
THE INTEL IA-64 COMPILER CODE GENERATOR

BY MAKING USE OF POWERFUL FEATURES TO GENERATE HIGH-PERFORMANCE CODE, THE IA-64 ARCHITECTURE ALLOWS THE COMPILER TO EXPLOIT HIGH INSTRUCTION-LEVEL PARALLELISM. THE AUTHORS DESCRIBE THE IMPORTANT PHASES OF THE PRODUCTION COMPILER'S CODE GENERATOR.

Jay Bharadwaj
William Y. Chen
Weihaw Chuang
Gerolf Hoflehner
Kishore Menezes
Kalyan Muthukumar
Jim Pierce
Intel

..... In planning the new EPIC (Explicitly Parallel Instruction Computing) architecture, Intel designers wanted to exploit the high level of instruction-level parallelism (ILP) found in application code. To accomplish this goal, they incorporated a powerful set of features such as control and data speculation, predication, register rotation, loop branches, and a large register file. By using these features, the compiler plays a crucial role in achieving the overall performance of an IA-64 platform. Here, we describe the electron code generator (ECG), the component of Intel's IA-64 production compiler that maximizes the benefits of these features.

The ECG consists of multiple phases occurring in the order shown in Figure 1. The first phase, translation, converts the optimizer's intermediate representation (ILO) of the program into the ECG IR. Predicate region formation, if conversion, and compare generation occur in the predication phase. The ECG contains two schedulers: the software pipeliner for targeted cyclic regions and the global code scheduler for all remaining regions. Both schedulers make use of control and data speculation. The software pipeliner also uses rotating registers, predication, and loop branches to generate efficient schedules for integer as well as floating-point loops.

The ECG's register allocator must handle several IA-64 specific issues. These include not-a-thing (NaT) bit maintenance during spill/fill, advanced load address table (ALAT) awareness for data-speculative registers, correct rotating register allocation for the software pipeliner, and predicate awareness. NaT bits are associated with a control speculation mechanism. The ALAT is the hardware mechanism enabling data speculation.¹

The predicator

Predication, or conditional execution of an instruction based on a predicate, is one of the key IA-64 architectural features. Using predication, the compiler can merge the execution of multiple control flow paths. This increases ILP by removing the penalty of mispredicted branches² and nonsequential control flow in pipelined regions. In addition, predication increases code motion freedom by allowing instructions to be moved upward across branches in a nonspeculative manner and to be pushed downward into subsequent join blocks.

Region formation and if conversion

Predicate region formation selects a group of connected basic blocks—a predicate region—to be if-converted, that is, to remove control flow edges within the region using

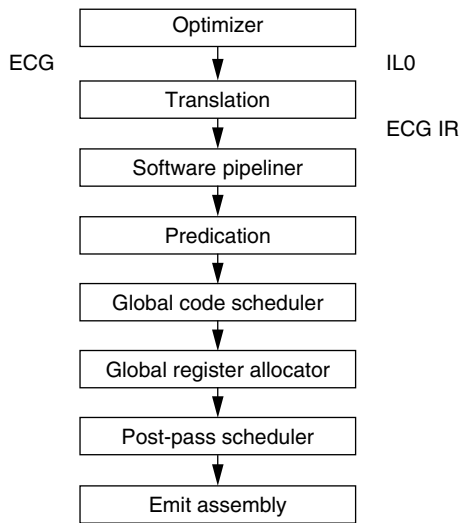


Figure 1. ECG phase order. IR designates intermediate representation.

predicates.³ The basic characteristic of the selected predicate region is that the total number of static branches within the predicate region should be reduced after if conversion. The predicate region selection criteria depend on the availability of dynamic profile information. Without dynamic profile feedback, the selection algorithm focuses on the availability of processor resources and the compatibility of individual critical paths. The algorithm avoids including basic blocks in the predicate region if they cause processor resource over-subscription or if they significantly increase the critical path through the region. With dynamic profile feedback, the selection criteria are extended to include the cost of branch misprediction and the weight of individual critical paths. The algorithm focuses on the branches that produce the most misprediction penalties and chooses the surrounding blocks to form a predicate region.

