An Advanced Optimizer for the IA-64 Architecture

THE IA-64 ARCHITECTURE'S RICH SET OF FEATURES ENABLE AGGRESSIVE HIGH-LEVEL AND SCALAR OPTIMIZATIONS—SUPPORTED BY THE LATEST ANALYSIS TECHNIQUES—TO IMPROVE INTEGER AND FLOATING-POINT PERFORMANCE.

Rakesh Krishnaiyer
Dattatraya Kulkarni
Daniel Lavery
Wei Li
Chu-cheow Lim
John Ng
David Sehr
Intel

Intel's IA-64 architecture has a rich set of features, including control and data speculation, predication, large register files, and an advanced branch architecture.[1,2] These features allow the compiler to exploit instruction-level parallelism (ILP) and optimize applications in many new ways. Intel's IA-64 compiler incorporates the latest optimization techniques already known in the compiler community. In addition, some known techniques have been extended, and new techniques have been designed specifically for the IA-64 architecture.

High-level optimizations include loop and data transformations to improve cache locality and parallelism. Compilers typically apply these techniques to program structures at a higher level of abstraction, compared with the program representation for scalar optimizations. For example, loop unrolling and loop unroll-and-jam transformations exploit the large register file to eliminate redundant references to array elements and to expose more parallelism. Scalar replacement of memory references replaces array references with register references. Linear loop transformations, loop fusion, loop tiling, and loop distribution improve cache locality. Finally, data prefetching overlaps memory access latency with computation.

A primary objective of scalar optimizations is to minimize the number of computations and the number of references to memory. The primary scalar optimization that achieves this objective is partial redundancy elimination (PRE),[3] which minimizes the number of times an expression is evaluated. We extended PRE to use control and data speculation to eliminate more loads. PRE's counterpart, called partial dead-store elimination (PDSE), serves to remove redundant stores to memory.

State-of-the-art analysis techniques[4] support optimizations in the IA-64 compiler. Memory disambiguation determines whether two memory references potentially access the same memory location. This information is critical in hiding memory latency, because knowing that a load doesn't interfere with an earlier store is essential to scheduling the load earlier. The memory disambiguator includes support for data speculation. Interprocedural analysis and optimization (IPO) has proven effective in optimizing applications by exposing opportunities across procedure call boundaries. IPO gains importance for a processor with many functional units and registers.

IA-64 enables more aggressive optimizations. For instance, effectively using the large register file eliminates memory operations. Optimizations use rotating registers to reduce the overhead of software register renaming in loops. Predication serves in situations such as removing hard-to-predict branches and implementing an efficient prefetching policy. The compiler uses control and data speculation to

0272-1732/00/$10.00 © 2000 IEEE

eliminate redundant loads, stores, and computations.

This article provides a partial list of the analyses and optimizations that enable IA-64 performance.

Memory disambiguation

The effectiveness and legality of many compiler optimizations rely on the ability to accurately disambiguate memory references. Examples include code scheduling, loop transformations, and elimination of redundant loads and stores. The IA-64 compiler provides different kinds of analyses for memory disambiguation.

State-of-the-art memory disambiguation

The simplest disambiguation cases are direct scalar or structure references. The compiler may disambiguate two structure references *a.field1* and *b.field2*, either by determining that *a* and *b* are different memory objects or that *field1* and *field2* are nonoverlapping.

To disambiguate a pair of indirect memory references such as **p* and **q*, the compiler usually must perform points-to analysis,[4] which determines the set of memory objects that each pointer could possibly point to. Because pointer *p* or *q* could be a global variable or a function parameter, the compiler's points-to analysis is interprocedural.

For arrays, the compiler performs data-dependence analysis using a series of dependence tests, and it determines accurate dependence direction and distance information.

Various simple language rules help in providing disambiguation information, even when the more expensive analyses are turned off. For example, parameters in programs conforming to the Fortran standard are independent of each other and of common block elements. As another example, an indirect reference cannot access the same location as a direct access to a variable that hasn't had its address taken.

