Fracture Mechanics on the Intel® Itanium™ Architecture (A Case Study)

Gerd Heber*
Cornell Theory Center
638 Rhodes Hall
Ithaca, NY 14853

Andrew J. Dolgert†
517 Clark Hall
Cornell University
Ithaca, NY 14853

Maxim Alt‡
Intel Corporation
SC12–411
3600 Juliette Lane
Santa Clara, CA 95054–1513

Karen A. Mazurkiewicz§
Intel Corporation
CH7–401
5000 W. Chandler Blvd.
Chandler, AZ 85226–3699

Lynd Stringer¶
Intel Corporation
CH7–401
5000 W. Chandler Blvd.
Chandler, AZ 85226–3699

September 25, 2001

Abstract

We optimized our fracture mechanics code to achieve a 3.4-fold speedup on Itanium processors. The computational core of the code is a linear equation solver using preconditioned conjugate-gradient method. The dense and sparse matrix-vector computations are both floating point and memory bandwidth intensive. We found that the EPIC architecture depends heavily on the compiler to find instruction level parallelism to achieve maximum performance.

1 Introduction

The Intel® Itanium™ processor is the first commercially available implementation of Intel's EPIC technology. A natural question is, given a legacy code, what optimization is necessary to achieve reasonable performance on an Itanium processor-based system.

In this paper, we describe our effort to tune a fracture mechanics code, called Crack Propagation on Teraflop Computers (CPTC) [1, 2]. The CPTC software is a joint effort of engineers, computer scientists, and numerical analysts to simulate the growth of arbitrary cracks in 3D solids based on linear elasticity. Figure 1 is a high-level view of the interaction of major components in CPTC. After creation of an initial

*Tel. (607) 255–7885, Fax (607) 254–8888, heber@tc.cornell.edu, *corresponding author*
†Tel. (607) 255–6066, Fax (607) 254–8888, ajd27@cornell.edu
‡Tel. (408) 765–2996, Fax (408) 765–6688, maxim.alt@intel.com
§Tel. (480) 554–4073, Fax (480) 554–7774, karen.a.mazurkiewicz@intel.com
¶Tel. (480) 554–6777, Fax (480) 554–7774, lynd.stringer@intel.com

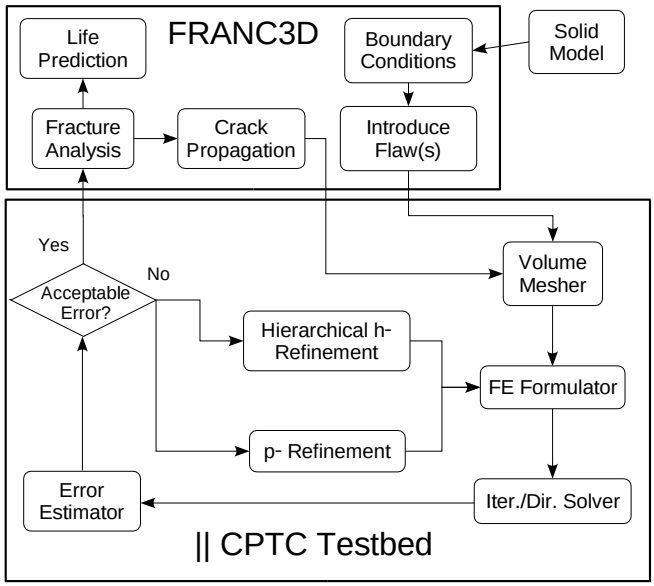

Figure 1: High-level view of the CPTC simulation environment.

model the spatial domain is decomposed through volume meshing into simple elements such as tetrahedra or hexahedra. Thereafter, a discretized version of the underlying (continuous) elasticity equations is derived using the finite element formulator. After solving the equations, a fracture analysis is performed to predict the growth of cracks and make a lifetime prediction. Typically, tens or hundreds of crack growth steps, each involving volume meshing and equation formulation and solving, must be performed for a single lifetime prediction.

CPTC is written in C/C++, and the performance-critical part is the linear equation solver with both dense and sparse matrix-vector computations. While the CPTC software runs on parallel computers with distributed memory using the Message Passing Interface (MPI) [3] API, the focus of our tuning effort was sequential or per-process(or) performance. No changes were made to the MPI code.

The CPTC code is characterized by loops, few branches, and is floating-point and memory bandwidth intensive. The key to maximum optimization was helping the compiler find the maximum instruction level parallelism in the computationally intensive loops. To do this, we changed source code and also compiler flags. To measure progress, we used the assembly language output, high-level optimizer (HLO) report, and software pipelining (SWP) reports. Our optimizations resulted in an 3.4-fold speedup on Itanium processors.

The paper is organized as follows: In Section 2, the computational kernels underlying the solver and a general discussion of the main implementation issues are presented. Section 4 contains some remarks about the tuning methodology. This is followed by Section 5 where we present the key steps of our tuning effort in a condensed form. In Section 6, we summarize the main results and lessons that we learned along the way. Conclusions are drawn in Section 7, Appendix A lists technical specifications of the test configuration, and a glossary in Appendix B lists a few key terms.

2 Computational Kernels—A Priori Considerations

As mentioned in the introduction, the solver is the most time consuming part in a linear fracture analysis. It uses conjugate gradient method (CG) [4] with global extraction element-by-element (EBE) preconditioning [5]. In the next two sections, we describe two main components, the iterative solver and the preconditioner, and how their computational kernels challenge the architecture.

