(TypeCnt, M, 1)                 // No choice in this case. 2nd position in M;MI is M
                                       // The decrement function is described in table 2
  if (slot > 0)                        // The only possible templates are M:MI and MI;I
    DEC(TypeCnt, I, 1)                 // In both of these cases, the 3rd position is an I

  DEC(TypeCnt,M,TypeCnt[L])            // Only one template for LX instructions: MLX

// At this point, only A,I,M,F,B remain
  if (TypeCnt[I]≥ TypeCnt[M])          // For Fs, choose between MFI and MMF
    DEC(TypeCnt, I, TypeCnt[F])        // Choose all MFIs
    DEC(TypeCnt, M, TypeCnt[F])
  else
    ExtraM = TypeCnt[M] − TypeCnt[I]
    if (2 * ExtraM ≥ F)                // Check for all MMFs
      DEC(TypeCnt, M, 2*TypeCnt[F])    // Choose all MMFs
    else
      DEC(TypeCnt, M, 2*(ExtraM/2))    // Remove an even number of M types
      TypeCnt[F]− = ExtraM/2           // With MMF, 2 M's for each F

      DEC(TypeCnt, M, TypeCnt[F])      // Use MFIs for those above the limit
      DEC(TypeCnt, I, TypeCnt[F])
      TypeCnt[F] = 0                   // Finished with F
    end else
  end else

// At this point, only A,I,M,B remain
  if (TypeCnt[I] > TypeCnt[M])         // Are there more Is or Ms left?
    Big = I; Sml = M
  else
    Big = M; Sml = I

  gap = (2 * TypeCnt[Sml] − TypeCnt[Big])/3  // Find how many MII-MMI pairs
  DEC(TypeCnt, M, 3*gap)               // Each MMI-MII pair has 3 Ms
  DEC(TypeCnt, I, 3*gap)               // Each MMI-MII pair has 3 Is

// Now, TypeCnt[Big] ≥ 2 * TypeCnt[Sml]
  DEC(TypeCnt, Sml, TypeCnt[Big]/2)    // Fill Big as biased as possible (MMI for M & MII for I)
  TypeCnt[Big]=0

// At this point, only A and B remain
  TypeCnt[A] = 0)                      // MII += NumOfInt[A]/3, since As can be either M or I
  update slot                          // indicate the slot where this instruction group ends

end   // B types naturally go at the end. Trailing NOPs will fill out any remaining slots in the last
      // bundle. The next instruction group is able to overwrite these NOPs
```

Table 1: Optimal Template Selection Algorithm

```
DEC(TypeCnt, Type, Num   // This function decreases the count of the given instruction type by Num.)
                         // A is a super type for either I or M. The strategy is to first decrement the
                         // counter of the more restrictive type, until empty, before removing from A.
  if (Num < TypeCnt[Type])                // See if there are enough of Type
    TypeCnt[Type]− = Num                  // Decrease the count for this type
    return
  end if

  Num− = TypeCnt[Type]                    // There are not enough of the requested Type,
  TypeCnt[Type] = 0                       // but remove as many as can be done
  if ( (Type = I) or (Type = M) )         // The I and M types can be filled with A types
    if (Num < TypeCnt[A])                 // See if there are enough A types
      TypeCnt[A]− = Num                   // Decrease the A count
      return
    end if
    Num− = TypeCnt[A]                     // Remove all of the A types
    TypeCnt[A] = 0
  end if
  Insert #Num NOPs into remaining templates   // No instruction of the right type found, use NOPs
end
```

Table 2: The DEC function called by FIND_TEMPLATES

```
SCHEDULE_RECURSIVE(U)                     // U = the set of not-yet-scheduled instructions
  ADVANCE_CLOCK(Clk)                      // Advance the clock
  define Best = MAXINT                    // Initially, no best solution
  if (U = ∅)                              // See if finished
    return 0
  define R = READY(U)                     // The set of ready-to-schedule instructions
  define S = NOT_YET_CRITICAL(R,Clk)      // All R which could be delayed
  define M = R − S                        // All R which must be scheduled this cycle
  for each C combination of elements of S // (including C = ∅)
    P = C + M                             // C+M = this instruction group
    CurTypes = COUNTS(P)                  // Counts number of instructions of each type in P
    (Ccost, T) = FIND_TEMPLATES(CurTypes) // Finds cost of current, and the templates used
    LeftTypes− = CurTypes                 // Update the # of each type, still left unscheduled)
    (Ucost, x) = FIND_TEMPLATES(LeftTypes) // Ignoring dependencies, find minimum cost for U
    if ( not PRUNE(T,Ccost,UCost,S,C))    // Only explore reasonable choices
      Rcost = SCHEDULE_RECURSIVE(U-P)     // Cost of rest (U-P)
    cost = Ccost + Rcost)
    if (Best > cost)                      // Is this solution the best so far?
      Best = cost
    if (Best = Ucost)                     // Is this a minimum-cost solution?
      return Best
    end if
  end for
  return Best                             // All possibiltities have been explored
end
```

Table 3: Optimal Instruction Scheduling Algorithm

