Performance Tuning with PA-RISC Compilers

Carl Burch

HP Development Environment Solutions Lab

Interworks 2000

April 10 - 13, 2000

Outline

Build-Process Tuning For Integer Applications

- How much performance can the compilers get for me?
- What's wrong with -O?
- Why do compilers have all those optimization options?
- The option tuning process.
- Profile-Based Optimization.
- Specialized Options
- What if my application stops working?

Topics NOT Addressed

- Source-Code Tuning
- Multiprocessor and Loop-Oriented Application Performance
- System Configuration Issues
- Shared Library Performance
- All Available at: http://devresource.hp.com/devresource/Topics/Optimization/Perf.html

How much performance can the compilers get for me?

Large percentage factors for most applications. Orders-of-magnitude speedups are almost always due to algorithmic or data structures tuning of the source code.

Unfortunately, "your mileage will vary". What works on different applications varies widely, though there are some optimizations that almost always deliver major performance wins.

What's Wrong with -O?

Nothing, really. -O is the industry standard, without a doubt. Occasionally we will be called in to consult on applications that are not optimized and should be, but the norm is that unoptimized applications either are not performance-critical or spend all their time in system or third-party (e.g., database) libraries.

The universal use of -O is also why it's inadequate - your competition is using it, too. If the performance of your application is to be a competitive advantage, it has to beat what your competition has done about performance - and -O is the least they'll have done.

Why do compilers have all those optimization options?

There are quite a few optimization options in your manuals, any of which might be a major performance boost for your application. How do you know which to try? Should you bother?

Given that -O is the industry standard, compiler engineers obviously put every optimization we can into -O. There are four reasons why all those options exist:

- Incorrect results if the users' assertion proves false.
- Slows down the application if it does not have the performance problem this optimization addresses.
- Large compile-time compared to the probable runtime performance gained.
- Maintenance options, to turn off components of -O to work around defects - yours or ours.

The Options Tuning Process

1. Get an Unoptimized Version Working.
2. Freeze the Source.
3. Profile.
4. Identify the Crucial Operations.
5. Select Compiler Options that Address the Operations Crucial to your Application.
6. Rebuild, Test, then Benchmark.
7. Reprofile. If slower, to determine why. If faster, to identify the next bottleneck.
8. Iterate back to Step (4) until the schedule runs out.

The Options Tuning Process - In the Real World

Lets' face it, nobody profiles anything. Try these (each level includes its predecessors):

1. -O
2. The "Safe" options.
3. Link-Time PBO.
4. Real PBO and +Ostatic_prediction.
5. PBO and +O3 +Onolooptransform.
6. PBO and +O4 -Wl,+Oselectivepercent=3.

The “Safe” Options

Optimization Options Safe for ALMOST All Applications

- +DA1.1 +DS2.0a: Optimizes for the fastest PA-RISC chips (PA8x00) while still compatible with the entire line.
- +ESlit +Oentrysched +Ofastaccess +Olibcalls +Oprocelim: Almost always safe.
- +Onolimit: Better runtime at the cost of more compile time.
- +ESfic: Inlines function pointer calls for programs NOT using shared libraries.
- ld +pd size/+pi size: specify a data/instruction page size.
- +Odataprefetch +Osignedpointers: Can produce better performance of loop intensive applications.
- +FPD +Onofltacc: Better performance for floating point applications, at the cost of letting results vary from those obtained without optimization.

Telling the Compiler about the Machine

Architecture Level and Scheduling Models

The +DA option tells the compiler the sets of instructions it may choose from. Code will not run on a machine of a lower architecture level than that specified in the +DA option.

The +DS option specifies the machine the optimizer should assume for scheduling purposes. Any scheduling option's code will run on any PA-RISC machine, though with varying performance.

The default for both the +DA and +DS options is to target the compilation server.

In general, more recent machines care more about the scheduling model than earlier systems. The best combination of options for applications that must run one binary across all PA-RISC machines is the highest +DA option that will run on all the target machines, and the most recent +DS option for any machine. Applications requiring maximal performance should investigate shipping a separate binary for each architecture level supported.