2.1 The Solver

In CPTC, fracture mechanics is modeled in terms of linear elasticity equations. These partial-differential equations are discretized using the Finite Element Method (FEM). The result is a system of linear equations $Ax = b$. The main characteristics of A can be summarized as follows:

- A is an $N \times N$ matrix where N is large, typically on the order of 10^6.
- A is *sparse*, i.e. it has only $\mathbf{O}(N)$ nonzero entries.
- A is symmetric and positive definite.
- A has a distinct *structural pattern*, e.g. most blocks of three consecutive rows have nonzero entries in the same column positions.

The preconditioned conjugate gradient method [4] is a standard iterative solver for symmetric and positive definite problems. An extensive suite of iterative solvers, including CG, is part of the Portable, Extensible Toolkit for Scientific computation (PETSc) [6, 7, 8] which is used in CPTC. The main computational labor per iteration consists of the following:

- One matrix-vector product $y_i := \sum_j a_{ij} x_j$
- The application of the preconditioner (see Section 2.2)
- Two dot products $x \cdot y := \sum_i x_i y_i$
- Three DAXPY type operations $y \leftarrow y + \alpha x$.

The matrix-vector product, dot products, and DAXPYs are very memory bandwidth demanding. There is no temporal locality and the ratio between floating point operations and load/store operations is rather low. This is problematic because of the great imbalance between the available memory bandwidth and the (high) rate of execution in the processor's floating point units. For the matrix-vector product, depending on nonzero structure, spatial locality may be poor and memory latency critical. Toledo's exemplary discussion in [9] demonstrated a combination of blocking, prefetching, and reordering as an effective remedy to make the best use of memory bandwidth. For CPTC, however, blocking and reordering are not directly applicable without major code changes.

2.2 The Preconditioner

The global extraction element-by-element preconditioner (EBE) was first described in [5] and all details can be found there. Here, we restrict ourselves to a brief description of the main idea.

Within the FEM, the global matrix A is the result of an assembly process $A = \sum_e A_e$ where each finite element e (such as tetrahedra or hexahedra in 3D) contributes a small (dense) elemental matrix A_e. Because of the adjacency of these elements, e.g. two elements sharing a vertex or an edge, there is a certain overlap between those contributions. There is a unique submatrix A_e^{ex} in A associated with each finite element. (A_e^{ex} consists of A_e plus the overlapping contributions from adjacent elements. For CPTC, a typical size of A_e^{ex} is 30×30 when using 10-noded tetrahedral elements with 3 degrees of freedom per node.)

The key idea of EBE is to use the family $\left\{(A_e^{ex})^{-1}\right\}_e$ as a preconditioner. The practical implementation has three separate kernels:

1. Since the A_e^{ex} do not change during the CG iteration, the inversion (factorization) has to be done only once as part of the preconditoner setup (using LAPACK's [10] `DPPTRF` routine).

2. In each CG iteration, for each element a *solve* $z = (A_e^{ex})^{-1}x$ must be performed (LAPACK's `DPPTRS`, which amounts to two calls to BLAS' `DTPSV`).

3. In each CG iteration, for each element a *gather* operation must be performed before the solve and a *scatter* operation after the solve.

```
/* gather */
for (i = 0; i < ndof; ++i)
  x[i] = y[pos[i]];
...solve...
/* scatter */
for (i = 0; i < ndof; ++i)
  y[pos[i]] += z[i];
```

There is a vast amount of parallelism in EBE, because the gatter/scatter and solves can be performed independently for all elements. (Typically, there are on the order of 10^6 elements in a finite element mesh). On the other hand, locality may be poor for the gather/scatter operations, the solve (equal to two triangular solves) involves divisions or reciprocals, and the average loop-trip-count in the inner-most loop of the triangular solver is small.

3 Features of the Itanium Architecture

Among the novel features of the Itanium architecture (compared to Pentium architecture) that are the most relevant to a CPTC-type code are the following:

- Support for software pipelining, through
  - Predicate registers
  - Rotating registers

- Counted loops with zero cycle branch latency
- An issue rate of two `fma` instructions per cycle
- Issue of six instructions per cycle in two bundles of three instructions each.

It remained to be seen to what extent our code would benefit from these new features. The following subsections contain a quick rundown of two traditional architectural measures.

3.1 Memory Bandwidth

Memory bandwidth appears to be crucial in many parts of CPTC. The STREAM [11] benchmark gives us a rough estimate of how much bandwidth we can expect on our target system. Table 1 shows the results for the OpenMP [12] versions of STREAM with one, two, and four threads, respectively. All results are in megabytes per second with parameters `N=5000000, NTIMES=30, OFFSET=8`. (Technical specifications for the test system can be found in Appendix A.1.) For each number of threads, there are two columns. The left column is for the version compiled with `-O2` and the right for the `-O3` version. In the `-O2` version, only all four CPUs seem to saturate the memory bus. The sole difference between the `-O2` and `-O3` versions is that in the latter case all STREAM loops are prefetched, which accounts for the higher bandwidth requirements.

Test	**1 Thread**		**2 Threads**		**4 Threads**	
Copy	782.0	1160.1	1148.9	1160.6	1227.4	1187.1
Scale	772.7	1169.0	1128.7	1169.6	1223.3	1161.2
Add	528.5	1319.0	945.3	1316.5	1388.0	1314.2
Triad	518.7	1339.3	925.4	1337.1	1394.8	1335.4

Table 1: STREAM (MB/s), `efl -O2/-O3`.

