

HP Compilers for HP Integrity Servers

HP-UX



©2002–2008 Hewlett-Packard Development Company, L.P.

July 28, 2008

Legal Notices

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend. All rights are reserved. No part of this document may be copied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this material is subject to change without notice.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
United States of America

Copyright Notices.

Copyright © 2002–2008 Hewlett-Packard Development Company, L.P., all rights reserved.

Reproduction, adaptation, or translation of this material without prior written permission is prohibited, except as allowed under the copyright laws.

Trademark Notices.

HP-UX Release 11i and later (in both 32- and 64-bit configurations) on all HP Integrity servers are Open Group UNIX 95 branded products.

Intel® and Itanium® are registered trademarks of the Intel Corporation.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Table of Contents

HP Compilers for HP Integrity Servers

Understanding HP Compilers	7
Optimizing for Integrity Servers	9
Predication	9
Control speculation	10
Data speculation	11
Explicit Parallelism	12
What's New in the Compilers for AR0709 and AR0809	13
Performance	15
Understanding Key Features of the HP Compilers	16
Standards compliance.....	16
Compatibility	17
Extensive application availability	17
Faster development and debug.....	17
Advanced low-level optimization	18
Profile-based optimization	18
Powerful high-level optimization.....	20
Precise floating-point control.....	25
Extensive inline assembly	27
Application Tuning.....	28
Profiling	28
Include header files	29
Scheduling for the processor.....	29
Choosing the link mode	30
Increasing the page size	30
Describing application characteristics	30
Tuning with profile-based optimization	33
Tuning across program modules	33
Tuning floating-point numerical code.....	33
Allowing optimization flexibility.....	35
Using inline assembly	35
Troubleshooting optimization problems	36
Additional Information.....	38
References	39

List of Figures

Figure 1:	Internal structure of the HP compilers	8
Figure 2:	Build model for interprocedural optimization	24

List of Tables

Table 1	Floating-point type suffixes and macros.....	25
Table 2	General purpose tuning options and when to use them	32
Table 3	Options and pragmas for troubleshooting problems in optimized code.	37

HP Compilers for HP Integrity Servers

This document provides a technical overview of the key features of HP compilers for HP Integrity servers running the HP-UX 11i v2 or later operating system.

Understanding HP Compilers

HP Integrity servers use the Intel® Itanium® architecture, co-developed by HP and Intel, which uses Explicitly Parallel Instruction Computing (EPIC). EPIC enables processors to take advantage of advanced compiler techniques and massive processor resources to execute instructions faster and more in parallel than traditional RISC. HP compilers have been designed in parallel with the architecture to exploit the benefits of this architecture for real applications. In addition to the key features of HP compilers, this paper suggests how developers can use HP compilers to unlock the performance advantages of Integrity servers.

HP offers a family of compilers for Integrity servers, supporting the C, C++, and Fortran languages. HP compilers share an overall design structure that facilitates the integration of common functional components such as the code generator, optimizer, linker, and debugger.

The HP compiler structure (see Figure 1) begins with a language-dependent front end which includes components for lexical, syntax, and semantic analysis of the incoming source code. Each front end produces an intermediate-level representation of the program. The high-level optimizer performs performance-enhancing optimizations of the intermediate code. The code generator converts the intermediate representation into an instruction sequence nearly appropriate for the target system. Finally, the low-level optimizer completes the generation of machine code and performs additional transformations which improve performance.

HP-UX system libraries on Integrity servers are generally supersets of their counterparts on HP 9000 systems based on the PA-RISC architecture. For example, the HP-UX C/C++ math library for Integrity servers provides an API that is a major superset of the PA-RISC library. It includes all the C99/Unix 2003, Unix 95, and PA-RISC math functions for four floating-point types. The functions provided for Integrity servers are generally faster, more accurate, and more consistent in the treatment of exceptional cases than their PA-RISC counterparts.

All HP compilers share a common implementation of the code generator and optimizers in order to maximize inter-operability between languages.

The HP compilers for Integrity servers have already compiled millions of lines of C, C++, and Fortran source code for HP-UX 11i, including HP-UX 11i itself. Much of this compilation is at high levels of optimization. Hundreds of independent software providers have used the HP compilers to make thousands of applications available on HP-UX 11i. HP uses its compilers to tune performance of the SPEC2006 benchmark, which comprises a total of 29 applications, using advanced levels of optimization. These programs make extensive use of sophisticated Itanium processor family features such as predication, speculation, and data prefetching.

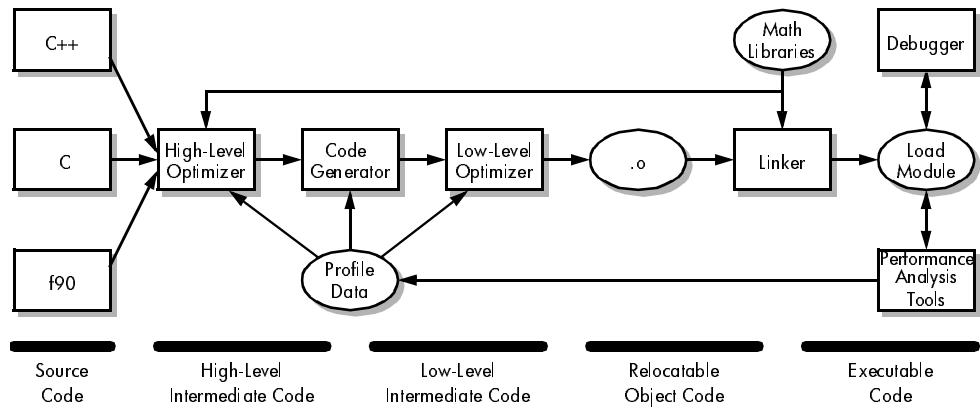


Figure 1: Internal structure of the HP compilers

Optimizing for Integrity Servers

The Intel Itanium architecture seeks to reduce execution time by maximizing instruction-level parallelism—the concurrent execution of multiple instructions. It provides three key features that enable the compiler to maximize instruction-level parallelism (ILP):

- predication
- speculation, both of control and data
- explicit parallelism

While support from the architecture for these features is critical, the compiler must exploit these features to their utmost in order to deliver superior application performance.

Predication

Predication is the conditional execution of an instruction based on the setting of a boolean value contained in a predicate register. The Intel Itanium architecture provides 64 predicate registers that can be used to control the execution of nearly all instructions.

In the example below, both assignments to `x` can execute in the same cycle (because both predicates are never simultaneously true), saving two instructions and at least one execution cycle, and avoiding any risk of branch misprediction.

Example 1: Using predication

```
if (a == 0) {
    x = 5;
} else {
    x = *p;
}
```

The compiler can use predication to transform control dependencies on branch instructions into data dependencies on compare instructions.

Example 2: Code from Example 1 generated using branches

```
        cmp.ne.unc p1,p0 = a,0
(p1) br      L1 ;;
        mov      x = 5
        br      L2 ;;
L1:  ld      x = [p]
L2:
```

The assignment to `x` that is executed is control dependent on the predicate (`p1`) in the first branch instruction.

Transforming from control dependence to data dependence has two principal benefits:

- Removal of branches, which increases the number of instructions per cycle. For example, by eliminating the branches in Example 2, both assignments to `x` can be executed in the same cycle.
- Elimination of the misprediction penalty associated with branches. In a pipelined processor, a branch presents a potential disruption in the pipeline flow. The processor must predict whether a conditional branch is taken and must predict the target, if it is indirect. An incorrect prediction flushes and restarts the pipeline. With a deep pipeline and wide issue bandwidth, this represents a significant loss of performance. On the Intel Itanium processor, for example, a branch misprediction penalty is 9 cycles, representing 54 lost instruction issue opportunities. Even with sophisticated branch prediction techniques, a small percentage of mispredicted branches can translate into significant performance cost.

Example 3: Code from Example 1 generated using predication

```
        cmp.ne.unc p1,p2 = a,0 ;;
(p2)   mov        x = 5
(p1)   ld         x = [p]
```

In Example 3, all branches have been eliminated and the assignments to `x` are now data-dependent upon the compare that defines the qualifying predicate.

Control speculation

Control speculation is the execution of an instruction before the execution of all of the conditions controlling its execution. Using control speculation, the compiler can generate code which causes the program to execute conditional code concurrently with a guarding condition, even ahead of a guarding condition, rather than waiting for the result of the guarding condition evaluation. Such concurrency can significantly improve runtime performance.

Example 4: Control speculation code

```
int a,b;
extern int *p;
extern int global;
if(condition) {
    a = global;
    b = *p + 2;
}
```

Using control speculation, the HP compiler can cause the program to execute portions of the two assignment statements in the `then` clause ahead of the condition.

Loads and arithmetic operations on integer variables are ideal candidates for control speculation because they do not cause undesired side effects. A source code statement may utilize loads, arithmetic operations, and stores. If the source code statement is guarded by a condition evaluation, the loads and arithmetic operations can be speculated; any store operations are dependent on the condition evaluation and cannot be speculated. Calls are also exempt from control speculation.

Because the speculative load is being performed ahead of the guarding condition, it is possible (for a variety of reasons) that the load might not be successful. Unsuccessful speculative loads result in a speculation token (called a NaT) in the target register. This NaT token propagates through subsequent instructions. After the guarding condition is finally evaluated, the results of the speculated instructions are used if they are still needed. If any NaT token is found, a recovery code sequence is executed which recomputes the needed results.

Example 5: Code from Example 4 using control speculation

```
        ld.s      t1 = [p] ;;
        add      b = t1,2
        cmp.ne.unc p1,p0 = condition,0 ;;
(p1)   chk.s     t1, L2
L1:
...
L2:   ld        t1 = [p] ;;
      add      b = t1,2
      br      L1
```

If the NaT bit on register `t1` is set, the `chk.s` instruction branches to the recovery code located at `L2`. Recovery code reloads `t1` without speculation, then recomputes the result in `b`. `(p1)` is a predicate used to determine whether the result is needed.