Compare generation materializes all the necessary predicates through the use of predicate generation instructions. It computes control dependence information for all basic blocks within the predicate region and the predicate region's exit basic blocks. Basic block B depends upon another basic block (A) for control when the condition computed in A dictates whether B gets executed. A controlling predicate, using the assigned virtual predicate name, is generated for all the controlled

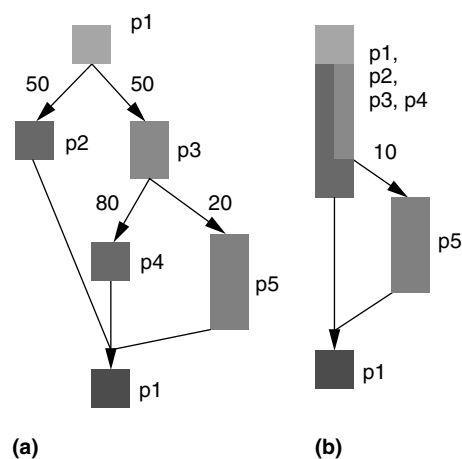


Figure 2. Three control flow paths (a) are reduced to two using predication (b). The decision to predicate is based on a combination of dynamic profile information, resource availability, and critical-path length compatibility. Merging the control flow paths accomplishes two tasks. The unbiased conditional branch is eliminated, resulting in a highly biased conditional branch. A larger basic block is formed from otherwise small basic blocks. This offers more opportunity for ILP to fill the issue bandwidth and hide long latency instructions.

basic blocks within the predicate region and all the region exit basic blocks.

Path collapsing merges the control flow paths within the predicate region into a minimal set of control flow paths. The merged basic blocks are physically placed next to each other. Each basic block within the predicate region is guarded with the correct virtual predicate name. A side exit from the predicate region occurs when the exiting basic block isn't contained within the predicate region. When there are identical branch targets outside the predicate region, these exit flow edges are merged to remove duplicates.

Predicate optimizations

Predicate registers are virtually named and materialized only when it becomes necessary. Predicate name assignment generates a virtual predicate name for all basic blocks within a function (Figure 2). Basic blocks exhibiting identical control flow behaviors are assigned the same virtual predicate name. For all critical edges (edges from a node with multiple

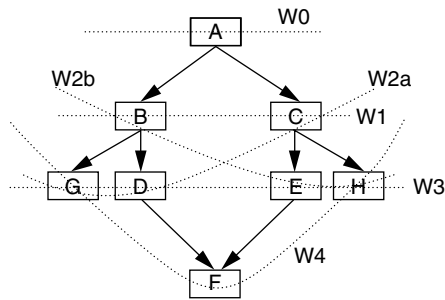


Figure 3. Wavefront advancement in top-down scheduler.

successors to a node with multiple predecessors—such as a missing else-block within an if-then statement—predicate name assignment creates a nonvisible basic block holder and assigns a virtual predicate name. Later due to control flow changes, the predicate name assignment creates new virtual predicate names.

To execute efficiently and correctly, compare optimization examines the collapsed region for opportunities to insert, merge, and replace predicate-generating instructions within the predicate region. The first step is to replace predicate-generating instructions into parallel semantics, if possible. This reduces the critical path through the predicate region. The remaining conditional compares are examined to discover unconditional compare opportunities. When a conditional compare is converted into an unconditional compare, it removes the necessity for predicate initialization instructions. In general, predicate initialization instructions are inserted when the predicate register isn't defined on all paths to the predicate register.

Predicate query system

PQS is a predicate relational database accessed by later phases of the code generator. It contains information such as predicate disjointness, predicate dominance and post-dominance, predicate promotion,⁴ and predicate addition and subtraction. Without accurate predicate information, the scheduling, software pipelining, and register allocation must make conservative decisions that would result in suboptimal code. PQS is based upon work proposed by HP Labs.⁵

Global code scheduler

GCS is capable of scheduling the code in any acyclic control flow subgraph. No restrictions are placed on the number of entries into or exits from the subgraph. GCS may generally move code in either direction across a visible edge. To schedule loops, GCS removes back edges to render the subgraph acyclic. After the region of code is scheduled, it's

abstracted away by nesting it. Thus, after scheduling code in an inner loop, the nested loop is represented as a single node containing summary data flow information. This information enables code motion across the nested inner loop. The code schedule in the inner loop is not disturbed when scheduling the outer loop. Also, GCS takes into account all latencies for live-in and live-out values of the inner loop.