3.2 Floating Point Performance

The Itanium processor can issue 2 `fma` instructions (in separate `mfi` bundles) per cycle. On an 800 MHz processor, the theoretical peak performance is 4 floating point ops (= 2 `fma`) times 800 per second, in other words, 3.2 Gflops. A more realistic estimate for the sparse matrix-vector operations should be based on memory bandwidth. The peak read-only memory bandwidth for the Itanium processor-based system is roughly 2 GB/s. An L2 cache line is 64 bytes, or eight double-precision floating-point numbers. Assuming that there is no locality and that we have to go to memory for each floating point operation, we arrive at 250 Mflops. Over-estimating the memory subsystem and under-estimating the locality in our code, 250 Mflops becomes a rough estimate of the possible performance on our Itanium processor-based system.

4 Methodology

The process of optimizing changes because of EPIC technology. EPIC puts the entire burden of Instruction Level Parallelism (ILP) detection on the compiler's shoulders. Since the Itanium processor is an in-order machine, all the available parallelism must be present in the assembly. Thus, the compiler's translation from our source code to assembly is critical.

We chose for the CPTC performance metric the per iteration time in the iterative solver. We established a *baseline* by compiling the entire code with the `-O2` flag and ran it on a sample problem of about 30,000 degrees of freedom on a single processor. (This is a typical workload for a single processor in a parallel job.) On our test configuration, which is specified in Appendix A, the execution time per solver iteration was 0.486 seconds. Using the VTune Performance Analyzer, we identified the following hotspots (percent of execution time):

- 47% in `DTPSV`
- 30% in the (sparse) matrix vector product
- 8% in the gather/scatter operations.

Qualitatively this confirmed our expectations, though it was not clear at this point why, for example, we would spend 50% more time in `DTPSV` than in the matrix-vector product.

The following (simplified) pseudo code shows our methodology after establishing a baseline. It proceeds in loops, but there are a lot of branches in some of them.

```
_hotspots:
  identify new hotspot using VTune;
_change_source:
  make changes to the source code;
_check assembly:
  recompile;
  satisfied = check(assembly output, HLO and SWP reports);
  if (satisfied) { /* the assembly output looks okay */
    progress = run test problem;
    if (progress) /* faster */
      create new VTune sample;
    else { /* (unexpectedly) slower */
      retry = analyze changes made;
      if (retry)
        goto _check_assembly;
      else {
        discard source changes;
        goto _hotspots;
      }
    }
  } else /* the compiler doesn't get it */
    if (cflags_changed) { /* played with flags */
      reset compile flags;
      cflags_changed = false;
      goto _change_source;
```

```
} else { /* lets change compilation flags */
  change compile flags;
  cflags_changed = true;
  goto _check_assembly;
}
```

This loop terminates when either the architectural limits have been found, or the performance goals have been met.

Notice that the analysis of assembly code and compiler reports is not a late optimization step but, rather, a common first measure of the code's clarity (to the compiler). In fact, we did no hand-coding of assembly during our tuning effort. Assembly code is time-consuming to write and maintain and is unnecessary in most cases beacuse source code changes and compiler flags offer easy adequate control to achieve high performance.

5 Examples

In this section, we showcase the optimizations that yielded the largest performance gains in CPTC. We did not make the optimizations in this order. The five of us were simultaneously exploring different parts of the code. The "true story" can be found in Section 6.

5.1 The `DTPSV` Function

The CPTC code uses the CLAPACK distribution [13] which, as described in the release notes, was created using the f2c [14] Fortran-to-C translator. `DTPSV` is a BLAS function which solves systems of equations $Ax = b$ or $A^t x = b$ where A is an upper or lower triangular matrix. There are four core loops in `DTPSV`. (There are four other loops dealing with increments in the vector x that are different from one. For CPTC, however, the increment is always one.) The loops are discussed in the following subsections. We use the original line numbers in `dtpsv.c` from the Netlib [13] CLAPACK distribution.

In the following discussion, we assume that the reader has a reasonable understanding of software pipelining (SWP). Reference [15] is a good introduction and Volume 1 of reference [16] has a nice account of the Itanium processor's support for SWP.

A note on how to interpret the assembly code. Below, we show a few code examples with their assembly output. The compiler dumps key information about the instruction schedule in form of comments (the stuff after the `//`) into the assembly file. The comments are of the form

```
(predicate) instruction (;;) // cycle number : source line number
```

Note that the cycle number is zero based. For example, assuming that the following `fnma` instruction is part of a software pipelined loop, it would be issued on cycle 9 which is in stage 5 of the software pipeline (stage one is controlled by `p16` [16]) and it corresponds to source line 131.

```
(p20) fnma.d  f42=f6,f36,f41     //9:131
```

5.1.1 Loops 1 and 2

Because of the loop-carried memory dependencies in lines 131 and 173, the loops shown in the left parts of Figures 2 and 3 are not software pipelined when compiled with `ecl -O2`. For CPTC, the `ap` and `x` arrays do not overlap and we can disambiguate by declaring `double* restrict ap` and `double* restrict x`. When compiled with `ecl -O3 -Qrestrict`, the assembly code shown right is produced.

```
130 for (i__ = j - 1; i__ >= 1; --i__) {
131   x[i__] -= temp * ap[k];
132   --k;
133 }
```

```
.b1_47:
 {   .mmi
  (p16) ldfd    f37=[r33]          //0:131
  (p16) ldfd    f32=[r2],-8        //0:131
  (p16) add     r32=-8,r33 ;;      //0:130
 } { .mfi
  (p16) lfetch.excl.nt1 [r44]      //1:130
  (p20) fnma.d  f42=f6,f36,f41     //9:131
  (p16) add     r42=-16,r44        //1:130
 } { .mib
  (p24) stfd    [r41]=f46          //17:131
        nop.i   0
        br.ctop.sptk    .b1_47 ;;  //1:130
}
```

Figure 2: Loop 1. Disambiguation permits the compiler to software pipeline.