A variety of factors could cause a speculative load to result in a NaT token and potentially trigger the execution of recovery code. Loads from invalid addresses usually generate an exception on a traditional architecture; speculative loads from invalid addresses result in NaT. In addition, a miss in the translation-lookaside buffer (TLB) on the control speculative load usually results in a NaT when recovery code is present. A TLB is a cache of translations from virtual memory addresses to physical addresses and physical addresses to virtual memory addresses.

Data speculation

Data speculation involves the early execution of a load prior to one or more store instructions that both:

- preceded the load in original program order.
- might write to the same memory location as is read by the load.

Data speculation can reduce the overall number of required cycles because it increases instruction level parallelism.

Example 6: Using data speculation

```
int a,b
extern int *p;
extern int *q;
*p = a;
b = *q + 2;
```

Use of data speculation moves the last statement performing a load of `*q` above the store of `*p`, even though `*p` and `*q` may be the same memory location.

The Intel Itanium architecture allows the compiler to exploit this type of speculation safely by providing a facility to dynamically identify address conflicts and by allowing the compiler to trigger execution of a recovery code sequence when an address conflict is discovered during runtime. The compiler utilizes the advanced check (`chk.a`) instruction that checks to see if there have been any conflicting writes to the address accessed by the advanced load. If such a conflict occurs, the advanced check branches to compiler-generated recovery code where the load is reexecuted to ensure the correct value.

Example 7: Generated code for Example 6 using data speculation.

```
ld.a      t1 = [q] ;;
add       b = t1,2
st        [p] = a
chk.a    t1, L2

L1:
...
L2: ld     t1 = [q] ;;
add       b = t1,2
br        L1
```

The load of `*q` is moved ahead of the store of `*p`. The `chk.a` checks for conflicting writes to the location whose contents were loaded into `t1`, and branches to `L2` to correct them.

Explicit Parallelism

Explicit parallelism allows the compiler to take advantage of its knowledge of the program semantics, combined with a model of the processor resources, to generate groups of instructions that can be executed in parallel (without requiring hardware dependency analysis). Current processors can execute up to 6 instructions per cycle (2 bundles of 3 instructions each). Supported by a large register file and multiple execution units, the compilers are able to schedule multiple computations in parallel. With explicit “stop bits” in the instruction stream, the compilers indicate exactly which groups of instructions may be executed in the same cycle. The compilers optimize the code to maximize the number of parallel computations performed in each cycle, using the techniques of predication, speculation, and modulo scheduling of loops to increase the opportunities for parallelization.

What's New in the Compilers for AR0709 and AR0809

The **AR0809 (A.06.20)** release of the HP compilers adds a number of new features, including:

- Support for decimal floating-point arithmetic for C on HP-UX 11i V3 (11.31) Integrity systems, following the IEEE 754-2008 floating-point standard and the draft ISO/IEC Technical Report 24732, "Extensions for the programming language C to support decimal floating-point arithmetic."
- Version C.02.00 of HP Code Advisor. New features in this version include:
 - support for detecting violations of pre-defined or user-defined coding guidelines
 - enhancements to +w64bit diagnostics
 - enhancements to the program complexity metrics
 - reports in XML format.
- Further improvements in optimizations, and the availability of +Oautopar for C++.
- Availability of pragmas to save and restore severities of diagnostics. The #pragma diag_push directive saves the severity state of all front-end and back-end diagnostics, and the #pragma diag_pop directive restores the diagnostic severity state to the state before the previous #pragma diag_push directive.
- New suboptions to +check such as:
 - The +check=lock command-line option, which enables the checking of C and C++ applications which employ pthreads and reports violations of locking discipline when appropriate locks are not held while accessing shared data by different threads. For more information about the technique used, see Reference 22.
 - The +check=thread option, to enable gdb thread checking features at runtime. See Reference 23 for more details.
 - Enhancements to the +check=uninit option, to enable runtime checks to detect uninitialized reads of local variables and heap objects.
- Enhancements to the +inline_level option, which now accepts values in the range of 0.0 to 9.0 with 0.1 increments to allow for finer-grained control on the aggressiveness of the inliner. The +Oinline_budget option is now deprecated.
- Enabling of some loop optimizations such as fusion, unrolling, unswitching and rerolling at the optimization level two (+O2). The loop optimizer is also now more aggressive at recognizing more kinds of loops.

The **AR0709 (A.06.15)** release of the HP compilers delivers a number of new features, including:

- Optimization of printf and fprintf calls into puts and fputs calls respectively, at optimization levels two or higher.

- Several new diagnostics and command-line options for controlling them:
 - The `+Wmacro` option for disabling the specified diagnostics in the expansion of the specified macro.
 - The `+Wcontext_limit` option for limiting the number of instantiation contexts emitted in diagnostics about template instantiations.
 - The `+wperfadvice` option for enabling and controlling the verbosity of performance advisory diagnostics.
 - The `+Wv` option for detailed descriptions of the specified diagnostics.
 - The `+wlock` option for enabling compile-time diagnostics for potential incorrect usages of pthread library-based lock/unlock calls in programs.
- Ability to automatically parallelize loops with `+Oautopar`. When used at optimization levels three or higher, the compiler will transform serial loops into multi-threaded parallel code when the loop transformer considers such a conversion profitable, thus allowing applications to exploit otherwise idle resources in multicore or multiprocessor systems. `+Oautopar` can be combined with `+Oopenmp` and OpenMP directives for manual parallelization. When used in combination, `+Oautopar` considers automatically parallelizing only those loops which are not manually parallelized with the OpenMP directives.
- Optimization to improve data cache locality with `+Oloop_block`, which transforms nested loops operating on arrays too large to fit in the cache, into loops which operate on strips of these arrays such that they fit.
- A change at optimization levels of three or higher, where `+Oloop_unroll_jam` is now enabled by default.
- A change in the C mode of the compiler, where `+Olit=all` is now the default, placing all string constants in read-only memory.
- The `+macro_debug` option to control the emission of macro debug information into the object files.
- The `+pathtrace` option to save program execution control flow into global and/or local path tables, permitting the HP WDB debugger to assist with crash path recovery from the core files, or to assist when debugging the program by showing the executed branches.
- The `+annotate=structs` option to annotate all accesses to the fields of C/C++ structs for use by other performance analysis tools such as Caliper to produce data-centric reports.
- New suboptions to `+check` such as:
 - The `+check=globals` option, to enable runtime checks to detect corruption of global variables.
 - The `+check=truncate` option, to enable runtime checks to detect data loss in assignments when integral values are truncated.
 - Enhancements to the `+check=bounds` option, to provide the option of checking for out-of-bound references to buffers through pointer access as well as to array variables.

- The `-Bhidden_def` option, which is similar to the `-Bhidden` option, but assigns the hidden export class to only locally defined symbols.
- The `-dM` option, which when used with the preprocessor option `-P` or `-E`, lists all macros that are in effect at start of compilation.
- Enhancement of the `#pragma OPT_LEVEL` directive to accept an argument `INITIAL`, thus allowing the optimization level in effect at the start of the compilation to be restored subsequent to changes caused by other `#pragma OPT_LEVEL` directives.
- Support for debugging code compiled with optimization levels two or higher.

Performance

In addition to these new features, the AR0709 and the AR0809 releases deliver significant application performance improvements. Compared with the AR0606 release, the AR0709 release improves performance by 5-10% and the AR0809 release shows a further 4-7% performance improvement over the AR0709 release.

Understanding Key Features of the HP Compilers

The following sections describe each of these key features:

- “Standards compliance” on page 16
- “Compatibility” on page 17
- “Extensive application availability” on page 17
- “Faster development and debug” on page 17
- “Advanced low-level optimization” on page 18
- “Profile-based optimization” on page 18
- “Powerful high-level optimization” on page 20
- “Precise floating-point control” on page 25
- “Extensive inline assembly” on page 27

Standards compliance

Each HP compiler adheres to defacto industry and international standards to enhance link and runtime compatibility and source code portability. This adherence to open standards protects the investment of application developers and provides rapid development and deployment of new applications. The HP C compiler is branded for compliance to ISO/IEC 1990 and on 11i Version 3 (11.31) Integrity systems, the UNIX 2003 standard, which includes ISO/IEC 9899:1999; the HP Fortran compiler adheres to ISO/IEC 1539-1: 1997; the HP aC++ compiler is largely compliant with the ISO/IEC 14882 standard for the C++ language (including the C++ standard library).

All HP compiler code generation and data layout conforms to the *Common Software Conventions and Runtime Architecture* (see Reference 9) for the Intel Itanium architecture. In addition, the HP aC++ compiler code generation and data layout largely conforms to the Common C++ ABI for the Intel Itanium architecture (see Reference 10)—with a few documented exceptions.

The math library conforms to the specification in the ISO/IEC C99 standard, including the annex for IEC 60559 (IEEE 754-1985) implementations and the annex for IEC 60559-compatible complex arithmetic. The compilers and math library adhere to the IEC 60559 (IEEE 754-1985) binary floating-point standard. They fully support three real IEC 60559 binary floating-point types: float (32-bit), double (64-bit), and long double or quad (128-bit PA-RISC-compatible type). The C and C++ compilers and libraries additionally support the IEC 60559 compatible, 80-bit, extended type in the Intel Itanium architecture (see Reference 1). The C compiler and libraries for 11i Version 3 (11.31) support three IEEE 754-2008 decimal floating-point types, to the extent specified by draft ISO/IEC Technical Report 24732: `_Decimal32`, `_Decimal64`, and `_Decimal128`.

