IA-64 and SSL Performance

Dave Holt
Mark Mayotte
Hewlett-Packard
3404 E Harmony Road
Fort Collins, Colorado 80528
(970) 898-4141
mark_mayotte@hp.com

Abstract

SSL operations underlie most financial and private transactions on the Web. SSL server performance is critical, since a single SSL transaction can take as many server resources as hundreds or thousands of normal requests. HP's Itanium Processor Family (IPF – AKA IA-64) offerings bring unprecedented performance to this critical piece of the Internet infrastructure.

This paper presents an overview of the generic software/system tuning process and its importance to obtaining the best performance on IPF systems. It also provides more background information with a review of the SSL protocol at a high level, with attention to performance aspects. The most aggressive tuning approach is explored through the development, leveraging existing open-source code, and application of a simple benchmark methodology for measuring SSL performance. This benchmark was used to drive significantly improved SSL performance on HP's new IPF platform. Comparative results on other systems have also been collected and are presented to place the leadership results obtained on HP's IPF systems in context.

1 IPF’s Sensitivity to Tuning

The Itanium Processor Family (IPF – AKA IA-64) is the next generation of processors based on a collaboration between Intel and HP. IPF promises to provide a base for high-performance, stable, scalable platforms required for use in demanding, mission critical enterprise applications. IPF is based on a new architecture, Explicitly Parallel Instruction Computing (EPIC) that is designed to extract the highest performance from the most demanding computational loads. In order to extract the maximal benefits of EPIC you must take the time to analyze and tune your workload.

In the design of IPF, an explicit trade-off was made that placed a larger burden on the compilers and optimization of the software in order to effectively use the advanced capabilities of the architecture. While all architectures benefit from tuning efforts, this effort provides greater benefit for applications targeted for IPF systems. As IPF will grow to be the dominant enterprise architecture, the efforts made to tune IPF solutions will provide a payback far into the future.

1.1 Performance Tuning in General

The amount of effort you, or your software vendor, put into tuning an application depends on how critical its performance is to your needs. If the application represents the critical path of a business critical task… then the extra work required to extract the best performance will be more than worth it.

Tuning efforts can be broken into 3 major types or phases:

1.2 General, “Outside Focused” Tuning Efforts

This is where you manipulate the environment around the application to maximize performance. General tuning can sometimes be as simple as buying a faster, more capable system or dedicating a system to execute just the workload you are interested in optimizing. It also includes kernel and system tuning, memory upgrades and other methods to provide a platform that meets the unique needs of the targeted application.

A good understanding of the targeted workload is required to learn what system resources are at a premium. Performance measurement tools that provide memory, disk and cache usage statistics and other system resource measures (such as semaphores, shared memory etc.) can direct you in these tuning efforts. For ISV supplied solutions, optimal system settings are usually available from the vendor.

1.3 Basic Profiling and Tuning of the Critical Sections

Here you break the application into its major parts, analyze each to determine bottlenecks, and work to remove those bottlenecks. Basic profiling and tuning requires the identification of, and access to the components that make up the application (not necessary source code). Critical paths are determined through simple measurement and you then focus effort on the pieces that represent the largest part of the workload.

If you do not have access to source code, you can substitute other functionality equivalent modules (if this flexibility is present) or work with the vendor to obtain a higher performing version of key components. You can break a critical task into multiple parallel paths and dispatch the load to a (logical) cluster of servers. If you do have access to source code, tuning often starts by compiling with successively higher levels of optimization and employing Profile Based Optimization[1] techniques.

PBO is a technique where statistics are gathered on the execution profile of the object code. The statistics are then fed back to the compiler and optimizer and used to generate an execution image that more closely matches the demonstrated behavior of the application. Improvements on the order of 30% are not uncommon with this approach.

1.4 Iterative Analysis and Tuning in a Controlled Environment

The Iterative approach employs a deeper understanding of the applications being optimized in addition to the same approaches as #1 and #2. It goes further by establishing a test bed to facilitate measurement and iteration. The design of the test bed is a critical piece: a poorly constructed benchmark will lead you to tune your application incorrectly.

Tuning is not a magical process. The best results are often obtained by stubbornly iterating until the target level of performance is obtained… or you just run out of time. Another important benefit to taking the time to develop or adopt a benchmark that is meaningful for your environment is the ability to evaluate the capabilities of different vendor's offerings separately from their marketing claims. "Bake-offs" based on realistic benchmarks that you care about are a critical part of the buying cycle.

We now present a specific example that applies the most aggressive approach just described, starting with some insight into the importance of SSL in today's Internet based business environment.

2 SSL Motivation

When you make a credit card purchase on the web, access your brokerage or retirement account, or receive other confidential information, you are probably using SSL.

