



SunNetwork
2003 Conference and Pavillon

Writing fast apps on Solaris

Bart Smaalders

Senior Staff Engineer

Solaris Kernel Group



Writing fast apps on Solaris

- Introduction
 - Who I am.
 - What this talk is about:
 - Measuring performance & observing code behavior
 - Modifying algorithms & data structures
 - Exploiting the compiler & linker
 - Optimizing at run time.
 - “Correctness is a constraint, performance is a goal. Of course, if you can't be right, you might as well be fast.”

Writing fast apps on Solaris

- Measuring application performance
 - Selection of benchmark(s) critical to success.
 - Mind the 3 Rs: Repeatable, Realistic, Runnable
 - Overall metric should be understandable by VP
 - The easier & faster it is to run, the more likely it is that performance work will happen.
 - Results should be comparable between different application versions – and it's even better if your results can be compared to your competitor's results.
 - “If you can't measure it, you can't improve it.”
 - “That which is measured, improves.”

Writing fast apps on Solaris

- Measuring application performance
 - Performance work is essentially iterative and must be a core part of development – it can't be “sprayed on” afterwards.
 - Root causing performance problems can require a lot of investigation - hence observability is very important to success.
 - Distillation of complex benchmarks into a set of microbenchmarks can be critical to focused performance improvement.
 - GUI apps require a automated method of driving application through benchmark.

Writing fast apps on Solaris

- Measuring application performance
 - Tools for initial performance observations/analysis
 - Ptime: accurate real/user/system process timings using microstate accounting.
 - Truss: lots of data on system calls, faults. Particularly useful features include:
 - truss -cf to follow forks and count system calls.
 - truss -u to investigate calls to and from shared libraries.
 - truss -d to time stamp calls.
 - truss -m fltpage to trace page faults

Writing fast apps on Solaris

- Measuring application performance
 - Tools for initial performance observations/analysis(cont.):
 - Pmap: investigate the address space and working set of a process. Take a look at `pmap -x`.
 - Gprof, prof – traditional simple tools, require recompilation.
 - [ld.so.1](#)'s LD_PROFILE – allows you to generate gprof data for any shared library. See Linker and Libraries Guide.
 - Forte Performance Collector/Analyzer – powerful integrated GUI tool. Has performance counter support. Available via Try & Buy...

Writing fast apps on Solaris

- Measuring application performance
 - Tools for more detailed examination:
 - Cputrack (UltraSPARC & P3 (P4 soon)): program and read the performance counters.
 - Cpustat: system wide CPU performance counter measurements.
 - Trapstat (SPARC): measure count, impact of traps.
 - Helps find TLB problems
 - Track US-III floating point denorm traps.
 - Forte Collector/Analyzer

Writing fast apps on Solaris

- Changing the code
 - Algorithms: many severe performance issues are algorithmic problems – find and fix these first, preferably during the design process.
 - Correctly sized & hashed hash tables are $O(1)$
 - AVL trees are $O(\log N)$
 - Badly configured hash tables are $O(N)$
 - Design for and use prefetch – overlap memory wait time with execution.
 - “The only way to go faster (usually) is to do less work”.

Writing fast apps on Solaris

- Changing the code
 - Work for increased data locality
 - Reduce D\$, E\$ misses
 - Reduce DTLB trap/table walk rates
 - Watch out for:
 - Oft-traversed linked lists of randomly allocated items
 - Excessive indirection
 - Consider:
 - Can I block the data a cache line or a page at a time?
 - Can this data structured be imbedded rather than referenced?
 - Can I coalesce 3 malloc calls into 1?
 - Can I use prefetching?

Writing fast apps on Solaris

- Changing the code
 - Work towards improved code locality:
 - Reduce I\$, E\$ misses/stalls.
 - Reduce ITLB traps.
 - Watch out for:
 - small hot functions spread over many shared libraries.
 - Large complex functions with many seldom taken branches.
 - Many repeated sequences of calls to manipulate each item in a data set.
 - Consider:
 - Can I process data as a set? Combine/inline functions?
 - Can I use either profile-directed optimization or `#pragma rarely_called` to move unused basic blocks out of line?