Compatibility

HP provides near-complete source code and makefile compatibility between its current line of HP 9000 PA-RISC compilers and its new generation of compilers for easy migration of source code and makefiles to Integrity servers. The PA-RISC compiler options are nearly unchanged in the new compilers, while several new options provide access to the new features available only on Integrity servers. Occasionally, it is necessary to introduce a minor incompatibility in order to comply with standards or repair a defect. For more information, see Reference 18 and Reference 19.

Built to run on Integrity systems running HP-UX 11i v2 and later, the AR0809 compilers continue to provide significant compatibility with frequently-used features of the Tru64 C and C++ compilers. In particular, the Tru64 C++ ARM dialect is provided, as this was the default Tru64 C++ dialect through Version 5.

Extensive application availability

HP compilers support the compiler options and pragmas most often chosen by software developers as essential for preserving and extending application availability. For example, the HP compilers support both the widely used and traditional 32-bit data model and the newer 64-bit data model where longs and pointers are 64 bits wide. The traditional 32-bit data model is appropriate for many legacy applications which may not be 64-bit clean. Many other compilers require the application to comply with the 64-bit data model which usually requires a separate 64-bit migration step for legacy applications.

To extend the lifetime of new applications for Integrity servers, HP compilers provide several code scheduling options. These options allow software providers to target a specific processor model or to use a blended model that is suitable for all members of the processor family.

Faster development and debug

Traditionally, compilers perform minimal optimization by default and no optimization when debugging is specified. This approach is inappropriate for Itanium-based systems, where unoptimized programs generally run about two to three times slower than when optimized at +O1 and four to five times slower than when optimized at +O2.

Some optimizations are also required for a debug build since 30 to 50% of the instructions in an unoptimized code sequence are `no-op` instructions. This relatively large number of `no-op` instructions is due to the need to form three-instruction bundles, and the limited number of bundle templates available. With optimization, the compiler is able to make much more effective use of the bundle templates.

HP has significantly enhanced performance of code compiled for debugging by providing +O1 level of optimization by default. Optimizations performed at +O1 include common sub-expression elimination, constant propagation, load store elimination, copy elimination, register allocation and restricted basic block scheduling. Care has been taken to ensure that the program can still be debugged correctly; that is, that breakpoints are at expected places and variables have expected values at breakpoints corresponding to source lines.

Advanced low-level optimization

At optimization level 2 (option +O2), HP's low level optimizer takes full advantage of the key features of the architecture. In addition to the local optimizations applied at +O1, the optimizer applies Static Single Assignment (SSA)-based global value numbering (see Reference 6), global code motion, value congruent instruction elimination to reduce the static and dynamic number of instructions, aliased scalar promotion (see Reference 7), a fast version of interprocedural inlining using "tuned-down" heuristics, and SSA-based partial redundancy elimination. The loop optimizer performs data prefetching, sum reduction, scalar replacement, strength reduction, post-increment synthesis and loop unrolling. Data prefetching is automatically performed on loops where the optimizer is able to discern an array reference pattern or linked-list traversal.

HP compilers divide application code into regions which form the unit of operation for instruction scheduling. The instruction scheduler employs control speculation, data speculation, and predication to schedule the region as efficiently as possible, maximizing instruction-level parallelism (see Reference 3). Where possible, given reasonable constraints on compile time, innermost loops are subject to software pipelining. The software pipeliner takes advantage of the special branches and rotating registers provided in the architecture to generate software pipelined loops with little or no code expansion, even in the presence of control flow and non-counted loops (see Reference 5).

Profile-based optimization

HP is a leader in the delivery of profile-based optimization (PBO) (see Reference 2). PBO data provides the compiler with branch-taken and routine execution frequency information as an additional guide to optimization. In addition, it provides the compiler with data access address strides and data cache miss information, used to guide data cache optimization and scheduling. It also provides the compiler with loop iteration counts, used to guide loop optimization. PBO can provide as much as a 30% performance improvement over +O2 optimization by tuning applications according to their typical execution characteristics. The performance impact of PBO is even higher on Itanium-based systems than on traditional RISC systems because the architecture provides a larger number of mechanisms to increase instruction level parallelism based on application behavior.

Many compiler optimizations are enhanced by knowledge of the execution behavior of the application.

- Certain optimizing transformations are performed on code regions. Profile data helps these transformations select target regions to minimize region crossings within high frequency execution paths.
- Selection of instructions within a region to speculate or predicate is more effective when the compiler has more accurate information on relative execution frequencies.
- High-level optimizations such as loop optimization and procedure inlining can greatly benefit from profile data to select particularly hot loops and call sites for optimization.
- The optimizer can insert more efficient prefetches for linked-list recurrences, if the PBO data indicates that the accesses have a regular stride.

- Cache utilization is enhanced by ordering global and static variables within the data segment such that frequently accessed variables are placed close together.
- For loops that commonly iterate only a few times, as indicated by the loop iteration count PBO data, the optimizer can “peel” off that number of iterations into straight-line code. This can improve instruction level parallelism by allowing greater scheduling freedom for the peeled instructions.
- Scheduling is enhanced by accounting for data cache misses on integer accesses and either reordering the loads or scheduling uses farther away.

On Integrity servers, the two-step PBO process can be done through the `+Oprofile=collect` build, followed by the `+Oprofile=use` build, similar to the process on PA-RISC systems. The first step of the build (`+Oprofile=collect`) inserts instrumentation code to collect edge weights, data access address strides, and loop iteration counts. When the binary is subsequently run, in addition to the profile data collected by the instrumented code, HP Caliper samples load data cache profile information using the performance monitor unit (PMU). All collected profile information is written into a data file, which is used by the compiler for the subsequent `+Oprofile=use` build.

New for Integrity servers, HP compilers also provide profile-based optimization using compiler options and source code pragmas. These options and pragmas fine-tune profile data or substitute for profile data in situations where the collection of a typical application input might be difficult. Some sources of difficulty include:

- Representative input data sets might not be readily available.
- Application or system configurations representative of all customer usage profiles might not be practical to duplicate.

The option `+Ofrequently_called` indicates to the compiler those functions that are called relatively frequently. The option takes a `list` or a `filename` as an argument; the file should contain a white-space separated list of function names. The `file` option allows function name lists to be generated through an automated tool.

Similarly, `+Orarely_called` identifies those functions that are relatively rarely called. Alternatively, this information can be expressed through the source code using the `FREQUENTLY_CALLED` and `RARELY_CALLED` pragmas.

The `ESTIMATED_FREQUENCY` pragma is a block-scope pragma that indicates the estimated relative execution frequency of the current block as compared with the immediately surrounding block. This can be used to indicate the average trip count in the body of a loop, or to indicate the fraction of time a `then` clause is executed. The pragma accepts a constant argument which is the expected execution frequency or loop count.

Example 8: Typical use of the `ESTIMATED_FREQUENCY` pragma

```
if (condition) {
#pragma ESTIMATED_FREQUENCY 0.99
    ...
    for (...) {
#pragma ESTIMATED_FREQUENCY 4.0
        ...
    }
} else {
    ...
}
```

In Example 8, the code in the then clause of the `if` statement is expected to execute 99% of the time (implying that the `else` clause is executed 1% of the time). The loop is expected to execute four iterations, on average.

The `ESTIMATED_FREQUENCY` pragma gives the developer fine-grain control over the degree of control speculation used by the compiler around any given source code condition. In addition, knowledge of the average loop iteration count can cause the compiler to determine that data prefetching would not be effective.

The `NO_RETURN` pragma asserts to the compiler that the specified function does not return. This allows the optimizer to simplify the control flow graph, enabling more aggressive optimization and reduced pressure on the register stack.

Powerful high-level optimization

The HP high-level optimizer contains an interprocedural optimizer, a high-level loop optimizer, and a scalar optimizer.

The interprocedural optimizer is enabled with the option `-ipo` at optimization levels two or higher (e.g. `+O2 -ipo`). Optimization level four (option `+O4`) implies `-ipo`. The loop optimizer is enabled at optimization levels three or higher (options `+O3` and `+O4`). The scalar optimizer is enabled along with the other high level optimizations.

The option `-ipo` can be used to compile some or all of an application's source files. Compiling only some modules with `-ipo` enables intermodule optimizations between those files. In this mode, only parts of the application are analyzed during IPO by the compiler and therefore the compiler has to make conservative assumptions about the rest of the application. This can result in lost optimization opportunities.

For highest performance, it is beneficial to compile all of an application's source files with `-ipo`; this is called "whole program mode." In this mode, the compiler can perform precise analysis of an application, potentially resulting in better performance.

The high-level optimizer makes use of PBO information and is more effective when used in combination with PBO (option `+Oprofile=use`), for example, PBO data improves function inlining. PBO data can reveal the most likely callee at an indirect call site, allowing the high-level optimizer to transform the indirect call into a test and a direct call.

The inliner framework has been designed to scale to very large applications. It uses a novel and fast underlying algorithm and employs an elaborate set of heuristics to guide its inlining decisions.

The inlining engine is also employed at +O2 for intra-module inlining. At this optimization level the inliner uses tuned down heuristics in order to guarantee fast compile times.

Application performance benefits from interprocedural optimization in the following ways:

- Interprocedural analysis of memory references and function arguments enables and improves many optimizations; for example, it yields additional opportunities for register promotion.

Consider this example:

```
void foo( int *x, int *y )
{
    ... = *x; // load 1
    *y = ... // store 1
    ... = *x; // load 2
}
```

Without any additional knowledge about the properties of the pointers *x* and *y*, the compiler has to issue a second load instruction (load 2), since the store (store 1) may overwrite the content of the pointer *x*.

If, as a result of interprocedural analysis, the compiler was able to determine that *x* and *y* never alias (point to the same memory location), the compiler can promote the value of **x* into a register and just reuse this register (load 2).