SSL[2] is a security protocol more recently standardized in as TLS[3]. It uses a combination of public-key and private-key cryptography to allow private communication between consumers and businesses over a public network.

Popular browsers include a padlock icon to indicate that the connection is secure (it can't be observed by a third party). This icon indicates that SSL is being employed, and users are encouraged not to transmit sensitive information (such as credit card numbers) without it.

Our focus is on the performance of a server handling SSL transactions.

[1] A listing of the available tools to assist the developer and optimizer is available at http://devresource.hp.com/devresource/Tools/ToolLibrary.html. The Caliper tool can be used as both a PBO and a performance analysis tool.

[2] SSL 3.0 Specification. http://home.netscape.com/eng/ssl3/

[3] Transport Layer Security. http://www.ietf.org/html.charters/tls-charter.html

3 SSL Performance

For a merchant trying to make money on the Web, probably the most important exchanges with the customer are those involved in accepting an order. All the pretty graphics and fancy interface is valuable because it leads to a financial transaction with the user.

Similarly, sites that collect user information (e.g. product registration), find that the most useful information (name, address, income range) are most likely to be considered confidential by the user. Using SSL to protect the data increases the likelihood of receiving useful information.

SSL is also being used to keep other information secure, such as brokerage and 401K transactions, travel requests, employee transactions (health plan signup, time vouchering), and password entry to remote sites. Over time it is likely to be used for other applications such as browsing websites.

SSL transactions are surprisingly expensive for the server. A single SSL request takes processing power much greater than a more ordinary web request. In late 1999, we saw SSL operations taking over 1000 times as long as a SPECweb96 operation. In mid 2001, we saw improvement to closer to 10 to 1 compared to SPECweb99.

Studies have shown that customers are willing to wait a small number of seconds for the result of a secure transaction. If they see no response they will often 'reload.' effectively doubling the server workload! If they give up, the merchant has wasted the server resources used so far, lost a sale, and quite possibly lost a customer. On the web "your competitor is only a click away."

Obviously it is critical for servers to provide adequate SSL performance. Web traffic is "bursty," producing large spikes in access rates. Since SSL requests are comparatively so expensive, there is little to be gained by delaying other (inexpensive) requests to handle the sales. Web server needs to be able to handle significant SSL load.

Other studies have focused on the need for hardware accelerators[4], since some platforms are weak at the computations needed to handle SSL requests[5].

Our results show that some platforms perform quite well without additional accelerator hardware.

4 How SSL Works

SSL uses the almost-magical tricks of Public Key Cryptography to allow two parties who have never met to communicate securely over a public network.

The user typically follows a link to a well-known site such as www.amazon.com. Once they are ready to make their purchase, they click on a link taking them to the order section. This is the beginning of their secure transaction.

In response to the secure request (identifiable by the icons mentioned above, and by the transition from an "http:" to an "https:" URL), the server sends its public *certificate*. This certificate is *signed* by

[4] Information Security Magazine, January 2000.
http://www.infosecuritymag.com/articles/january00/cover.shtml

[5] This reference covers the architectural capabilities of different microprocessors with an emphasis on IPF: Intel Itanium Processor – High Performance on Security Algorithms (RSA Decryption Kernel)
http://www.intel.com/eBusiness/pdf/prod/ia64/30045WP.pdf

one of a small set of trusted authorities (such as VeriSign). The user's browser checks the *signature* against its built-in list, and verifies that the certificate is indeed valid.

The valid certificate assures the customer that the server is indeed who they say they are, and avoids certain masquerade attacks.

Besides the identifying information, the certificate includes a 1024-bit *public key*.

The browser now generates a random number, and performs a one-way encryption[6] of it using the server's public key.

The client next sends the encrypted random number back to the server. The server uses its never-divulged *private key* to decrypt the message and retrieve the client's random number.

The magic just happened. The client and server who never met now share a secret (the random number), established via communication that was entirely open to public view.

Unfortunately, this exchange required millions of cycles of computation (particularly on the server).

Now that they have a shared secret, however, the customer and server can use an older, more efficient technique called *symmetric* or *private-key* encryption. Using identical algorithms, the browser and server derive a 128-bit key from the shared secret.

This key may now be used to securely and efficiently transfer data between the two parties.

One final piece of the protocol is the use of a secure hash on the data payload to ensure that it is not altered during transmission.

The SSL protocol allows the use of alternate cryptographic algorithms for the initial public-key exchange, the symmetric encryption, and the secure hash. The client browser (using parameters that can be set by the user, but are most likely defaults) and the server negotiate a set of algorithms to use (called a *crypto suite*) during the initial phase of the SSL transaction.