Disambiguation support for data speculation

In some cases, the compiler cannot determine whether two memory references are definitely independent. Such cases arise from a lack of information at lower levels of optimization, an inability to analyze the whole program because of dynamically linked libraries, or the high complexity of certain types of disambiguation problems.

For example, consider the loop in Figure 1, from the SPEC CPU2000 benchmark 254.gap. Each source line in the loop body contains the same chain of dependent instructions. Pointer *r* is read from a field of a structure that is passed in as a formal parameter. Pointer *p* is

```
for (k = SIZE(hdR)/(4*sizeof(TypDigit)); k !=0; --k) {
   c = L * *r++ + (c>>16); *p++ = c;
   c = L * *r++ + (c>>16); *p++ = c;
   c = L * *r++ + (c>>16); *p++ = c;
   c = L * *r++ + (c>>16); *p++ = c;
}
```

Figure 1. Disambiguation of pointer references in SPEC CPU2000 benchmark 254.gap.

IA-64 background

Readers of this article should have a basic knowledge of such IA-64 features as predication, speculation, and rotating register support for software pipelining. Details of these features appear in "Introducing the IA-64 Architecture" in the September-October 2000 issue of *IEEE Micro*.[1] Elsewhere, Dulong et al. provide an overview of the Intel IA-64 compiler.[2] A companion article on code generation, also appearing in the September-October 2000 issue of *IEEE Micro*,[3] provides details on the predicator, global code scheduler, software pipeliner, and register allocator.

This article focuses on both high-level and scalar optimizations,[4] emphasizing the uniqueness of design and implementation made possible by IA-64 features. In many cases, we briefly introduce the optimization but don't include the complete algorithms. Many of these basic algorithms are available in the literature (see Muchnik[4] for a comprehensive list of references). Extensive literature is available on ILP compilation.[5-8]

References

1. J. Huck et al., "Introducing the IA-64 Architecture," *IEEE Micro*, Sept.-Oct. 2000, pp. 12-23.
2. C. Dulong et al., "An Overview of the Intel IA-64 Compiler," *Intel Technology J.*, Q4 1999.
3. J. Bharadwaj et al., "The Intel IA-64 Compiler Code Generator," *IEEE Micro*, Sept.-Oct. 2000, pp. 44-53.
4. S. Muchnik, *Advanced Compiler Design and Implementation*, Morgan Kaufman, San Francisco, Calif., 1997.
5. J.C. Dehnert and R.A. Towle, "Compiling for the Cydra-5," *J. Supercomputing*, Jan. 1993, pp. 181-228.
6. W.W. Hwu et al., "Compiler Technology for Future Microprocessors," *Proc. IEEE*, Vol. 83, No. 12, Dec. 1995, pp. 1,625–1,640.
7. P.G. Lowney et al., "The Multiflow Trace Scheduling Compiler," *J. Supercomputing*, Jan. 1993, pp. 51-142.
8. B.R. Rau and J.A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *J. Supercomputing*, Jan. 1993, pp. 9-50.

```
ld r1 = *r
mul r3 = L * r1
                ld.a r21 = *r
                mul r23 = L * r1
                                ld.a r31 = *r
                                mul r33 = L * r1
------------------------------------------------
shift r4 = (c>>16)
add c = r3 + r4
st *p = c       shift r24 = (c>>16)
                chk.a r21, L3
                add c = r23 + r24
                st *p = c
                                shift r34 = (c>>16)
                                chk.a r31, L4
                                add c = r33 + r34
                                st *p = c
```

Figure 2. Overlapping possibly dependent chains of instructions using advanced loads.

read from a field of a similar structure dynamically allocated by a memory management function that is part of the 254.gap program.

At the default level of optimization, most compilers don't perform interprocedural points-to analysis. For example, in the Intel IA-64 compiler, this analysis occurs only if interprocedural optimization is turned on. Without interprocedural analysis, the compiler can't be sure that the loads of **r* are independent of the stores to **p*, even though they're likely independent. Even interprocedural points-to analysis is very difficult in this case because of the special memory management function and because the program uses the same field to point to many different types of objects. If the compiler must assume that the loads and stores are dependent, then the scheduler cannot overlap the different source lines. This lack of overlap, combined with the long-latency multiply operations, results in poor performance.