+DAportable

To generate code compatible across PA-RISC 1.1 and 2.0 workstations and servers, use the +DAportable option.

If you specify +DAportable and do not specify +DS, instruction scheduling will currently be for +DS1.1d, the PA7200 processor.

For best performance on all currently supported machines without providing multiple executables, use "+DAportable +DS2.0a" (+DS2.0a schedules for the PA8x00 processors).

The definition of +DAportable will be updated when the lowest supported architecture level changes in the future.

Profile-Based Optimization

The single most effective build process improvement you can make for most integer compute-intensive applications is to use Profile-Based Optimization (PBO). Profile information about which paths were taken (and not) is now used by every code generator and optimizer component, as well as the linker.

The bad news is that PBO is not just a matter of adding some option to the list and recompiling. PBO is a three-phase build-run-build process. The requirement to run the instrumented executable to generate the profile database means it can't be transparent - we don't know how to run your application.

Loop-oriented codes tend to get less advantage from PBO than branchy integer applications. The technical reason for this is that the compiler can often calculate the iteration count for loops, which tend to be the only branches that count - so the profile data doesn't really add any information. That being said, many otherwise loop-oriented codes still profit from PBO.

PBO Process

Conceptually, PBO is a three phase process:

1. Build the instrumented executable from source.
2. Run the instrumented executable to generate the profile database.
3. Rebuild the profile-optimized final executable from the source again, using the profile database.

It is more efficient to do it as a four-phase process:

1. Compile the source files to .o files with the +I option. This makes .o files containing the compilers' intermediate code (ISOMs) instead of PA-RISC machine code (SOMs).
2. Link the instrumented executable with the +I option.
3. Run the instrumented executable to generate the profile database.
4. Link the profile-optimized final executable, using the profile database via the +P option.

PBO Process Example

1. Compile the source files to .o files with the +I option:

   ```
   % cc -c -O +I *.c
   ```

2. Link the instrumented executable with the +I option:

   ```
   % cc -O +I *.o
   ```

3. Run the instrumented executable to build the profile database in the flow.data file:

   ```
   % a.out < input.file > output.file
   % ll flow.data
   -rw-r-----   1 cdb      lang      371571 Apr  4 13:40 flow.data
   ```

4. Link the profile-optimized final executable:

   ```
   % cc -O +P *.o
   ```

PBO: Why Do We Care About Branches?

As computers get faster, deciding where to fetch the next instruction from gets greatly harder. Superscalar processors fetching up to four instructions at a time like the PA8x00 see a branch instruction on almost every fetch. If the fetcher guesses wrong about whether the branch will be taken (or to where), the penalty is flushing all the instructions in the pipeline behind the mispredicted branch.

PA8x00 chips may have to flush up to 24 instructions worth of work when a conditional branch is mispredicted. The same number for PA7200 processors is 4 instructions.

What's the antidote to mispredicting branches? Profile-Based Optimization. With PBO the compiler can lay out code to fall through most branches, or eliminate them altogether.

Branch Prediction on PA8x00 Processors

Mispredicted branches are so costly that the PA8x00 processors provide two methods of predicting their behavior, dynamic and static.

- The default is dynamic prediction, where the PA8x00 hardware tracks the behavior of recent branches and uses that history to predict their current action.
- For large applications with too many branches too seldom executed for the small hardware history buffers to be effective, PBO can be amplified by the +Ostatic_prediction option. It uses the history of a branch over the PBO instrumented run to encode predictions into the branch instructions themselves.
- PA8500 and later processors use “agrees mode”, combining the advantages of both static and dynamic prediction.

PBO/linker Interactions

PBO can be used even for products with sophisticated link processes:

- Mixtures of ISOMs and SOMs.
- Shared Libraries.
- Archive Libraries.

See the “HP-UX Linker and Libraries User's Guide” at http://docs.hp.com/dynaweb/hpux11/dtdcen1a/lnkuen1a/@Generic__BookView for details.