Different algorithms are available to trade off computation costs, patents (RSA used to be patented), ease of government wiretapping, etc. In practice, only a small fraction of the many possible suites are commonly used.

5 Benchmark Design

A good benchmark can be useful as a workload to drive intelligent tuning decisions, and to do product positioning, both between product lines and with the competition. The benchmark described below has been helpful in these areas.

With additional calibration against specific customer workloads and configurations, benchmarks can also be used for sizing and capacity planning. We have not yet performed this calibration.

5.1 Two key benchmark objectives

There are two features that are important in developing a quality benchmark:

[6] http://www.rsasecurity.com/rsalabs/faq/2-3-2.html It is computationally infeasible for an eavesdropper to reverse the encryption using only the public key.

5.1.1 It must give reproducible results

This first point obviously means you should be able to reproduce your own measurements, but it also means that an independent observer can verify them as well.

A proprietary benchmark is at a severe disadvantage because it limits the number of people that can reproduce it. Some benchmarks have licensing restrictions that require permission before results can be reported.

Freely distributable benchmarks (such as netperf[7]) have potentially the largest base of potential testers. This increases the range of results available beyond what any one organization could provide. The results will also be produced by people with a broad variety of axes to grind.

When designing benchmarks it is important to be decisive. A benchmark with dozens of options is unlikely to produce useful results. When a user wants to compare systems, he will likely find that one system has been measured with option B, while the other was measured with option Q, so no useful comparison is possible.

A related problem is when a measurement produces many separate results (e.g. separate timing results for each of the hundreds of Unix syscalls). These measurements can be useful for engineering purposes, but they provide little direction about what should receive tuning attention. For a customer, they provide mountains of data, but little useful information.

A benchmark that is HP-UX specific is less valuable than one that runs on a variety of Unix's, while one that also runs on non-Unix servers is even better. The benchmark described can be run on any system that runs a web server able to process SSL. Additionally, a benchmark is more useful if it can be driven by a variety of clients.

5.1.2 It must represent a meaningful workload.

Unfortunately, this second point is often violated. A benchmark that does not represent any customer need (e.g. nops/sec) is useless. On the other hand, a benchmark need not represent all possible customers to be valuable.

5.1.2.1 Temptations

When developing a benchmark, it is tempting to tweak the benchmark so that it focuses on the aspects that your system (whether commercial or an academic project) is best at. Keeping a customer application in view helps avoid the temptation to head down that path. For example, some of the architectures that HP sells are best at the initial public-key negotiation, but not as strong at the symmetric encryption. Trying to make ourselves look good would lead to decreasing the amount of data transferred, or even to skipping the data transfer entirely (A secure transaction with no data is about as useful as a fast nop). We've stayed with the 14KB average transfer size that SPEC derived[8] years before optimizing SSL results became an issue.

---

[7] http://www.netperf.org/netperf/NetperfPage.html

[8] http://www.spec.org/osg/web96/web96q+a.html - question 6

5.1.2.2 You have to start somewhere

This isn't the last SSL benchmark that ever needs to be written. It doesn't cover all interesting areas, but it does cover an interesting, important area that isn't adequately handled by other benchmarks. It's reproducible, and it represents a reasonable first approximation of a secure user workload.

5.2 Choosing parameters for SSL Benchmark

With the above objectives in mind, it's clear that we need to be customer-focused when we make decisions about what crypto suite to measure,how much data to transfer, and how much session reuse to have. We want to represent good user practices – ones that are commonly used, and secure.

Five important parameters for SSL measurements:

5.2.1 Public Key Algorithm + length

The choice here is fairly obvious: 1024-bit RSA[9]. RSA is very commonly used. Now that RSA Security's patent has expired, there's no reason not to use it.

1024-bit is considered a secure length for normal usage[10]. 768 or less would be insecure, and >1024 is currently only needed for specialized applications (such as PKI management). To keep a level playing field, we consistently used 2-prime RSA (i.e. not using RSA's multiprime technology).

5.2.2 Symmetric Key Algorithm + length

128bit RC4[11] is both fast and secure.

Shorter RC4 is insecure – it was used because of (now-obsolete) government regulations.

DES is currently insecure[12].

3DES[13] (or triple-DES) is more secure than RC4-128, but it is **much** more expensive.

AES[14] (formerly Rjindael) will likely replace RC4 over the next few years but is only rarely used today.

For today's workload we chose 128bit RC4.

5.2.3 Cryptographic Hash Algorithm

SHA1 and MD5 are both plausible choices. SHA1 is commonly used, not much more costly than MD5, and notably more secure[15] (160 vs. 128 bit hash), so we used SHA1.