However, an IA-64 compiler can use data speculation, as shown in Figure 2, where the code for three of the four source lines in the loop body appears. In the figure, *L* and *c* reside in registers. The *mul* pseudoinstruction represents a sequence of instructions to implement integer multiply in the IA-64 architecture. The compiler can use advanced loads (written *ld.a*) to schedule the loads and multiplies of the later source lines ahead of the stores from previous lines. This greatly reduces the time needed to execute the loop body. To check whether the stores from the earlier source lines wrote to the same memory location as the loads, a check instruction (written as *chk.a*) executes for each of the speculated loads. If any overlap is found, control transfers to the recovery block at *L3* or *L4* to reexecute the appropriate load and the dependent multiply.

Using advanced loads entails several cost considerations. First, the check instructions consume additional resources. Second, if the load and store do access the same location, recovery code will be executed. Therefore, the probability of independence times the number of cycles removed from the schedule length must be less than the probability of dependence times the number of cycles required to branch to and execute the recovery code. Finally, the supporting advanced load address table (ALAT)[2] has limited size and associativity. The scheduler and register allocator must take this into account to avoid replacement of live ALAT entries.

The disambiguator can aid the compiler by providing information about the dependence probability. It can use heuristics and knowledge of the memory references to identify cases of unlikely dependence. The disambiguator can then notify the optimizer and code schedulers about an unlikely dependence to signal an advanced load opportunity.[5] Researchers have also proposed enhancements to array data dependence analysis to return a probability of dependence.[6]

Memory reference elimination

One very important way to reduce memory bandwidth demands is to use registers effectively and eliminate as many memory references as possible. Therefore we employ various methods for using speculation and predication in PRE and PDSE. A further optimization—scalar replacement—lets us remove loads and stores across loop iterations. This optimization makes special use of IA-64's rotating registers to eliminate the register-to-register copy overhead traditionally associated with scalar replacement.

Speculative partial redundancy elimination

PRE works by exploiting expression availability at points in the program control-flow graph. An expression *e* is *available* at a program point *p* if along every control-flow path from the program entry to *p* there's an

instance of *e* that's not subsequently killed. An expression at point *p* is fully redundant if the same expression is already available. A fully redundant expression may then be replaced by using the previous computation's value.

An expression *e* is *partially available* at a point *p* if there's an instance of *e* along only some of the control-flow paths from the program entry to p. An expression *e* is partially redundant at a point *p* if it's already partially available. The compiler removes the partial redundancy by inserting a copy of the redundant expression on the control-flow paths where it's not available, making it fully redundant.

Figure 3a illustrates a potential problem the compiler must seek to avoid. Insertion of an expression *e* at a program point *p* is said to be *down-safe* if along every control-flow path from *p* to the program exit there's an instance of *e* such that the inserted expression is available at each later instance. In Figure 3a, the expression **q* is partially redundant, and the insertion of *t1 = *q* above the test for null wouldn't be down-safe. Down-safety has two aspects. First, an unsafe insertion may create an incorrect program. For example, in Figure 3a the inserted load **q* would be executed before determining whether *q* is a null pointer. Second, an unsafe insertion may increase the number of instructions along some paths. In Figure 3a, the redundancy would be eliminated for the left-most path, but an extra instruction, *t1 = *q*, would be executed on the right-most path.

The standard PRE algorithm removes all the redundancies possible with down-safe insertions. Control speculation lets PRE remove redundancies on one control-flow path, perhaps at the expense of another, less important control-flow path. In the example in Figure 3a, assume that the left-most control-flow path is executed much more frequently than the right-most path. If the redundancy on the left-most path can be removed without producing an incorrect program, performance will improve even though an extra instruction executes on the right-most path. Determining the relative importance of different paths requires profile feedback or a static heuristic.

Figure 3b shows how using control speculative load *ld.s* removes the redundancy in Figure 3a. Check instruction *chk.s* replaces the original redundant load to determine whether the load caused a page fault, a segmentation violation, or some other exception. In doing so, chk.s uses no memory system resources. If the load caused an exception, control transfers to the recovery block to reexecute the load and any speculated instructions that depend on it. Eliminating the redundant load may expose further redundancies in the instructions that depend on the load.

