Coding Considerations

Practical Methods to Maximum Efficiency for Intel® Itanium® Architecture

Author: Max Alt

June, 2004

Intel® Solution Services is Intel Corporation's worldwide professional services organization, helping enterprise companies capitalize on the full value of Intel architecture through consulting focused on architecture transitions.

Backed by the largest silicon manufacturing company and one of the largest e-Business corporations in the world, Intel Solution Services uses its foremost expertise in Intel architecture and next generation technologies, as well as its relationships with key industry alliances, to design cost-effective, leading-edge solutions that help deliver superior business results.

Our services are available through on-site consulting, as well as at Intel Solution Centers located worldwide. The Intel Solution Centers are state-of-the-art environments for designing and testing high-performance solutions using Intel's best-known methods and technologies.

For more information about Intel Solution Services, visit our Web site at www.intel.com/internetservices/intelsolutionservices or email us at solution-services-questions@intel.com. To contact us by telephone, call toll free in the U.S. at 866-268-9812; Europe, Middle East and Africa at +44 118 944 7931; Asia Pacific at +852 2844 4555 and Japan at +81 3 5208 5375.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Abstract

All computer programs can benefit from specific tuning efforts aimed at extracting maximum performance on any given processor architecture. While it is relatively easy to derive peak performance values from the characteristics of the architecture and the implementation, it is difficult to approach that peak performance without extensive experience in tuning applications.

Mature compilers can intuitively make many decisions. Yet "out-of-the-box" does not mean "black-box" for even mature compilers, and developers should be aware of what is going on behind the scenes. Developers still need to know how to evaluate what theoretical maximum may be and what should be done in the source code to encourage the compiler to generate close-to-theoretical maximum code.

This paper provides a distillation of the most applicable techniques for tuning numerically-intensive codes on Intel® Itanium® architecture. It may repeat some traditional coding techniques, but primarily it highlights a different angle in high-performance computing practices. It explores issues around mature compilers, with the goal of achieving near-to-optimal theoretical scheduling. It explains how the compiler works so the developer can measure the effectiveness of the tuning effort, and further, how to offer hints to the compiler for it to make even better decisions and come even closer to the theoretical maximum code.

Table of Contents

Introduction

This paper targets any software developer who strives to analyze, characterize, and tune the performance of a "High Performance Computing" (HPC) application, especially ones that are memory or floating–point intensive. This paper is not a tutorial, but rather a survey of what on the surface may appear to be traditional coding techniques. However, most of the techniques presented here, as well as some new concepts, showcase optimization efforts from a different angle.

The optimization techniques, however, have been placed in the Appendix because the goal of this paper is to demonstrate that most of the optimizations can and should be done by the compiler. This paper highlights compile-time analysis and advocates "out-of-the-box" performance, minimizing developer's efforts to manually change the source code to obtain better performance. It explains what optimizations a developer should expect from a compiler and how to influence them to occur. However, "out-of-the-box" does not mean "black-box" to the compiler, and developers should be aware of what is going on "behind the scenes." In this paper, the reader should be able to relate to both sides: not requiring any changes to obtain "out-of-the-box" performance by the compiler and at the same time, understanding the compiler's internals to guide the compiler to generate the optimal code

In the beginning of this paper, the concept of "efficiency" is introduced. The discussion of application efficiency will lead to five levels of efficiency on Intel® Itanium® processors; each characterization may assist in moving an HPC application to a theoretical maximum. Central to the performance of many HPC applications is floating point performance. On the Itanium architecture, it is necessary both to balance the ratio of potentially long latency instructions (such as memory access to floating point operations) and to balance the mixture of instructions to exploit the multiple units that function in parallel. Definitions of five key aspects of efficiency will be provided, and an exploration of the role IPC (Instructions Per Cycle, see Glossary) plays in measuring these various types of efficiency.

Later on, this paper emphasizes the importance of compile-time analysis, showing how to evaluate and analyze the Intel compiler's output for the Itanium architecture, and how to manipulate that compiler's high-level optimizer to generate more efficient code. Additionally, several source code transformation techniques are presented, with the goal of presenting the compiler the correct mix of operations to allow the High-Level Optimizer (HLO) to effectively schedule the instructions. The performance benefit of techniques presented in the paper is focused around the concept of "loop balancing" with respect to pressure on functional units of Itanium processors, but primarily with respect to floating-point-to-memory ratio. In addition, the in-depth compiler discussion explores some considerations in applying commonly used optimization techniques, discussing potentially negative impact on performance from loop unrolling, prefetching, and software pipelining, both individually and in combination.

The knowledge of assembly language becomes less important as a developer attempts to evaluate the compiler's generated code. The compiler provides an interface to trace

invoked optimizations and therefore verify assumptions made. However, along with understanding the main optimization techniques and common compiler algorithms, it remains important to know the architecture, the implementation, and how to evaluate whether the compiler used appropriate heuristics and generated efficient code. Along with the compiler flags, directives and optimization reports, the source code still remains the main interface to the compiler.

This paper gives a glimpse into the art of tuning an HPC application, with a carefully selected set of techniques and guidelines. At the same time, while we are trying to emphasize the importance of “out-of-the-box” application performance, the paper conveys ways and techniques to guide mature compiler engine algorithms to generate an efficient code without manual code transformations by using extensive compiler interface features.

Definitions of Efficiency

Efficiency is defined *as the fraction of time the application is doing useful work.* However, this statement is trite unless "useful work" is properly defined, as it is clear that a highly utilized CPU does not necessarily imply that anything useful is being accomplished. Instead, measurements must be defined and then used to help guide the tuning of the application. The following set of definitions will be sufficient:

Instruction Count: The number of instructions retired by the processor during a measured execution interval. *N* in the formula below denotes Instruction Count over a code section.

Instructions Per Cycle (IPC): The number of cycles it takes to complete a single instruction. *c* in the formula below denotes the average number of instructions retired per cycle.

Thus, if *f* is the processor frequency,

$$t = N / (c \times f)$$

Floating-Point/Memory (F/M) ratio: The ratio of floating point operations to memory operations (supported by the architecture) over a particular code section.

If a developer were to improve time *t* in the formula above, the traditional methods would imply minimizing instruction count *N*, and maximizing average instructions retired per cycle *c*.

The instruction count can first be improved by a better algorithm and matching the algorithm to a platform. Many books have been written on the subject of how to model a better algorithm. This paper emphasizes that creating an optimized algorithm is the primary step in performance tuning. It is an absolutely useless task to tune a non-efficient algorithm.

A good choice of instructions, compiler, compiler optimization and guiding compiler heuristics may also help to minimize instruction count *N*, and will be discussed later in the paper.

To maximize *c*, the average instructions per cycle, an environment should efficiently execute instructions on the processor pipeline. This can be done by:

- Making available a large number of the processor's resources
- Lowering latency for resources and instructions
- Overlapping (concurrent) instruction stream execution by improving instruction level parallelism
- Having a good instruction mix and instruction schedule

Iterative Approach to Efficiency

The methodology defined in this paper for optimizing and maximizing efficiency consists of five major steps, with each step subject to iteration and feedback from succeeding steps. The five steps are:

- Algorithm selection
- Algorithm mapping
- Optimization of microscheduling
- Balancing the F/M ratio
- Run-time analysis.

The progression through these steps is linear and iterative. Earlier steps should not be skipped, as each activity depends on the results of its predecessors.

a. **Algorithm selection**, or optimal mathematical modeling of the algorithm, as mentioned earlier, is the most important step, as tuning an inefficient algorithm rarely gives acceptable results.

b. **Algorithm mapping** (implementation) is the realization of the selected algorithms into source code. Some of the source code may leverage the platform architecture through specific structuring of the code to take advantage of platform strengths, while other parts of the source code may be generic. Both types of source code should reflect good coding practices, which will ease the code tuning process. General discussion of coding practices is outside the scope of this paper; however, specific examples may be given as needed.

c. **Optimization of microscheduling** (maximal instruction-level parallelism) is clearly separated into two steps:

   - The first step is related to algorithm implementation (discussed above), where a developer maximizes instruction level parallelism and optimizes IPC using optimization techniques and coding practices, attempting to address the platform architecture on a high level.

   - The second step is the compiler's responsibility and maximizes instruction level parallelism through better code generation, larger scope compilation, global optimizations, and steering correct assumptions and heuristics in the compilation (assuming optimal compiler flags and directives were used). The last step implies code generation with optimal scheduling, according to the rules of the processor microarchitecture, laying the base for theoretically maximum performance.

d. **Balancing the F/M ratio** to match the processor microarchitecture is done through modifying the source code to balance memory operations with floating point operations or integer operations. The goal of this step is to enable optimal microscheduling with minimal synchronization between execution units in execution hotspots identified through run-time profiling. It will be necessary to iterate the previous step (c) to achieve the highest efficiency. Based on advanced abilities of the compiler to optimally schedule a code, where memory operations are balanced with floating point operations, a developer may assist the compiler's microscheduler to generate the code close to the theoretical maximum.

e. **Run-time execution** is instrumented and measured to discover data access irregularities: DTLB (Data translation look-aside buffer) misses, cache misses, bank conflicts, secondary misses, OzQ cancels, etc. Some of these issues can be handled with work-arounds by using appropriate directives to the compiler, while some problems will require iteration through earlier steps to achieve the highest performance. The goal of this step is to supply the processor pipeline an instruction stream for efficient execution, considering potential data access irregularities which may occur during execution of the best possible code derived from stages a – d.

In conclusion, improving the efficiency of an application becomes an iterative methodology of these steps to compose a closed loop, starting from compile-time analysis and code transformation based on hardware synthesis and resource balancing (see Figure 1), moving to run-time analysis and detection of data access irregularities, and then back to the compiler to feed back the run-time data obtained, via additional human guidance, to the compiler engine.