---

[9] http://www.rsasecurity.com/rsalabs/faq/3-1.html
[10] Applied Cryptography, Bruce Schneier, tables 7.6 + 7.9
[11] http://www.rsasecurity.com/rsalabs/faq/3-6-3.html
[12] http://www.eff.org/descracker.html
[13] http://www.rsasecurity.com/rsalabs/faq/3-2-6.html
[14] http://csrc.nist.gov/encryption/aes/
[15] http://www.rsasecurity.com/rsalabs/faq/3-6-5.html

In practice, the RC4-SHA suite (shorthand for SSL3_TXT_RSA_RC4_128_SHA) is commonly used for secure SSL transactions.

5.2.4 Data Transfer Size

Choosing the data transfer size is an opportunity to play "benchmark games". The benchmark designer is trading off the importance of the Public Key computation (done once, very computational intense) with the symmetric computation and hash (done on a per byte basis). The smaller the transfer size, the greater is the relative importance of the Public Key computation.

We decided to stick with the 14KB average size used by SPEC[16] in their SPECweb96 and SPECweb99 benchmarks. They did significant research[17] involving access log analysis at a number of sites to come up with this figure. Secure traffic may have a different average size than non-secure traffic, but until a similar-quality analysis is done on secure traffic, we're hesitant to change the value.

Also, the 14K figure may be interpreted as the total payload per RSA negotiation. That way it also approximates session reuse (see below) with a total of 5 3KB transfers per session.

5.2.5 SSL Session Key reuse rate

Here the right answer is less clear, but there is still value in making a choice.

Some early benchmarks got this totally wrong, simulating a thousand customers each making a purchase by having one customer make a thousand purchases. That is, they had an infinite reuse rate, and only a single RSA negotiation was done for the entire benchmark run. Although the RSA is no longer the only important performance component it is still very significant, and ignoring it is wrong.

The conservative zero-reuse assumption was chosen – SSL sessions are never re-used. This represents the "worst case" situation.

Measurements across a variety of sites should be made to derive session reuse and transfer sizes that more accurately reflect SSL usage patterns.

The (presumably) smaller transfer sizes and higher reuse rates will cancel each other out to a degree, giving results similar to those measured in this first-generation benchmark.

The major concern at present is that the reuse path through the Web server and crypto code isn't being measured, and therefore hasn't received as much tuning attention.

5.3 Our Benchmark

Having made careful decisions about the parameters of the secure workload, we consider how to drive that load. The mechanics of the benchmark do not matter much, as long as meaningful load parameters are maintained.

[16] Standard Performance Evaluation Corporation. http://www.spec.org/
[17] http://www.spec.org/osg/web96/web96q+a.html - question 6

With that in mind, we kept the actual benchmark[18] very simple. Client systems drive the load with invocations of cURL[19]. Throughput is determined from (the average of) the requests recorded in the server log file.

The tested system can use any web server capable of handling secure traffic, as long as it can produce standard log files. The system tested can run any appropriate OS. No additional programs need to run there. We tested HP-PA (1.1 and 2.0), UltraSPARC (2+3), IA-32, and IPF architectures. We've used HP-UX, Solaris, and Linux operating systems, and the Zeus and Apache web servers.

Client software should run on any type of Unix. We've tested with HP-UX and Linux clients.

Besides testing the appropriate crypto routines, we want to exercise the system components that a real request would use: interrupt handler, network code, operating system and web server – a short but meaningful solution stack.

By initiating the requests on separate client machines, we exercise those paths used in real life. Since client efficiency is relatively unimportant, we have considerable freedom of choice in implementation.

Other benchmarks should produce similar results as long as the configuration is similar to that described above.

5.3.1 What it doesn't do

The benchmark does not check for errors. The person running the benchmark is responsible for checking error logs. Failure to do so will result in a loss of credibility when others cannot duplicate the results. This leverages the open-source nature of the benchmark.

The benchmark uses a steady-state request arrival rate. A Poisson or heavy-tail distribution (along with a response time metric) would better represent customer workloads for sizing and capacity-planning purposes. The simple steady-state approach is nonetheless useful to drive system tuning and provide a level playing field for comparing different systems.

5.4 Other Benchmarks

5.4.1 Micro benchmarks

Micro benchmarks, or component-level benchmarks (such as cycles per 1024-bit RSA decrypt) are useful primarily to engineers who are tuning specific sections of code. I believe there are benchmarks of this type included in the OpenSSL code base[20]. These benchmarks do not require client systems (since they do not test the network, OS, or web server).

5.4.2 System Level Benchmarks

A system level benchmark shows the combined impacts of the areas that affect real-life performance: OS, networking, interrupt processing, data transfer, user libraries, crypto libraries, web server, etc. As individual bottlenecks are tuned, new ones appear.