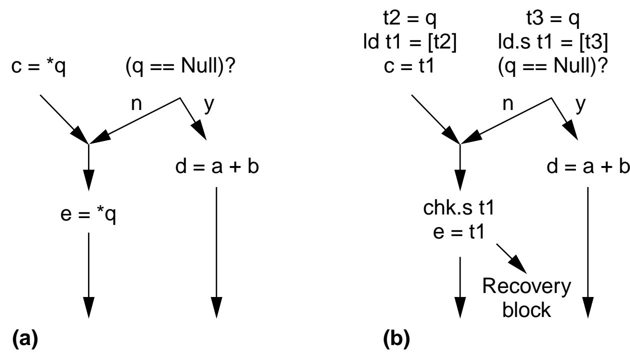

Figure 3. Redundancy elimination using control speculation: original control-flow graph (a) and after speculative PRE (b).

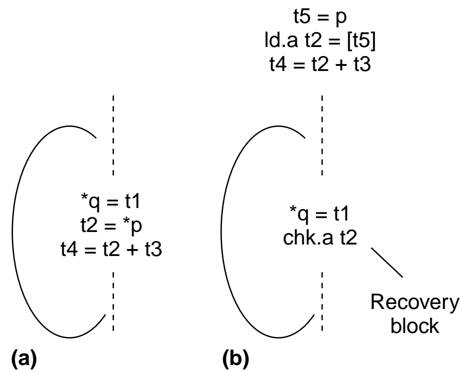

Figure 4. Removal of loop-invariant load using data speculation: original control-flow graph (a) and after speculative PRE (b).

Redundant load removal is sometimes inhibited by intervening stores rather than by lack of down-safety. In Figure 4a, the loop-invariant load **p* can't be removed unless the compiler can prove that the store **q* doesn't modify the same memory location. If the memory disambiguator can determine that there's a small but nonzero probability that **p* and **q* access the same memory location, the loop-invariant load and the add that depends on it can be removed using data speculation, as shown in Figure 4b. Advanced load *ld.a* facilitates the insertion of **p* in the preheader, and *chk.a* replaces the original

```
for (i=2;i<n;i++)
{
a(i) = a(i-1) + 1
b(i) = a(i)+a(i-1)
}
```

(a)

```
t2 = a(1)
for (i=2;i<n;i++)
  t1 = t2 + 1
  a(i) = t1
  b(i) = t1+ t2
  t2 = t1
}
```

(b)

```
b1:
  add r32 = r33, 1
  st8 [r2] = r32,8
  add r4 = r32,r33
  st8 [r3] = r4,8
br.ctop b1
```

(c)

Figure 5. An example of a scalar-variable replacement: original loop (a), transformed loop (b), and assembly code (c).

redundant load. If the store writes to the same address as the load, the check transfers control to the recovery block. This block contains code to reload **p* and reexecute *t4* = *t2* + *t3*. If the store and load access different memory locations, execution continues normally.

Scalar replacement

A general method for eliminating loads of values computed by previous iterations is scalar replacement,[4] which can perform the equivalent of store motion using dependence analysis to identify (usually total) redundancy. Scalar replacement also performs some redundant-store elimination similar to PDSE.

Scalar replacement replaces memory references with compiler-generated temporary scalar variables, which are eventually mapped to registers. Most back-end optimization techniques map array references to registers when there's no loop-carried data dependence. However, the back-end optimizations don't have accurate dependence information to replace memory references with loop-carried dependence by scalar variables. Scalar replacement, as implemented in the IA-64 compiler, also replaces loop-invariant memory references with scalar variables defined at appropriate levels of the loop nesting.

For an example of scalar replacement of memory references, consider the loop in Figure 5a. In the transformed loop in Figure 5b, all the loads from array a are replaced by compiler-inserted temporary scalar variables. In particular, note that the loop-carried data reuse of $a(i-1)$ is replaced by a scalar variable saved from a previous iteration. A scalar variable also replaces later uses of $a(i)$ and $a(i-1)$ in the same iteration (loop-independent reuse).