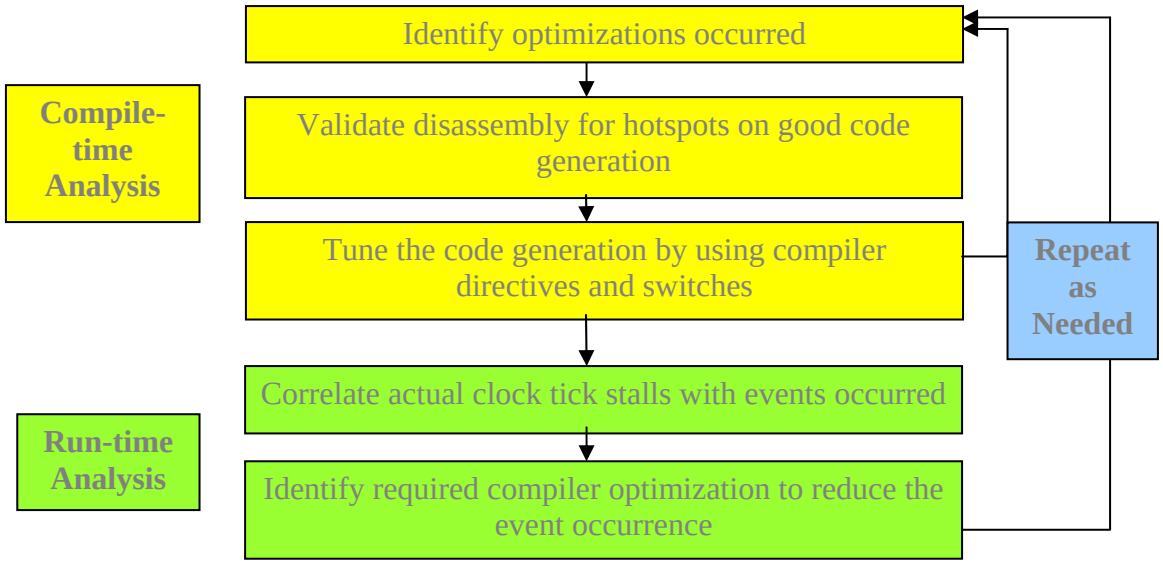

Figure 1. Our approach to application optimization and tuning

Compile-time analysis

Before discussing these steps further, it is important to assess what role the IPC plays in evaluating the efficiency. The optimal IPC is only one of the necessary conditions for efficient execution; however, it is far from being sufficient. Since the Itanium microarchitecture can execute six instructions per cycle, the optimal IPC for Itanium processors is 6:

- Having the optimal maximum IPC within a code block does not imply any conclusion about execution efficiency nor the algorithm, nor its implementation efficiency. For instance: IPC cannot reflect how efficient the math model of the algorithm is.

- Wrongly speculated and retired instructions are counted in IPC.

- Falsely predicated and retired instructions are counted in IPC. In other words, falsely predicated instructions are not NOP instructions, but they turn into NOP instructions when executed.

- Branch-mispredicted instructions do not affect IPC as they were not retired. In other words, if an algorithm is branchy and the hardware frequently mispredicts branches, the stalls it causes do not affect the IPC count.

Beyond the discussion of IPC and improving the instruction level parallelism with the coding practices below, it is important to introduce the concepts of advanced compile-time analysis and research, aimed at improving efficiency steps.

Once again, the F/M ratio compares the number of floating-point operations to the number of memory operations. This concept requires a scope, or a context, in which the F/M ratio was measured. For example, having *F/M = 1* per hotspot means that within a code region, we have, on average, one executed floating-point operation per one executed memory operation.

Loop balancing based on F/M ratio ("loop balancing") means that, within a scope of loop kernel, the floating-point execution unit and memory unit are synchronized in such a way that they both can execute without stalls.

The main property of a balanced loop is that it does not introduce particular pressure on any of the execution units. The compiler's microscheduler is responsible for generating optimally scheduled code, based on the platform's microarchitecture rules for balanced loops.

Dense code for a balanced loop would mean generating close to the maximum possible performance loop. The execution of a perfectly balanced loop by a parallel architecture can be compared to a complex watch mechanism or a model of asynchronously broadcasted waves with a variety of amplitudes.

The F/M ratio and loop balancing are more theoretical concepts than practical implementations. The actual metrics of the execution and performance would be the number of bytes per FLOP (Floating point operation) or bytes per IOP (Integer operation) that can be processed. The “byte per FLOP” concept and analysis of how to distinguish the F/M ratio during execution will be part of the discussion in this paper.

Coding Practices

When developing or modifying code, the developer will be rewarded by certain coding practices that ease the characterization of performance and allow the compiler's HLO heuristics to recognize, automatically or through hints, possible code transformations. This section relates to "Algorithm mapping," step b (see section above) of improving the efficiency.

When characterizing the performance of an application, one of the most important steps is estimating loop trip counts. Loop trip counts directly correlate to the applicability and usefulness of many of the HLO heuristics. The compiler does not have information about trip counts, while the developer knows (or can derive) this information. Loop trip counts are crucial for the effectiveness of the compiler's HLO heuristics because an incorrectly estimated count can cause the compiler to make incorrect decisions regarding code transformations. While the compiler may create versions for the code to separate a favorable assumption from the general case, it is important to understand an algorithm optimization opportunity to modify the iteration space so the inner-most loop has the majority of iterations.

The following coding practices are attempts to trigger HLO optimizations, invoking specific heuristics for generating efficient code:

- **Data alignment specification**. Correct data alignment must be explicitly provided, or hints must be given to the compiler on actual alignment.

- **Data caching awareness**. Data must be properly blocked to fit properly in the cache, or the cache blocking provided by the compiler must be validated. In the event this is not feasible, hints can be provided for the compiler to perform pre-fetching.

- **Long latency instructions**. Careful thought must be given to control long latency instructions. A developer should be aware of the long latency operations used in the code. This cannot always be avoided, but there are techniques to address long latency instructions and to ease the execution unit pressure they cause. When long latency instructions must be used, a good practice is to guide the compiler code generation to leverage other functional units as well. Look for opportunities to vectorize such operations, or choose a different implementation of the operation to increase the throughput.

- **Run-time memory management**. Data allocations, data structures, data layout and access patterns. Close attention to run-time data allocation and to memory access patterns helps avoid memory access irregularities at run-time. Simple data structure references and using arrays instead of dynamically allocated data are always preferable for achieving better performance. The attention to data layout, memory reference expressions and memory access patterns helps prevent issues such as bank conflicts, false sharing, DTLB misses, misalignment and secondary cache misses.

- **Analyze memory reference increments**. Or in other words, analyze prolog (preparation steps) to compute memory references for the next loop iterations. Keep increments ungeneralized or create special cases for known frequent values.

  Examples of performance-degrading generalizations:

  ```
  ix+=ix + isize;

  jx+=jx +jsize;

  j=j->next
  ```

  Dynamically computed increments would somewhat limit the ability to unroll the loop, generate loadpairs and make prefetching predictions more difficult. It also requires the scheduler of software pipelined loops to generate more code to compute the increments and new memory addresses.

- **Generalizations and complex code**. One of the most common examples of introducing performance tradeoff is when attempting to create a general usage model for an algorithm. Generalizing code often requires using more computing resources and dynamic decision-making logic. Such code defeats the heuristics of the HLO by introducing code complexity or excessive control flow inside loops.

  Examples of performance degrading generalizations:

  ```
  For (i=0; i < n; i++) {

  if (I ==1) {….}

      Else { if (J==I) { …. } } }
  ```

- **Complex memory references**. Or in other words, analyzing references such as complex pointer computations, strain the ability of compilers to generate efficient code. Places in the code where the compiler or the hardware is performing a complex computation in order to determine where the data resides, should be the focus of attention. Pointer aliasing and code simplification assist the compiler in recognizing memory access patterns, allowing the compiler to overlap memory access with data manipulation. Reducing unnecessary memory references may expose to the compiler the ability to pipeline the software. Many other data location properties, such as aliasing or alignment, can be easily recognized if memory reference computations are kept simple. Use of strength reduction or inductive methods to simplify memory references is crucial to assisting the compiler.

  Example 1: Indirect access A[B[i]] – many decisions could not be made and heuristics cannot be applied when the compiler or hardware seeks for access patterns or attempts to predict memory references.

Example 2: A piece of Fortran code which was converted to C code (or a piece of C code which was converted to Fortran), which contains 2-D memory accesses – after the translation you can see accesses in the form of:

A[n × dim + i - 1], or pointer annotation of a similar expression.

Example 3: A notation of form:

n->calc->ttt = n->calc->bbb + d->calc->offset, where n=d->next

- **Detection of possible invariants** relative to the loop index variable. The situation may get worse if these values are referred by a pointer, which may inhibit the ability of the compiler to determine whether the expression is a constant.

Example 1: Checking on an environment within the loop to trigger cases:

```
For (i=0; i < n; i++) {
  If (env->off) { ….}
  Else {… } }
```

Example 2:

```
While (nodes--)  {
if (env->isisolated)
   n->calc->xxx = d->calc->yyy + t->dd->f->x;
   d = n;
   n = n->calc;
}
```

Similar to Example 1, *env->is isolated* is not changed in the loop, and the test can be carried outside of the loop. The expression *t->dd->f->x* is a constant value relative to the loop indexing, and being referred by a pointer assumes that the value may be changed during each iteration of the loop. And therefore, the compiler automatically assumes it needs to reload the value of *x* with each loop iteration.

Example 3: Computing a constant or referring to a constant memory location within the loop:

```
For (J=0; J<M; J++)
For (I=0; I<N; I++) { …
if (j == NN)
   S(J) = S(J) + A(I) × B(I); } }
```

- **Identifying the possibility of using an array of data structures** rather than a dynamically allocated linked list of structures.

Example: The loop described above (see Example 2) iterates on a linked list of nodes. The compiler would have an impossible task to predict what length the loop is, and would have an extremely limited set of heuristics to apply: Automatic unrolling would be an impossible task, and prefetching would be difficult as well, since memory references cannot see beyond one iteration, etc. Complex memory references and ambiguity might prevent the loop from being pipelined.

If instead of dynamically allocated "calc" structures there could be a pre-allocated array of structures whose sizes are equal to the size of the nodes, the comprehensive set of the compiler's heuristics can be applied, and scheduling the loop above in the software pipeline would generate much more efficient code.

- **Detecting generalized handling of non-similar cases** within the kernel of the high-performance loop. Consider using the loop peeling technique instead of splitting the loop on individually handled cases.

  Example:

```
For (i=0; i < N; i ++) {
  If (i == 0)  F(i) = b;
    else if (i==1) F(i) = c;
    else { F(i) = F(i-1) – F(i) };
  }
```

  Loop peeling will resolve the issue of having a control flow within the loop:

```
F(0) = b; F(1) = c;
For (i=2; i < N; i ++) {
  F(i) = F(i-1) – F(i) ;
  }
```

- **Detecting complex "if" statements in the loop**, as it hinders the ability to pipeline the software and creates more opportunities for the compiler to generate speculative or advanced load instructions.