---

[18] Publicly available at ftp://ftp.cup.hp.com/dist/networking/benchmarks/SSL_rate.tgz (instructions included)

[19] http://curl.haxx.se/

[20] http://www.openssl.org/

Before creating a new benchmark, existing readily available benchmarks were explored. Intel had a very polite and useful commentary[21].

5.4.2.1 Web Bench

Web Bench uses insecure key lengths, so it is not a good choice for SSL benchmarking. Today, RSA has been tuned enough that symmetric encryption (and hence key length) **does** have a significant impact. Apparently Web Bench was developed when US security export restrictions were still in place. As a very general benchmark, there is no one way to run it, so results on different systems are likely to be made with different parameters, and are thus not comparable.

Web Bench is Windows-based.

5.4.2.2 Web Application Stress

The biggest problem with the Web Application Stress benchmark is that it doesn't control session reuse. Reuse varies with the type of client used. This makes it very easy to get meaningless results (e.g. with infinite session reuse).

WAS is Windows-based.

5.4.2.3 SPECweb SSL

SPEC is currently working on developing a SPECweb99 variant that will include SSL usage. It will not be open-source (SPEC charges for access to the benchmark code). They do have a formal review and publication process.

SPEC has a good reputation for developing quality benchmarks.

5.4.3 Clients: Windows vs. UNIX

Serious performance measurement requires multiple clients (We used 120 clients to saturate the 4-way Itanium). Controlling these clients requires a platform with good support for remote access. Any Unix variant would suffice, Windows clients become difficult to manage after the first few.

5.5 Further Benchmark Development

5.5.1 More Load per Client

When measuring very high performing systems such as a 4-way Itanium, the number of clients required becomes awkward. If extremely fast systems such as this were frequently tested, it would be appropriate to revise the benchmark to generate more SSL load per client. Such a new version would likely not create a new process for each request, so it would need to be extremely careful to handle SSL session reuse correctly.

5.5.2 Customer Studies of SSL session reuse and transfer size

It would be valuable to conduct studies at various sites currently using SSL to gain a good understanding of real-life SSL session-reuse rates and SSL transfer sizes. Once determined, these

[21] Designing a Secured Website – What you need to know about SSL Benchmarking. http://www.intel.com/network/white_papers/ssl_benchmarking/benchmark.htm

parameters could be used to make the benchmark load more representative. Additionally, calibration of benchmark results against live SSL systems would provide information needed for system sizing and capacity planning.

5.5.3 New Crypto Suites

AES is likely to become prevalent over the coming years. When desired, the current benchmark can easily use AES instead of RC4.

6 Benchmark Results

6.1 Test Setup

With a benchmark in hand, we built a test harness leveraging older clients that had been used for SPECweb96 + 99 testing. These were rack-mounted B160 + B180 workstations connected to the test system by a private, switched 100t network. Each workstation also had a second 100t port on our site network for control.

6.2 The Tuning Cycle

In the simplest case of a tuning cycle, you set up your server, measure it, then change a single component and try again.

Although this description is simple, it doesn't give you any hint about what to change. In practice, you also run analysis tools on the server to give you those hints.

The tools available for analysis are different on different target platforms.

vmstat (standard on UNIX systems) will report user, system + idle time. In many benchmark situations you are trying to saturate the CPU, so idle CPU suggests that you need more load or you have a contention problem.

Profiling (prof, gprof, etc) will help you identify routines within an application that are bottlenecks. We used an internal program called Prospect to get prof-style output without requiring application recompile. (We did not have access to the web server source code).

Although Prospect is not generally available, a similar tool called Caliper is now available. After manually sorting + grouping the profile output, we were able to identify promising areas and estimate potential improvements.

Kernel profiling and syscall traces may also be helpful in situations where the system time is large.

Next you evaluate the opportunities, considering how quickly and easily they can be implemented, and decide what change to try on the next measurement cycle.

6.3 Over Time

6.3.1 Initially Un-tuned

Initial measurements were made in November 1999 with a different, custom-built SSL benchmark. It showed about one SSL op/sec on a 9000/K580 (240 MHz PA8200).

6.3.2 A Surprising Improvement

Attention to system-level SSL performance began in October of 2000. (Teams involved in the crypto code had been using micro-benchmarks and intuition to guide their tuning.) Measuring with a new benchmark (described above), new OS version, new web server version, and new processor lead to a surprising 40x improvement. We began tracking down the source of the improvement.

One of the easiest tests to run was changing back to an older version of the web server. It turned out that at Version 3.3.7 (December 1999), the Zeus[22] Web Server had changed to a different crypto library. This accounted for roughly 20x of improvement, with an additional 2x from the processor change.

6.3.3 Tuning on Itanium

Initial measurements on the Itanium system, showed a **very** clear bottleneck.