To support global analysis in the presence of control flow within the scheduling region, a novel path-based data dependence representation is used. This representation merges control flow and data dependence information in one structure.⁶

Wavefront scheduling and deferred compensation code

Compensation is duplicated code that needs to be inserted when moving code up across control flow joins or down below control flow splits. Empty blocks called join-split (JS) blocks⁷ are added on each edge that both starts from a node with multiple successors (split) and ends at a node with multiple predecessors (join). These are placeholders for scheduling compensation code. Blocks called interface blocks are created for this same purpose at side entries/exits, that is, blocks that conditionally enter or exit the region. Adding JS and interface blocks to an acyclic region forms a scheduling region. We define a wavefront in the scheduling region as a strongly independent cut set.⁷ Figure 3 shows an acyclic scheduling region and all possible wavefronts in it.

In wavefront scheduling, the wavefront is the partition between scheduled and unscheduled code. If scheduling top down, the nodes above the wavefront are already scheduled, and the schedule in these nodes won't be changed (unless we resort to backtracking). The nodes below the wavefront don't contain scheduled code. The wavefront represents the set of blocks into which instructions are currently being scheduled. The candidates for scheduling into a block on the wavefront include unscheduled data-ready instructions originating from the same block or another block that is either above or below the wavefront. In a top-down scheduling scheme, the wavefront first passes through the region entry

blocks. When code scheduling into a block is completed, the scheduler declares it closed and advances the wavefront across it. This block now lies above the wavefront and represents a fully scheduled block. Thus the wavefront advances down the region until it finally passes through all the exit nodes in the region. When block B is closed, any unscheduled instructions from B, or above, are implicitly moved below B to be scheduled later. Hence before declaring a block closed, the scheduler examines both the correctness and profitability of such downward code motion.

Figure 3 numbers the wavefronts to show a wavefront's advancement in a top-down scheduling scheme. Note that there can be multiple alternatives to the way the wavefront is advanced, as shown in the figure by W2a and W2b.

When performing an upward or downward code motion that requires code duplication, GCS postpones the generation of this compensation code until it is actually scheduled. In doing this, we provide scheduling freedom for the compensation code since its destination block is not fixed a priori. We also ensure that at most one copy of the instruction is executed on any path through the region. Figure 4 illustrates how this is done.

When the wavefront is W1, copy I' of instruction I originating from block G is first scheduled in block F. This motion requires compensation since F doesn't dominate G. Instead of immediately generating and inserting the compensation code for I in block B, instruction I is left behind in block G along with some bookkeeping information that indicates that it remains to be scheduled along control flow paths ABDEG and ABCEG. The bookkeeping data also indicates that the compensation code must be scheduled on or before E to avoid execution of I (or its copies) multiple times along path AFG.

The scheduling heuristics may not select I for scheduling in block B if it isn't an important enough candidate in B. Assume that this is the case and that the wavefront is advanced to W2. It is then selected for scheduling in block C, so copy I'' of instruction I originating from block G is scheduled into block C. Since F and C don't collectively dominate G—the block of origin of I—this scheduling choice requires compensation. Now the

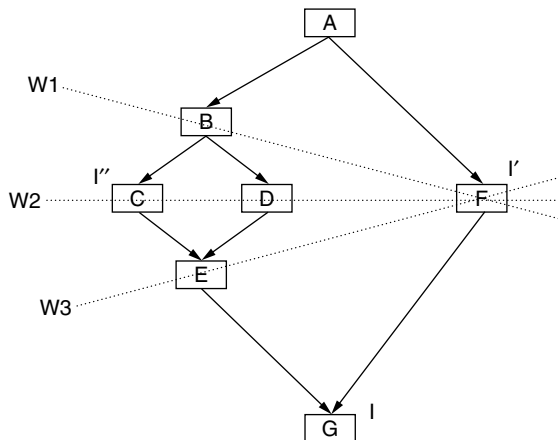


Figure 4. Deferred compensation code generation.

remaining instruction I in G represents the compensation that needs to be scheduled in block D. To avoid multiple executions of I along path ABCEG, advancement of the wavefront past W2 is constrained until I is finally scheduled in block D. Bharadwaj et al.⁶ describe the bookkeeping scheme used to keep track of this deferred compensation code in greater detail.