```
PRUNE(T,Ccost,Ucost,S,C)                          // Inputs: templates used, current cost, rest cost bound, the set of ready-
                                                  // but-delayable instructions, and the set of instructions chosen from S
   define PartialCost = Cost of the partial solution of those already scheduled (everything except P ∪ U)
   define L = S − C                               // The set of instructions that are ready, but not chosen

// COST_PRUNING: rejects solutions which are more expensive than the current best
   if (PartialCost + Ccost + UCost ≥              // Performs true branch and bounding of any path
       BestCompleteSolutionSoFar)                 //     known to take longer than a previous solution
          return 1

// FILLABLE_NOP: looks at the chosen templates for a NOP that could have been filled
   if (∃ an instruction, i ∈ L and a NOP, n ⊂ T,  // Checks whether the alternative solution C = C + i
       such that i could have filled the slot of n) //     is better
          return 1

// SPARSE_TEMPLATE: looks for templates that contain only one instruction, where that instruction is able to be delayed
   if (∃ an instruction, i ∈ C and a template, t ∈ T, // Checks whether the alternative solution C = C − i
       such that SCHEDULE_THESE(C-i) = T-t)       //     is better (Doesn't actually call SCHEDULE_THESE)
          return 1
   return 0
end
```

Table 4: Pruning techniques that run in O(1) time

then there is no need to search for a better solution. This optimization is indicated in table 3 by means of the early return from inside the loop.

6. NON-OPTIMAL HEURISTICS TO REDUCE COMPILE-TIME

Non-optimal heuristics are also considered. While branch-and-bound greatly reduces the optimal search space, the compile time for some basic blocks is still unmanageable. In order to reduce the compile time further, non-optimal approaches must be used. Such approaches all operate on the same premise: reduce the compile time by only searching a small portion of the solution space, without guaranteeing that a good solution is found. We use such heuristics only for larger basic blocks, which have already taken a long time to compile, or for those that are longer than a certain threshold. Therefore, most small basic blocks, and some easy-to-schedule larger ones, are scheduled optimally.

In this section, we consider a mixed-bag of such heuristics, the first of which is basic block splitting. Large basic blocks require the longest time to compile, because the number of potential schedules grows exponentially with basic block size. Therefore, it makes sense to break large blocks into smaller pieces that can be optimized separately. Among various partitioning heuristics, we choose the simplest: using the ordering provided originally by SGICC's list scheduler, the basic blocks are split into pieces of the maximum allowed size. In the results section, we evaluate several possible values for the maximum allowed block size, so as to determine the best.

The second heuristic prevents scheduling instructions on certain, non-promising cycles. We only consider those cycles which correspond to the deadlines of any instructions, unless those instructions are schedulable on another selected cycle.

The third heuristic is to limit the for-loop of the base algorithm in table 3. If the statement: "for each C combination of elements of S" is replaced by: "C = all elements of S" then we have an eager scheduling algorithm. If it is replaced by: "$C = \emptyset$" then we have a lazy scheduling algorithm. (All of the instructions are still scheduled, because once their deadlines are reached, the instruction goes into set M instead of S.) Both the eager and the lazy algorithms are very fast, because they only examine one possible solution each. It is more interesting to replace the for statement with: "for each C = either {all of S or $\emptyset$}" In this case, exactly 2 solutions are examined at each level of recursion. Thus the number of searched solutions is in the order of $2^{\#\ of\ allowed\ cycles}$, which is potentially large, but not nearly as large as the original for-loop. We call this approach eager-lazy, because the final solution is an amalgam of eager scheduling for some instruction groups and lazy for others. Our results show that eager-lazy is highly effective: it is able to find much better solutions than either eager or lazy, with only a slightly longer compile time than either.

An alternative to eager-lazy is our fourth heuristic; while eager-lazy examines a binary tree subset of the entire search tree, this method searches the entire tree in a non-optimal manner, by allowing a certain amount of performance loss in the solution. For instance, if the current solution is found to be nearly optimal, then the algorithm might decide that it is good enough and quit. The issue here becomes, what formula to use for the allowed performance loss. In our implementation, the allowed performance loss is based on the slack in the basic block, and the length of time spent in the for loop. The slack of an instruction is defined as the number of allowed cycles on which it could possibly be scheduled and still maintain dependencies and minimum solution cycles; the slack of a basic block is defined as the product of the slacks for the individual instructions. Table 5 details our algorithm for calculating allowed performance loss. This method includes an experimentally determined *multiplying factor* that is discussed in our results.

Our fifth heuristic is a timeout switch. If any basic block, after splitting, takes more than a specified number of seconds in order to compile, then the search simply terminates, and the best solution found thus far is chosen. The selection of the cut-off time is discussed in the results.

Applying the Method Across Basic Blocks Future research may evaluate how this method can be integrated into instruction scheduling across basic blocks. It is possible to modify the algorithm to schedule a trace or a superblock,