Writing fast apps on Solaris

- Changing the code
 - Eliminating unnecessary stack frames (SPARC)
 - Reduce register window spill/fill traps.
 - Watch out for:
 - Recursion
 - Lots of tiny nested function calls
 - Consider:
 - Iterating rather than using recursion
 - Using tail calls
 - Avoiding unnecessary layering

Writing fast apps on Solaris

- Changing the code
 - Avoiding the Global Offset Table tax.
 - Watch out for references to global data in small PIC (shared library) functions.
 - Consider using TLS or other context to point to need information, or make data static to library.
 - Very important in small “accessor” functions.

Writing fast apps on Solaris

- Changing the code
 - Avoid false sharing
 - Important for multi-threaded or shared memory multi-process workloads.
 - Watch out for:
 - Arrays of locks or statistics.
 - Small malloc'd areas frequently updated from multiple threads.
 - Fields in structures that are written from multiple threads.
 - Consider:
 - Allocating cache line sized (64 bytes) areas.
 - Grouping structure members by use.

Writing fast apps on Solaris

- Changing the code
 - Use large pages to reduce DTLB & DTSLB misses (detect this problem via `trapstat(1m)`).
 - `Memcntl(2)` allows selection of large pages for portions of application address space.
 - Most often useful for heap.
 - Doesn't work on text or x86(yet).
 - Use `madvise(3C)` to advise kernel of access patterns.
 - Use `madvise(3C)` to advise kernel of locality information (MPO).

Writing fast apps on Solaris

- Compiling for performance
 - Use higher levels of optimization on heavily used functions.
 - Watch out for space/time tradeoffs – unrolling seldom-called code is a waste of I\$, ITLB space.
 - Consider using architecture-specific flags (SPARC).
 - Use pragmas to help compiler: `no_side_effect`, `nomemorydepend`, `does_not_read_global_data`, etc.
 - Read the compiler documentation!

Writing fast apps on Solaris

- Optimizing using the linker
 - The Linkers and Libraries Guide is excellent.
 - Avoiding excessive numbers of shared libraries
 - Reduce TLB pressure, calls through PLTs.
 - Reduce startup time (symbol lookup overhead)
 - Consider limiting shared libraries to FRUs.
 - Make sure shared libraries you build use PIC code; the `-ztext` flag to `ld` insures this.

Writing fast apps on Solaris

- Optimizing using the linker
 - Getting rid of LD_LIBRARY_PATH via \$ORIGIN and ld -c.

```
cc -o myapp ... -R$ORIGIN/../../lib -lmylib -c  
$ORIGIN/../../etc/ld.config
```
 - Using processor-specific libraries: use the \$PLATFORM token to expand library paths to match processor type:

```
cc -o myapp ... -R$ORIGIN/$PLATFORM/lib -lmylib
```

Writing fast apps on Solaris

- Optimizing using the linker
 - Reduce symbol scope [symbolic, direct, local] via mapfiles.
 - Local symbols are library private.
 - Symbolic bound symbols are visible externally, but the library itself treats them as if they were local. Disables interposition.
 - Direct binding causes the linker to record in which library the symbol is to be found. Disables interposition unless the interposition library is compiled with `-zinterpose`.
 - Some care needed with C++; better tools on the way.

Writing fast apps on Solaris

- Optimizing using the linker
 - Reorder text and data via mapfiles
 - Text reordering at function level when compiling with -xF. Order can be developed using Forte Analyzer, or done using ad-hoc tools. More work to be done here.
 - Data segments can be reordered when code is compiled with latest Forte compilers. Colocation of read-only items, items written to frequently, etc. can result in significant reductions in
 - Working set
 - Page faults
 - More tool development needed here.

Writing fast apps on Solaris

- Optimizing at Run time
 - Use ufs logging – largely eliminate fsck, increase speed of meta-data updates.
Example: file creates are 7x-20x faster.
 - Use ppgsz get to use large pages and reduce DTLB misses.
 - Minimize startup times with crle.
 - Decrease contention for resources with processor sets.

Writing fast apps on Solaris

- Conclusion
 - Maximum performance is obtained when all aspects of application development are optimized: design, implementation, compilation, linking and run time.
 - Many tools and utilities exist in Solaris to observe and improve performance.
 - A copy of this presentation, more information and examples of use of these tools and techniques can be found here:
<http://playground.sun.com/~barts/fastapps.html>



SunNetwork
2003 Conference and Pavillon

Writing Fast Applications on Solaris

Bart Smaalders

bart.smaalders@sun.com