Speculation and predication

GCS selects an instruction to schedule into block B from a candidate list of data-ready instructions. These instructions may originate from any block in the region connected to B by at least one control flow path. The speculation support in IA-64 allows many dependencies to be ignored when determining data readiness. Control dependencies and data dependencies on qualifying predicates can be broken using control speculation. Data speculation allows unlikely memory dependencies to be broken.

GCS selects the best candidate from all the speculative and nonspeculative candidates based on a cost-benefit analysis. Among other things, this analysis takes into consideration the instruction's global critical-path length, its resource requirements, and its speculation and code duplication costs. Speculation check instructions and recovery code⁸ are generated as a byproduct of speculation; load safety information⁹ avoids unnecessary control speculation.

Figure 5 (next page) provides an example illustrating the costs of speculation and some ways to avoid it. The cmp instruction sets the

Instructions	Cycle	
cmp p1,p0=...	// 1	
(p1) br B10	// 1	
ld v2=[v1] // safe load	// 2	
ld v4=[v3]	// 2	
ld v8=[v7]	// 3	
add v6=v2,v4	// 4	
st[v8]=v6	// 5	
(a)		
Instructions	Cycle	
ld v2=[v1]	//1	REC: ld v4=[v3]
ld.s v4=[v3]	//1	add v6=v2,v4
cmp p1,p2=...	//1	br B11
(p2) ld v8=[v7]	//2	
(p1) br B10	//2	
add v6=v2,v4	//3	
chk.s v6, REC	//4	
B11 :		
st [v8] = v6	//4	
(b)		

Figure 5. Recovery code example for control speculation: before (a) and after (b) scheduling.

first predicate and clears the second if the condition is true and vice-versa if the condition is false. In this example three loads are moved up across a controlling branch. One load, ld v2=[v1], is known to be safe, so the corresponding chk and its associated recovery code aren't necessary. Another load, ld v8=[v7], can be guarded by the p2 nonbranch predicate, which is complementary to branch predicate p1. This load can be scheduled as soon as p2 is ready for use and treated as a normal load since it won't be executed speculatively. The only load that ends up needing a speculation check and recovery code is ld v4=[v3].

There are two kinds of costs associated with generating speculation checks and recovery code. One is incurred regardless of whether the chk instruction fails and branches to recovery, while another is incurred only on each chk failure. Chk consumes an execution unit resource on the main execution path.

Because they are used in the recovery code, registers v2 and v3 must be kept alive until chk executes. Thus speculation can extend live ranges and thereby increase register pressure. Addition of the recovery code increases the static size of the binary. If chk fails and branches to

recovery, the penalty is high since chk may be mispredicted and incur an additional cost to jump to recovery. The recovery code may not be in the cache and might prove costly to execute. So speculation must only be used when based on a sound analysis of the likelihood of speculation failure. Despite these costs, we speculated the load in Figure 5's example because it has a high critical-path length and we had determined that speculation failure was unlikely.

GCS uses predication support in the IA-64 architecture to convert a control speculative instruction to a nonspeculative one. Sometimes an instruction being moved across a branch is scheduled after the compare operation that controls the branch. If the predicate produced by the compare is available for use, the instruction is predicated. This eliminates the need for a check instruction and recovery code, as depicted in the example in Figure 5.

In addition, when a predicate is false, the adverse effects of a speculative operation on the data cache and TLB (translation look-aside buffer) are avoided. When an instruction is moved across multiple branches, the predicate controlling execution of the block of origin may not be available. This may be so because the compare generating the predicate hasn't been scheduled, or the compare's latency hasn't yet expired. However, a predicate for an intermediate block in the control dependence chain may be available.