  Example (molecular dynamic code):

```
!DIR$ swp
      do 6002 m=1,max1
      jat=idv(m)
      jdx=id(jat)
      if (jdx.ne.1) goto 6002
      if(m.eq.im) goto 6002

      dxt=x0v(m)-x0v(im)
```

```
          if(dabs(dxt).gt.acutal) goto 6002
          dyt=y0v(m)-y0v(im)
          if(dabs(dyt).gt.acutal) goto 6002
          dzt=z0v(m)-z0v(im)
          if(dabs(dzt).gt.acutal) goto 6002
          rijsq=dxt*dxt+dyt*dyt+dzt*dzt
          if(rijsq.gt.acutal2) goto 6002
          sk=rijsq*csi
         index=int(sk)
         skf = sk - dble(index)
         dk = skf/csi
         index = index + 1

         if(index.gt.ngrid-1) then
            index = ngrid-1
         endif

         denspi = rhop(idx,index) +
         +    (rhop(idx,index+1)-rhop(idx,index))*skf
          denspj = rhop(jdx,index) +
         +    (rhop(jdx,index+1)-rhop(jdx,index))*skf
          fcp = denspj*embfp(iat)+denspi*embfp(jat)
          atpe(iat) = atpe(iat) + 0.5d0*pp
           fx(iat) = fx(iat) + dxt*fcp
           fy(iat) = fy(iat) + dyt*fcp
           fz(iat) = fz(iat) + dzt*fcp
          skij = 0.5d0
          st11 = fp*dxt*dxt*skij
          st22 = fp*dyt*dyt*skij
          st33 = fp*dzt*dzt*skij
      if(iat.le.nn1) then
           atomstt(iat) = atomstt(iat) - st11
           atomstt(iat) = atomstt(iat) - st22
          atomstt(iat) = atomstt(iat) - st33
           endif
6002       continue
```

In this example, fairly large code is being software pipelined. Complex and frequent control flow generates non-optimal scheduling. The beginning of the loop checks whether the processed point is in the sphere. If the overhead is not high and it is algorithmically correct to process all the points without filtering, it is recommended to eliminate all the control flow related to the point location check. When the loop is software pipelined, the microscheduler may hoist latencies of floating-point computations related to points outside of the sphere. Microscheduling the loop becomes complex and inefficient if the software pipelining engine attempts to schedule the entire control flow, not knowing what case will be frequently executed.

Another group of “if” statements may be eliminated or reduced when applying loop peeling techniques (see above) or scalar replacement. In the same example, the *if (iat.ne.nn1)* statement is not necessary within the loop, since the *iat* variable is not changing with loop iterations. It is external to a loop environment variable, which is an invariant relative to the loop index. The statement with the control flow can be carried outside of the loop without implication on the algorithm.

- **Detecting unnecessary memory references** in the loop notation:

  Example:

```
for (i = j + 1; i <= *n; ++i) {

  X(i) -= temp * AP(k); }
```

The notation for the loop boundaries contains the pointer or memory reference. The compiler does not have any means to predict whether the value referenced by a pointer *n* is being changed with loop iterations by some other assignment. This causes the loop to reload the value referenced by *n* for each iteration. The code generator engine also may deny scheduling a software pipelined loop when potential pointer aliasing is found. Since the value referenced by pointer *n* is not changing within the loop and it is invariant to the loop index, the loading of **n* has to be carried outside of the loop boundaries for simpler scheduling and pointer disambiguation.

- **Recurrence and induction.** Recurrent relations and inductive solutions should be utilized as alternatives for straightforward computations and vice versa. Use of recurrence often occurs naturally as a result of avoiding complex code and complex memory references. The inductive approach often implies the introduction of a loop construct with some resource dependencies in every loop iteration. If a loop was already there, an additional inductive computation may cause loop imbalance.

  Sometimes, a case for deduction will arise, replacing a straightforward computation with an inductive solution. Both methods are valid, and strictly depend on the particular situation.

  Example: Fibonacci numbers can be done either as:

  A loop:

  For (i=2, i<N; i++)

  A(i) = A(i-1) + A(i-2)

  or directly:

  A(k) = (½)^k × ((sqrt(5)-1)^k + (sqrt(5)+1)^k)

- **Inlining** of frequently used and relatively small functions. This eliminates the overhead of call/return and dead code, and last but not least, exhibits new opportunities for compiling code and loop transformations beyond function call boundaries. Inlining may be automatic or manual. Automatic inlining uses built-in tools to determine what the compiler chooses to inline and what not to. The compiler decision for inlining may be subject to a developer's evaluation and may be a base for manual inlining. It is a good practice to use IPO (inter-procedural optimization) reports or any automatic inlining decisions to determine what

should be inlined manually if the user does not want to use special automatic flags for inlining.

- Detecting or **explicitly exhibiting the short code component operating on vectors,** which may be implemented by performance libraries now or in the future, or the compiler may be able to recognize and highly optimize certain code patterns. Those components should have sequential memory access and be either memory or ALU/FP intensive or both. Such a code component should be inlined in your code and should not be logically separated from your main computation. To do this a developer may need to introduce temporary variables or temp arrays, increase memory consumption, and distribute loops.

  Example: Matmul, DAXPY, DDOT scatter and gather operations, square root for a vector, cosine for a vector, etc.

The coding practices above are not sufficient to obtain a well-generated code. However, they can assist the compiler in making maximal usage of the HLO heuristics by recognizing optimization opportunities in the source code.

High-Level Optimization Techniques and Practices — Code Transformations

Modern compilers have the ability to perform automatic (and hint-driven) code transformations analogous to manual code transformations, often included in a compiler's High-Level Optimizer. The most common techniques include *loop interchange, loop unrolling, loop distribution, loop fusion, loop peeling, loop skewing, strength reduction and cache blocking*. In most HPC applications, these techniques can yield significantly improved performance. However, the compiler by itself frequently cannot determine whether a particular code transformation should be invoked. Additional information from the developer (transferred to the compiler via source code or compiler interface) is often required.

In the previous section about improving efficiency through **algorithm mapping**, the coding practices discussed can exhibit this information about the source code. This may potentially allow the compiler to detect opportunities to use high-level optimization techniques to provide better scheduling by invoking built-in hints or heuristics in the compiler.

HLO is the next step in the "drilling down" technique to improve efficiency of the generated code, and relates to the following steps of improving application efficiency: **Optimization of microscheduling**, and **Balancing the F/M ratio** to match the processor microarchitecture.

Assuming that the reader is quite familiar with traditional optimization techniques, their definitions and purposes, this paper focuses on the interaction between those techniques and presents the techniques from a different angle. In addition to the most commonly used techniques described in the Appendix, the technique of *loop balancing* has been introduced to assist developers to approach a solution for the compiler to achieve optimal scheduling.

The *loop balancing* technique can be found in advanced compiler literature related to the **hardware synthesizer**. There are rules of the microarchitecture that define the theoretical maximum performance or theoretical maximum pressure on execution units during loop kernel execution. The key algorithms that usually need to be synthesized are matrix multiply, dot product, or other simple and common vector computations. Another example is FFT-related algorithms. The hardware synthesis is, in the end, a run-time characteristic, but it has to be supported by ideal scheduling by the compiler, assuming theoretically optimal scheduling to exploit maximum performance of the architecture.

When an HPC cluster runs those algorithms for thousands of computing hours, it is highly desirable for metrics like timing, area and power consumption to be as close to the theoretical maximum performance as possible. Despite the complexity of providing the automatic hardware synthesis – it is still very much a compiler problem, with at times, human-provided guidance – loop distribution is one of the best techniques for achieving balanced pressure on execution units.

It is important to detect or exhibit balanced code sections in your code which generate the hardware synthesis for a period of the application run. Modifying scheduling for this code section, one would be able to control the CPU power consumption, thermal effects or intensity of execution units, cache or memory access.

By applying the loop distribution technique we change the pressure on the execution units to "our favor," and as implied, we can resynchronize the execution of the functional units in a balanced manner.

The loop distribution technique also can be used to separate operations that are using the same processor's functional units. Doing so may increase parallelism by allowing overlap of non-conflicting operations within distributed loops.

There are other side effects or undesired events, related to hardware design, which can occur on loops with a kernel that has a clear pressure on one functional unit (especially if it is the memory unit). These aspects will be discussed in detail with run-time performance and data access irregularities.

To summarize, the loop distribution code transformation is also used to:

- Create tighter code, maximizing instruction level parallelism and the IPC.
- Reduce empty cycles.
- Assist in reaching the hardware synthesis or theoretical maximum per single loop by guiding the compiler's microscheduler to produce a balanced number of FLOPs and IOPs per load/store (according to scheduling theory and the hardware's microarchitectural rules of operations that can be combined and scheduled within one cycle).
- Reduce absolute pressure on a single execution unit, or reduce **relative pressure** to other functional units during execution of every loop iteration. For instance, reducing the number of memory references, which reduces pressure on the memory unit, implies lowering potential primary and secondary misses for L2, and then minimizes pressure on L2 OzQ.

  Example 1: rmatmul.c

```
for ( kk = 0 ; kk < kmax ; kk++ ) {

        for ( jj = 0 ; jj < jmax ; jj++ ) {

                for ( ii = 0 ; ii < imax  ; ii++ ) {

                        i = ii + jj*jp + kk*kp;

                b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] + dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i]
* xdcr[i] +   dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i] + cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i]
+ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xccr[i] + cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i] + ubl[i] *
xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i] +  ucl[i] * xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i] +  ufl[i] * xufl[i] +
ufc[i] * xufc[i] + ufr[i] * xufr[i] ;
```

```
} }
```

There are over 30 memory references accessed through one loop iteration, and only 14 FMA (Floating-point Multiply Add) operations executed during loop iteration. This loop clearly generates a pressure on the CPU memory unit and L2 OzQ. The loop above is not balanced relative to F/M ratio.

Since the kernel represents the sum of FMA terms, the author experimented with the number of FMAs computed within one loop iteration.

Whether the iteration kernel is:

```
b[i] += dbl[i] * xdbl[i];
```

or:

```
b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i];
```

or:

```
b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] + dcl[i] * xdcl[i];
```

It impacts the F/M ratio because adding each FMA term adds two additional memory accesses. The added pressure on the memory unit leads to greater deviation from a balanced loop, and therefore may generate less dense code with more empty cycles in the loop as the number of FMA terms increases.

Some high-level optimizations would be limited to being invoked in the loop above: the compiler may run out of rotating registers when they decide to software pipeline, and then, it will not be able to unroll this loop. The number of static registers required to implement one iteration of the loop increases significantly with the unrolling factor.