The IA-64 architecture provides registers that rotate one register position each time a special loop branch instruction executes.[2] This hardware feature enables the compiler to map the compiler-inserted scalars directly onto the rotating registers. In the assembly code in Figure 5c, *t1* and *t2* are assigned registers *r32* and *r33*, respectively. The assignment *t2* = *t1* (*r33* = *r32*) is done implicitly by the br.top instruction. Rau provides more details on the use of rotating registers to eliminate copies.[7]

Scalar replacement of memory references uses the direction vectors and dependence types in the data dependence graph to determine the memory references that should be replaced by scalars and how to perform the bookkeeping such replacement requires. The compiler examines the data dependence graph for each loop and partitions the memory references on the basis of whether the corresponding data dependences are input, flow, or output dependences. Memory references within each group are sorted by dependence distance and topological order. Those with loop-independent and loop-carried flow dependence are processed first, followed by memory references with loop-carried output dependence.

Cache optimizations

Processor speed has been increasing much faster than memory speed, so along with optimizations to make better use of registers, the compiler must be very aggressive in cache optimizations to bridge this gap. IA-64's compiler memory optimizations include loop transformations, blocking, and data prefetching. The latter technique tolerates memory latency, while the other locality optimizations try to make best use of the cache to reduce latency.

Prefetching with predication

Data prefetching can effectively hide memory access latency. It works by overlapping the time to access a memory location with computation time as well as with the time to access other memory locations.[4] The compiler inserts prefetch instructions for selected data references at carefully chosen points in the program, so that referenced data items are moved as close to the processor as possible before the data items are actually used. Prefetch instructions (named *lfetch* in IA-64) have one argument: the address to be prefetched. The instruction's effect is to move the cache line containing the address to a higher level of the memory hierarchy. The address itself has no cache alignment requirement.

The data-prefetching technique employs data reuse analysis to selectively prefetch only those data references likely to suffer cache misses. For example, if a data reference within a loop exhibits spatial locality by accessing locations that fall within the same cache line, then only the first access to the cache line will incur a miss. If the cache line size is 64 bytes and the array element is 8 bytes, then this reference can be selectively prefetched under a conditional of the form (*i mod 8*) == *0*, where *i* is the loop index. When multiple references access the same cache line (group locality), only the leading reference needs to be prefetched. Similarly, if a data reference exhibits temporal locality, only the first access must be prefetched.

In the example in Figure 6, the compiler inserts prefetches for arrays *a* and *b*. The references to array *a* have spatial locality, whereas the references to array *b* have group spatial locality. Here, *k* is the prefetch distance computed by the compiler. The conditional statements that control the data-prefetching policy can be removed by loop unrolling, strip-mining, and peeling. However, this may result in code expansion and additional instruction cache misses. Predication support in IA-64 provides an efficient way to add prefetch instructions. The conditionals within the loop are converted to predicates through if-conversion. The two *lfetch* instructions are predicated, and compare instructions that calculate the value of these predicates appear within the loop.

When multiple array references with spatial locality are prefetched within the same loop, the prefetches can be spread across the iterations. For example, the predicates for prefetching *a* and *b* in Figure 6 are chosen such that the prefetches are equally spaced in the eight-iteration window. This spacing tries to minimize the number of outstanding prefetches at any time and reduces contention for resources in the memory subsystem.

Apart from the prefetch instructions, the overhead associated with this kind of prefetching includes the instructions required for prefetch address calculation and predicate computation. If the loop is memory bound or floating-point bound, these additional instructions (which occupy integer instruction slots) don't increase the resource requirement within the loop. But each *lfetch* instruction uses a memory slot and may lengthen the schedule for the loop.

```
for (i=1;i< n;i++) {
  a(i) = b(i-1) + b(i+1)
}
```

(a)

```
for (i=1;i< n;i++) {
  a(i) = b(i-1) + b(i+1)
if (iand(i,7) == 0)
   prefetch(&a(i+k))
if (iand(i,7) == 4)
   prefetch(&b(i+k+1))
}
```

(b)