- Function inlining exposes traditional benefits, such as the reduction of call overhead, the improvement of the locality of the executing code and the reduction of the number of branches. More importantly though, inlining exposes additional optimization opportunities because of the widened scope, which also enables better instruction scheduling.
- The whole call graph is constructed, enabling indirect call promotion, where an indirect call is converted to a test and a direct call. Depending on the application characteristics, and in the presence of PBO data, this can result in significant application speedups (we have observed up to 20% improvements for certain applications).
- Dead variable removal allows the high level optimizer to reduce the total memory requirements of the application by removing global and static variables that are never referenced.
- Recognition of global, static and local variables that are assigned but never used allows the optimizer to remove dead code (which may result in additional dead variables).
- Conversion of global variables that are referenced only within a module allows the high level optimizer to convert the symbol to a private symbol, guaranteeing that it can only be accessed from within this module. This gives the low-level optimizer greater freedom in optimizing references to that variable.

- Dead function removal (functions that are never called) and redundant function removal (for example, duplicate template instantiations) help to reduce compile time and improve the effectiveness of cross module inlining by reducing the working set. Additionally, as the application's total code size reduces, it will incur fewer cache and page misses (resulting in potentially higher performance).
- Short data optimizations. Global and static data allocated in the short data area can be accessed with a more efficient access sequence. In whole program mode (`-ipo`) the compiler can perform precise analysis to determine if all global and static data fits into the short data area and allocate it there. If the data doesn't fit, the compiler can determine the best safe short data size threshold, enabling a maximum amount of data items to be addressable more effectively.

This is an advantage over `+O2` alone (without `-ipo`). At optimization level `+O2` the same optimization can be enabled with the option `+Oshortdata`, `+Oshortdata=<threshold>`. However, this method is typically not adaptive to application change and evolution.

- For calls to external functions (function not residing in a binary) the linker introduces a small call stub. If the compiler knows that a function call is a call to an external function, it can inline the call stub, resulting in better performance.

The HP compilers support a mechanism that allows annotating function prototypes with a pragma (`#pragma extern`) marking those functions as external functions. When used with the compiler option `-minshared` (see "Choosing the link mode" on page 30), the compiler can perform call stub inlining.

All this is no longer necessary with `-ipo` in whole program mode. In this model the compiler knows which functions are defined by the application and which are external and automatically marks functions appropriately.

- Interprocedural constant propagation enables more efficient code.
- Data layout optimizations, including structure splitting and dead field removal, can help reduce the working set of an application and thereby improve data cache behavior. In its framework for interprocedural data layout optimizations, if the compiler is able to determine that a given structure type can be modified safely, the compiler may split a structure type into hot and cold parts, with the goal of reducing cache and TLB penalties. This optimization has been greatly improved in the current compiler and handles many additional cases; the `+Oinfo` option can be used to determine whether this optimization has been performed.

Although full interprocedural optimizations are only available in the presence of `-ipo` or `+O4`, the compiler performs "lightweight" interprocedural optimization at `+O2` and above. This phase can improve performance of applications with frequent use of static variables and functions. Lightweight IPO enables the following optimizations:

- Dead function elimination for static routines.
- Dead variable elimination for static variables.
- Address exposure analysis for static variables.

The interprocedural analysis phase is also able to expose and warn on additional source problems, for example, for variables that are declared with incompatible attributes in different source files.

The HP compilers deliver many high-level math functions as high-performance implementations in intermediate form (as IELF object files), allowing the compiler to inline these functions. This results in the usual benefits gained from inlining, such as avoidance of call overhead or broadening of scope, and can help the instruction scheduler hide load and store latencies.

The high level loop optimizer has been significantly improved, and performs the following classic loop optimizations based on array access patterns (the loop optimizer is enabled at optimization levels two (+O2) and higher):

- Loop interchange
- Loop distribution
- Loop fusion
- Loop unrolling
- Loop unswitching
- Loop cloning
- Parallelization
- Loop blocking
- Loop unroll-and-jam
- Scalar replacement
- Recognition of memset/memcpy type loops
- Loop rerolling

These optimizations are designed to improve locality of array access, improving the utilization of the data cache. Parallelization distributes the work of a loop body among available processors.

The loop optimizer also performs some new optimizations:

- Automatic parallelization. This optimization allows applications to exploit otherwise idle resources on multicore or multiprocessor systems by automatically transforming serial loops into multithreaded parallel code. When the +Oautopar option is used at optimization levels three (+O3) and above, the compiler automatically parallelizes those loops that are deemed safe and profitable by the loop transformer. With +Oautopar, the parallelized application will utilize all the processors or the number of desired processors indicated by the environment variable OMP_NUM_THREADS. The default is +Onoautopar, which disables automatic parallelization of loops. Automatic parallelization can be combined with manual parallelization through the use of OpenMP directives and the +Oopenmp option. When both +Oopenmp and +Oautopar are specified, then any existing OpenMP directives take precedence, and the compiler will only consider auto-parallelizing other loops that are not controlled by those directives.

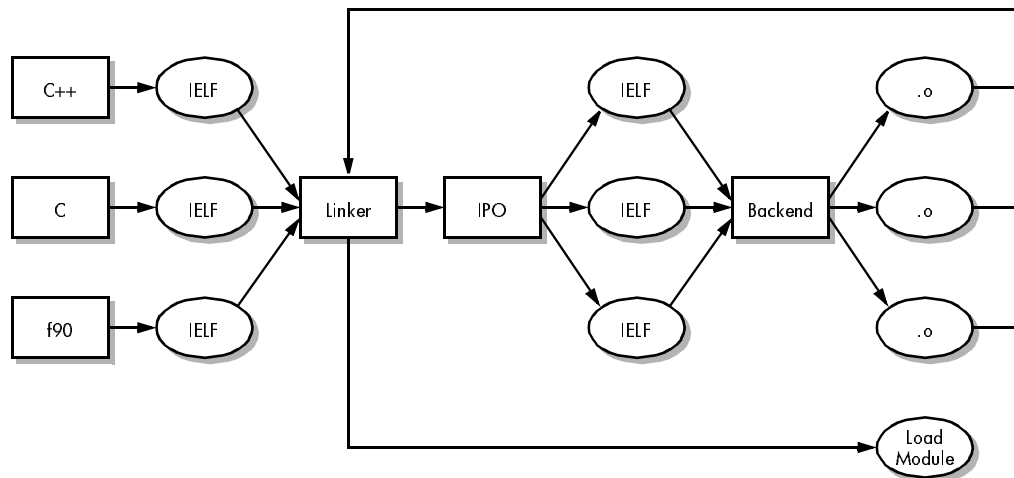


Figure 2: Build model for interprocedural optimization

- Loop multiversioning. Some loops can be optimized more aggressively by assuming certain conditions, all of which may not be known at compile time. The loop optimizer can clone these loops, introduce some runtime checks and optimize the cloned loops more aggressively. At the executable runtime, the assumed conditions are checked and the correct loop is executed.
- malloc combining. The optimizer can combine several small block allocations into a single large block allocation. This improves locality and reduces the cost of calling the allocation routine.

The high level scalar optimizer performs expression simplification and canonicalization, SSA-based dead code removal, copy propagation, constant propagation, and register promotion, as well as control flow optimizations and basic block cloning.

The interprocedural optimization framework (enabled with `-ipo` at optimization level `+O2` or higher) has been designed to scale to very large applications.

Fortunately, nothing changes from a user's perspective; in particular, existing build processes do not have to be modified. Since the IPO and code generation are performed at link time, the link time may increase significantly.

The internal build model differs slightly from the default build model and is illustrated in Figure 2 (simplified for clarity).

The object files generated with `-ipo` contain an intermediate representation of the user code; these object files are called IELF files. IELF files have been designed for fast access.

Compared to regular object files containing debug information (option `-g`), IELF files are typically larger by a factor of 3x. Compared to object files containing no debug information, IELF files can be significantly larger, as they have to include, for example, complete type information. This can be a strain on file systems for very large applications. The utility `elfdump` allows determining whether a given object file is an IELF file (generated with `-ipo`) or a real object file: the option `-f`, which displays the ELF file header, will report a file type of "HP_IFILE" for IELF files).

IELF files are not guaranteed to be compatible from one compiler release to the next. If your application attempts to make use of old IELF files, a full recompilation of your application may be necessary after a compiler update.

IELF files are consumed by the interprocedural analysis and optimization phase which as result generates a set of final temporary IELF files containing the transformation results. Great care has been taken to minimize the amount of core memory needed during IPO and to ensure that the fastest algorithms are chosen (see Reference 11 and Reference 12).

The IPO phase also generates a temporary Makefile containing targets for translating the temporary IELF files into real object files. This translation is done with a standalone backend called `be`, which contains the code generator and the low-level optimizer (in addition to the high level optimizer). The IPO phase executes `make` on the generated Makefile in parallel mode, generating final object files required for the link of the application. This mechanism is transparent to the user.

The default number of parallel `be` processes is set to the number of processors on a machine. This number can be overridden by setting the environment variable `PARALLEL` (see the man pages for `make` for more details).

This parallelization speeds up the time spent in code generation and low-level optimization greatly on machines with multiple processors. For several serial build processes (no parallel `make` for the frontend parts) `+O4` has been observed to be faster than `+O2`. However, in general, we would expect `+O4` to be no worse than 2x slower than `+O2` (depending on the application's build mechanics and the build machines).

Precise floating-point control

HP compilers are designed to provide complete developer access to the uniquely powerful floating-point features of the architecture. These features enable HP compiler-generated floating-point code and the math library to be both highly accurate and well optimized under default and general compiler options.

Under the `-fpwidetypes` option, the C and C++ math library headers define names for functions and macros involving the 80-bit floating-point type, and they define alternative names for functions and macros involving the 128-bit long double type, as shown in Table 1. The names `extended` and `quad`, which are industry convention, facilitate portability by providing type names that are not dependent on the particular format of the `long double` type, which differs in implementation among different operating systems.