A better, optimized and software pipelined version is presented below. The developer is suggesting to the compiler where a line for the optimal number of FMA terms should be drawn and where the loop needs to be distributed. The compiler is using the “introducing temporary array” technique to internally implement the loop distribution:

```
for ( kk = 0 ; kk < kmax ; kk++ ) {

        for ( jj = 0 ; jj < jmax ; jj++ ) {

double tmp;

#pragma ivdep

                        for ( ii = 0 ; ii < imax  ; ii++ ) {

                                i = ii + jj*jp + kk*kp;
```

```
                    tmp = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] + dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i]
* xdcr[i] +   dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i] + cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i]
+ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xccr[i];

#pragma distribute point

 b[i] = tmp + cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i] + ubl[i] * xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i] +
ucl[i]

* xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i] +  ufl[i] * xufl[i] + ufc[i] * xufc[i] + ufr[i] * xufr[i] ;

} }
```

The notation above is equivalent to two distributed loops (iterating on ii) with identical bounds separated at the point of “pragma,” where the data is passed from one loop to another through the “tmp” array. The variable tmp is expanded from variable to the array in order to support the loop distribution implementation.

Example 2: smg2k.c

```
for (loopk = 0; loopk < nz; loopk++) {

  for (loopj = 0; loopj < ny; loopj++) {

    for (loopi = 0; loopi < nx; loopi++)        {

                    rp[ri] -= Ap[Ai] * xp[xi];

                    Ai++; xi++;ri++;

        } //for loopi

Ai += sy1 - nx; xi += sy2 - nx; ri += sy3 - nx;

        } // for loopj

    Ai += sz1 - ny; xi += sz2 - ny; ri += sz3 - ny;  } //for loopk
```

It may be beneficial to distribute the loop in order to generate denser code per loop kernel and get closer to a balanced loop:

```
for (loopk = 0; loopk < nz; loopk++) {

  for (loopj = 0; loopj < ny; loopj++) {

double tmp;

#pragma ivdep

    for (loopi = 0; loopi < nx; loopi++)        {

                tmp = rp[ri+loopi] - Ap[Ai+loopi] * xp[xi+loopi];

#pragma distribute point

                    rp[ri] = tmp;

                    Ai++; xi++;ri++;
```

```
        } //for loopi

Ai += sy1 - nx; xi += sy2 - nx; ri += sy3 - nx;

        } // for loopj

    Ai += sz1 - ny; xi += sz2 - ny; ri += sz3 - ny;  } //for loopk
```

The opposite of the loop distribution technique is loop fusion. A developer may use loop fusion for similar reasons as in loop distribution, such as improving F/M balancing, maximizing instruction level parallelism and better utilization of the CPU's functional units. Both techniques have the same advantage – reducing **relative pressure** on a single processor's functional unit.

Loop unrolling is a type of loop fusion, serving the same purpose of better utilizing the functional units during execution of loop iteration. Loop fusion and unrolling also affect software pipelining parameters: vary the Initiation Interval and the number of stages in the software pipeline (the pipeline depth). Since the loop unrolling technique is in fact repeating the loop kernel's operation times the unrolling factor, this type of loop fusion increases the pressure on the same functional units, and as a result, does not reduce the relative pressure. Therefore, in this paper the loop fusion technique will be discussed from the angle of reducing relative pressure on a single execution unit, or introducing utilization of other units.

Example 3: Consider the loop fusion when a loop kernel is a very small (simple) computation:

```
for (i = 1; i <= j-1; ++i)  {

  X(i) -= temp * AP(k);   ++k; }
```

In this example, there are two load operations (X and AP), one store operation to place the result into X, and one FMA operation. The F/M ratio is 1/3, or if loadpair instructions were generated, the F/M ratio is ½. The depth of software pipelining, or number of stages, may be relatively high compared to the number of operations occurring within one loop iteration (#stages=9). And if the loop has a relatively low trip count, it would be necessary to decrease the depth of software pipelining or the number of stages through loop unrolling or loop fusion. Since unrolling does not improve F/M ratio (F/M=1/3),

```
for (i = 1; i <= j-1; i+=2)  {

     X(i) -= temp * AP(k);

      X(i+1) -= temp * AP(k+1);  k+=2; }
```

pulling into the loop some floating-point operations from outside offsets relative pressure and decreases the number of software pipelined stages. Also, if some integer operations are available, pulling them into the loop can improve utilization of the functional units:

```
for (i = 1; i <= j-1; ++i)  {

        X(i) -= temp * AP(k) + B(k);   ++k; }
```

The technique of “pulling” operations from outside of the loop may automatically occur as a result of outer loop unrolling, changing iteration space of the loops, loop interchange, constant propagation, scalar replacement or inlining.

Example 4: A long latency instruction with pressure only on one functional unit of the processor, for instance, sqrt(). The code example can be as follows: square root of vector elements.

```
for (i = 1; i <= j-1; ++i)  {

        X(i)-= sqrt(A[k[i]]);  }
```

In this case, any coding style will express the pressure on the floating-point functional unit, where all other functional units will be executing empty cycles for 11 cycles while the square root is being computed, as there are no other operations to execute but the floating-point square root computation; memory, integer and other functional units will be idle.

The loop fusion technique is entirely contained within the High-Level Optimizer. Unlike loop distribution or loop unrolling, there is no compiler flag or a directive to indicate which loop statements may be improved by being fused into one loop.

As the loop distribution assists the compiler’s microscheduler to generate code close to a theoretical maximum by subtracting operations from a loop kernel, for the loop kernel in Example 4 above, the opposite technique (loop fusion) will be required to add operations to it.

The relative pressure on a functional unit, or deviation of a loop to be balanced (according to the microscheduler), is partially measured by F/M ratio metrics. The loop distribution and loop fusion are directly manipulating the F/M ratio. In order to quantify the metrics for F/M ratio improvement, it is important to emphasize that relative pressure on a floating-point unit (FPU) versus a memory unit directly affects performance. A high relative pressure on FPU would indicate extremely low parallelism. A high relative pressure on the memory unit potentially could cause poor scaling, run-time data access irregularities, randomly imprecise scheduling, and unpredictable memory performance (see details in the run-time issues section below). Balancing the number of floating-point operations to match memory operations requires the CPU to execute within one cycle without a stall to wait for resources. This has become the necessary practice to synthesize the hardware platform and to get closer to a theoretical maximum performance for loop kernels.

According to the Itanium microarchitecture rules, the F/M ratio ideally should be close to **one byte per FLOP**, where the FPU does not wait on freed resources from memory units, and execution of functional units does not need to be

synchronized to avoid stalls on dependencies (this is explained in more detail in later sections).

To summarize, most common loop-based transformations, which also help increase execution efficiency, include the following:

1. Loop fusion
2. Loop distribution
3. Loop unrolling
4. Loop software pipelining
5. Loop interchange
6. Inlining

The only loop-based transformation we have yet to discuss is the well-known software pipelining. Software pipelining is the main tool used to increase instruction level parallelism and maximize IPC.

The code transformations discussed above may expose to the compiler new opportunities to enable loop software pipelining (which might not be software pipelined before applying those transformations).

The compiler's code generator decides whether it would be beneficial to software pipeline or use a regular global acyclic scheduler. If the compiler decides to pipeline the loop, many heuristics are invoked to determine the most efficient microscheduling of the loop, for example: loop trip count; average load latency (integer or float); which internal algorithms to use for long latency operations (division, square root); whether to use prefetching; if control flow is present, what path is more frequently executed; when to use speculative or advanced loads; etc. If the compiler decides not to software pipeline a loop, it might occur anyway due to an incorrect assumption based on which global acyclic scheduler may be more efficient than software pipelining. For example, if there is a complex control flow within a loop, or a loop has a small trip count, or a loop has carried dependencies.

- *Loop carried dependency.* This is the one of most common reasons why the compiler would not software pipeline a loop. First, let's focus on how a developer can validate the compiler's decision on loop-carried dependencies. A good rule of thumb: *If you can execute the loop backward, forward, or in any order without any impact on the result, it means your loop does not carry load-store dependencies, and can be software pipelined.* Earlier versions of Intel® compilers would detect an inter-iteration loop-carried dependency in a simple data reduction function such as *S=S+A[I]*. However, in fact, in the explicitly commutative operation loop *S=S+A[I],* there is no carried dependency among iterations, as the addition can be performed in any order without any impact on the final result, regardless of the fact that some false dependencies can be observed on the surface.

  A coding technique called *introduction of a temporary array* (see Example 2 above) may help here:

```
for (loopk = 0; loopk < nz; loopk++) {

  for (loopj = 0; loopj < ny; loopj++) {

double tmp[nx];

    for (loopi = 0; loopi < nx; loopi++)       {

                tmp[loopi] = rp[ri+loopi] - Ap[Ai+loopi] * xp[xi+loopi];

    }

    for (loopi = 0; loopi < nx; loopi++)       {

                rp[ri+loopi] = tmp[loopi];     }

       Ai += sy1; xi += sy2; ri += sy3;

       } // for loopj

    Ai += sz1 - ny; xi += sz2 - ny; ri += sz3 - ny;  } //for loopk
```

In this manual code transformation, the *tmp* array usage is shown explicitly, unlike in Example 2 above where the compiler does the same transformation using the *distribute point* directive.

If the loop does have some carried dependencies or the compiler assumes it does (as in the example above), a developer could apply some coding construction which may assist the compiler to resolve those dependencies. The simplest way to resolve dependencies is to be able to store intermediate values in temporary storage between iterations, which exposes the dependencies. Thus, explicit introduction of a temporary array can be used to assist the compiler to software pipeline, because:

- It resolves carried dependencies, providing a temporary storage place for the compiler to hold the dependent values between loop iterations.
- It can be used as a temporary storage location when distributing the loop.
- It can assist the compiler to artificially balance memory operations vs. floating-point and integer operations, and improve F/M ratio. What may not be obvious is that in most cases, introducing additional storage space and memory operations can be more efficient than not having a balanced loop and a relatively high pressure on one of the functional units.

Example:

Variable = F(Variable, loop index)

will be transformed to:

Array Element = F(Array Element, loop index)

There are some cases when software pipelining (SWP) can decrease performance. Most of those cases relate to the overhead of rolling software pipelining in and out versus a relatively small loop length (low trip count), for example:

- When a loop has an implicitly low trip count.
- Or when a loop has a relatively low trip count in comparison to SWP depth (number of stages) or the Initiation Interval (II).
- Or when loops have very long latency kernels, which exercise only one functional unit (sqrt, sin, cos, etc.). These loops have a very high initiation interval. One could consider loop unrolling as an alternative to software pipelining.
- Or when loops have relatively simple kernel, when the II is of the same magnitude as the loop trip count.
- Or when a loop has a control flow that causes many of the loop kernel instructions not to be executed. In other words, the rule of thumb is: profile-guided optimization (PGO) data assists an SWP engine to software pipeline a loop considerably better than plain SWP without any knowledge of a data set.