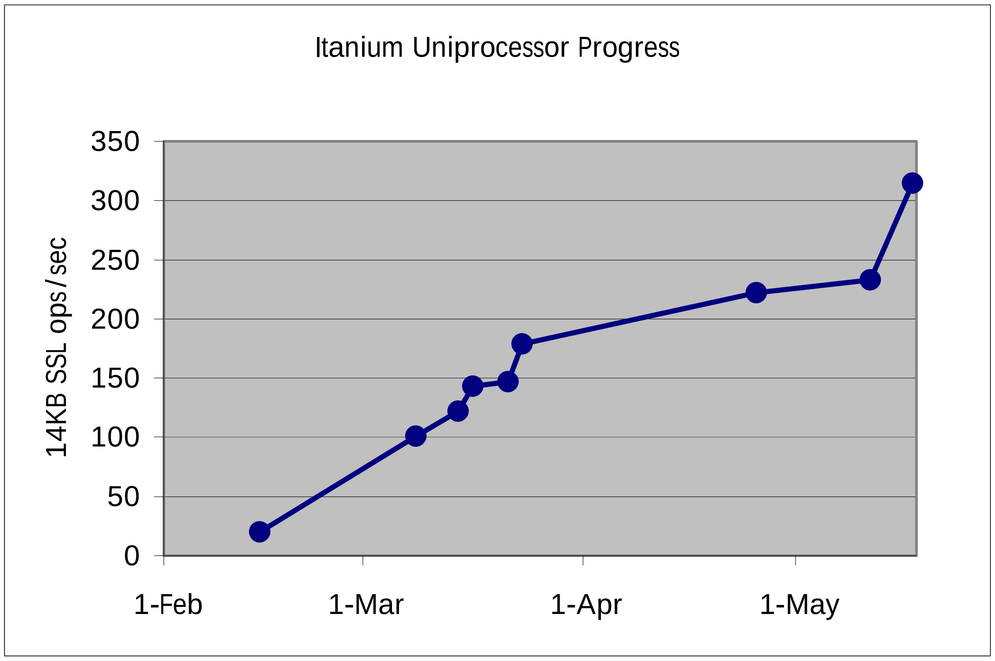
- Almost all time was spent in user mode (not system or interrupt).
- Almost all user time was spent in zeus.web (unsurprising).
- Almost all time in zeus.web was spent in crypto library routines.
- Almost all time in the crypto routines was spent in the code used in the initial private-key RSA decryption operation.

Because there was such a strong bottleneck that could be significantly improved on this architecture we were able to make excellent progress in improving performance in a short period of time.

Successive versions of RSA decrypt code that were better tuned lead to other crypto routines (RC4 and SHA1) becoming more significant in the overall performance. As these were tuned, other areas became tuning targets (libc, general Zeus and other crypto code). Other opportunities for tuning that we can be addressed over the coming months were also identified.

The graph below shows how system-level SSL performance improved on Itanium as a result of the tuning efforts of HP, Zeus, and RSA.

[22] Zeus Technology. http://www.zeus.co.uk/



Our first measurement with prototype Itanium hardware and an alpha version of Zeus Web Server for HP-UX/Itanium gave us 20 SSL ops/sec in February of 2001. This version included very early crypto libraries.

March brought a beta version of ZWS with better crypto libraries. This delivered 100 SSL ops/sec.

Tuning during March investigated different boot options (workarounds for early chip features), and an improved HP-UX kernel. These changes brought us to 180 SSL ops/sec.

April brought a new version of Zeus with improved RC4 (crypto library) routines. This brought us up to 220 SSL ops/sec.

May brought a new optimized kernel, and a faster, more final version of the Itanium chip with a larger cache. This brought us to the current 315 SSL ops/sec.

6.3.4 Real Life Tuning

Producing the progress described above involved other support activities and efforts that didn't "pan out." They are described to help set expectations for others planning tuning efforts.

The process included

- making measurements for comparison
- researching various tuning options (Who's allowed/able to change the code? What's their schedule?)
- managing the clients
- deciding when to upgrade (or delay upgrade of) hardware and software
- running the tests
- checking for errors (and re-testing when needed)
- coordinating with partners (internal and external to HP)