Predication with such a predicate will not render the instruction nonspeculative but will reduce the instruction's speculativeness when the instruction executes (that is, the qualifying predicate is true). To do this effectively, GCS maintains a predicate promotion list and keeps track of predicates that become available as compare instructions are scheduled. This list is essentially a form of control dependence information, but in the predicate domain. GCS also uses the predicate promotion list when speculating predicated instructions. GCS speculates these instructions by promoting the qualifying predicate to an available predicate in the predicate promotion list. If no predicate is available, the instruction is unpredicated. Instruction predication, unpredication, and predicate promotion are thus achieved on the fly while scheduling.

Downward code motion and branch generation

GCS benefits from downward code motion

in several ways. First, operations that cannot be speculated—such as stores and speculation check instructions—tend to stay in their block of origin. It is advantageous to move these instructions downward if they don't fit into the block's schedule. Second, downward motion can empty a block, eliminating an unconditional branch or exposing an opportunity for multiway branch generation. To do this effectively, GCS monitors and updates block layout during scheduling. Finally, downward code motion helps reduce the amount of speculation or compensation code needed to expose ILP.

To move stores down to a join node, GCS must ensure they are predicated. Predicates are only available for use in blocks dominated by the compare instructions that generate them. This places a limit on downward code motion.

The GCS design elements permit effective use of the architectural features of IA-64 while being sensitive to practical considerations such as code size and compile time. Bharadwaj et al.⁶ give further details on GCS features and implementation.

Software pipelining

Software pipelining¹⁰ improves the performance of a loop by overlapping the execution of several iterations. The IA-64 architecture provides extensive support for software-pipelined loops, such as register rotation, and special loop branches and registers.¹ These features enable efficient software pipelining of loops without the accompanying increase in code size seen in other architectures.

Register rotation provides a renaming mechanism that eliminates the need for loop unrolling to support software register renaming. Special software-pipelined loop branches support register rotation and, combined with predication, reduce the need to generate separate code blocks for the prolog and epilog phases.

The following example illustrates register rotation. A `swp_branch` pseudoinstruction (either `br.wtop`, `br.ctop`, `br.wexit`, or `br.cexit`) represents a software-pipelined loop branch.¹

```
Loop:stage1: ld4 r32 = [r10],4
// post increment by 4
stage2:      // empty stage
stage3:      st4 [r11] = r34,4
```

```
// post increment by 4
swp_branch Loop ;;
```

The pipelined loop has three stages. The value that the load writes to `r32` is read by the store's two iterations (and two rotations) later as `r34`. Meanwhile, two more instances of the load execute. Because of register rotation, those instances write their results to different registers and don't destroy the value needed by the store.

The number of cycles between the start of successive iterations in a software-pipelined loop is called the Initiation Interval, or II. A software-pipelined loop executes in three phases: prolog, when the pipeline is ramped up; kernel, when the pipeline is at steady state; and epilog, when the pipeline is drained.

The input to the ECG software pipeliner is the set of basic blocks in the loop. If the loop has multiple basic blocks, it's converted into a loop consisting of a single basic block using if conversion.

Instructions are overlapped across iterations by using the modulo scheduling technique.¹¹ Consider the following loop, shown with virtual registers (prefixed with a "V"):

```
Loop:ld4 Vr1 = [Vr2], 4
      add Vr3 = Vr1, Vr4
      st4 [Vr5] = Vr3, 4
      br.cloop Loop ;;
```

This loop has no dependencies across iterations; that is, it has no loop-carried dependencies. Therefore multiple loop iterations can be executed in parallel. A possible pipeline schedule for a source iteration of this loop, assuming a load latency of two cycles, is

```
Stage 1: (p16) ld4 Vr1 = [Vr2], 4
Stage 2: (p17) // empty stage
Stage 3: (p18) add Vr3 = Vr1, Vr4
Stage 4: (p19) st4 [Vr5] = Vr3, 4
```

A predicate is assigned to each stage of the software pipeline to control the execution of the instructions in that stage. This predicate is called the stage predicate. For counted loops, we define `p16` as the predicate for the first stage, `p17` as the predicate for the second stage, and so on. Register rotation takes place at the end of each stage, when the software-pipelined loop branch is executed in the kernel loop.