Sometimes, when a loop has a control flow, the Global Acyclic Scheduler may not correctly estimate opportunities for optimization and may decide that software pipelining would not benefit performance. Use the PGO technique as your guide to what the compiler should in fact decide based on knowledge obtained from the run-time analysis. If the compiler software pipelines the loop after PGO, but did not when based on the static scheduler, then consider enforcing software pipelining with the directive *#pragma swp*. Use this technique if you notice that after PGO the Scheduled II is better than without PGO information. This indicates that the run-time data does provide data to the compiler for invoking heuristics, which the user may want to trigger during the static scheduling via code transformations.

These various code transformations, arrived at by measurement, guide the compiler to make better decisions about software pipelining potential, thereby creating kernels that are optimally software pipelined with maximum instruction level parallelism.

It is important to note that the loop unrolling technique and software pipelining are two techniques that complement each other. The loop distribution technique can help to software pipeline as it enables removal of loop carried dependencies, and the loop fusion technique may assist the compiler's code generator and microscheduler to create more balance, relative to the F/M ratio of software pipelined kernels.

Summary of performance analysis practices

In order to support the automatic code transformations performed by the HLO and to guide the compiler's ability to generate optimally scheduled code with maximum instruction level parallelism, a developer needs to consider the following:

- Analyze the loop trip count for high-performance loops.
- Analyze whether high-performance loops might have justified a loop-carried dependency, according to the algorithm.
- Analyze the shape of the iteration scheme for nested high-performance loops, i.e. iteration space.
- Analyze whether your code uses denormal values. If there is a high probability that the value will be a very small number, it is recommended to perform computations with scaled-up numbers.
- Minimize or avoid divisions. Replace divisions by multiplications, reciprocals, pre-compute divisions, look-up-tables, etc.
- Minimize or avoid integer multiplication of 32-bit or 64-bit numbers. For the Itanium® processor family, the integer multiplication of such numbers requires translation to floating point, computation using the FPU, and then translation back to an integer.
- Use single precision floating-point computation when you can. This is beneficial because it uses shorter computations to compute all the precision bits, and all the math library calls get inlined.
- Avoid generalized handling of non-similar cases within kernels of high-performance loops.
- Avoid complex "if" statements in the loop, as they hinder the ability to software pipeline or schedule efficiently.
- Avoid loop bounds being expressed as values at a memory reference, or any value expressions as pointer references which are not variants within loop iterations.
- Avoid routine calls within the loops.
- Lower the loop dimensions, or **consider modifying the iteration space** to propagate computations into the inner loop from the outside when the inner loop:
    - Has a low trip count.
    - Has a small amount of work to execute.

- Is unbalanced based on the F/M ratio.
- Is hard to software pipeline.

- Analyze and be aware of **data alignment:**
  - 16-byte alignment is required for the load pair instruction generation
  - Misaligned load/store ratio to the data size may cause an exception
  - Do not assume alignment when explicitly declaring a local variable.
  - Incorrect array alignment can cause L2 bank conflicts while accessing the data on some versions of Itanium architecture.

Evaluating the Compiler's Code Generation

Before moving on from the discussion of the compile-time analysis to the run-time analysis, let us formulate a list of simple rules which can help us determine and evaluate how well the compiler did its work. There are built-in compiler tools (see the Appendix section "Optimization reports") that give developers ways to evaluate the compiler's output. Later, the discussion of run-time issues emphasizes that it takes a conscious human decision to compromise between potential compile-time theoretical maximum and resolving run-time irregularities. However, alternatives need to be available to enable a developer to both create solutions and allow research on better performance abilities:

- One solution is to generate the code absolutely close to the theoretical maximum based on rules of the microarchitecture, optimal microscheduler decisions, and good algorithms.

- Another solution is to guide the final usage of optimization techniques by actual run-time performance based on solving data access irregularities on the code level (or by the tools built into the compiler).

In some cases it is important for the developer to sit with a pen and paper and calculate the theoretical minimum II for a given loop. The compiler decision on the minimum II is quite complex, but one can follow the rules of thumb presented below for the Itanium architecture to evaluate how code should look and what to expect the compiler to generate:

- Use microarchitecture and dispersion rules:

    - Two bundles with a total of 6 instructions.

    - Some instruction combinations (templates) cannot be permitted, such as MMM (three memory instructions per bundle).

    - 11 functional units: 4 memory/ALU (4 FP loads – 2 integer load and 2 store), 2 ALU units, 2 FMACs and 3 branch units.

    - Latencies: 1 cycle for most of the INT ops; 0 cycles to compare branch and dependent memory; 4 cycles for FMAC; 1 cycle per integer load if hit L1D; 6 cycles per FP load if hit L2D, or 13 cycles if hit L3; > 16 cycles if hit main memory.

    - In a SWP loop the average number of cycles for the FP load is assumed and set to a hard-coded value, reconfigurable with a compiler switch.

    - Some long latency instructions, such as division and square root, can be automatically replaced in a SWP loop with a different algorithm which

provides a better throughput; however, intermediate results may be inhibited for a number of cycles.

- Per one cycle, or within 6 consecutive instructions (2 bundles without a stop bit in the middle), the following combinations are permitted:
    - Two loads and two stores (4 memory operations)
    - Six integer operations (ALU)
    - Two FMAs (4 floating-point operations)
    - Maximum of two loads or two stores per bundle
    - Two FP operations per bundle
    - Three branches per bundle
    - Six ALU+MEM operations.

  Or in other words:

  Minimum II = max (ceil (#loads/2), ceil(#stores/2), ceil (#memops/4), ceil (#FPops/2), ceil(#INTops/6),ceil(#ALU+MEM/6), ceil(#branches/3), ceil(#instructions/6))

Looking at the loop kernel of a code it is fairly easy to spot integer operations, floating point operations or branches. However, understanding the memory operations may be more of a challenge. The compiler may perform some code transformations to reduce memory operations, and explicitly specified memory access may not end up being translated to actual load or store. All of the code transformations discussed above may lead to inhibition of some load and store, which is the ultimate goal of minimizing II.

Note that using the unrolling technique may increase the minimum II as the number of operations per iteration may increase, less whatever operations overlap between iterations. The minimum II also changes with other optimization techniques, as usage of loadpairs and prefetching will impact the number of operations required to execute the loop iteration.

The ability to reduce the complexity of memory reference computation may reveal to the compiler new opportunities to reduce memory operations when generating code. The Itanium microarchitecture introduced the capability to reduce memory instructions when a stream of values needs to be loaded with a feature called *loadpair*, when 2 double-precision FP values can be loaded with one instruction.

- Evaluate F/M ratio for the high-performance loop kernel:

- For number-crunching code, use the rule that the number of floating-point operations per cycle has to be greater than the number of memory operations per cycle.

- Loadpair instruction generation may provide a better F/M ratio.

Run-time issues and data access irregularities

Discussing the compile-time alternatives to achieving optimal scheduling and influencing functional units to perform to their maximum, we have to note that certain compromises are necessary to avoid run-time performance problems, which can overshadow those performance improvements. The in-depth performance analysis and re-iterative compile-time testing based on run-time execution should be the focus of moving an application to its theoretical maximum performance.

Regardless of perfect and optimal scheduling, the run-time issues may imply undoing some optimization techniques. The main focus of this section is to understand what the maximum performance possible of an analyzed application is, and what prevents us from reaching it.

Most run-time issues are related to data or instruction access irregularities, such as DTLB and ITLB (Instruction Translation Look-aside Buffer) misses, cache misses (primary and secondary) for all levels of cache, L2 bank conflicts, L2 OzQ cancels, over-subscription, and so forth.

As an example to support this point, let’s consider the possibility of the compiler generating four floating-point loads per one cycle, which is supported by the Itanium 2 microarchitecture. This type of code generation could be dangerous at places where bank conflicts would start to appear because of these more intensive loads. One can evaluate what is more beneficial for an HPC loop – avoid bank conflicts and generate at most three loads per cycle, or stress the memory unit and not worry about the overhead related to bank conflicts.

Data Event Address Register (EAR) events and long latency loads
In order to follow the data-access irregularities, Itanium processors provide built-in hardware events to profile data latencies. These hardware counters are intrinsically exact and almost the only hardware counters which could map exactly to the place where an event has occurred. Using data EAR events, the developer can monitor all the data accesses and see their total distribution based on latencies and how long it took to get a particular piece of data.

Data EAR events are precise, are triggered by load instruction, and record actual delivery latency associated with Instruction Pointers initiated by the load (IP recorded by hardware). All Intel performance tools support Data EAR monitoring abilities – VTune™

Performance Analyzer, EMON, etc. For example, in the VTune Performance Analyzer, the “Dear_latency_gt_64” event indicates a data delivery that took longer than 64 cycles.

Data EARs allow unambiguous localization of long latency data accesses. Those events can easily point to irregularities that occurred within loads. For example, data expected to be delivered from L2 cache, which takes 16 cycles to complete, can be considered an irregularity; or data which takes 200 cycles to be delivered also can point to some strange memory activity. The developer and compiler may know about average data delivery latencies and approximate distributions: 1-4 cycles for L1 cache, 4-8 cycles for L2 cache, 8-16 for L3 cache access, and over 16 cycles is most likely to be originated by the memory bus. Bank conflicts and OzQ cancels can definitely cause L2 cache access to increase to more than 8-cycle latency. The Intel compiler microscheduler has a built-in average memory latency parameter, which takes into account possible data access irregularities when it schedules the code. For L2 data, it is set to 11-cycle latency. This average latency parameter can be modified by a compiler switch, and can be a great tool to research how alternating the memory latency parameter can influence the microscheduler to create denser code for a computation-intensive loop.

There are many ways to investigate and reduce long latency loads – such as avoiding L2 bank conflicts, high-level optimizations, manual prefetching to ensure that data will be in the cache, and many more.

L2 bank conflicts
Because L2 bank conflicts are well-discussed in existing literature, we will mention it here just briefly. The L2 cache has a 256-byte cyclical structure and is constructed of 16 banks, each of which is 16 bytes wide. The L2 cache is 8-way associative and has 128-byte access cache lines.