Loop 1 is software pipelined. The software pipeline has 9 stages and the initiation interval (II) is 2 cycles. The latency of a load (`ldfd`) is 9 cycles and the latency of the `fma` that feeds a store (`stfd`) is 8 cycles. The loads are on cycle 0 in stage 1, the `fma` on cycle 9 in stage 5, and the store on cycle 17 in stage 9. Each stage of the SWP is II (=2) cycles long. Notice that stages 2, 3, 4 , 6, 7, 8 of the SWP are empty (the predicates `p17, p18, p19, p21, p22, p23` are absent). They are there to fill the gap between the small II and the high latencies for the load/stores and the `fma`. As we mentioned earlier, for 10-noded tetrahedra, the triangular solves are done on 30×30 matrices. For such a matrix, loop 1 has an average trip count of 15. With a spin up and down of 9 stages, the SWP overhead is considerable.

Loop 2 behaves a little better. The assembly output is shown in the right part of Figure 3.

The loop is software pipelined, there are 5 stages in the SWP, and the II is 4. Stages 2 and 4 are empty. Furthermore, the compiler did unroll the loop by two. Why wouldn't the compiler unroll loop 1? The only qualitative difference between loops 1 and 2 is that loop 1 is executed in a decremental, rather than an incremental fashion. A brief inspection of loop 1 shows that it can be rewritten with a positive increment. The loop is then almost identical to loop 2 and will be unrolled by two by the compiler and the assembly code is identical to the one shown right in Figure 3.

5.1.2 Loops 3 and 4

The loop shown in the left part of Figure 4 is software pipelined when compiled with `ecl -O2` (II of 5, 2 stages). When compiled with `ecl -O3 -Qrestrict`, the assembly code shown right is produced.

It is software pipelined (II of 10, 2 stages) and unrolled by two.

With loop 4, not shown here, we face the same problem as with loop 1. It is almost identical to loop 3, its SWP is a little top-heavy, and it is not unrolled because the compiler does not like decremental loops.

```
172 for (i__ = j + 1; i__ <= i__2; ++i__) {
173   x[i__] -= temp * ap[k];
174   ++k;
175 }
```

```
.b1_76:
 {   .mmi
  (p16) ldfd    f32=[r8],16        //0:173
  (p16) ldfd    f37=[r33]          //0:173
        nop.i   0 ;;
 } { .mmf
  (p16) ldfd    f40=[r3],16        //1:173
  (p16) ldfd    f43=[r38]          //1:173
  (p18) fnma.d  f46=f7,f34,f39 ;;  //9:173
 } { .mmf
  (p20) stfd    [r37]=f48          //18:173
  (p20) stfd    [r42]=f36          //18:173
  (p18) fnma.d  f34=f7,f42,f45 ;;  //10:173
 } { .mii
  (p16) add     r32=16,r33         //3:172
  (p16) add     r37=16,r38         //3:172
  (p16) add     r42=32,r44         //3:172
 } { .mfb
  (p16) lfetch.excl.nt1 [r44]      //3:172
        nop.f   0
        br.ctop.sptk    .b1_76 ;;  //3:172
 }
```

Figure 3: Loop 2. A similar loop with positive increment is also software pipelined and prefetched, but here the compiler chose to unroll the loop by two.

The fix for loop 1, rewriting as incremental loop, does the job in this case as well.

Two remarks. For CPTC, the MKL version of DTPSV turned out to be slower than our hand-compiled version.

As we pointed out in Section 2.2, DTPSV is invoked twice per call to DPPTRS. Since DPPTRS is invoked very often (once for each element in each CG iteration!), overheads in the SWP might considerably degrade performance. The average workload in DTPSV is still relatively small when compared to the spin up and down costs of the SWP. These loops have a relatively small trip count and fairly long SWPs. To make things worse, there are expensive divisions (reciprocals) in the outer loops. An algorithmic change seems to be appropriate. We wrote two functions, DPPTRS2 and DTPSV2, with the same functionality as DPPTRS and DTPSV except that they are capable of processing two matrices and right hand sides simultaneously. This way we increase the ILP, decrease the number of stages (hence decrease overhead), and increase the number of parallel requests on the front side bus (FSB). (The CPTC performance improved by almost 20%.) From a software perspective, this is a departure from the standard BLAS/LAPACK which generally is not advisable. A more aggressive approach would be to abandon the use of BLAS/LAPACK and rewrite EBE in a way that the backsolves are done in a "super-loop" for all elements at the same time.

5.2 Sparse Matrix-Vector Multiplication

The CPTC code uses PETSc's CG implementation. In PETSc, the default storage format for sparse matrices is compressed sparse row format (CSR). Although PETSc has support for blocked formats (block compressed row and block diagonal storage) the FEM formulation in the current version of CPTC is done in a way that destroys the block structure (condensation of essential boundary conditions) and prevents us

```
214 for (i__ = 1; i__ <= i__2; ++i__) {
215   temp -= ap[k] * x[i__];
216   ++k;
217 }
```

```
.b1_107:
 {   .mmi
  (p16) ldfd    f35=[r18],16        //0:215
  (p16) ldfd    f36=[r17],16        //0:215
        nop.i   0 ;;
 } { .mmi
  (p16) ldfd    f32=[r16],16        //1:215
  (p16) ldfd    f37=[r15],16        //1:215
        nop.i   0 ;;
 } { .mii
  (p16) lfetch.excl.nt1 [r34]       //2:214
        nop.i   0
        nop.i   0 ;;
 } { .mfi
        nop.m   0
  (p17) fnma.d  f39=f33,f38,f34     //14:215
        nop.i   0 ;;
 } { .mfb
  (p16) add     r32=32,r34          //9:214
  (p16) fnma.d  f33=f35,f36,f39     //9:215
        br.ctop.sptk    .b1_107 ;;  //9:214
 }
```

Figure 4: Loop 3.

from using a blocked format. Nevertheless a certain structure, in the form of nonzeroes in identical column positions in consecutive rows, is preserved. This kind of structure is supported in PETSc by *i-nodes*. PETSc scans the matrix for sufficiently many i-nodes and automatically chooses optimized (for i-nodes) versions of matrix operations. For CPTC problems the natural i-node size is three (three displacements per FE node). We simplify our discussion of the sparse matrix-vector product by considering the "i-node free" version. All the optimizations which we discuss in the following carry over to the i-noded case, the latter actually being simpler than the case we are about to discuss (because there's more work in the loop body and better spatial locality).