Table 1 Floating-point type suffixes and macros

Type	Name	Function Suffix	Macros
<code>__float80</code> (80-bit-type)	<code>extended</code> (defined in <code>math.h</code> , <code>float.h</code> , <code>complex.h</code> , and <code>stdlib.h</code>)	<code>w</code> (e.g., <code>logw</code>)	<code>W</code> (e.g., <code>HUGE_VALW</code>) <code>EXT_</code> (e.g., <code>EXT_MAX</code>)
<code>long double</code> or <code>__float128</code> (128-bit type)	<code>quad</code> (defined in <code>math.h</code> , <code>float.h</code> , <code>complex.h</code> , and <code>stdlib.h</code>)	<code>q</code> (e.g., <code>logq</code>)	<code>Q</code> (e.g., <code>HUGE_VALQ</code>) <code>QUAD_</code> (e.g., <code>QUAD_MAX</code>)

HP C and C++ compilers provide a choice of three binary floating-point evaluation methods, indicated by the `-fpeval={float|double|extended}` option. The option `-fpeval=extended` can facilitate importing programs from IA-32 platforms that employ wide range and/or precision, and can generally improve robustness where narrower evaluation would be sensitive to rounding error, overflow, or underflow.

- `float`, the default, selects the C99-specified evaluation method which evaluates binary floating operations and constants to their semantic type.
- `double` selects the C99-specified evaluation method which evaluates float operations and constants to the wider range and precision of double and other operations and constants to their semantic types.
- `extended` selects a method that evaluates float and double operations and constants to the wider range and precision of extended, and other operations and constants to their semantic types.

In addition, the HP C compiler provides a choice of three decimal floating-point evaluation methods, indicated by the `-fpevaldec={_Decimal32|_Decimal64|_Decimal128}` option, analogous to their binary floating-point counterparts.

- `_Decimal32`, the default, evaluates decimal floating operations and constants to their semantic type.
- `_Decimal64` evaluates `_Decimal32` operations and constants to the wider range and precision of `_Decimal64`, and other operations and constants to their semantic type.
- `_Decimal128` selects a method that evaluates `_Decimal32` and `_Decimal64` operations and constants to the wider range and precision of `_Decimal128`, and other operations and constants to their semantic types.

Several HP compiler options allow the developer to control the accuracy of floating-point computation and the treatment of special values.

- `+Ofltacc={strict|default|limited|relaxed}` controls the accuracy of floating-point computations.
 - `strict` disallows contractions, such as Floating Multiply-Add (FMA) synthesis. This can also be expressed using `#pragma STDC FP_CONTRACT OFF` in the source code at the desired scope.
 - `default`, the compiler's default, allows contractions (FMA synthesis) as with the C99 `#pragma STDC FP_CONTRACT ON`, but disallows any other floating-point optimization that might change result values. Contractions are acceptable in most applications, but can break those that depend on operations being rounded to specific range and precision and (rarely) some that do not.
 - `limited` is like `default` except that it also allows floating-point optimizations (such as substitution of `0.0` for `x * 0.0`) that might impact the generation and propagation of infinities, Not A Numbers (NaNs), and the sign of zero.
 - `relaxed` indicates the characteristics of `limited` and allows floating-point optimization (such as reordering of expressions, even if parenthesized) that might change rounding errors. The option `relaxed` allows the compiler to

invoke slightly less accurate math functions to improve performance. The relaxed option now implies `+Ocxlimitedrange`; an explicit `+Onocxlimitedrange` option overrides the implication.

- The `+Osumreduction` option allows the optimization of sum reductions, regardless of the floating-point accuracy. Normally, sum reductions are only optimized under `+Ofltacc=relaxed` for the C/C++ compilers (the Fortran language standard allows them to be optimized by default). This option can be used to allow sum reduction optimization under any setting of the `+Ofltacc` option. It is useful for programs that do not require program ordering of the partial sums involved in the sum reduction, but that require accuracy in other computations. Alternatively, the `+Onosumreduction` option will disallow the sum reduction optimization under any setting of `+Ofltacc`.
- The `+Ocxlimitedrange` option indicates complex multiply, divide, and cabs operations are not required to satisfy C99 infinity properties, and allows extended and long double versions to be more likely to encounter undue over/underflow. This functionality can also be chosen with `#pragma STDC CX_LIMITED_RANGE ON` in the source code at the desired scope, and is implied by `+Ofltacc=relaxed`.
- The `+FPD` option or the library call `fesetflushzero(1)` set the flush-to-zero underflow mode.

Other options provide specialized characteristics of floating-point code and math library functions.

The option `+Ofenvaccess` provides reliable use of `<fenv.h>` functionality to access floating-point control modes and exception flags. This functionality is also activated with the `#pragma STDC FENV_ACCESS ON` in the source code at the desired scope.

The `+Olibmerrno` option provides math functions which set `errno`, and return values documented for HP PA-RISC and Unix 95 where these differ from C99 for IEC 60559 implementations. Alternatively, `+Onolibmerrno`, the default, provides functions that do not set `errno`.

Extensive inline assembly

HP allows users to embed machine-level code within a C or C++ program using inline assembly intrinsics. These intrinsics have an easy-to-use interface defined in `<machine/sys/inline.h>`. When this header file is included, the compiler ensures the correct values and use of inline instruction arguments.

A paper describing the use of the inline assembly intrinsics is available online (see Reference 17).

Application Tuning

Tuning an application is basically an iterative process with just two steps:

1. Determine the hot spots in the application or general performance issues with the application.
2. Optimize the hot spots, attacking performance issues.

HP has developed several new tools for finding hot spots and characterizing application performance on Integrity servers. The following sections describe the use of these tools and other techniques for tuning application performance.

Profiling

HP provides two performance analysis tools:

- **HP Caliper** – provides access to several types of performance data. HP Caliper provides three levels of performance measurements, from application call graphs to instruction-level events. Global measurements report total values for critical performance elements such as cache and TLB misses, branch mispredictions, pipeline stalls, instructions executed, and so on. Global measurements are a quick way to find performance problems. Sampled measurements report the same performance metrics as global, but they are sampled during application runtime and correlated to program locations. Precise measurements provide exact function call counts, function coverage, call graph and basic block arc counts. HP Caliper operates during application runtime and does not require the application be built with any enabling compiler options. (For more information, see the references listed in “Additional Information” on page 38)
- **HP-UX gprof** – shows the hot procedures in the application and which hot paths call those procedures. Using this information, the developer can decide which procedures or sections of the application could most benefit from tuning, either with compiler options and pragmas or algorithmic improvement. The `-G` compiler option is used to instrument an application for `gprof`.

HP Caliper includes two modes of operation (`cgprof` and `scgprof`) that perform the same measurements and reporting as `gprof`, but without the need to compile the source code specially for measurement.

Include header files

Legacy C applications frequently do not include header files for system libraries such as `libc` and `libm`. For example, `<stdio.h>` should be included in any file that calls `printf`, but it frequently is not. The default argument and return types in C allow calls to many C library functions with no explicit prior prototype or declaration, and these calls usually work fine in the traditional 32-bit data model where the default `int` type and pointer types are the same size. Default types do not work for calling most `libm` functions. Some codes include their own declaration instead of the header file for a `libm` function. This is allowed, but will inhibit optimization (including inlining) of the call.

HP has included in its HP-UX system header files a variety of pragmas that allow the HP compiler to statically bind and optimize calls to library functions. The compiler binds library calls at compile-time where it is safe to do so, as identified by a `#pragma BUILTIN` or `#pragma BUILTIN_MILLI` in the system header. Calls to these functions can be optimized by the compiler; the compiler can inline the function, substitute a call to a faster routine, or apply more aggressive optimizations around the call. The system header files also include pragmas that declare system library routines external so that calls to shared system libraries may be optimized. These transformations provide significant performance gains in some applications.

Scheduling for the processor

Different members of the Intel Itanium processor family can have different resource constraints, instruction latencies, and other scheduling criteria. HP compilers allow the developer to optimize the application for a specific member of the processor family, or to create applications which are suitable for any Itanium processor, using the `+DS{blended|itanium|itanium2|montecito|native}` compiler option.

- The default option `+DSblended` specifies code scheduling that runs reasonably well on all implementations.
- The `+DSitanium`, `+DSitanium2`, and `+DSmontecito` options select code optimized for these processors.
- The option `+DSnative` asks the compiler to schedule code optimally for the system type on which compilation is occurring.

HP will add new choices to the `+DS` series of options as new processors are introduced.

Use these options with care. An application compiled for one processor may run sub-optimally on another processor. Code scheduled for the Intel Itanium 2 (`+DSitanium2`) processor may run noticeably slower (5–40%) on an Intel Itanium processor than code compiled with the `+DSitanium` option. The relative performance difference will vary with the application; floating-point intensive codes tend to be more sensitive to the scheduling model than integer codes. The `+DSblended` scheduling model is a hybrid model that attempts to generate code that runs reasonably well on all existing implementations, and it will continue to evolve as new Itanium implementations are released. In the AR0809 compilers, the `+DSblended` model is equivalent to the `+DSitanium2` model.

It might be necessary to recompile applications for a future member of the Intel Itanium processor family in order to obtain optimal performance. Binary compatibility, however, is assured regardless of the choice of scheduling option.

Choosing the link mode

By default, all HP compilers assume the `-dynamic` option. The resulting object file uses dynamic linking and can be included in a shared library. When the object file will be linked into an executable rather than a shared library, the option `-exec` is appropriate. The `-exec` option tells the compiler that all defined global symbols are resolved within the executable itself, usually resulting in faster loads and stores. The option `-minshared` directs the compiler to use archive libraries (when available), rather than shared libraries, to potentially improve performance. It tells the compiler that all symbols will be resolved within the executable itself, except for those symbols declared with the appropriate pragmas in system header files.