Bank conflicts could occur when:

- Two load/stores attempt to access the same bank on a single cycle
- Accessing adjacent data within a cache line
- Accessing data which is 256-byte aligned

To avoid bank conflicts, the developer can adjust memory access patterns and memory data structure layout accordingly, while ordering allocation and data.

Future Itanium architectures may modify L2 cache design to eliminate this problem, but currently, the Intel compiler can automatically avoid bank conflicts.

L2 secondary misses
Another data irregularity issue is an L2 secondary miss. This event can occur when an L2 primary miss was identified (data was not in L2), and after bringing the data to the cache and re-accessing it again, the L2 miss occurs again. The secondary miss can occur when the data is being accessed but not located in the cache, the cache line has been transferred but not completed, and at the same time there was an attempt to access data within the

same cache line. An L2 secondary miss can also occur when prefetching is not working correctly. For instance, if the prefetch distance was not calculated correctly, when prefetch is issued, the load to be accessed is not in the cache because it was accessed too soon.

To reduce L2 secondary misses, the developer and compiler should compute prefetch distances correctly and provide correct alignment for the data.

L2 OzQ cancels and OzQ over-subscription

The L2 bank conflicts subject is a well-known and frequently occurring data access irregularity event. However, there are other major architecture-dependent data access irregularity events, called OzQ cancels and OzQ over-subscription, related to OzQ design. The OzQ is the 32-entry queue in Itanium architecture that controls L2 data accesses and provides out-of-order data return and data load to the FP register file directly from L2. There are about 25 events related to OzQ and its overflow.

Minimum latency for L2 data access is 5 cycles for integer data, and 6 cycles for floating data. However, this latency increases if we have an OzQ cancel or over-subscription event, or a recirculate event. The overhead of these events can be anywhere from 6-18 cycles. The OzQ is a cyclic queue, and when it fills up or the tail hits the head – a data access irregularity event occurs. OzQ entries can be removed when data leaves the cache; however, the hardware does not collapse the queue, and with empty entries in it, the OzQ tail hits the head more frequently.

Therefore, when within the memory intensive loop there is a high L2 cache activity, L2 OzQ over-subscription and recirculate events are likely to occur. The solution is to reduce the activity, i.e. the L2 cache misses. In order to do so, one can use prefetching as the main tool to reduce frequent recirculation. A variety of techniques (such as loop balancing and loop distribution) can also be used to lower the pressure on the memory unit and reduce OzQ over-subscription and OzQ cancel events.

In this section it is appropriate to mention that loop balancing affects performance in an unexpected way. Consider the following example (pseudo-code):

Loop 1:

For I do

$A[i] \mathrel{+}= B[i] \times C[i] - D[i] \times E[i]$

Loop 2:

For I do

$V[i] = SQRT(A[i])$

Both loops are software pipelined and unbalanced. The first loop clearly has 2 FMAs, 5 floating-point memory loads, and 1 store, and most likely one prefetch instruction generated by HLO. The second loop has 1 load, 1 store and multiple FMAs to converge to the right precision of square root. The first loop is memory intensive, and the second loop is floating-point intensive. The OzQ events will be an issue in the first loop, but it will not affect the second loop because of a long-latency floating-point instruction in it. Assuming the compiler generated a correct prefetching for the first loop, we will still encounter L2 OzQ over-subscription and recirculation events.

The loop distribution technique in this case may balance the loops, and may also be a very strong tool to reduce OzQ events. It seems the following loop transformation should not affect anything in terms of performance, as operations themselves and their quantity stay the same following redistribution:

Loop 1:

For I do

$A[i] \mathrel{+}= B[i] \times C[i]$

Loop 2:

For I do

$V[i] = SQRT(A[i] - D[i] \times E[i])$

However, the performance is better in the second structure because it is easier for the compiler to schedule operations to get closer to a theoretical maximum in Loop 1, and balance the loop through redistributing the memory intensive parts amongst Loop 1 and Loop 2. Loop 1 is no longer memory intensive, and the loop kernel can be theoretically scheduled in one cycle. Also, the OzQ over-subscription will not be an issue in either the first nor second loop and will not generate any additional run-time stalls. The Loop 2 performance will not be affected, as it remains unbalanced with added memory operations being compensated by a long latency floating-point instruction (sqrt).

This example demonstrates why loop balancing and achieving theoretical maximum are beneficial for the performance improvements, even though it does not change the number of operations we have to perform (as the run-time sets the “final tone” of actual performance).

By analyzing the run-time performance we can list the issues that occurred during the run that depend on the data flow, data access patterns, dynamically computed data references, data layout, actual execution paths, dynamically computed branches, and so forth. The result of this analysis could be fed back to the compiler to guide the compiler’s heuristics for more precise decisions about code generation.

First, the need for data prefetching can be easily identified. Based on the discussions above related to OzQ and cache misses, it is clear that correct data prefetching can help generate code that performs better.

There are many ways to compute a future reference address to prefetch. The simplest way is to let the compiler guess, and during compile-time, generate an immediate value for the prefetch distance, when this computation is quite simple and straightforward. The developer, however, must make sure the correct pointers have been prefetched, and the prefetch distance generated by the compiler looks correct. In more complex examples, when the compiler cannot compute the reference to determine what to prefetch, the developer could use profile-guided optimization, or the compiler could spawn a thread to compute prefetch data references. In this paper, we will discuss the simplest method of guiding and controlling the automatic prefetching mechanism built into the compiler's HLO.

Here are some things to note: prefetch is not a load operation and it does defer the exceptions to a later stage when the load operation is actually performed. The prefetching operation is the subset of load operation components. If one tries to prefetch a pointer that does not exist, calling lfetch explicitly avoids causing an exception. Please see the Appendix for load operation hierarchy and where the lfetch operation stands in the list. The prefetch operation is not without cost – it does take cycles.

For the Itanium architecture, floating point data never resides in L1 cache – it can only be put in L2 or L3 caches. A wrongly computed prefetch can cause cache thrashing and massive OzQ cancel and recirculation events.

In software pipelined loops you can only see one prefetch operation, which may relate to number of arrays within a single instruction, since the data references can be stored in a rotating register and switches to a different array during each iteration.

It would be very beneficial if the compiler could enhance prefetch-related directives to allow users to explicitly specify and define memory layout, data access patterns, prefetch initialization points and assist in computing prefetch distances, but these functions are not yet available.

There are issues to be aware of when prefetching is generated by the compiler. In some cases it may cause high-level optimization (–O3) to generate code that performs worse than the default –O2 optimizations. Also be aware that excessive prefetching in the wrong places may thrash the cache or cache registers, when cache utilization is already being tuned by a developer. Another issue with prefetching is that an extra memory slot may be needed to accommodate an additional lfetch instruction in already densely generated code. Adding a memory slot may also require adding an extra cycle to a loop kernel or increasing II in SWP loops.

Scaling

Scaling is another environment change which can move us farther away from reaching theoretical maximum during run-time performance on perfectly balanced loops. Let's discuss how to work around those environment scaling.

Scaling with multiple processors on an SMP system
With an increasing number of CPUs on an SMP system, a memory intensive loop generates a substantial number of cache misses and as a result, initiates memory bus cycles. The perception on Itanium processors is that when we scale from 2 CPUs to 4 CPUs, most memory-intensive computational loops may saturate the memory bus.

To understand the program bandwidth limitations, first we have to specify that maximum front-side bus bandwidth for Itanium 2 processor-based platform is 6.4 GB/sec, distributed over half of each load and store.

The program bandwidth required (per processor) by a high-performance loop is:

> *Bus_Bandwidth = Cache lines per iteration × 128 bytes × frequency/cycles per iteration, where lines per iteration must include read and 2x write output lines*

Therefore, setting program bandwidth to < 6.4 GB/sec would determine minimum cycles per iteration for one processor, which is:

> *cycles per iteration = lines_per_iteration × 128 × frequency x Num CPUs / 6.4*

When scaling, our goal is to provide a balanced computational loop, which is also optimally scheduled, a resource for computation without stalls. As we already specified in the microarchitecture rules, 4 FLOPs can be originated by one cycle, and also 4 loads/stores can be done in one cycle (regardless of load pairs, which should load more data with the same amount of load instructions). The ideal F/M ratio per balanced loop is 1 byte per FLOP – this is required by some users. Having an SMP system should not interfere with this requirement, which implies that on an SMP system, when the loop is balanced none of the CPUs are stalled or waiting for resources.

To provide 1 byte per FLOP while scaling, the developer has to ensure that data on all of the processors resides in L3 cache, or can be brought to the cache by prefetching in advance. It implies that accumulation of all the cache misses on all the CPUs cannot originate as a memory bus saturation issue. As the formula above illustrates, the scaling metric of byte per FLOP is dependent on the frequency and the cache size. It is clear that the cache size can compensate for memory bus saturation, since we can decrease the misses. However, increasing the frequency or number of CPUs worsens the problem once again, as the cache is required to supply the data much faster, and then requires an even larger L3 cache.

To summarize, with a fixed frequency and number of processors, we can specify the cache size at which a balanced loop with an F/M ratio of 1 would not stall, or have 1 byte per FLOP to compute.

Scaling with the frequency

When scaling with the frequency, the best expectation is to get linear scaling in performance. This is a rather rare phenomenon. Most likely the performance does not scale linearly with frequency, and we would like to model the “non-linear” part of the scaling which is represented by stall events that occur due to memory accesses. Based on the previous scaling-related deductions, we will briefly list the issues related to scaling with the frequency:

- Faster clock or faster ALU computations may cause bottlenecks in memory accesses
- Memory performance (or better chipset) compensates frequency scaling
- Cache size very often compensates memory performance while scaling
- Some algorithms cannot easily take advantage of cache compensation, such as sparse matrices computations, as data hardly resides in cache regardless of its size.

Frequency scaling can be split into two components: frequency scale for computation and stalls, which represents a linear dependency, and frequency scale with given memory latency, where scaling represents the function: *F(data EAR distribution vector)*. The data EAR distribution vector is the discrete function of memory latencies distribution defined by DEAR_latency_xxx events.

The entire data EAR distribution vector represents all the stall events that occurred during memory accesses. We can consider that, in a way, the data EAR distribution vector represents a function of cache size, memory access pattern, memory latency and cache latency.

While scaling with frequency, certain things will remain variant, such as stalls and memory access patterns. However, the following aspects could be constant: the compiler microscheduler’s decision to schedule the code differently, or use a different cache size.