Thus a 1 written to p16 enables the first stage, then rotates to p17 at the end of the first stage to enable the second stage for the same source iteration. Each sequential 1 written to p16 enables all the stages for a new source iteration. This behavior enables or disables the execution of the pipelined loop stages during the prolog, kernel, and epilog phases.

After modulo scheduling and rotating register allocation,¹² LC, EC, and the rotating predicate registers are appropriately initialized in the loop's preheader. Assuming a loop trip count of 100 iterations, the example loop is transformed as follows:

```

mov LC = 99 // LC = loop trip
count - 1
mov EC = 4 // EC = epilog stages
+ 1
mov pr.rot = 1 << 16 // p16 = 1,
rest = 0
Loop:
  (p16) ld4 r32 = [r5], 4
  (p18) add r35 = r34, r9
  (p19) st4 [r6] = r36, 4
      br.ctop Loop ;;

```

Note that an iteration starts and completes every cycle at a steady state. This leads to a throughput of one iteration per cycle, even though each iteration takes four cycles to complete.

In a counted loop, the br.ctop instruction doesn't depend on any other instruction in the loop, allowing a quick decision about whether to start a new source iteration. In while loops, the decision to execute another source iteration depends on a more general computation ultimately terminating in a compare that sets the qualifying predicate for the br.wtop branch. The computation may be complex and contain loads or other long latency instructions. Thus the br.wtop branch may not be ready for execution until late in the source iteration, resulting in limited overlap of the iterations.

To address this problem, we speculatively start execution of subsequent iterations¹³ using IA-64's support for control speculation. The software pipeline stages prior to the one containing the compare that sets the br.wtop's qualifying predicate are called the speculative stages of the pipeline. It isn't possible for the compare and the br.wtop branch to control

the execution of these stages. Therefore, these stages aren't assigned stage predicates. The result of the compare—in addition to being the qualifying predicate for br.wtop—also acts as a stage predicate to control the first non-speculative pipeline stage. The compare, rather than the br.wtop, generates the predicate that enables the nonspeculative portion of the next source iteration.

Consider the following while loop:

```

Loop:
  ld4 Vr1 = [Vr2], 4 // (1)
  add Vr3 = Vr1, Vr4 // (2)
  st4 [Vr5] = Vr3, 4 // (3)
  cmp Vp1, p0 = (Vr1 != 0) // (4)
  (Vp1) br Loop

```

Without control speculation, this loop has a recurrence cycle caused by a 4 → 1 → 4 sequence of edges.

Assuming a two-cycle load latency, this loop can be pipelined with a minimum II of three cycles. However, using control speculation, the load of the next iteration can be speculatively executed even before we know the next iteration will execute. This enables us to pipeline the loop with an II of 1, with the following schedule:

```

Loop:stage 1: ld4 Vr1 = [Vr2], 4
stage 2: // empty stage
stage 3: (p18) add Vr3 = Vr1,
Vr4
      (p18) cmp Vp1, p0 = (Vr1 != 0)
stage 4: (p19) st4 [Vr5] =
Vr3, 4

```

Here, stages 1 and 2 are speculative stages, and stage 3 is the first nonspeculative stage. After register allocation, we generate the following pipelined loop, adding a check instruction to the loop for the speculated load:

```

mov EC = 3 // #epilog stages +
#spec. stages
mov pr.rot = 1 << 16 ;; / /
p16 = 1, rest = 0
Loop:
  ld4.s r32 = [r5], 4 // specu-
lative load
  (p18) chk.s r34, recovery //
check r34 for exception

```

```
(p18) add r35 = r34, r9
(p18) cmp p17, p0 = (r34 != 0)
(p19) st4 [r6] = r36, 4
(p17) br.wtop Loop;
```

There is no stage predicate assigned to the load because it is speculative. The check, however, is predicated by stage predicate p18. Therefore, the deferred exception, if any, for a load of iteration N will be serviced only if iteration N is known to execute.