Increasing the page size

If your application incurs a high data or instruction TLB miss rate, requesting a larger virtual memory page size for data or instructions can provide an additional performance gain. HP Caliper can tell you if your application is experiencing a high TLB miss rate. You can specify large pages using the linker `+pd` and `+pi` options, or using the `chattr(1)` command. It is often worth testing a wide range of page sizes, as application performance can vary unpredictably.

Describing application characteristics

HP compilers support several options and function attributes that describe the coding style used by the application. These options allow the compiler to make assumptions about the behavior of the application. Following these coding guidelines can be time-consuming but rewarding because these options often yield substantial performance gains.

- The option `+Otype_safety={off|limited|ansi|strong}` describes the type of safety rules used by the code being compiled.
 - `off` is the default. It indicates aliasing can occur freely across types.
 - `limited` specifies an observance of the ANSI aliasing rules with unnamed objects treated for aliasing as though they are of unknown type.
 - `ansi` specifies an observance of ANSI aliasing rules with unnamed objects treated as named objects.
 - `strong` specifies the ANSI aliasing rules, except that accesses through values of character types are not permitted to touch other non-character objects and the compiler assumes field addresses are not taken.
- The option `+Onoptrs_to_globals` declares to the compiler statically-allocated data (including file-scoped globals, file-scoped statics, and function-scoped statics) will not be accessed through pointers. Conversely, the option `+Optrs_to_globals` assumes that statically-allocated data can be accessed through pointers.

- The options `+Oparmsoverlap` and `+Onoparmsoverlap` declare whether or not function parameters may overlap with others. For Fortran, the default is `+Onoparmsoverlap`, which says that arguments will not overlap with one another or with any variables in common blocks. The option `+Oparmsoverlap` allows such overlap. For C and C++, the default is `+Oparmsoverlap`, which allows memory accessed indirectly through a pointer argument to overlap memory accessed indirectly through another pointer argument, or to overlap statically-allocated data. The option `+Onoparmsoverlap` declares that no such overlap will occur. The `__restrict` keyword (or `restrict` in C99 mode) may also be used.
- The `malloc` attribute indicates that the return value of the given function either points to a memory location or is a null pointer, that the returned memory can be pointed to only by the returned pointer (not, e.g., by any global variable), and that no other `malloc` calls can return the same memory location or a pointer to it. This enables the compiler to make more aggressive aliasing assumptions about addresses returned by the given function. For example:

```
void *mymalloc(int i) __attribute__((malloc));
```

Many wrappers around `malloc()` obey these rules.

- The `non_exposing` attribute indicates that the given function does not cause any address it can derive from any of its formal parameters to become visible after a call to the function returns. An address becomes visible if the function returns a value from which it can be *directly* derived, or if the function stores it in a memory location that is visible (can be referenced directly or indirectly) after the call to the function returns. This indicates that the compiler can make more aggressive aliasing assumptions about addresses passed to the given function. For example:

```
void foo(int *pi) __attribute__((non_exposing));
```

Many wrappers around `free()` obey these rules. Many function that have nothing to do with memory allocation also obey these rules.

The compiler alone cannot determine when these guidelines are followed, nor can it diagnose violations of these guidelines. However, in whole program mode, the option `+O[no]ptrs_to_globals` is not necessary any more since the compiler will perform analysis to detect whether statically allocated data are accessed through pointers. In addition, the compiler will detect and warn about wrong uses of `+Onoptrs_to_globals` if whole program mode is specified.

Table 2

General purpose tuning options and when to use them

Optimization Type	When to Try	Options
Include system header files	Legacy C code which does not include headers for all the system library functions called	
Optimizing for the processor	Optimal performance on a specific processor Willingness to support multiple processors	+DSblended +DSitanium +DSitanium2 +DSmontecito +DSnative
Choosing the link mode	Creating an executable rather than a shared library Preference for available archived libraries.	-exec -minshared
Increasing the page size	High data (or instruction) TLB miss rate.	-Wl , +pi -Wl , +pd
Removing recovery code	Non-numerical application which does not rely on signal handling related to memory accesses. PBO data available to guide control speculation.	+O[no]recovery
Describing application characteristics	It is known that application follows specific guidelines.	+O[no]ptrs_to_globals +Otype_safety +O[no]parmoverlap

Tuning with profile-based optimization

Profile-based optimization (PBO) is likely to be worthwhile if:

- The application contains a large number of control-flow branches.
- The application contains a large number of indirect branches (for example, C++ virtual function calls) and the `-ipo` option is used.
- HP Caliper data indicates high branch misprediction rates, high numbers of case statement layout optimization opportunities, or large numbers of `if-convert` opportunities for hot branches.
- Representative input sets are readily available or the available profile data can be translated into PBO options and pragmas.
- HP Caliper data indicates poor data cache performance, and the application contains linked-list traversals and/or large numbers of global or static variables.
- The application contains loops that tend to iterate only a few times.

For integer code, PBO can be expected to achieve a 5–40% improvement in application performance; floating-point code will generally see more modest improvements.

Tuning across program modules

The compiler option `-ipo` requests cross-module optimization (optionally in conjunction with PBO). If HP Caliper data collected after using `-ipo` shows an increase in instruction cache and/or TLB misses, this probably indicates a bit too much inlining was performed. In this case, the option `+inline_level` can be used to limit inlining.

Tuning floating-point numerical code

The performance strategies already mentioned can improve floating-point performance. For example, optimization at `+O2` or with PBO will speed up most floating-point code. Optimization at `+O3` further speeds up some code and can dramatically speed up loop-intensive code. With option `+O3`, the compiler performs additional optimizations such as loop transformations (interchange, fusion, distribution, and so on) and more inlining of math library routines into user code. In particular, if HP Caliper indicates high data cache or TLB miss rates, the optimizations performed at `+O3` can be highly beneficial.

At optimization levels `+O2` and higher, the compiler inlines the more commonly used math functions, including `log`, `exp`, `sin`, and `cos`, provided that the proper header files are included. This can substantially improve performance, particularly for calls in loops, and does not affect function behavior.

The general optimization strategies heretofore can be applied without loss of floating-point quality. Here are some additional suggestions:

- Specific optimizations involving math library functions are done only if the source file includes the math headers, such as `<math.h>`, that declare the function.
- The compiler optimizes even under controls for special floating-point semantics (see “Precise floating-point control” on page 25); however, these controls do restrict optimization and may degrade the performance of code that does not require the

behavior provided. For best performance, use `#pragma STDC FENV_ACCESS ON` in the smallest blocks that enclose the code for which it is needed, rather than using the compile option `+Ofenvaccess` for the entire compilation unit. Similarly use `#pragma STDC FP_CONTRACT OFF` in the smallest sensitive blocks and compile with `+Ofltacc=default`, rather than compiling with `+Ofltacc=strict`.

- `+Olibmerrno` is best used only if the compilation unit requires the math functions to set `errno`. Consider querying exception flags instead of `errno`.
- The `-l:libm.a` option will link in an archive version of `libm` and result in more efficient calling sequences. (Using the `-Wl,-a,archive_shared` option when linking will have a similar effect, but may cause the linker to select other archive libraries where shared libraries may be preferred.)

The following techniques can provide significant performance gains, but can degrade the application's ability to deal with unusual or unexpected inputs. They are best suited to performance-hungry applications that are known to run correctly on systems with a relaxed floating-point model or that can be well tested.

- `+Ofltacc=limited` is appropriate when the application does not depend on a specific treatment of infinities, NaNs, or the sign of zero. This option will not substantially improve performance of most codes.
- `+Ocxlimitedrange` or `#pragma STDC CX_LIMITED_RANGE ON` are appropriate in a C application which uses complex multiply, divide, or a `cabs()` function and the textbook formulas (without special consideration for overflow, underflow, or infinite values) for these operations are acceptable. In HP-UX, the `float` and `double` complex operations are implemented using wider internal range and precision, alleviating problems of premature overflow or underflow.
- `+Ofltacc=relaxed` can provide a performance gain over `+Ofltacc=limited` when the application meets the criteria for `+Ofltacc=limited`, the application is known to run correctly with looser floating-point models, and reproducibility of low order result bits is not essential.
- `+Osumreduction` can provide a performance gain when the application contains sum reductions that do not require strict ordering of their partial sums, but cannot use `+Ofltacc=relaxed`.
- `+FPD` or a call to `fesetflushzero(1)` are suitable when the application is tolerant of zero being delivered in lieu of denormal result values. Flush-to-zero mode can significantly speed up some computations with the `float` type.

The following techniques can provide significant performance gains when algorithms can be redesigned or re-implemented in new code:

- The 80-bit extended type arithmetic is essentially as fast as `float` or `double`. The speed of an extended math function is typically about 0.7 times that of the corresponding `double` function. There may be an overall performance gain if the extra precision and range allow the removal of branches to special code for handling rounding errors and underflow and overflow conditions. In addition, the extra range and precision of the extended type can result in simpler, more robust application code that is easier to maintain. Even the 128-bit long `double` (`quad`) type, whose

functions are typically within 0.25 times as fast as corresponding routines for extended types, can be considered in high performance code where extreme precision is needed locally.

- Replace portions of the implementation with inline assembly.

Allowing optimization flexibility

The compiler options `+Ofast` and `+Ofaster` direct the compiler to use typical collections of aggressive optimization options that are safe for most applications. While the features included in `+Ofast` and `+Ofaster` may evolve from release to release, `+Ofast` currently implies the following:

- `+O2` requests level two optimization.
- `+Onolimit` allows full optimization of large procedures, possibly at the expense of longer compile time.
- `+Ofltacc=relaxed` (see “Precise floating-point control” on page 25).
- `+FPD` enables the flush-to-zero rounding mode on the hardware.
- `+DSnative` directs code scheduling specialized for the type of system on which compilation is taking place (see “Scheduling for the processor” on page 29).
- `-Wl, +pi, 1M` and `-Wl, +pd, 1M` causes the application to utilize 1Mbyte instruction and data virtual memory page sizes, respectively.
- `-Wl, +mergeseg` causes the dynamic loader to merge the data segments of shared libraries loaded at runtime, which allows the kernel to use larger size page table entries. Note that the use of this option increases the size of the Resident Set Size (RSS) and may degrade the performance of short-lived programs.