```
FIND_ALLOWED_PERFORMANCE_LOSS()          // Finds the number of NOPs tolerated in the solution
   if (iteration# in the current for loop <size(S))   // If the search space has not yet been explored well
      return 0                                        // 0 means no performance loss is allowed
   if (size(S < 4)                                    // Be optimal for small search spaces
      return 0                                        // 0 means no performance loss is allowed
   return itteration# * (log10(Slack)/3 - 1)*         // Slack grows exponentially, so use logarithm.
            MultiplyingFactor)/(size(S))              // Allow more performance loss as the algorithm goes along
end
```

Table 5: Allowed performance loss algorithm

just as the current method schedules a single basic block. In this approach, instructions are moved across basic blocks, in order to improve the code density, and as a by-product, uncovering multi-way branch opportunities. Such an approach does not preclude code motion for other reasons as well.

7. RESULTS

Our algorithm is implemented into the SGICC (v 0.11) research compiler for IA-64. SGICC employs list scheduling with an earliest-deadline-first priority function. Templates are only considered at the last stage, after the order of instructions is already determined. Hence, when the current instruction does not fit into the next slot, a NOP is inserted. Stop bits are inserted whenever the current instruction has a dependency with any instruction already in the instruction group. This is a greedy strategy. SGICC (v 0.11) is run with full optimization. For instruction scheduling across basic blocks, it uses hyperblocking [11], a technique that converts control flow into predication.

Since our algorithm is inserted after the existing instruction scheduling phase of an existing complier, we are occasionally able to take advantage of SGICC solution. Since the list scheduling solution is generated prior to our algorithm, we are able, using theorem 2, to compare the existing solution to our lower bound. If the list-scheduling solution is found to be optimal, our algorithm is skipped. Otherwise, our search algorithms are applied, but the original solution is retained, in case it turns out to yield a better result.

Performance is measured by the number of static NOPs. Static, rather than dynamic NOPs are used, because our algorithm does not currently use consider block frequencies, so the dynamic NOP count would make comparing compile time to performance difficult. Moreover, it is unlikely to be very different. The rational for measuring the results in terms of NOPs is given in section 5.3.

Before running experiments, three parameters from Section 6 must be empirically determined: the size at which to split a basic block, the multiplying factor in determining the amount of allowed performance loss, and the allowed search time. To this end, the equake benchmark was chosen for evaluation. Parameter values are found by the following four steps. First, initial guesses are made for the parameters. Second, one parameter is chosen, and is swept through a range of values, while the other parameters remain fixed. Third, the best value is chosen based on its performance in reducing code size, and its compile-time cost. More emphasis is placed on performance than cost. Fourth, this parameter is updated to the chosen value, and the process is repeated for the next parameter. Table 6 presents the results of the process. It is interesting that only a small percentage of basic blocks time-out, even with a one second cut off. Further, these blocks tend to continue running for a long period of time, if allowed to, without yielding much improvement. Therefore using the cut-off heuristic is worthwhile.