Figure 5 shows the sparse matrix-vector multiplication from the NAS CG benchmark [17].

```
571 do j=1,lastrow-firstrow+1
572    sum = 0.d0
CCC assembly starts here
573    do k=rowstr(j),rowstr(j+1)-1
574       sum = sum + a(k)*p(colidx(k))
575    enddo
576   w(j) = sum
577 enddo
```

```
.b2_15:
 {   .mmi
  (p16) ld4     r40=[r9],4          //0:574
  (p16) ldfd    f32=[r8],8          //0:574
  (p17) shladd  r33=r2,3,r30 ;;     //3:574
 } { .mmi
  (p17) add     r65=-8,r33          //4:574
  (p16) lfetch.excl     [r28],4     //1:573
        nop.i   0 ;;
 } { .mfi
  (p17) ldfd    f37=[r65]           //5:574
  (p20) fma.d   f41=f36,f40,f43     //14:574
  (p16) sxt4    r2=r40              //2:574
 } { .mib
  (p16) lfetch.excl.nt1 [r27],8     //2:573
        nop.i   0
        br.ctop.sptk    .b2_15 ;;   //2:573
 }
```

Figure 5: Sparse matrix-vector product from NAS CG benchmark.

The assembly output (`efl -O3`) for the core loop is shown in the right part of Figure 5. It is software pipelined (5 stages, stages 3 and 4 empty, II of 3) and prefetched (stride-one accesses). On our test system, for the class A problem (the matrix A is a $14,000 \times 14,000$ matrix and 15 iterations), the performance for the entire CG algorithm is 88 MFlops, which is poor when compared to our conservative estimate of 250 Mflops in Section 3.1. There are two basic issues:

1. There is not much work done in line 574, which can be helped by unrolling.
2. The array `p` is not automatically prefetched.

Figure 6 shows an optimized version of the matrix vector product. The loop was unrolled by two, the loads are hoisted, and the array `p` was manually prefetched. The assembly output (`efl -O3`) for the core loop is shown in the right part of Figure 6. It is software pipelined (2 stages, II of 8 cycles) and the pipeline is much shorter this time. On our test system, for the class A problem, the performance for the entire CG algorithm is 187 MFlops.

PETSc's i-noded version of sparse matrix-vector multiplication is already unrolled. It only needs to be manually prefetched (5 prefetches for i-node size 3). The file to be modified is

```
$PETSC_DIR/src/mat/impls/aij/seq/aijnode.c
```

and the function to be modified is called `MatMult_SeqAIJ_Inode`.

5.3 The Gather/Scatter

In the left part of Figure 7 the core loop of the EBE preconditioner is shown. When compiled with `ecl -O3`, an inspection of the assembly code revealed that none of the loops was software pipelined.
A closer examination raises the following issues:

1. The accesses to the C++ vector `z` in lines 14 and 33, though the read and write access operators `operator[](size_t)` are declared as `inline`, are treated as function calls, and that is the main reason why these loops are not software pipelined.
2. The expressions in lines 14 and 32 should be simplified, and the pointer indirections should be eliminated.
3. The array `iebe->S` can be aliased and precalculated.
4. The accesses to `iebe->x_locp` in line 15 and to `iebe->y_locp` in line 32 are not stride-one, and the compiler will not prefetch them. (Arrays with stride-one access are automatically prefetched by the compiler. This is currently not done for arrays with indirect indices. Intel's compiler group is working to add this optimization.)

In the right part of Figure 7 an optimized version of the core loop of the EBE preconditioner is shown. The following changes were made:

1. We removed all C++ syntactic sugar. The compiler cannot handle it, for now (as of version 5.0.1B-30.2). We introduced the static `z` array in line 6 and invoke the triangular solve directly in line 44.

2. The expressions in lines 14 and 32 were simplified and the pointer indirections were eliminated.

3. We introduced the array `S` to alias and precalulate `iebe->S`.

4. Using intrinsics (lines 1 and 2), `iebe->x_locp` and `iebe->y_locp` are manually prefetched.

5. Alignment to 16 byte boundaries for automatic variables is not guaranteed by the compiler and is enforced in lines 6 and 7. It also enables the compiler to issue a load-pair instruction (`ldfpd`) in line 52.

All loops (lines 27, 33, and 50) are now software pipelined.