Figure 6. Prefetching with predicates: original loop (a) and loop with prefetches (b).

```
r33 = 80+ r16
r34 = 80+ r18
for (i=1;i< n;i++){
a(i) = b(i-1) + b(i+1)
r32 = r34 + incr
lfetch.nt1 [r34]
   r34 = r33
   r33 = r32
}
```

(a)

```
   add r33 = 80, r16
   add r34 = 80, r18
Loop:
   (p16) ldfd f32 = [r8], 8
   (p16) ldfd f37 = [r3], 8
   (p24) stfd [r2] = f46, 8
   (p20) fma f42 = f36,f1,f41
   (p16) add r32 = 16, r34
   (p16) lfetch.nt1 [r34]
         br.ctop Loop
```

(b)

Figure 7. Prefetch using rotating registers: as explicit assignments (a) and after transformation (b).

Prefetching with rotating registers

IA-64's rotating registers can alleviate the increase in resource requirements while prefetching. Multiple arrays accessed uniformly within a loop can be prefetched with a single *lfetch* instruction using a rotating register that rotates the addresses of the different arrays that must be prefetched. This obviates the need for predicate calculations within the loop and saves memory slots that would otherwise be occupied by multiple *lfetch* instructions.

This transformation for the loop in Figure 6 appears in Figure 7. Here, *incr* is a function of the cache line size, the prefetch frequency, and the number of arrays that need to be prefetched within the loop. The rotation of registers r32 → r33 → r34 is shown as explicit assignments in Figure 7a. These assignments are transformed to use rotating registers, as shown in Figure 7b. Note that just one *lfetch* instruction serves to prefetch both arrays in this loop. This method can prefetch up to eight such arrays with one instruction. The addresses of the two arrays that require prefetching are initialized before the loop.

```
for (j=1;j<1000;j++){
   y(j) = y(j) + a*x(j)
}
```

(a)

```
for (j=1;j<1000;j+=2){
   y(j) = y(j) + a*x(j)
   y(j+1) = y(j+1) + a*x(j+1)
}
```

(b)

Figure 8. Using the load-pair instruction: original loop (a) and unrolled loop (b).

```
for (i = 1;i< n;i++){
    a[i] = x*a[i] + b[i]
    *p = …
}
```

(a)

```
          t = ld.a n
          for (i = 1; i< t; i+=2){
          a[i] = x*a[i] + b[i]
              *p = …
              chk.a t, recover_1
          b1: a[i+1] = x*[i+1] + b[i]
              *p = …
               chk.a t, recover_2
               }
          b3: …
recover_1:
               t = ld n
               if (i>t) goto b3
               else goto b1
```

(b)

Figure 9. An example of speculative unrolling: original loop (a), unrolled loop (b).

Within the loop, an *lfetch* instruction issues in every iteration. Once an address has been prefetched, it's incremented by *incr*, and the addresses are rotated. If eight arrays must be prefetched, then after eight iterations the next cache line of the first array will be prefetched. For fewer than eight arrays, the same cache line will be prefetched more than once. This is an overhead associated with this scheme, and it must be weighed against the benefits. This method also requires that extra rotating registers be allocated for rotating the prefetch addresses. These registers can't serve any other purpose within the loop. Nevertheless, this scheme reduces the schedule length by eliminating the extra *lfetch* instructions and predicate computations used in the previous method.

In general, if there are multiple arrays having different stride requirements, or if there are more than eight arrays (assuming the same data size and cache line size as in the example given above), more than one *lfetch* instruction may need to be issued per iteration using this scheme.

The IA-64 compiler uses heuristics to choose the technique for prefetching on a loop basis, that is, whether to use unrolling, explicit predication, or conversion to rotating-register mode. These heuristics take into account factors such as the number of references that must be prefetched, prefetch frequency, and actual increase in schedule length.

Bandwidth optimizations

IA-64 provides instructions that load two floating-point numbers at a time into a pair of registers. Such load-pair instructions take only a single memory issue slot and thus provide high bandwidth for memory accesses.