Having determined these parameters, experiments are run on eight SPEC benchmarks, described in table 7. Five of the benchmarks are integer and three are floating point.

Parameter under study	Parameters used in the test: Split Size	Multiply Factor	Allowed Time	Best Value Found
Split Size	*varied*	10^{-3}	60 sec	35
Multiply Factor	35	*varied*	60 sec	10^3
Allowed Time	35	10^3	*varied*	1 sec
Final choices:	35	10^3	1 sec	

Table 6: Choosing Parameters. With initial guesses of 10^{-3} for the multiplying factor, and 60 seconds for the allowed time, the split size is evaluated. A value of 35 is found to yield the best result. Next, the multiplying factor is varied, with the split size now set to 35. The chosen parameter values are shown.

Figure 3(a) shows a comparison of the improvements obtained by three of our methods, as compared to the original SGICC result. Improvement is measured as the percentage of NOPs removed, $1-\#NOPS_{final}/\#NOPS_{original\ SGICC}$. Figure 3(b) shows a comparison of the corresponding compile-time costs. The cost is the percentage increase in compile time, $compile\ time_{original\ SGICC}/compile\ time_{final}-1$. The three schemes are eager, lazy and eager-lazy searching methods. The eager method schedules every instruction that has its dependencies satisfied, while the lazy method does not schedule any instructions that have not reached their deadline. These two approaches do not require any searching for a best solution, since each yields only one solution. Eager-lazy searches for the best among every combination of schedule where the individual instruction groups are either eager or lazy. The x-axes of figures 3(a) and (b) are each divided into three regions: integer results, floating point results and averaged results. From figure 3, it is seen that all of these techniques increase compile time by a modest amount. The performance improvement of even the simpler methods of eager or lazy is 46% for integer and 25% for floating point benchmarks. Eager-lazy must perform better than the other two, since all-eager and all-lazy are two of the many solutions tested by eager-lazy. It gives an improvement of 56% for integer and 35% for floating point benchmarks.

The algorithms to implement the three methods in figure 3 use most of the optimizations outlined in the paper: branch and bound, basic block splitting, skipping unneeded cycles, and timing out. In particular, basic block splitting proved necessary to achieve good compile times for eager-lazy. What is not used in the eager and/or lazy ap-

Name	Description	Type	lines of code	# Basic Blocks*	# Useful Operations*
mcf	Combinatorial optimization / vehicle scheduling	Integer	1909	465	3173
parser	Parses a user's input sequence	Integer	10924	6068	31658
gap	Implements a language and library for group computing	Integer	59481	27651	154362
twolf	placement and global routing package for lithography	Integer	19748	8864	72813
vortex	Single-user object-oriented database transactions	Integer	54550	21764	135335
equake	Simulation of seismic wave propagation	Float	1513	561	5428
lucas	tests the primality of Mersenne numbers	Float	674	279	2375
ammp	Molecular dynamics of a protein-inhibitor complex	Float	13263	4250	38995

* *found by examining the output of SGICC*