Getting Started With PBO - Link-Time PBO

There is a form of PBO that only involves the linker. Actually, most of the "Linker and Libraries User Guide" Chapter 8 PBO section is written from that point of view. It builds the ISOM/ basic-block-level PBO on top of the linker PBO architecture.

- New adopters of PBO can debug your PBO process using SOMs without the long "link" times to compile ISOMs.

The target optimization of link-time-only PBO is procedure positioning guided by the call-graph profile.

- Does nothing inside procedures.
- On 11.x (where we have large page support), it primarily serves as an i-cache optimization for direct-map caches.
- Used to get occasional home runs (~15% or more) by TLB miss reductions, but large pages do a better job at that.
- Procedure positioning also reduces dynamically taken long-call stubs.

What Do +O3 and +O4 Do?

Above -O, the High-Level Optimizer (HLO) is invoked to optimize compilation units or entire programs across procedure boundaries.

The main interprocedural optimizations done by the HLO are inlining and cloning. If the code for a call site can be profitably replaced by a copy of the callee's code, the call will be inlined. If the call cannot be profitably inlined, it may be cloned if one or more of its arguments are constants - the call redirected to a local copy of the cloned function, with the constant argument replaced by the constant. Both inlining and cloning are greatly more effective when guided by Profile Based Optimization.

At +O3, the HLO looks at all the calls within a compilation unit (the source files that result in a single .o file).

At +O4, the HLO's scope is all the .o files that were compiled with the +O4 option. +O4 code is not optimized until link time, when ld invokes a background process to compile the +O4 .o files.

Selective +O4

The combination of PBO and +O4 is the most powerful of all build processes. Giving the High-Level Optimizer a profile to guide its inliner instead of using heuristics is one of the most effective ways known of saving procedure call overhead.

Normally at +O4, the HLO's scope is all the .o files that were compiled with +O4. For large products, this can imply the use of very large amounts of virtual memory - in some cases, far more than any plausible build machine.

In this case, we recommend the use of a new linker option that pre-selects the ISOMs to be presented to the HLO according to the call-arc profile in the flow.data file. The remaining ISOMs (those not involved in heavily-executed calls) are compiled at +O2.

ld +Oselectivepercent <N>

N is a variable parameter, with values about 1 to 5. A value of 2 would mean that the ISOMs containing either the calling or called functions from the top two percent of the profiled call-arcs would be optimized at +O4.

Within the set of ISOMs presented to the HLO, the HLO uses the profile (including the basic-block profile inside functions) to further refine the functions and call sites to be inlined or cloned.

+Oselectivepercent is a linker option, not a compiler driver option. It amplifies -P, which must also appear on the linker command line.

Selection of Advanced Optimization Options

What your application is NOT doing can be as profitable as profiling what it is doing. The following options may not be correct for programs that violate their assumptions.

+ESlit: Though named for its use in Embedded Systems, +ESlit is also a data-cache optimization for programs that do not write on strings or const data. The d-cache optimization results from a smaller total amount of data (collapsing redundant strings to one copy), and the segregation of the literal data from the read-write data (typically more heavily accessed) which leads to more synergistic data cache line fetching.

+ESfic: Embedded Systems Fast Indirect Calls. For executables built without shared libraries. This option is incorrect if any shared libraries are referenced, which aborts by jumping into data space and trying to execute data as code.

Options Selection (continued)

+Osignedpointers: For programs that do not manipulate pointers into shared memory or mapped files, +Osignedpointers can greatly accelerate pointer-manipulation loops.

+Odataprefetch: Can produce better performance of loop intensive applications with many data-cache misses.

+Oentrysched: If you do not use elaborate stack unwind, mixing the entry/exit code of each procedure with the user code reduces the time to call the function, and is safe. The sole dependence on unscheduled entry and exit code is "context restoration", an Ada-style unwinding of the stack up to a parent caller and restarting from the return point of the call. Printing of stack tracebacks and setjmp/longjmp use are unaffected by +Oentrysched.