6 Summary of Results

We achieved a 3.42-fold speedup in the time per solver iteration, reducing the execution time from 0.486 seconds to 0.142 seconds. Table 2 shows an extract from our "CPTC logbook." It reflects the actual optimization progress. In the following, we look at the optimizations from a different angle and summarize the

Change	**Time (s)**	**Speedup**
Baseline	0.486	1.0
SWPipeline all relevant loops in `DTPSV`	0.340	1.43
No C++ exception handling and aliasing of pointers in EBE	0.244	1.99
SWP and prefetch gather/scatter	0.273	2.05
Remove C++ syntactic sugar from EBE	0.217	2.24
Alignment in EBE and load pair	0.207	2.35
MKL 5.0 BLAS	0.247	1.97
Hoist loads and prefetch (16 ahead) sparse matrix-vector product	0.186	2.61
`DTPSV2`	0.155	3.14
Better prefetch in EBE	0.142	3.42

Table 2: Extract from CPTC logbook. Note that the only algorithmic change was the creation of the routine `DTPSV2`.

lessons we have learned along the way.

6.1 CPTC Improvements

The Itanium processor comes with extensive performance monitoring capabilities. The architectural and microarchitectural events on the Itanium processor whose occurrences are countable through performance monitoring mechanisms are well documented [16, 18].

Table 3 shows performance monitoring results for CPTC and compares the baseline and the final versions of the code. The results were obtained using the powerful EMON tool developed by Intel. Unfortunately, EMON is not available publicly. However, similar utilities are available from SGI and HP.

The second row in Table 3 is based on the DATA_ACCESS_CYCLE counter which counts the number of cycles that the pipeline is stalled or flushed due to instructions waiting for data on cache misses, L1D way mispredictions, and DTC misses. "Data access cycles" is the ratio between this counter and the total cycles. Figure 8 shows the miss rates as a function of their latency in a diagram.

	Baseline	**Final**
Clocks per instruction	1.56	0.75
Data acess cycles	0.87	0.31
Average intructions between all stop bits	3.2	4.8
Misses with latency between 4-8 clocks	80.1%	80.3%
Misses with latency between 8-16 clocks	11.7%	14.6%
Misses with latency between 16-32 clocks	5.9%	2.0%
Misses with latency between 32-64 clocks	1.0%	0.9%
Bus bandwidth	129 MB/s	506 MB/s
Mflops	57	207

Table 3: Selected performance counters.

The significant decrease in clocks per instruction indicates a greater ILP which is confirmed by a higher average of instructions between stop bits. There is a remarkable decrease in data access cycles resulting in higher throughput. There is also a noticeable shift from the 16-32 towards the 8-16 clock latency range in the cache misses. Finally, our bandwidth utilization almost quadrupled. However, it is still only about 50% of what STREAM told us would be available. The Mflops rate appears reasonable in the light of our crude estimate in Section 3.1.

The VTune Performance Analyzer's output of the execution time distribution for the initial and final versions is shown in Table 4.

Function	**Baseline**	**Final**
DTPSV	47.0%	8.5%
DTPSV2	N/A	35.1%
Sparse Matrix-Vector Product	30%	23.6 %
Gather/Scatter	8%	20.7%

Table 4: Distribution of execution time according to VTune Performance Analyzer.

6.2 Lessons Learned

The compiler technology for the Itanium architecture is relatively new and has not yet (and cannot be expected to have) reached the level of maturity the technology has for the Pentium architecture.

Considerable performance gains can be achieved when standard optimization techniques, supported by a diagnostic tool like the VTune Performance Analyzer, are applied.

For "loopy" codes the following remarks apply.

1. Pay special attention to expression simplification, aliasing and disambiguation.
2. Analyze load/store dependencies and other issues that could make the compiler prefer an overly conservative resolution for potential ambiguities or false dependencies.
3. Be aware that the compiler (in level `O3`) will prefetch arrays only for stride-one accesses. Significant performance gains are possible through manual prefetching with the `__lfetch(*,*)` intrinsics.
4. Examine the assembly output and estimate the number of cycles needed for the loop using as many functional units of the CPU as possible. Check the number of loads, stores, and prefetches in the (C or Fortran) source code, divide it by two and compare it to the length of the loop scheduled by the compiler.
5. Examine the SWP output for your core loops carefully. Questions to ask:
   - Why does the compiler fail to software pipeline a loop?
   - What is the loop's trip count?
   - What is the ratio between the workload in the loop and the overhead introduced by SWP? Loops with low trip counts and many stages in the SWP should be avoided. Can I squeeze more work into the loop's body?
   - Why didn't the compiler unroll this loop? Can I help it by unrolling manually?
6. Check the alignment of your data and use the `__declspec(align(*))` primitive, if necessary. Cache line splits and expensive misaligned accesses are the potential problems. Also, the compiler will not issue load pair instructions for misaligned data.

6.3 A Few Remarks on Pentium Architecture

This paper is not about comparing Pentium and Itanium architectures. Without additional "boundary conditions" in form of a specific application, cost, amount of porting/tuning involved, any high-level comparison is a fairly futile undertaking. The two architectures are as different as they probably can be, and they are also targeted at completely different audiences. *If the obscurity of the Pentium architecture may appeal to a few people, the Itanium architecture certainly is a contemporary computer-architectural EPIC.*

After folding the changes for the Itanium processor back into the CPTC source tree, the performance improved only marginally on various Pentium-based systems. This can be partly attributed to the maturity of compilers for the Pentium line. On the other hand, for CPTC, vectorization alone will not yield dramatic performance gains and SWP is clearly the more flexible and powerful technology. Furthermore, we believe in the greater advantage of EPIC over runtime ILP detection mechanisms.

The performance of the CPTC on the Itanium processor, as the first implementation of EPIC, is certainly competitive with that of current generation Pentium 4 systems.

7 Conclusions

Although the Itanium processor is the first implementation of the Itanium architecture, it is a great processor for scientific computing. The Itanium architecture offers tremendous resources in the form of numerous functional units and a wealth of general purpose and floating point registers. We have found it advantageous that the compiler exercises full control of the utilization of those resources. The job of optimization becomes that of enabling the compiler to exploit parallelism in the code and optimize utilization of the memory subsystem. Compilers for the Itanium architecture are maturing, but, with significant programmer help, achieve top performance. Future performance increases will be available with Itanium architecture follow-ons (McKinley) and compiler improvements.