Table 7: Benchmarks. All are in C and from the SPEC 2000 suite, with the exception of lucas. The SPEC 2000 version of lucas is in Fortran, so we have used a C version instead.

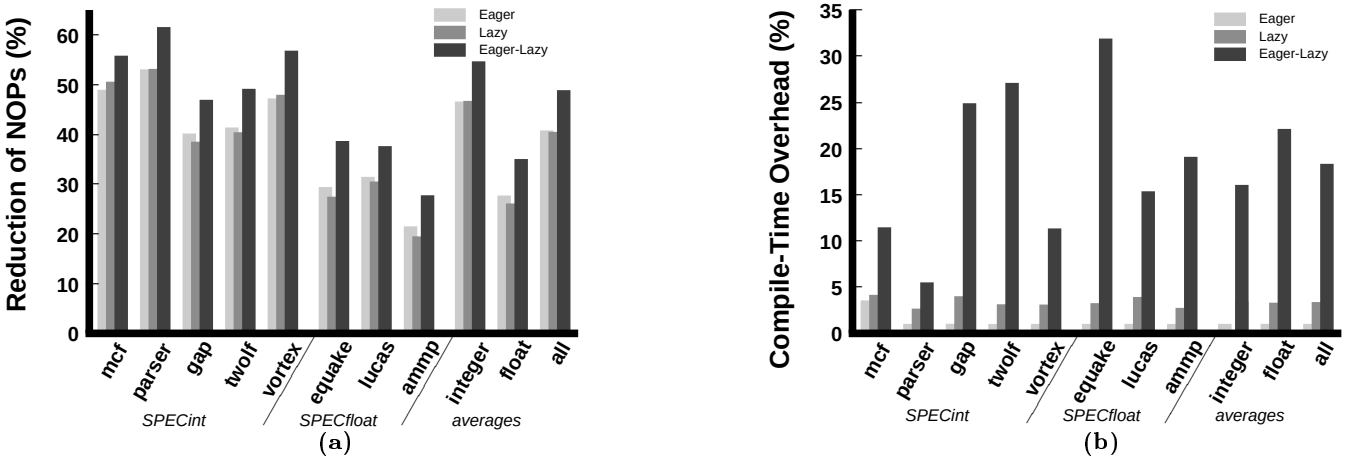

Figure 3: NOP improvement and compilation cost results for eager, lazy and eager-lazy. The first 5 benchmarks are SPEC integer programs, the last 3 are SPEC floating point. Averages are also shown.

proaches are: the allowed-performance-loss technique and two of the pruning methods of Table 4: the *Fillable_NOP* and *Sparse_Template* methods.

Figure 4 shows the performance-loss allowing alternative to the eager-lazy approach, which allows more flexibility in choosing instructions, For this method, the behavior with and without either the *Fillable_NOP* or *Sparse_Template* pruning methods of Table 4. Since the performance for all of these four cases turns out to be so similar as to be almost indistinguishable on a plot, figure 4(a) does not distinguish 4 separate bars. The compile time, however, is markedly different among the cases, especially for the floating point applications. However there is no clear winner among the cases. In particular, equake and ammp show wide differences among the cases, but the preferred cases are opposite for the two benchmarks. The cause of this is unclear.

On examination of figure 4 it may seem odd that the no-pruning technique has the potential to run faster than with pruning; after all, pruning is intended to allow more quickly reaching the solution, and this type of pruning is shown to be optimal. The cause however, is interaction between this optimization and another optimization- the bounding of all solutions known to be worse than the best so far. These prunings are optimal– which means that an optimal solution is guaranteed to remain in the search space after pruning– but this does not indicate that the solution space about to be pruned away does not also contain an optimal solution, since there may be many equally good schedules. In pruning away potentially good, or even optimal solutions, the best-solution-found-so-far variable may not update as quickly as without pruning, leading to a larger search space.

It is interesting that, in both sets of figures, the integer benchmarks result in a greater performance improvement, and therefore also a reduced compile time, due to establishing a better bound on the solution. Table 8 indicates a poorer coded density for floating point programs as well, which is likely due to the limited number of templates available for floating point instructions (see table 1).

Table 8 gives some final numbers for the two most promising techniques. It is seen that code density improvement is similar between floating point and integer cases, but not for compile time and NOP improvement. The NOP improvement and code density improvement measure different things; NOP improvement measures the effectiveness of removing NOPs and the code density improvement measures the relative size of the program, and therefore, roughly approximates the performance.

8. CONCLUSIONS

Constant time algorithms for selecting templates and for finding an upper bound on code size have been described. With these results, an instruction scheduling algorithm that is based on optimal approaches is presented. This method

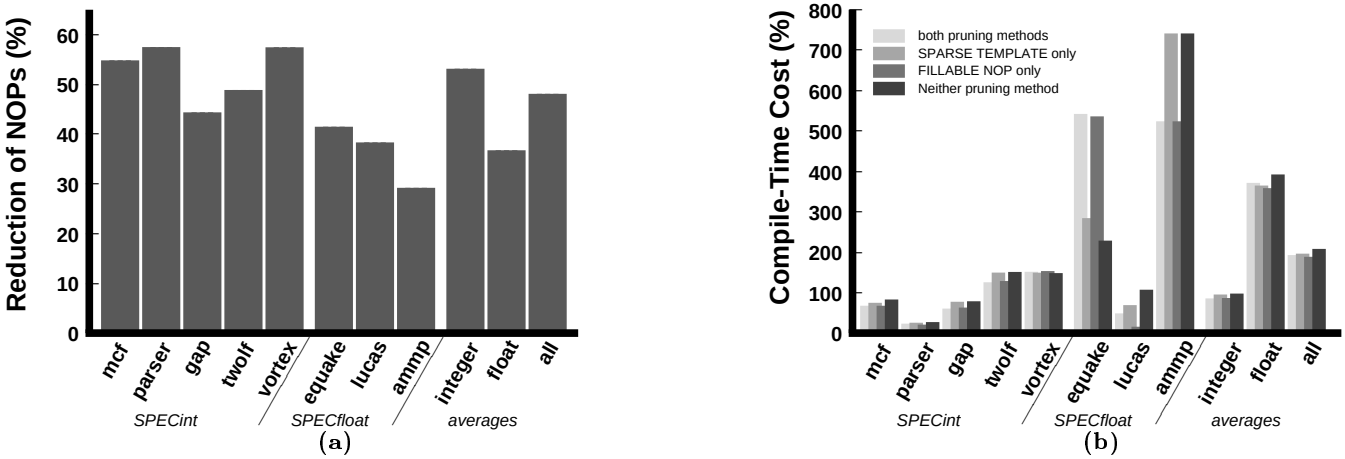