The compiler identifies and pairs up accesses to consecutive memory locations. The paired data accesses can be found, for example, within a basic block or from different loop iterations. To exploit opportunities across loop iterations, the compiler may unroll the loop by a desired factor. Unrolling exposes the accesses to consecutive memory locations within the loop body. The unroll factor must be chosen carefully to fully utilize the resources.

Consider the loop in Figure 8. If the array elements are single- or double-precision floating-point, after unrolling the loop we will identify [y(j), y(j + 1)] and [x(j), x(j + 1)] as potential candidates for the load-pair instruction. On the other hand, if the array elements are complex numbers, a load-pair instruction can serve to access the real and imaginary parts from the same iteration. The consecutive memory locations identified depend on the data layout and needn't be obvious from the program source. For example, in C/C++, if we have x → sum = y → f1 + y → f2, where f1, f2 are adjacent fields with the correct alignment, load-pair can be applied to y → f1, y → f2 as well. In the case of loops, we must determine the unrolling factor carefully. For example, the loop in Figure 8 only needs to be unrolled by a factor of two. When load-pair is not used, there are two loads and a store per iteration. If there are two memory ports, they're not fully utilized in every iteration. With the use of load-pair, we have two load-pairs and two stores per unrolled iteration, resulting in full memory port utilization. For another example, if the loop body had one load and one store, the loop would be unrolled by a factor of four, to give two load-pairs and four stores for every unrolled iteration.

To identify loops to be unrolled for load-

pair, the compiler uses results from the data dependence analysis. For example, if the loop in Figure 8a had the statement $y(j) = y(j - 1) + a*x(j)$, load-pair couldn't be used for array *y*, since there's a loop-carried dependency of distance 1. However, if the loop had the statement $y(j) = y(j - 2) + a*x(j)$, then load-pair could still be used for array *y*.

Optimizations for ILP

Three techniques—speculative unrolling, register blocking, and loop fusion—are useful in exposing ILP.

Speculative unrolling

Loop unrolling exposes parallelism across instructions in adjacent loop iterations. The unrolled loop body is much larger than the original loop, so it exposes larger regions for scheduling. Since the unrolled loop iterates fewer times, loop unrolling reduces the number of dynamic branches. The large number of registers in IA-64 enables the compiler to unroll loops by significantly larger factors without register spills than compilers for other contemporary architectures. More importantly, predication and speculation enable the compiler to generate efficient code for unrolled loops.

Figure 9 illustrates speculative unrolling of a loop by a factor of two. In the original loop in Figure 9a, we can't determine the number of loop iterations because *n* and **p* may share the same location. Therefore, when the loop is unrolled, checks for the exit condition must remain each time the original loop body is copied. However, when we know that the loop modifies the loop bound very rarely, the compiler can use IA-64's speculation support to produce an efficient unrolled loop. The upper bound of the unrolled loop in Figure 9b is data speculative, and appropriate checks monitor the changes to the upper bound. The recovery code, which we expect to execute rarely, takes care of loop exit conditions, updating of the new loop bound, and return to normal execution.

Register blocking

This technique turns loop-carried data reuse into loop-independent data reuse and exposes more ILP from outer loop iterations. Register blocking is similar to loop blocking or tiling, with relatively smaller tile sizes, followed by an unrolling of the iterations in the tile. Figure 10 demonstrates register blocking. This process takes advantage of the large register file to map the references to many of the common array elements in adjacent loop iterations onto registers.

```
for(j=1;j<2*m; j++){
  for(i=1; i<2*n;i++){
    a[i,j]=a[i-1,j]+a[i-1,j-1]
  }
}
```

(a)

```
for(j=1;j<2*m; j+=2){
  for(i=1; i<2*n;i+=2){
     a[i,j]=a[i-1,j]+a[i-1,j-1]
    a[i+1,j]=a[i,j]+a[i,j-1]
    a[i,j+1]=a[i-1,j+1]+a[i-1,j]
    a[l+1,j+1]=a[i,j+1]+a[i,j]
  }
}
```

(b)

Figure 10. An example of register blocking: original loop (a), register-blocked loop (b).

The original loop in Figure 10a has two distinct array read references in every iteration. The register-blocked loop in Figure 10b has only six distinct array read references for every four iterations in the original loop. Note that two of the six references are loop-independent reuses.