One could use the compiler’s microscheduler parameters to generate code targeting various CPU frequencies. For example, by increasing or decreasing the average load latency parameter (virtually modifying actual data EAR event distribution for the compiler), one could linearly determine performance while frequency scales up and down, also requiring setting smaller or larger cache sizes.

Summary

Throughout this paper, we have discussed three major trends and methodologies for performance characterization of high-performance codes:

- Closed-loop methodology compile-time analysis, first applying basic coding practices to reveal opportunities for optimization to the compiler; then influencing automatic code transformations based on hardware synthesis and resource balancing; then detecting run-time data access irregularities; and finally feeding back to the compiler the run-time data adding human guidance for the compiler's HLO, such as PGO or prefetching control.

- New enhancements that enable compilers to generate dense, balanced code, according to the best possible schedule derived from microarchitecture rules. Manual code transformations are not permitted; however, a developer can specify to the compiler certain guidance via directives or compiler switches to manipulate or trigger invocation of the specific compiler's HLO heuristics.

- Discussed concepts, terminology, practices and specific technical information to assist the developer to get the most out of the Itanium processor-based platform.

The following key takeaways are stressed:

- Use visual inspection of assembly to your advantage. It is quite easy to determine the theoretical maximum performance for a hotspot and assess the actual compiler's work.

- It is actually possible to use the compiler's flags and directives to get closer to the theoretical maximum. In the past 2 years, the Intel Compilers for Itanium processors has made a tremendous leap in the ability to generate efficient and close-to-optimal code out-of the-box, without manual code changes.

- The compiler's HLO and SWP are main tools to maximize IPC; however, "good" IPC is a necessary condition for better efficiency but not sufficient alone. The five efficiency levels for Itanium processors are: algorithm selection, algorithm mapping, optimization of microscheduling, balancing the F/M ratio, and run-time analysis.

Appendix

Software pipelining

Software pipelining is the main method available to increase instruction level parallelism and maximize IPC. Software pipelining is not a feature of the Itanium microarchitecture, but rather a software feature. However, it is supported by certain microarchitectural features, such as rotating registers, which assist a software pipelined loop to have packed representation. Software pipelining is a method of loop implementation which allows overlapping several loop iterations within one CPU cycle. A number of cycles between two consecutive iterations of the loop is called the initiation interval. Note that software pipelining is not considered as part of the high-level optimizer; it belongs to –O2 default optimizations.

Using one of the software-related compiler directives below can guide the compiler to generate better code, and sometimes even enable pipelining of high-performance loops:

#pragma ivdep,

ignores vector dependencies for the scope of the next loop

#pragma swp,

software pipelines for the scope of the next loop

#pragma noswp,

does not software pipeline for the scope of the next loop

#pragma loop count (<tripcount>) ,

indicates a magnitude of possible loop trip counts.

Here is the list of useful software pipelining-related compiler flags:

-*ivdep_parallel* to resolve loop-carried dependencies by ignoring vector dependencies

-*fno-alias* to disambiguate pointers throughout the module (for C programmers)

-*ansi_alias* to assert that the program adheres to the language-specified type aliasing rules (such as float, int, double, short; all independent based on a type; asserts out of bounds access arrays; on by default in Fortran)

-*alias_args*- to assert that C function arguments passed by reference do not alias (on by default in Fortran)

or, in Fortran, programmers who are using Cray pointers: use -*safe_cray_ptr* flag.

The Intel compiler has a built-in technique to resolve pointer ambiguity by generating advanced load instructions with *ld.a*. Generating advanced loads within a software pipelined loop kernel may negatively impact performance.

There are also compiler switches to control code generation's software pipelining methods, as well as switches to disambiguate separate classes of the pointer based on their type, location, scope and declarations (see the Intel compiler reference guide that comes with the compiler).

Loop Interchange

Loop Interchange is *a code transformation that changes the nesting order in nested loops.*

Loop interchange is used for:

- Improving spatial locality, such as providing sequential and blocked data access for the innermost loop. Such situations can arise from codes ported from Fortran with a column-major storage for arrays, to C, which use a row-major storage scheme.

- Improving the effects of inner loop optimization by moving higher trip-count loops inside lower trip-count loops.

- Reducing memory operations or computations via data reuse. The results of memory operations or computations can be stored in registers when nested loop order changes.

- Hoisting invariants relative to the innermost loop index variable. When loops are interchanged, data that was variable relative to the inner loop index can become a constant, or may need to be calculated only prior to entering the inner loop.

Loop Skewing

Loop skewing is a loop transformation *that reshapes an iteration space to make it possible to express the existing parallelism with conventional parallel loops*.

At times, this reshaping also lowers the dimension of the nested loops. This technique is complementary to loop interchange, but can also be used when loop interchange is not possible. The lowering of loop dimension also enables the compiler to invoke heuristics for prefetching and for tracking spatial accesses.

An interesting case of reshaping the iteration scheme is code transformation called *loop peeling*, occurring when we pull out the first or last iterations of the loop in order to extract possible singularities or special case handling mechanisms out from the loop body.

Loop Unrolling

Loop unrolling is a well-known technique used to improve performance. Loop unrolling is also a code transformation *that reshapes an iteration scheme to improve possible instruction level parallelism and allow execution of multiple loop iterations at the same time*.

Loop unrolling to a factor of *n* consists of unrolling the loop body *n-1* times to create *n* copies of the loop body and then fusing those copies together to achieve optimal execution. Manual code transformations targeting unrolling are typically not necessary with modern compilers.

The benefits of loop unrolling include:

- More effective usage of functional units, as more data and operations are manipulated per iteration, allowing the compiler's instruction scheduler maximal opportunity to "fit" the instructions into execution slots.
- Allowing extensive data reuse to reduce or eliminate memory accesses.
- In conjunction with software pipelining, loop unrolling is the major technique used to hide the latencies of long-latency operations.

Loop unrolling, as with any other optimization technique, can affect performance negatively. Limitations of loop unrolling include:

- Low trip count (or small loop length) can introduce excessive overhead. Loop unrolling introduces overhead at the beginning and at the end of the loop. This overhead needs to be amortized over a large number of loop trips. If the number of loop trips is small, the payoff does not justify the extra time needed to transform the code.
- The number of available functional units can limit loop unrolling.
- Unrolling may cause more cache usage per iteration.
- The Itanium architecture's 96 general-purpose rotating registers may become a limitation for the unrolling factor:

  Total number of registers used = Number load/store references (stored in GR) × Unrolling factor × Software Pipelined stages

There are tools and interfaces to manipulate the compiler's heuristics and parameters when it is performing automatic loop unrolling. See the compiler user guide for following directives which can be placed in the source code before a loop:

#pragma unroll

#pragma nounroll

#pragma unroll(<factor>)

#pragma loop count(<loop trip count>)

The last directive represents a hint to the compiler on loop trip count and allows the compiler to invoke appropriate loop unrolling heuristics.

Part of the optimization report (a tool embedded within the compiler to monitor code transformations and automatic optimizations performed) relates to the loop unrolling. When the developer wants to monitor loop unrolling, the compiler's decisions and heuristics, it can be done using the compiler switch *–opt_report_phasehlo_unroll*.

Example: smg2k.c

```
for (loopk = 0; loopk < nz; loopk++) {
  for (loopj = 0; loopj < ny; loopj++) {
      for (loopi = 0; loopi < nx; loopi++)      {
            rp[ri] -= Ap[Ai] * xp[xi];
            Ai+=sx1; xi+=sx2;ri+=sx3;
          } //for loopi
      Ai += sy1 – nx*sx1; xi += sy2 – nx*sx2; ri += sy3 – nx*sx3;
          } // for loopj
      Ai += sz1 – ny*sy1; xi += sz2 – ny*sy2; ri += sz3 – ny*sy3;  } //for loopk
```

This code can be transformed into a loop that performs about three times better with denser, close-to-optimal scheduling if it is rewritten using coding practices described in this paper and also overwrites some compiler heuristics (based on algorithm knowledge):

```
for (loopk = 0; loopk < nz; loopk++) {
  for (loopj = 0; loopj < ny; loopj++) {
#pragma ivdep
#pragma unroll(2)
      for (loopi = 0; loopi < nx; loopi++)      {
```

```
            rp[ri] -= Ap[Ai] * xp[xi];

             Ai++; xi++;ri++;

        } //for loopi

Ai += sy1 – nx*sx1; xi += sy2 – nx*sx2; ri += sy3 – nx*sx3;

        } // for loopj

Ai += sz1 – ny*sy1; xi += sz2 – ny*sy2; ri += sz3 – ny*sy3;  } //for loopk
```

Strength reduction

Strength reduction is a code transformation *that replaces long latency operations by an equivalent, most likely inductive, series of short latency operations.*

This technique is an essential part of the HLO. However, manual strength reduction may be necessary to simplify complex expressions.

Strength reduction can be used to replace computations directly dependent on a loop index variable with inductive and recurrent computations. Most common cases known are replacing complex index calculations, such as 2-D array indices based on a multiplication operation, to an index calculation, which only contains additions.

Cache blocking

Once loop interchange has selected the best loop ordering for cache locality, performance can be further improved by blocking (also called tiling) some of the loops. Cache blocking is a complex code transformation technique that ensures effective cache utilization by grouping computations whose data references are in non-overlapping areas of cache. Currently some compilers (such as Intel Compilers) are L2 cache size aware, but not L3 size aware. (Intel's compiler engineering team is looking into ways address this in order to provide better code transformation for L3 cache spatial locality.)

Dense linear algebra codes benefit from cache blocking. However, the use of this technique is complicated by limited cache control. The cache size needs to be parameterized; otherwise, the code is likely to run poorly on a CPU with a different cache size.

Cache blocking can be fruitfully combined with other transformations. Typically, blocking with skewing, blocking with fusion and alignment, or blocking with any other code transformation which allows better register usage will result in performance improvements.

Loop distribution

Loop distribution is a code transformation *that extracts statements from a loop and distributes them into separate loops, usually with identical loop bounds.* Manually, the developer can place a hint to the compiler within the code, indicating where the loop distribution ought to occur. The following directive specifies that place in the code:

#pragma distribution point

The major reasons for using loop distribution on Itanium processors (especially when the compiler decides to software pipeline the loop) are to:

- Remove the limitation on rotating register usage when the rotating registers become the limitation for software pipelining of larger loops.
- Enable more effective usage of the CPU resources.
- Impact the compiler scheduler to reduce empty cycles in a software pipelined loop.

It is often necessary to introduce temporary arrays to store the data when splitting loops. These arrays are used to transport the data from one loop to another when performing a loop distribution transformation.