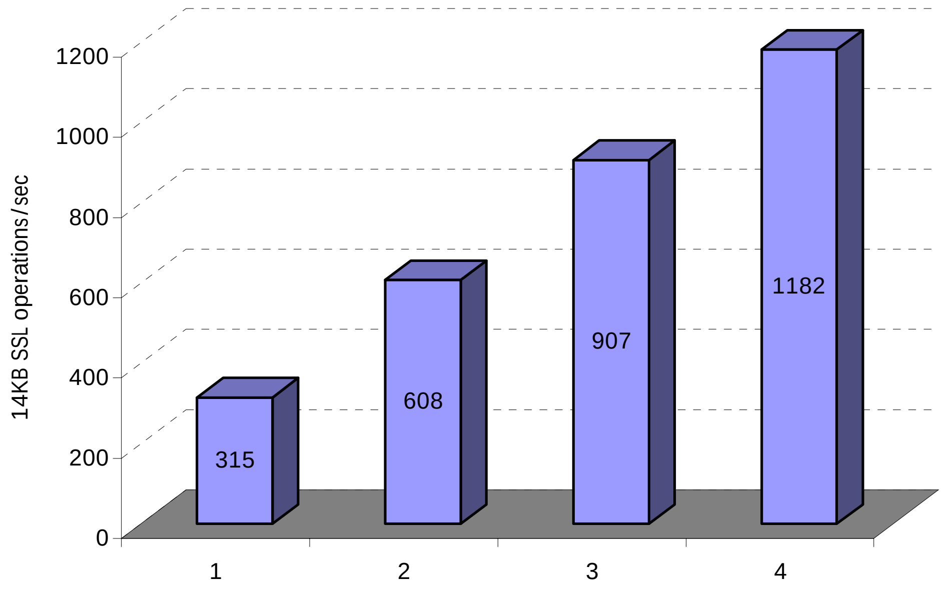
- collecting profile traces and aggregating the data
- locating bottlenecks and identifying tuning opportunities (short + long term)
- writing and publishing interim results
- convincing the partners to take action
- measuring prototypes for partners
- trying out dead ends

The aspects of the system we changed during testing included:

- new versions of Zeus
- new crypto libraries
- changed boot options
- changed the system firmware
- changed OS versions on the system
- increased number of clients
- upgraded the processor

Unlike some tuning efforts, we weren't able to get involved in a kernel-tuning cycle. We started too close to the OS release.

6.4 MP scaling



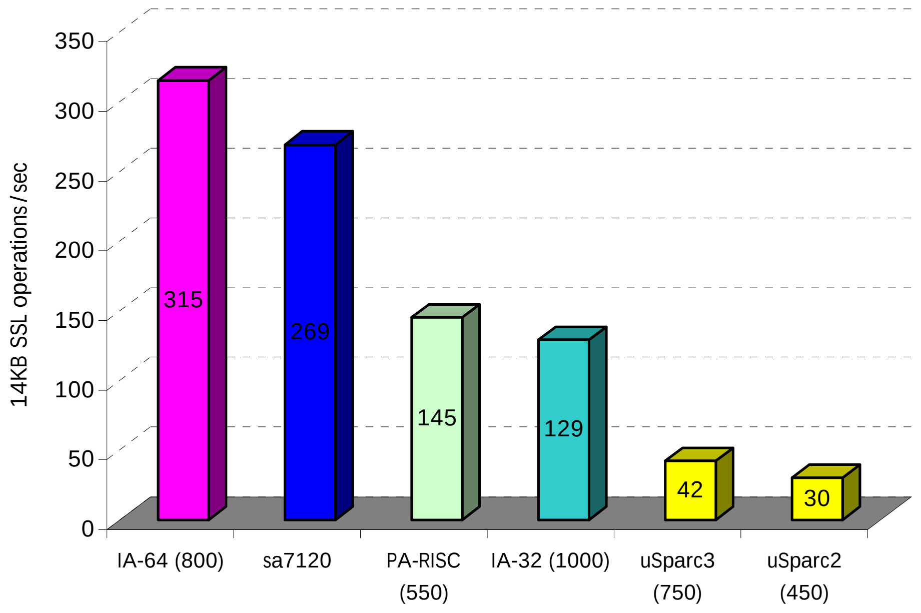
The SSL workload scales well on MP systems. There should be little interaction between independent requests, and in fact we saw excellent scaling. Zeus Web Server is good about batching access log updates to avoid excessive disk IO's.

One issue is maintaining a cache of recent SSL information in case the sessions are reused. This benchmark does not utilize this cache, but the web server maintains it in case reuse does happen. This is a good idea since unnecessarily repeating the public key exchange would be expensive. Zeus maintains a multi-level cache. The in-memory cache is managed efficiently, but maintaining the disk-level cache slowed the system significantly in the MP case. For the benchmark measurement we added 'tuning!ssl_diskcache no' to the Zeus configuration file /usr/local/zeus/web/global.cfg . This was the only configuration change made in any testing.

It would be useful for a future benchmark version to include a moderate amount of session reuse. This would help drive appropriate tuning decisions in maintaining the disk-level cache.