+Olibcalls: In C and C++ it is almost always correct to use +Olibcalls, and faster than the C library versions - if you don't provide your own versions of the system functions replaced.

Linker-Implemented Optimization Options

+Oprocelim: This option is included as the default with +O4, though there is some danger that it could eliminate a function the user intended to call from within a debugger or via a future shl_findsym(3) call. Such cases can be corrected by adding to the link command a -u option for the symbol you wish to retain.

+Ofastaccess: Programs that do not depend on the layout of data items declared independently of each other will not change results under +Ofastaccess. Since the PA-RISC linker by default lays out randomly the uninitialized data that +Ofastaccess sorts into order of data item size, the risk for most applications is very small. Like +Oprocelim, +Ofastaccess is the default with +O4.

Improving TLB Hit Rates

(From the 11.0 HP-UX Linker and Libraries User's Guide)

To improve Translation Lookaside Buffer (TLB) hit rates in an application running on a PA 8000-based system, use the following linker or chatr virtual memory page setting options:

- +pd size -- requests a specified data page size of 4K bytes, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, or L. Use L to specify the largest page size available. The actual page size may vary if the requested size can not be fulfilled.
- +pi size -- requests a specified instruction page size. (See +pd size for size values.). The default data and instruction page size is 4K bytes on PA-RISC systems.

Improving TLB Hit Rates (continued)

The PA-RISC 2.0 architecture supports multiple page sizes, from 4K bytes to 64M bytes, in multiples of four. This enables large contiguous regions to be mapped into a single TLB entry. For example, if a contiguous 4MB of memory is actively used, 1000 TLB entries are created if the page size is 4K bytes, but only 64 TLB entries are created if the page size is 64K bytes.

Examples:

- To set the virtual memory page size by using the linker:

  ld +pd 64K +pi 16K /opt/langtools/lib/crt0.o myprog.o -lc

- To set the page size by using chatr:

  chatr +pd 64K +pi 16K a.out

See also "Performance Optimized Page Sizing":

- http://www.unixsolutions.hp.com/products/hpux/pop.html

Floating-Point Options

+Onofltacc: If your application is floating-point intensive but not numerically unstable, this option may give you significant performance gains for relatively little increase in round-off error.

+FPD: For applications that frequently manipulate floating-point numbers falling in the IEEE denormalized ranges, the PA 7100 and later chips support a fast underflow mode allowing denormalized operands to be flushed to zero. This problem is more common for single-precision data than double, as single needs only be closer to zero than 10-23 to be denormalized. Fast underflow mode can improve performance substantially for applications that underflow frequently, because handling denormalized operands by trapping to the operating system is time consuming. The linker +FPD option sets up the hardware's fast underflow mode at link time. This option is also accepted by the cc(1) driver and passed to ld.**

What If My Application Stops Working?

If an application operates correctly when unoptimized but malfunctions when optimized, it is natural to suspect a defect in the optimizer. There are several other possibilities you should be aware of:

The application does not adhere to the language standards, or makes assumptions not guaranteed by the standards.

The application violates the compiler's assumptions; either the source code violates the default assumptions or a compiler directive is used inaccurately.

The application references uninitialized variables during execution. With optimization, variables are less likely to be accidentally initialized with a zero value.

What If My Application Stops Working? (cont.)

The application uses data that can be modified asynchronously without communicating this to the compiler. Invalid optimizations may be performed on variables residing in shared memory areas if such regions are not made known to the compiler.

Physical dependencies may change with optimization. An application relying on timing dependencies, for example, may break due to an idle wait loop taking less time per iteration. Optimization will change memory access patterns, which may result in different page fault or cache miss effects.

Lastly, the source of an optimization problem may indeed be a genuine compiler defect.

Use of Uninitialized Variables

Using a variable before it is assigned a value can be a potential source of optimizer problems. This is particularly true if the offending variable is promoted to a register by the optimizer.