Note that the third bullet above, "reduce empty cycles in software pipelined loop," means that according to the best knowledge of the compiler's microscheduler, one of the execution units does not have enough processing data and the loop is not balanced. Intuitively, it is clear that having dense code without empty cycles, when all the execution units are busy doing useful work, is better for the performance. Even in the latest versions of the compilers for the EPIC architecture, large software-pipelined loops have a high probability of being generated with empty cycles. This issue can be attributed to the exponentially growing complexity of scheduling in cases of a large loop body.

Loop Fusion

Loop fusion is exactly the opposite transformation from loop distribution. Loop fusion is a code transformation *that consolidates statements of several loops into a single loop*. The advantages of loop fusion, however, are similar to those of loop distribution:

- Reduces loop overhead, reducing the number of memory operations and providing better data reuse; enables more effective usage of CPU resources, such as register reuse.
- Removes dead code and stores data in registers instead of memory.
- Improves microscheduling and maximizes instruction-level parallelism.
- Improves SWP microscheduling.

- Allows manipulation or guidance of the depth of software pipeline kernels (number of stages).

Control of FP schedule

-IPF_relax_fp relaxes restrictions on FP precision and invokes faster algorithms, which may compromise precision and not qualify for IEEE standards (the last mantissa digit may be off).

-IPF_fma- generates a slower code without using FMA instructions, as precision and accurate FP computations may be extremely important. This does not allow combining floating-point add with multiply in one instruction. Default is FMA enabled.

-complex_limit_range enables use of basic algebraic expansions of some complex arithmetic operations. This can result in some performance improvement in programs that use a lot of complex arithmetic at the cost of some exponent range.

Control alignment

__declspec(align(16)) double A[1024]) declares array A is aligned to a 16-byte boundary.

–stack_temp flag in Fortran allocates temporary variables on the stack.

-ZpN is the compiler flag which aligns data structures to N-byte.

Control of load pair generation

The load pair instruction **ldpfd** is a *single Itanium® architecture-specific instruction, which loads two double-precision floating points into two adjacent FP registers*. Note that the microarchitecture has load-pair generation requirements, such as the loaded data location must align to 16 bytes. Benefits of load-pair generation:

- Increases the efficiency of data accesses without increasing bus traffic.

- It is a strong tool to balance floating-point operations vs. memory operations, improving the F/M ratio, and as a result, providing better relative functional unit pressure in memory-intensive loop kernels.

- It is a strong tool for decreasing the number of primary or secondary misses in L2 cache with increasing numbers of memory accesses, which gives better utilization of the L2 OzQ mechanism.

A developer has tools to monitor the compiler's decision regarding load pair generation via the compiler flag:

–opt_report_phasehlo_loadpair

Control automatic prefetching by HLO

The automatic software prefetching is enabled with the High Level Optimizer (-O3). Use following compiler directives to enable or disable prefetching:

#pragma prefetch ptr_a, ptr_b, ptr_c to specify to the compiler what pointers to prefetch.

#*pragma noprefetch* to disable prefetching for the next region.

The developer can track the compiler's prefetching decisions and what the compiler has decided to prefetch via optimizer reports, such as *–opt_report_phasehlo_prefetch*

One could also use intrinsics for prefetching manually:

In C: _lfetch (hint, pointer)

In Fortran: CALL LFETCH (hint, pointer)

where the hint specifies what level of cache is used to prefetch.

Optimization reports

Inlining reports can be viewed with *–opt_report_phaseipo*.

A developer can trace what the compiler has detected as automatically inlined functions, and what it could not inline because the function was external or inefficient to inline.

Example:

```
IP OPTIMIZATION REPORT:

WHOLE PROGRAM (SAFE) [EITHER METHOD]: FALSE

WHOLE PROGRAM (SEEN) [TABLE METHOD]: FALSE

WHOLE PROGRAM (READ) [LIBRARY READER]: FALSE

INLINING: (_behavior)


INLINING: (_current)

  -> _get_vsupply(EXTERN)

INLINING: (_stimulate)


INLINING: (_ACD_solve)
```

```
  -> INLINE: _ACD_update_nodeterm(13) (isz = 160) (sz = 165 (51+114))
  -> _ACD_update_stage_loop(9) (isz = 261) (sz = 267 (114+153))
  -> INLINE: _ACD_update_nodeloop(14) (isz = 163) (sz = 168 (61+107))
  -> _ACD_init_loops(8) (isz = 61) (sz = 66 (35+31))
  -> INLINE: _ACD_update_tree(15) (isz = 194) (sz = 200 (46+154))
  -> _ACD_update_trapezoid(6) (isz = 262) (sz = 268 (97+171))

INLINING: (_ACD_update_stage_loop)


INLINING: (_ACD_init_loops)


INLINING: (_ACD_update_trapezoid)


INLINING: (_initial)

  -> INLINE: _allocate_stage_data(17) (isz = 188) (sz = 193 (80+113))
    -> allocate(EXTERN)
    -> allocate(EXTERN)
    -> allocate(EXTERN)
    -> allocate(EXTERN)
    -> allocate(EXTERN)
  -> _wf_get_rf_time(EXTERN)
  -> INLINE: _behavior(20) (isz = 138) (sz = 143 (60+83))
  -> _prepare_stage(EXTERN)
  -> _wf_recalculate_dv_dt(EXTERN)
  -> INLINE: _advance_input(21) (isz = 58) (sz = 67 (19+48))
  -> _wf_limit_stepsize(EXTERN)
  -> INLINE: _allocate_stage_data(16) (isz = 188) (sz = 193 (80+113))
    -> allocate(EXTERN)
    -> allocate(EXTERN)
    -> allocate(EXTERN)
    -> allocate(EXTERN)
    -> allocate(EXTERN)
  -> INLINE: _behavior(19) (isz = 138) (sz = 143 (60+83))
  -> _prepare_stage(EXTERN)
  -> INLINE: _behavior(18) (isz = 138) (sz = 143 (60+83))
  -> _prepare_stage(EXTERN)
```

References

Optimizing compilers for modern architectures, Randy Allen and Ken Kennedy; Morgan Kaufmann Publishers: San Francisco, 2002.

Software Optimizations for High Performance Computing: Creating Faster Applications, Isom Crawford and Kevin Wadleigh; Hewlett-Packard Company: New Jersey, 2000.

Theory of scheduling, Richard W. Conway, William L. Maxwell, Louis W. Miller; Addison Wesley Publishing Company: Reading, Mass., 2003.

Cache Blocking and memory bandwidth discussions:

- The Cache Performance and Optimizations of Blocked Algorithms. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, April 1991. S. Lam, E. E. Rothberg and M. E. Wolf.

Intel® Itanium® 2 Processor optimization guides:

- Introduction to microarchitectural optimization for Itanium® 2 processors reference manual

- Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization

ASCI Purple benchmarks: http://www.llnl.gov/asci/purple/benchmarks

Glossary

Byte Per Flop – Performance metric measuring the balanced execution of floating point operations and memory accesses, so specified number of bytes are available for the floating point unit to execute a floating point operation without stalling on the resources.

DTLB – Data translation look-aside buffer

EAR – Event address register

F/M ratio – The ratio between the number of floating point operations and the number of memory operations per loop kernel.

FMA – Floating-point multiply-and-add

FP – Floating point

FPU – Floating point unit

HLO – High-level optimizer

HPC – High-performance computing

ILP – Instruction level parallelism

Initiation Interval (II) – The number of cycles between the start of successive iterations in the loop (if the II is *n* cycles, a new loop iteration will be completed every *n* cycles at steady state). These cycles are also called software pipeline loop kernels.

IPC – Instructions per cycle

IPO – Inter-procedural optimization

ITLB – Instruction translation look-aside buffer

L2 bank conflicts – The L2 cache is constructed of 16 banks, each 16 bytes wide (resulting in a 256-byte cyclical structure). Bank conflicts occur when two loads/stores attempt to access the same bank on a single cycle. Bank conflicts result from accessing adjacent data within a cache line and from accessing data which is 256-byte aligned. Bank conflicts can be a reason for OzQ cancel events.

L2 OzQ cancel – A series of events in Itanium 2 processors indicating that an OzQ queue entry was invalidated. There are about 25 various reasons for the cancel, such as L2 bank conflict or L2 secondary miss.

L2 OzQ over-subscription – An event in Itanium 2 processors indicating frequent overflow of the OzQ queue and unavailability of empty entries. It implies the processor unit will stall until an entry becomes available. A full OzQ causes the L1D micropipeline to shut down.

L2 OzQ recirculation – An event in Itanium 2 processors indicating OzQ queue entry retries to get data. Secondary misses to an L2 cache line are a common cause of OzQ recirculation events.

L2 secondary miss – An Itanium 2 processor event indicating double cache misses in L2, in other words, after missing the data in the L2 cache the first time, accessing the data from the same cache line caused the L2 cache miss event (there are many reasons for this event to occur, such as timing issues of delivering the data or OzQ cancel events).

Loop balancing, or hardware synthesizer – A code transformation which balances the loop kernel for a better F/M ratio for the particular microarchitecture.

Loop kernel – Instructions representing the body of the loop.

Loop skewing – A code transformation which equally changes iteration space of the nested loop.

Minimum II is the smallest initiation interval (II) that is feasible for a pipelined loop (according to the compiler's coded microarchitectural scheduling rules). Minimum II = Scheduled II means the loop is optimally scheduled according to the compiler's microscheduler.

OzQ – The queue built into the microarchitecture of Itanium processors, which tracks L2 cache misses and manages delivery of appropriate cache line data from the memory or L3 cache to L2 cache. This mechanism provides out-of-order data return. On the latest Itanium 2 processors, the queue size is 32 entries. The queue is built as a FIFO queue, without a "queue entries collapse" mechanism (the microarchitecture allows only entries overwrite).

PGO – Profiled-guided optimization

Recurrence II – Caused by loop-carried dependency edges (memory and register dependencies) from instructions in one iteration to instructions in subsequent iterations. If Recurrence II > 0, it means the compiler detected loop-carried dependencies.

Resource II – Indicates the percentage of utilization of a specific processor functional unit within a software pipelined loop kernel. The percent of Resource II used by memory operations, floating point operations, and integer operations shows the utilization of the corresponding execution units throughout the loop kernel. If floating point Resource II utilization is less than memory utilization, the software pipelined scheduler is not optimal for number crunching algorithms.

Scheduled II – The cycles per iteration of the pipelined loop.

SMP – Symmetric multi-processing

SWP – Software pipelining