The option `+Ofaster` is an alias for `+Ofast +O4` and is therefore ideally suited for cross-module optimizations.

Because both `+Ofast` and `+Ofaster` imply `+Ofltacc=relaxed`, they are not alone appropriate for tuning floating-point code that requires more rigorous floating-point behavior. However, they can be made appropriate by taking advantage of the compiler’s general left-to-right option processing. For example, the command-line options `+Ofast +Ofltacc=strict` tell the compiler that the latter `+Ofltacc=strict` overrides the earlier setting of `+Ofltacc=relaxed` imposed by `+Ofast`. Likewise, `+Ofast +FPD` enables the default gradual underflow mode.

Using inline assembly

HP C and C++ inline assembly support allows the user to directly exploit powerful assembly-level instructions that would otherwise be difficult for the compiler to generate from source-level constructs. Inline assembly is implemented as an extension to C/C++. Other than including an additional header file, no other changes are needed to use inline assembly. For certain applications, the use of inline assembly can improve performance or provide access to key functionality above and beyond that which the compiler alone can provide:

- The performance of multimedia applications can be significantly enhanced with inline assembly because the compiler cannot directly generate many of the most beneficial multimedia instructions.
- The HP-UX compilers and libraries make available most architectural floating-point features through standard language features and natural extensions. The 80-bit extended type, the `fma()` functions and the inquiry macros, such as `isinf` and `isunordered`, are all available using standard features and natural extensions. However, writers of low-level floating-point codes will still benefit from judicious use of inline assembly to access architectural features such as the `frcpa` and `frsqta` instructions, the 82-bit registers, and the alternate status fields.

For more information about inline assembly, see Reference 17.

Troubleshooting optimization problems

Occasionally, optimization can expose defects in an application that were hidden when the application was compiled without optimization. Here are some representative examples:

- Expressions that perform pointer arithmetic beyond the boundary of an object are undefined according to the language standards. Use of such non-standard pointer arithmetic to access data can result in failures in 32-bit mode due to the compiler's use of `addp4` (add pointer) instructions. The add pointer instruction computes addresses by adding an offset to a 32-bit pointer, which must point to the same address region as the resulting pointer. If an application uses non-standard pointer arithmetic, however, the compiler might not be able to enforce this condition, resulting in incorrect pointer accesses. Use the `+Ocross_region_addressing` option to prevent this problem when an application cannot be rewritten to avoid such pointer arithmetic.
- The option `+Oinitcheck` will direct the compiler to detect and initialize all uninitialized variables. Application code that contains uninitialized variables can show unexpected behavior after optimization. Without this option, the compiler attempts to detect and warn about uninitialized variables that are definitely uninitialized along all paths. If a variable is uninitialized along only some paths to its use, by default these will not be detected and can result in errors.
- Similarly, the option `+Oparminit` causes the compiler to initialize to zero any unspecified function parameters at call sites. Without using this option, these unspecified parameters can contain NaN tokens from previous computations that will result in incorrect behavior if they are consumed (see "Control speculation" on page 10).
- According to the C and C++ language standards, signed integer arithmetic overflow in user code results in undefined behavior. The compiler makes assumptions that such overflow does not occur during some optimizations. For example, the compiler will remove sign extension operations within loop bodies assuming that integer accumulations within the loop do not overflow. A program that relies on certain behavior for overflowing arithmetic operations may behave differently after optimization. The user can suppress assumptions made by some compiler optimizations regarding the lack of overflow with the `+Ointeger_overflow=conservative` option.

When compiling an application that relies on specific floating-point rounding behavior, `+Ofltacc=strict` is appropriate. By default, the only value-changing optimization the compiler performs is the synthesis of contractions. The resulting value is generally more accurate because it has not been subject to an intermediate rounding. However, some floating-point applications rely on the intermediate rounding for correct results.

A developer can narrow down the scope of a problem in optimized code by linking together subsets of the application's object files compiled at the failing optimization level and at a lower, working optimization level. A binary search on the object files with this method can quickly isolate the object file causing the execution failure. If the object file contains more than one C or C++ routine, another binary search on the routines within the source file using `#pragma OPT_LEVEL n` can usually identify the problematic routine. The `OPT_LEVEL` pragma should always be used at global scope; it reduces the optimization level to the value of the argument and that optimization level remains in effect for the rest of the file or until set again by another pragma. The `+O0=name` option can also be used to turn off optimization for selected functions. For Fortran applications, the `fsplit` tool will split a single source file into multiple single-routine files.

Table 3 Options and pragmas for troubleshooting problems in optimized code.

Option	Purpose
<code>+Oinitcheck</code>	Initializes all potentially-uninitialized variables with zero.
<code>+Oparminit</code>	Initializes to zero any unspecified function parameters at call sites, to avoid NaN values.
<code>+Ofltacc=strict</code>	Prevents all value-changing optimizations, even contractions.
<code>+Ocros_region_addressing</code>	In 32-bit mode, modifies the use of add pointer instructions so that non-standard-conforming pointer arithmetic works properly, incurring some performance cost.
<code>+O0=name[,name]...</code>	Turns off optimization for the named functions.
<code>#pragma OPT_LEVEL n</code>	C/C++ pragma which forces the compiler to compile subsequent code (up to the next <code>opt_level</code> pragma) at the given optimization level. Can only specify an optimization level which is equal or lower than the optimization level from the command-line.
<code>+Ointeger_overflow=conservative</code>	Suppress aggressive assumptions made by the compiler regarding the lack of integer arithmetic overflow in the program.

Additional Information

The HP Developer and Solution Partner Program (DSPP) web site contains useful information, including a number of “webinars” on related topics, such as HP-UX compilers, HP Caliper, and application performance tuning:

<http://www.hp.com/go/dspp>

Additional information about the ANSI C and C++ compilers is available at the following locations:

<http://www.hp.com/go/c>

<http://www.hp.com/go/cpp>

Extensive documentation and technical usage notes on floating-point and math functions, including the new decimal floating-point features, are available at:

<http://www.hp.com/go/fp>

Information about HP Code Advisor is available at:

<http://www.hp.com/go/cadvise>

Extensive documentation and technical usage notes on HP Caliper are available at:

<http://www.hp.com/go/hpcaliper>

Information about HP Wildebeest Debugger (WDB) is available at:

<http://www.hp.com/go/wdb>

References

- [1] Peter Markstein, Itanium processor family and Elementary Functions, Speed and Precision, Hewlett-Packard Professional Books, Prentice Hall, Inc., 2000.
- [2] Pettis, K. and Hansen, R.C., "Profile Guided Code Positioning," Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, SIGPLAN Notices, Vol 25, No. 6, June 1990.
- [3] R. Ju, K. Nomura, U. Mahadevan, and L-C. Wu, "A Unified Compiler Framework for Control and Data Speculation," Proceedings of PACT '00, October 2000, pp. 157-168.
- [4] Robert Hundt, "HP Caliper: A Framework for Performance Analysis Tools," IEEE Concurrency Magazine, Los Alamitos, CA Oct-Dec 2000.
- [5] U. Mahadevan, K. Normura, R. Ju, and R. Hank, "Applying Data Speculation in Modulo Scheduled Loops," Proceedings of PACT '00, October 2000, pp. 169-178.
- [6] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam, "Translating Out of Static Single Assignment Form," Proceedings of Static Analysis Symposium 1999: pp. 194-210.
- [7] V. S. Sastry and Roy D. C. Ju, "A new algorithm for scalar register promotion based on SSA form," Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), pages 15-25, Montreal, Canada, 17-19 June 1998. SIGPLAN Notices 33(5), May 1998.
- [8] David M. Gillies, Roy Dz-Ching Ju, Richard Johnson, and Michael S. Schlansker, "Global Predicate Analysis and Its Application to Register Allocation," Proceedings of MICRO 1996, pp. 114-125.
- [9] IA-64 Software Conventions and Runtime Architecture, <http://download.intel.com/design/Itanium/Downloads/245358.pdf>, 2001.
- [10] C++ ABI Summary, <http://www.codesourcery.com/public/cxx-abi>, 2001.
- [11] Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva Chakrabarti, Luis A. Lozano, Uma Srinivasan, Shin-Ming Liu, "SYZYGY—A Framework for Scalable Cross-Module IPO," International Symposium on Code Generation and Optimization (CGO), 2004.
- [12] Dhruva R. Chakrabarti, Luis A. Lozano, Xinliang D. Li, Robert Hundt, Shin-Ming Liu, "Scalable High Performance Cross-Module Inlining," International Conference on Parallel Architectures and Compilation Techniques (PACT), 2004.
- [13] J. Thomas, "Inlining of Mathematical Functions in HP-UX for Itanium(R) 2," Proceedings of the 2003 CGO, The International Symposium on Code Generation and Optimization, IEEE Computer Society, 2003.
- [14] T. Johnson, N. McIntosh, R. Hundt, "Optimizing Itanium-Based Applications," <http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=c208dd324de02110dd324de02110275d6e10RCRD>, 2004.
- [15] "Performance Tuning with HP-UX Itanium® Compilers" (Webcast), <http://www.presentationselect.com/hp/>, 2004.
- [16] "aC++ Version 6 Features to Improve Developer Productivity," <http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=a408d300f3f02110d300f3f02110275d6e10RCRD>, 2005.