Loop fusion

This technique combines adjacent conforming nested loops into a single nested loop.[4] Besides improving locality of data reference, loop fusion exposes more ILP by increasing the scheduling region in the loop body.

Loop fusion for IA-64 can be more aggressive than in compilers for other processors because it can take advantage of many available registers. When the original loop in Figure 11a is fused in Figure 11b, cache locality is also improved because the accesses to array *a* are reused within the same loop. Further, it enables the compiler to replace references to arrays *a* and *d* with references to compiler-generated scalar variables.

```
for (j=1;j<1000; j++){
  a(j) = x
}
for (j=1;j<1000;j++){
  c(j) = a(j-1)+ d(j-1)
  d(j) = c(j)
}
```

(a)

```
for (j=1;j<1000; j++){
  a(j) = x
  c(j) = a(j-1)+ d(j-1)
  d(j) = c(j)
}
```

(b)

Figure 11. An example of loop fusion: original loop (a) and after loop fusion (b).

The Intel IA-64 compiler incorporates the technology necessary to efficiently exploit the IA-64 architecture features for improved integer and floating-point performance. The compiler applies loop transformations, data transformations, and global scalar optimizations. All compiler optimization techniques are aware of profile information and effectively use interprocedural analysis information. MICRO

Acknowledgments

We thank the Intel IA-64 compiler team members for their contributions to the compiler technology described in this article. We thank the reviewers for excellent and useful suggestions for improving the presentation.

References

1. C. Dulong, "The IA-64 Architecture at Work," *Computer*, July 1998, pp. 24-32.
2. J. Huck et al., "Introducing the IA-64 Architecture," *IEEE Micro*, Sept.-Oct. 2000, pp. 12-23.
3. F. Chow et al., "A New Algorithm for Partial Redundancy Elimination Based on SSA Form," *Proc. ACM SIGPLAN 97 Conf. Programming Language Design and Implementation*, ACM, New York, 1997, pp. 273-286.
4. S. Muchnik, *Advanced Compiler Design and Implementation*, Morgan Kaufman, San Francisco, Calif., 1997.
5. D. Gallagher, *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*, PhD thesis, Dept. Electrical and Computer Eng., Univ. of Illinois at Urbana-Champaign, 1995.
6. R. Ju, J. Collard, and K. Oukbir, "Probabilistic Memory Disambiguation and Its Application to Data Speculation," *Proc. Third Workshop Interaction Between Compilers and Computer Architectures*, Univ. Minnesota, Minneapolis St. Paul, Oct. 1998.
7. B.R. Rau, "Data Flow and Dependence Analysis for Instruction-Level Parallelism," *Proc. Fourth Int'l Workshop Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, Springer-Verlag, Berlin, 1992, pp. 236-250.

Rakesh Krishnaiyer works on high-level optimizations for the IA-64 compiler at Intel. He received a BTech degree in computer science and engineering from the Indian Institute of Technology, Madras, and MS and PhD degrees from Syracuse University.

Dattatraya Kulkarni is on the Intel IA-64 compiler high-level optimizer team. He received a PhD degree in computer science from the University of Toronto, Canada.

Daniel Lavery is a member of the IA-64 compiler scalar optimizer team. He received a PhD degree in electrical engineering from the University of Illinois.

Wei Li leads and manages the high-level optimizer group for IA-64. He received a PhD degree in computer science from Cornell University.

Chu-cheow Lim is on the Intel IA-64 compiler high-level optimizer team. He received BS and MS degrees from Stanford University and a PhD degree from the University of California at Berkeley.

John Ng is on the Intel IA-64 compiler high-level optimizer team. He received a BSc degree in mathematics from Illinois State University and an MS degree in computer science from Rutgers University.

David Sehr is the group leader and manager for the IA-64 compiler scalar optimizer. He received MS and PhD degrees from the University of Illinois.

Direct questions and comments about this article to Wei Li, Microcomputer Software Laboratories, Intel Corp., SC12-305, 2200 Mission College Blvd., Santa Clara, CA 95052; wei.li@intel.com.