8 Acknowledgments

We would like to thank Intel® Solution Services for their hospitality at their Intel® Solution Center (formerly known as Intel Application Solution Center) in Chandler, AZ. Intel Solution Centers are state-of-the-art facilities for designing and testing high performance solutions. During two inspiring and extremely productive weeks of collaboration, we were able to test, tune and optimize solutions on Intel technology.

The first author was supported by the National Science Foundation's grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853, and ACI-0085969.

The second author was supported by the National Science Foundation's grants KDI-9873214 and EIA-9972853.

We would like to thank the head of the Cornell Fracture Group, Prof. Anthony R. Ingraffea, for his interest in and support of this project.

We gratefully acknowledge the support of the Dell Computer and Intel Corporations for providing Itanium processor-based hardware to port the CPTC environment at the Cornell Theory Center, and the support of the Intel and Microsoft Corporations for donating the necessary software.

A Configuration

A.1 Hardware

Processor(s): $4\times$ Itanium (C0) @ 733 MHz

Cache: See table 5

Memory: 4 GB RAM, 2×133 MHz FSB, 2.13 GB/s peak bandwidth

A.2 Software

OS: Microsoft Windows XP Advanced Server, 64-Bit Edition, Build 2462

SDK: Microsoft Windows Platform SDK Beta 2

	Size	Associativity	Security	Cacheline	Method	Latency (Cycles)
L1 (I)	16 KB	4-way	Parity	32 Bytes	Write Through	
L1 (D)	16 KB	4-way	Parity	32 Bytes	Read Only	2
L2	96 KB	6-way	ECC (64-Bit)	64 Bytes	Write Back	6 (INT), 9 (FP)
L3	4 MB	4-way	ECC (64-Bit)	64 Bytes	Write Back	21 (INT), 24 (FP)

Table 5: Itanium caches.

Compiler(s): Intel C/C++ and Fortran compilers 5.0.1B-30.2

Runtime: MPI Software Technology MPIPro 6.3, 64-Bit Edition

Profiler: Intel VTune Performance Analyzer 4.5 for Itanium

B Glossary

CPI: Cycles Per Instruction

EPIC: Explicitely Parallel Instruction Computing

HLO: High-Level Optimizer

II: Initiation Interval – is the number of cycles between the start of successive iterations. Each stage of a pipelined iteration is II stages long.

ILP: Instruction Level Parallelism

SWP: SoftWare Pipelining

Trip count: For a loop with limits `low` and `high`, and an increment of `inc` the trip count is given by `(|high-low|+|inc|)/|inc|`.

References

[1] Cornell Fracture Group home page. http://www.cfg.cornell.edu/, 2001.

[2] CPTC home page. http://www.tc.cornell.edu/Research/CMI/CrackProp/index.asp, 2001.

[3] MPI home page. http://www.mcs.anl.gov/mpi/, 2001.

[4] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[5] I. Hladik, M. Reed, and G. Swoboda. Robust Preconditioners for Linear Elasticity FEM Analyses. 40:2109–2117, 1997.

[6] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc/, 2001.

[7] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 2001.

[8] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[9] Sivan Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–726, 1997.

[10] E. Anderson et al. *LAPACK User's Guide*. SIAM, 1999.

[11] STREAM home page. http://www.cs.virginia.edu/stream/, 2001.

[12] OpenMP home page. http://www.openmp.org/, 2001.

[13] CLAPACK @ Netlib. http://www.netlib.org/clapack/index.html, 2001.

[14] F2C @ Netlib. http://ftp.netlib.org/f2c/index.html, 2001.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1996.

[16] Intel IA-64 Architecture Software Developer's Manual, 2000. Volumes 1–4, Revision 1.1.

[17] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/, 2001.

[18] Intel Itanium Architecture Software Developers Manual Specification update. http://developer.intel.com/design/itanium/manuals/248699.htm, 2001.

```
579 do j=1,lastrow-firstrow+1
580   i = rowstr(j)
581   iresidue = mod( rowstr(j+1)-i, 2 )
582   sum0 = 0.d0
583   sum1 = 0.d0
584   if( iresidue .eq. 1 ) then
585      sum0 = sum0 + a(i)*p(colidx(i))
586   endif
587   low = i+iresidue
588   high = rowstr(j+1)-2
CCC assembly starts here
589   do k=low,high,2
590      i0 = colidx(k)
591      a0 = a(k)
592      i1 = colidx(k+1)
593      a1 = a(k+1)
594      p0 = p(i0)
595      p1 = p(i1)
596      call lfetch_nt1(p(colidx(k+8)))
597      call lfetch_nt1(p(colidx(k+9)))
598      sum0 = sum0 + a0*p0
599      sum1 = sum1 + a1*p1
600   enddo
601   w(j) = sum0 + sum1
602 enddo
```

```
.b2_17:
 {   .mmi
  (p16) ld4     r35=[r10],8        //0:592
  (p16) lfetch.excl     [r34]      //0:589
        nop.i   0 ;;
 } { .mmi
  (p16) ldfd    f32=[r9],16        //1:591
  (p16) ld4     r36=[r8],8         //1:590
        nop.i   0 ;;
 } { .mmi
  (p16) ld4     r38=[r3],8         //2:596
  (p16) ld4     r39=[r2],8         //2:597
  (p16) sxt4    r40=r35 ;;         //2:595
 } { .mii
  (p17) lfetch.nt1      [r37]      //11:597
  (p16) sxt4    r35=r36            //3:594
  (p16) shladd  r41=r40,3,r30      //3:595
 } { .mmi
  (p16) ldfd    f34=[r29],16 ;;    //3:593
  (p16) add     r37=-8,r41         //4:595
  (p16) sxt4    r41=r39            //4:597
 } { .mii
  (p16) shladd  r36=r35,3,r30      //4:594
  (p16) sxt4    r40=r38 ;;         //4:596
  (p16) shladd  r38=r40,3,r30      //5:596
 } { .mmi
  (p16) add     r35=-8,r36         //5:594
  (p16) ldfd    f37=[r37]          //5:595
  (p16) shladd  r39=r41,3,r30 ;;   //5:597
 } { .mfi
  (p16) lfetch.excl.nt1 [r27],16   //6:589
  (p17) fma.d   f41=f35,f38,f42    //14:599
  (p16) add     r37=-8,r38         //6:596
 } { .mfi
  (p16) ldfd    f39=[r35]          //6:594
        nop.f   0
  (p16) add     r36=-8,r39 ;;      //6:597
 } { .mfi
  (p16) lfetch.nt1      [r37]      //7:596
  (p17) fma.d   f35=f33,f40,f36    //15:598
  (p16) add     r32=16,r34         //7:589
 } { .mib
        nop.m   0
        nop.i   0
        br.ctop.sptk    .b2_17 ;;  //7:589
 }
```