The optimizer can detect many kinds of references to definitely uninitialized local variables. In such cases a message is issued:

cc: line 4: warning 5004: Uninitialized variable "*xxx*" in function "*yyy*"

and the variable is preset to zero at procedure entry. It should be noted that such a warning is issued only if there is absolutely no definition of the variable along any path from the entry point of the routine to the use of the variable. If the variable is defined conditionally before being used (e.g. in a THEN clause but not in the ELSE clause), this warning will not be issued even though there is a potential for an optimization problem. Such situations will be reported if the +Oinitcheck option is used.

Use of Uninitialized Variables (continued)

Array, structure and pointer dereferences which may be picking up uninitialized values will in most cases not be flagged (however, these are also less likely to result in optimization-specific problems). In other words, absence of this warning does NOT imply that all is well.

This warning mechanism can be used to advantage for porting applications from other systems. Even without optimization, uninitialized variables can cause an application to fail. Turning on optimization will not cause the application to work, but it may point out instances where uninitialized variables are being used, facilitating their identification and correction.

Use of Shared Memory

If the application uses shared memory for inter-process communication without declaring it *volatile*, there is potential for an optimization problem. Depending on the circumstances, this problem can manifest itself in several ways.

In C and C++ care must be taken to indicate to the compiler which variables can be accessed asynchronously.

Use of shared memory also invalidates the +Osignedpointers option.

+O[no]fail_safe

The +Ofail_safe option allows compilations with internal optimization errors to continue by issuing a warning message and restarting the compilation at +O0.

You can use +Onofail_safe at optimization levels 1, 2, 3, or 4 when you want internal optimization errors to abort your build. The default is +Ofail_safe.

This option is disabled when compiling for parallelization.

What To Include On A Defect Report

If you're sure your problem is a compiler defect, the next question is what information the lab needs to fix the problem. The one promise I can make is that we very seldom fix defects we haven't seen - you control whether the defect is fixed.

If the compiler aborts or "hangs" while optimizing an application, the lab will have a much easier time tracking down the problem if the source code is available. The same is true when the application itself aborts, and attempts to narrow down the problem are unsuccessful. The version number of the compiler and operating system being used should be included.

If providing the source code has significant security issues, it is also possible to recompile with the +I option to create object files containing intermediate code (ISOMs). These files will suffice to allow the reproduction of the defect, but are no more reverse-engineerable than regular SOM object files.

What To Include On A Defect Report (cont.)

When an incorrect code sequence is suspected, the defect report should ideally include the shortest possible code sequence in which the problem can be demonstrated. A description of how the code was compiled, what options and directives were used, and the version number of the compiler should also be included.

In the cases where an optimized application is producing unexpected results, some effort should be taken to isolate the problem. Someone who knows how the application works is more likely to find the problem area than someone starting from scratch. If efforts to isolate the problem are unsuccessful, the source code should be submitted along with any necessary include files, data files, and instructions describing how to compile, link, and run the application.

Optimization: A Toolbox, Not a Push-Button

Application Performance can be a competitive advantage, and the PA-RISC Optimizer can help you win that advantage.

Like other toolboxes, it has many tools - and a few sharp edges.

HP has used the Optimizer to our competitive advantage. You can too, but your competition's Makefiles already have -O. Or maybe they're sitting next to you today ...

More Information ...

- The Optimization White Paper. On your system at /opt/langtools/newconfig/white_papers/optimize.ps.
- “Optimization for a Superscalar Out-of-Order Machine”, Anne Holler, Proceedings of the 29th Annual International Symposium on Microarchitecture, 1996. Also at: http://devresource.hp.com/STK/partner/pcxu_opt_compcon.pdf
- “Advanced Performance Features of the 64-bit PA-8000”, Doug Hunt, Proceedings of the Spring 1995 COMPCON.
- “PA-8500: The Continuing Evolution of the PA-8000 Family”, Greg Lesarte and Doug Hunt, Proceedings of the Spring 1997 COMPCON.
- “HP-UX Linker and Libraries User's Guide”, HP Part Number: B2355-90654.