- [17] “Inline assembly for Itanium®-based HP-UX,”
<http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=4308e2f5bde02110e2f5bde02110275d6e10RCRD>, 2005.
- [18] “aC++ standard conformance and compatibility changes,”
<http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=2708d7c682f02110d7c682f02110275d6e10RCRD>, 2004.
- [19] “C++ runtime environments (–AA and –AP) on HP-UX,”
<http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=eb08b3f1eee02110b3f1eee02110275d6e10RCRD>, 2004.
- [20] “ISO/IEC C Technical Report 24732,”
<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1312.pdf>, 2008.
- [21] IEEE Std 754-2008, Standard for Floating-Point Arithmetic, 2008.
- [22] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson,
“Eraser: A Dynamic Data Race Detector for Multithreaded Programs,” ACM Transactions on
Computer Systems, 1997.
- [23] “Debugging with GDB,”
<http://h21007.www2.hp.com/portal/download/files/unprot/devresource/Tools/wdb/doc/gdb58.pdf>, 2008.

Symbols

- #pragma diag_pop 13
- #pragma diag_push 13
- #pragma OPT_LEVEL 15
- +check 14
- +check=bounds 14
- +check=globals 14
- +check=lock 13
- +check=truncate 14
- +inline_level 33
- +macro_debug 14
- +Oautopar 13, 14
- +Olit=all 14
- +Oloop_block 14
- +Oloop_unroll_jam 14
- +Oopenmp 14
- +pathtrace 14
- +w64bit 13
- +Wcontext_limit 14
- +wlock 14
- +Wmacro 14
- +wperfadvice 14
- +Wv 14
- _Decimal128 26
- _Decimal32 26
- _Decimal64 26

Numerics

- 11i Version 3 (11.31) 16
- 128-bit long double type 25
- 32-bit 17
- 64-bit 17
- 80 bit floating-point type 25

A

- ABI 16
- address conflicts 12
- advanced check instruction 12
- algorithms 34
- analysis of incoming source code 7
- analysis, interprocedural 21
- application availability 17
- applications
 - performance hungry 34
- arithmetic and load operations 10
- attributes
 - malloc 31
 - non_exposing 31
- availability of applications 17

B

- basic block cloning 24
- benchmarks
 - SPEC2000 7
- Bhidden 15
- Bhidden_def 15
- branch misprediction penalties 10
- branches, removing 10

C

- C++ ABI 16
- C99 infinity properties 27
- cache utilization 18, 19, 21

Caliper. See HP Caliper.

- chatr 30
- checking for conflicting writes 12
- chk.a 12
- code
 - generator 7
 - options for substantial performance gains 30
 - scheduling options 17
- coding languages 7
- Common Software Conventions and Runtime Architecture 16
- compatibility 16
- compiler options 19
- complex arithmetic 16
- conditional execution, boolean values 9
- conditions, guarding 10, 11
- conflicting writes, checking for 12
- constant propagation 17, 22, 24
- constants, evaluating 26
- contractions, disallowing 26
- control dependence, transforming to data dependence 10
- control dependent execution 9
- control speculation 10
- controlling accuracy of floating point computation 26
- controlling the degree of control speculation 20
- copy elimination 17
- copy propagation 24
- cycles, reducing 11

D

- data dependence, transforming from control dependence 10
- data layout optimizations 22
- data prefetching 18
 - evaluating effectiveness 20
- data speculation 11
 - safety 12
- dead code removal 24
- dead field removal 22
- dead function removal 22
- dead variable removal 21
- debug
 - faster 17
- debugger 7
- debugging 17
- debugging code 15
- debugging code compiled with optimization 15
- decimal floating-point evaluation methods 26
- default optimization 17
- development, faster 17
- disallowing contractions 26
- dM 15
- dynamic linking 30

E

- E 15
- efficient scheduling 18
- elfdump 24
- eliminating misprediction 10
- embedding Assembly in C or C++ 27
- estimating execution frequency 19
- executing code concurrently 10
- explicit parallelism 12
- extending application availability 17
- Extensions for the programming language C to support decimal

floating-point arithmetic 13

F

facilitating portability 25
failures in 32-bit mode 36
features, key 16
fesetflushzero 27, 34
finding hot spots 28
fine-tuning profile data 19
floating operations, evaluating 26
floating point computation, controlling accuracy 26
floating point control 25
floating point optimizations 26
floating-point numerical code, tuning 33
flush-to-zero 27, 34
-fpevaldec 26
fprintf 13
fputs 13
fsplit 37
function arguments 21
function inlining 21
functions
 frequently called 19
 library, optimizing calls to 29
 library, statically binding calls 29
 rarely called 19

G

global code motion 18
GNU 16
gprof 28
guarding conditions 10, 11

H

high miss rate, resolving 30
hot spots, finding 28
HP Caliper 28
HP Code Advisor 13

I

IEEE 754 13, 16
IELF 23, 24
implementation, scheduling 29
improving locality 21
improving runtime performance 10
including header files 29
increasing instruction level parallelism 11
increasing the page size 30
indirect call promotion 21
industry standards 16
inline assembly, using 35
inlining 18, 21
 calls to external functions 22
inlining of math library routines 33
instruction
 level parallelism, increasing 11
 selection 18
instructions
 conditional execution 9
international standards 16
interprocedural analysis 21
interprocedural optimizer 20, 24
ISO/IEC 9899

1999 16

ISO/IEC Technical Report 24732 13

Itanium-based compilers, design structure 7

K

key features 16

L

languages, coding 7
level one optimization 17
level two optimization 18
libraries, math 7
libraries, system 7
link mode 30
linker 7
linking dynamically 30
load and arithmetic operations 10
load store elimination 17
locality, improving 21
loop
 low iteration count 19
 optimization 18, 23
 transformations 33
 unrolling 18

M

math library 7
math library inlining 23
math library specifications 16
maximizing instruction-level parallelism 9
memory references 21

N

NaN 11, 36
 tokens, unconsumed 36

O

open standards 16
operations
 load and arithmetic 10
optimization
 across modules 33
 calls to library functions 29
 debugging, for 17
 default 17
 floating point 26
 interprocedural 20
 level one 17
 level two 18
 loop 18, 23
 short data 22
 strategies 33
 transformations 18
optimizer 7
options
 +DSblended 29
 +DSitanium 29
 +DSitanium2 29
 +DSmontecito 29
 +DSnative 29, 35
 +FPD 27, 34, 35
 +mergeseg 35
 +O0 37

- +Ocross_region_addressing 37
- +Ocxlimitedrange 27, 34
- +Ofast/+Ofaster 35
- +Ofenvaccess 27
- +Ofltacc 26, 34, 35, 37
- +Oinitcheck 36, 37
- +Ointeger_overflow 35, 36, 37
- +Olibcalls 35
- +Olibmerrno 27, 34
- +Onolimit 35
- +Oparminit 36, 37
- +Oparmsoverlap 31
- +Oprofile=collect 19
- +Oprofile=use 19, 20
- +Optrs_to_globals 30
- +Orarely_called 19
- +Osumreduction 27, 34
- +Otype_safety 30
- +pd/+pi 30, 35
- a,archive_shared 34
- dynamic 30
- exec 30
- fpval 26
- fpwidetypes 25
- ipo 20
- minshared 22, 30

options for code scheduling 17

P

- P 15
- parallelism, explicit 12
- partial redundancy elimination 18
- penalties, branch misprediction 10
- performance
 - general tuning options 32
 - redesigning algorithms 34
 - techniques for improving 34
 - using coding guidelines for 30
- performance advantages
 - description 7
- Performance Monitor Unit (PMU) 19
- pipelining 18
- portability 25
- portability of source code 16
- post-increment synthesis 18
- pragmas
 - ESTIMATED_FREQUENCY 19
 - extern 22
 - FREQUENTLY_CALLED 19
 - OPT_LEVEL 37
 - RARELY_CALLED 19
 - STDC CX_LIMITED_RANGE 34
 - STDC FENV_ACCESS 27, 34
 - STDC FP_CONTRACT 34
- precise floating-point control 25
- predicate register 9
- predication 7, 9
- prefetching
 - data 7
 - linked lists 18
- preserving application availability 17
- printf 13
- profile-based optimization (PBO) 18
 - benefits of 33

- protecting your investment 16
- puts 13

R

- rarely called functions 19
- recovery code 11, 12
 - sequence 12
- reducing
 - execution time 9
 - number of branches 21
 - required cycles 11
- redundant function removal 22
- register allocation 17
- register promotion 21, 24
- removing
 - branches 10
 - dead functions 22
 - dead variables 21
 - redundant functions 22
- resource constraints 29
- restricted basic block 17
- rotating registers 18
- routine execution frequency 18

S

- scalar replacement 18
- scheduling efficiently 18
- scheduling implementation 29
- sequence, recovery code 12
- short data optimizations 22
- software pipelining 18
- source code
 - analysis of 7
 - portability 16
 - pragmas 19
- SPEC2000 benchmark 7
- specifications 16
- speculation 7
 - control 10
 - data 11
- standards
 - IEC 60559 16
 - ISO/IEC 14882 16
 - ISO/IEC 1539-1 1997 16
 - ISO/IEC 1990 16
- standards, coding 16
- Static Single Assignment (SSA) 18
- statically binding calls to library functions 29
- strength reduction 18
- stride 18
- structure splitting 22
- sub-expression elimination 17
- substituting profile data 19
- sum reduction 18
- synthesis, post-increment 18
- system libraries 7

T

- tokens, NaT 11
- tools for performance analysis 28
- transforming a control dependency 10
- troubleshooting
 - 32-bit mode 36

- optimization problems 36
- options and pragmas 37
- unconsumed NaT tokens 36
- uninitialized variables 36

Tru64 17

tuning

- cross-module optimization 33
- floating-point numerical code 33
- general purpose options 32
- including header files 29
- Itanium-based applications 28

U

- uninitialized variables 36

UNIX 2003 16

using inline assembly 35

V

value congruent instruction elimination 18

variables

- converting global to local 21
- never used 21
- ordering 19

virtual memory page size 30