Figure 4: NOP improvement and compilation costs for the allowed-performance-loss heuristic, with different pruning strategies. All of the above use COST_PRUNING, because it is clearly advantageous. It is the SPARSE_TEMPLATE and FILLABLE_NOP methods that are under study. In part (a), the four strategies yielded almost identical results, so the individual bars are not displayed.

Algorithm	average for integer				average for floating point			
	code density	code dens improvemnt	NOP improvemnt	compile time cost	code density	code dens improvemnt	NOP improvemnt	compile time cost
Original	66.87%	0.00%	0.0%	0%	60.23%	0.00%	0.0%	0%
Eager-Lazy	81.33%	14.46%	56.0%	17%	69.85%	9.62%	35.1%	22%
Performance loss	80.83%	14.96%	53.1%	87%	70.41%	10.18%	36.7%	359%

Table 8: Performance of the most promising techniques: Eager-Lazy and Performance-Loss Allowing (using COST_PRUNING and FILLABLE_NOP, as defined in table 4).

is able to remove more than half of the NOPs originally scheduled by the SGICC compiler, for integer programs. Since there are instances where eager-lazy out performs the performance-loss method, and visa versa, a hybrid approach could yield somewhat better results. Modification of the method to include real-machine constraints will also be of interest. These techniques are applied after global instruction scheduling, so that the technique does not replace, but rather supplements, methods such as trace scheduling.

Informal discussions with Intel researchers has identified that low code density is a serious problem, especially impacting the performance of integer programs. This research is a step toward addressing this problem.

9. REFERENCES

[1] J. N. Amaral and G. R. Gao. Using the SGI Pro64 Compiler Infra-Structure for R and D on Back-End Optimizations. In *International Conference on Parallel Architecture and Compilation Techniques (PACT2000)*, Philadelphia, PA, October 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, and K. M. Crozier. Integrated Predicated and Speculative Execution in the IMPACT EPIC Archtecture. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, July 1998.

[3] V. Bala and N. Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, Ann Arbor, Michigan, USA, November 1995. IEEE Computer Society.

[4] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. In *Proceedings of the 32nd Annual ACM/IEEE international symposium on microarchitecture on MICRO-32*, pages 262–271. IEEE Computer Society, November 1999.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] S. W. Daniel Kastner. ILP-based Instruction Scheduling for IA-64. In *In Proceedings of the ACM SIGPLAN Workshop on Languages*, Snowbird, Utah, USA, June 2001.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

[8] S. Hanono and S. Devadas. Instruction selection, resource allocation and scheduling in the AVIV retargeting code generator. In *Design Automation Conference*, June 1998.

[9] *Intel IA-64 Architecture Software Developer's Manual, Volumes I-IV*. Intel Corporation, January 2000. Also available at http://developer.intel.com.

[10] *Intel(R) Itanium(TM) Processor Hardware Developer's Manual*. Intel Corporation, August 2001.

[11] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann". "effective compiler support for predicated execution using the hyperblock". In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[12] W. mei Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1), Jan 1993.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[14] *The SGI Pro64(TM) compiler suite*. SGI Corporation, March 2000. http://oss.sgi.com/projects/Pro64/.

[15] P. Song. Demystifying EPIC and IA-64. *Microprocessor Report*, 12(1):21, January 26 1998.