6.5 Platform Comparisons

6.5.1 Comparative Results



Architecture	Clock (MHz)	OS	Web Server	14KB SSL ops/sec
Itanium	800	HP-UX 11.20	Zeus 3.4beta	315
Appliance	3chips	NA	NA	269
HP-PA8600	550	HP-UX 11.0	Zeus 3.3.8.3	145
IA-32	1000	Linux 7.0.91	Zeus 3.3.8.2	129
UltraSparc3	750	SunOS 5.8	Zeus 3.3.8.2	42
UltraSparc2	450	SunOS 5.8	Zeus 3.3.8.2	30

6.5.2 Itanium

The Itanium system does exceptionally well. The architecture is particularly good at the heavy computation needed for encryption work. These measurements were made with firmware revision 73.

6.5.3 HP-PA

The PA 2.0 architecture has good support for computation. This made up for the lower clock rates.

6.5.4 IA-32

The IA-32 performed quite well. The higher clock rate helped compensate for weaker arithmetic units.

6.5.5 UltraSPARC2 + UltraSPARC3

The Sun UltraSPARC2 showed weak performance even with Zeus web server (it was slower with Apache). This matched projections made by Intel based on instruction set architecture[23].

We expected a clock-speed improvement, plus an architectural boost from moving from UltraSPARC2 to UltraSPARC3. Surprisingly we didn't even get the full clock improvement. Our UltraSPARC3 system included Sun's firmware patch to work around a math problem. It would be interesting to know if the chip performs any better once that error is fixed.

6.5.6 Accelerator Hardware

We tried to measure a PCI accelerator card, but were unable to get one functioning correctly with Zeus + HP-PA.

We were able to test an accelerator appliance. This box, the HP sa7120, handles all the cryptographic work for the web server, accepting encrypted https requests from the client, and passing on unencrypted http requests to the back-end web server. The appliance was configured to not "spill" SSL requests when overloaded. This isolated the performance of accelerator from the system behind it.

Measuring 270 SSL ops/sec was very interesting since it provided a calibration between a measured, repeatable result and marketing claims of "600 connections per second". Inspecting the hardware internals showed that the system has 3 Rainbow[24] chips. Presumably the "200 connection per second" accelerator has a single chip. Apparently the 200 connections comes from a 5ms RSA decrypt time, so this is a micro benchmark, not a system benchmark. Perhaps the box could really hit 600 transactions, if they were of zero length ☺.

It appeared that the main CPU in the accelerator (IA32, presumably) was performing the RC4 + SHA1 computations. On a more difficult 3DES workload, the Rainbow chips might have been more involved and produced a comparatively better result.

7 Issues in SSL performance

This is a summary of system components and how they affected SSL performance.

7.1 Larger effects

As expected, the processor architecture had a large impact. Efficient support for integer multiplication is particularly important.

In most cases we saw a nearly linear speedup with increasing clock rate. This only applies within a processor family (e.g. a 550MHz PA-8600 outperformed a 1000MHz Pentium3).

Crypto libraries are critical to SSL performance. We saw significant improvement due to using the latest RSA BSAFE Crypto-C libraries[25]. (Note: we consistently avoided using multiprime).

[23] Intel Itanium Processor – High Performance on Security Algorithms (RSA Decryption Kernel) http://www.intel.com/eBusiness/pdf/prod/ia64/30045WP.pdf

[24] http://www.rainbow.com

[25] http://www.rsasecurity.com/products/bsafe/cryptoc.html

We originally expected the web server choice to play a fairly small role (other than the crypto library it is built with). In practice, we saw a significant difference between Zeus Web Server and generic Apache. This is likely due to the years of tuning that went into ZWS to provide leadership SPECweb96 and SPECweb99 results.

After working around a problem with the maintenance of disk-based SSL session cache, we saw nearly linear performance gains as we added processors (MP scaling). This makes sense, since almost all time is spent in user mode, and the requests are essentially independent of each other.

7.2 Smaller effects

We didn't run any controlled experiments changing the OS on a fixed hardware platform. Nonetheless, since the total system time is fairly small, it is likely to be a relatively minor effect.

Now that the RSA code has been fairly well tuned, the length of the data transferred has a significant (but not huge) effect. For example, one experiment in changing the 14KB transfers to 2KB increased the throughput from 315 to 390 – a 25% increase from a 7X decrease in data.

7.3 Very small effects

This benchmark is quite insensitive to disk and memory configuration.

Systems tested had a GB or less of RAM, and two disks. A real customer workload may have different requirements, but the SSL performance is basically driven by the CPU.

Network requirements are modest. Even the 4-way Itanium result used only 3 100t networks.

8 Conclusions

- Using a simple solution-stack benchmark, we have driven large improvements in SSL performance.
- Benchmark results show that IPF performs exceedingly well on this secure workload.
- The tuning approach described should be helpful in improving performance on other workloads.