Figure 6: Optimized version of the matrix vector product with manual prefetching and unrolling by 2.

```
for (int i=0 ; i<M.numberOfElements() ; ++i)
{
  int ndof = M[i].degreesOfFreedom(), k, offset;

  Vector  z(ndof) ;

  //---------------------------------------
  // Gather and apply S^{1/2}
  //---------------------------------------

  offset = iebe->ele_offset[i];
  for (k=0 ; k<ndof ; ++k)
  {
    z[k] = iebe->S[offset]
     *iebe->x_locp[iebe->ele_dof_to_loc[offset]];
    ++offset;
  }

  //---------------------------------------
  // Triangular solves
  //---------------------------------------

  M[i].solve(z);

  //---------------------------------------
  // Apply S^{1/2} and scatter
  //---------------------------------------

  offset = iebe->ele_offset[i];
  for (k=0 ; k<ndof ; ++k)
  {
    iebe->y_locp[iebe->ele_dof_to_loc[offset]]
             += iebe->S[offset]*z[k];
    ++offset;
  }
}
```

```
#include <ia64intrin.h>
#include <xmmintrin.h>

...

__declspec(align(16)) double z[30];
__declspec(align(16)) double S[30];
__declspec(align(16)) int    loc[30];

for (int i=0 ; i<M.numberOfElements() ; ++i)
{
  int ndof = iebe->ndof[i];
  int k, offset, info, nrhs = 1, *loc_ptr,
      n = ndof;
  char *uplo = "L";
  double xloc, yloc, *S_ptr, *x_locp_ptr,
         *y_locp_ptr;

  //---------------------------------------
  // Gather and apply S^{1/2}
  //---------------------------------------

  offset     = iebe->ele_offset[i];
  S_ptr      = &(iebe->S[offset]);
  loc_ptr    = &(iebe->ele_dof_to_loc[offset]);
  x_locp_ptr = iebe->x_locp;
  y_locp_ptr = iebe->y_locp;

  for (k=0 ; k<ndof ; ++k)
  {
    loc[k] = *loc_ptr++;
    S[k] = *S_ptr++;
    __lfetch(_MM_HINT_NT1, x_loc_ptr+loc[k]);
  }
  for (k=0 ; k<ndof ; ++k)
  {
    xloc = *(x_locp_ptr + loc[k]);
    z[k] = S[k] * xloc;
    __lfetch(_MM_HINT_NT1, y_loc_ptr+loc[k]);
  }

  //---------------------------------------
  // Backsolve
  //---------------------------------------

  cptc_dpptrs(uplo, &n, &nrhs, iebe->factor[i],
              z, &n, &info);

  //---------------------------------------
  // Apply S^{1/2} and Scatter
  //---------------------------------------

  for (k=0 ; k<ndof ; ++k)
  {
    yloc = S[k] * z[k];
    *(y_locp_ptr + loc[k]) += yloc;
  }
}
```

Figure 7: EBE core routine.

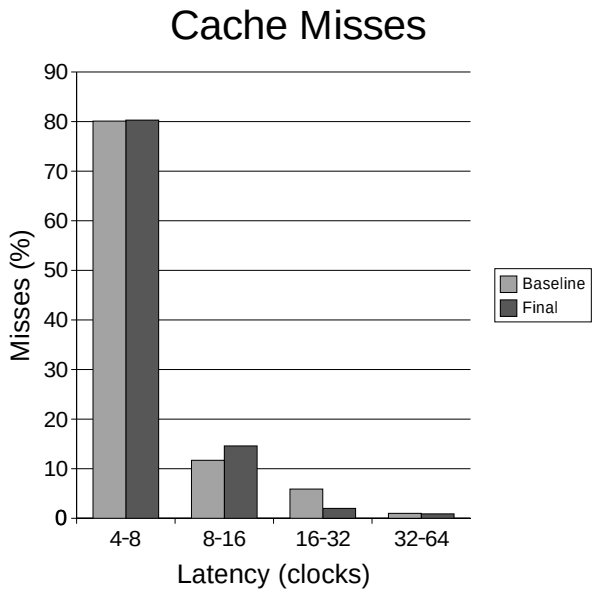

Figure 8: Percentage of cache misses as function of latency.