The compare sets p17. This is the branch predicate for the current source iteration, and, after rotation, the stage predicate for the first nonspeculative stage (stage three) of the next source iteration. During the prolog, the compare can't produce its first valid result until the kernel loop's third iteration. The predicates' initialization provides a pipeline of predicate values that keeps the compare disabled until the first source iteration reaches stage three. At that point, the compare is enabled and starts generating stage predicates to control the nonspeculative pipeline stages.

During the last source iteration, the compare result is zero. Therefore, during the next kernel iteration, the stage predicate for the compare is zero. The conditional compare stops writing results, and a stream of zeros written by the br.wtop branch rotates in to drain the pipeline.

The ECG software pipeliner has many other interesting features such as data speculation, back substitution, and riffing that enable pipelining of many loops and better ILP.

Global register allocation

The GRA in the IA-64 compiler is a region-based Chaitin/Briggs-style graph-coloring scheme.^{14,15} Special consideration for IA-64 features must be taken into account in the design.

NaT bits

Each general register has an associated NaT bit. When the register is spilled, the NaT bit is stored in the UNAT application register. The spill address determines the bit location. To obey this association, the register allocator spills registers that may contain speculated values into contiguous memory. In addition, since the UNAT is a 64-bit register, after 64 registers are spilled, the UNAT itself must be spilled. This mechanism requires extra bookkeeping for the register allocator.

Rotating registers

Traditional register allocators don't support register rotation. However, live ranges that span multiple pipelined loop stages can benefit from the use of rotating registers. A special allocator within the software pipeliner allocates these live ranges.¹² The GRA then allocates the remaining live ranges.

Advanced loads

The use of data speculation often increases register lifetimes. The example in Figure 6 shows v1-v11 as the virtual registers.

Without data speculation, the live range of v4 extends from instruction 11 to instruction 12. The live ranges of v1 and v6 end at instructions 11 and 12. With data speculation, since v1 and v6 are referenced in the speculated computation, their live ranges extend to within the recovery code. In addition, v4 is needed at chk.a, instruction 11; however, it's a special need. The physical register assigned to v4, say r4, in instruction 2 is needed to index into the ALAT in instruction 11;¹ that is, register index r4 is needed for chk.a but not the actual value in r4. Thus, r4 can be reused between instructions 4 and 11 but not as another advanced load destination.

Predication

In the presence of predication, a virtual register's liveness can be a function of multiple predicates because the qualifying predicate of an instruction guards its definitions and uses. Traditionally, liveness is represented as a bit vector. However, in our predicate-aware register allocator, it's represented as a bit matrix with predicates being the additional dimension. ECG's register allocator is based upon the work done by Gillies, Ju, Johnson, and Schlansker.⁵

We designed the ECG compiler to exploit the features provided in the IA-64 architecture. Many novel techniques were devised and implemented to make good use of predication, control and data speculation,

1) add v1=v2,v3	1) add v1=v2,v3
...	2) ld.a v4=[v1]
...	3) add v5=v4,v6
...	4) ...
10) st [v10]=v11	10) st [v10]=v11
11) ld v4=[v1]	11) chk.a v4, rec_code
12) add v5=v4,v6	rec_code:
	100) ld v7=[v1]
	101) add v5=v7,v6

(a) (b)

Figure 6. Comparing data speculation usage: without (a) and with data speculation (b).

rotating registers, and the register stack engine. As IA-64 is a new architecture, we'll learn much more about how to more effectively use these features over the next several years. Currently, the ECG compiler provides high performance and a solid base upon which to further leverage IA-64 strengths. MICRO

Acknowledgments

We acknowledge Kent Fielden, Dong-Yuan Chen, Youfeng Wu, Roland Kenner, and Chris McKinsey for their previous work in the design and implementation of parts of the ECG.

References

1. *IA-64 Application Developer's Architecture Guide*, Order No. 245188, Intel Corporation, Santa Clara, Calif., May 1999.
2. S.A. Mahlke et al., "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. 27th Int'l Symp. Microarchitecture*, IEEE Computer Society Press, Los Alamitos, Calif., Dec. 1994, pp. 217-227.
3. J.C.H Park and M.S. Schlansker, *On Predicated Execution*, Tech. Report HPL-91-58, HP Laboratories, Palo Alto, Calif., May 1991.
4. S.A. Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proc. 25th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, Dec. 1992, pp. 45-54.
5. D.M. Gillies et al., "Global Predicate Analysis and Its Application to Register Allocation," *Proc. 29th Int'l Symp. Microarchitecture*, IEEE CS Press, Dec. 1996, pp. 100-113.
6. J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs," *Proc. MICRO32, Proc. Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, Nov. 1999, pp. 262-271.
7. D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling," *Proc. MICRO24, Proc. Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, Nov. 1991, pp. 103-113.
8. S.A. Mahlke et al., "Sentinel Scheduling for Superscalar and VLIW Processors," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, N.Y., Oct. 1992, pp. 238-247.
9. D. Bernstein, M. Rodeh, and M. Sagiv, "Proving Safety of Speculative Load Instructions at Compile-Time," *Proc. Fourth European Symp. Programming*, 1992, pp. 56-72.
10. M.S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. ACM SIGPLAN 1988 Conf. Programming Language Design and Implementation*, ACM, June 1988, pp. 318-328.
11. B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. MICRO-27, Proc. Ann. Int'l Symp. Microarchitecture*, Dec. 1994, pp. 63-74.
12. B.R. Rau et al., "Register Allocation for Software Pipelined Loops," *ACM Programming Language Design and Implementation (PLDI)*, ACM, June 1992, pp. 283-299.
13. P. Tirumalai, M. Lee, and M. S. Schlansker, "Parallelization of Loops with Exits on Pipelined Architectures," *Proc. Supercomputing 90*, IEEE CS, Dec. 1990, pp. 200-212.
14. P. Briggs et al., "Improvements to Graph Coloring Register Allocation," *ACM Trans. Programming Languages, and Systems (TOPLAS)*, Vol. 16, No. 3, May 1994, pp. 428-455.
15. G. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proc. SIGPLAN 82 Symp. Compiler Construction*, ACM, Vol. 17, No. 6, June 1982, pp. 98-105.

Jay Bharadwaj is a senior staff engineer at Intel Corporation, where he has been involved with the IA-64 compiler since its inception. He is currently investigating usage of hardware performance monitors to enable continuous and transparent profiling, and user program optimization. His research interests include static and dynamic program optimization. Bharadwaj holds a master's degree in computer science from Rensselaer Polytechnic Institute in New York.

William Y. Chen is a research scientist at Intel, where he helped define the IA-64 architecture and worked on its compiler. He started a project for runtime optimizations on IA-64 binaries. His interests include architecture, compiler, and runtime compilation. Chen holds a PhD degree in electrical and computer science from the University of Illinois, where he worked on control speculation, data speculation, and predication.

Weihaw Chuang is a member of Intel's IA-64

product compiler team. Previously, he worked on the Itanium processor as a design automation engineer. Chuang received a BS degree in computer science from MIT.

Gerolf Hoflehner, a senior compiler engineer at Intel, works with the IA-64 product compiler team on register allocation and predication. His current interests are in compiler optimization, algorithm design, and computer architecture. Previously, he held various positions at Siemens. Hoflehner graduated from the Technical University Munich, Germany with a Diplom in mathematics.

Kishore Menezes is a senior compiler engineer, working on the global code scheduler with the IA-64 product compiler team. His general interests are in the field of compilers and computer architecture. Menezes obtained his PhD in computer engineering from North Carolina State University.

Kalyan Muthukumar, a senior compiler engineer, also works with the Intel IA-64 product compiler team. He has implemented several optimizations in the software-pipelining phase of the compiler. His interests are in compiler optimization and computer architecture. Prior to joining Intel, he worked at IBM and Apple Computer. Muthukumar obtained his PhD in computer science from the University of Texas at Austin.

Jim Pierce is an IA-64 architect at Intel, where he has focused on compiler performance. Currently, he is the code-generator manager of the IA-64 product compiler team. His current interests are in architecture, compiler technology, and dynamic compilation. Pierce received his PhD in computer science from the University of Michigan.

Direct questions about this article to Jim Pierce, Technology Research Labs, Intel Corporation SC 12-305, Santa Clara, CA 95054; jim.pierce@intel.com.