End-to-End Java Security Performance Enhancements for Oracle SPARC Servers

Luyang Wang
Oracle
4180 Network Circle
Santa Clara, CA 95054, USA
luyang.wang@oracle.com

Pallab Bhattacharya[1]
Facebook
1 Hacker Way
Menlo Park, CA 94025, USA
pllb@fb.com

Yao-Min Chen
Oracle
4180 Network Circle
Santa Clara, CA 95054, USA
yaomin.chen@oracle.com

Shrinivas Joshi
Oracle
4180 Network Circle
Santa Clara, CA 95054, USA
shrinivas.joshi@oracle.com

James Cheng
Oracle
4180 Network Circle
Santa Clara, CA 95054, USA
james.cheng@oracle.com

ABSTRACT

In this paper we investigate the performance of cryptographic operations, when used in Java applications. We demonstrate the advantage of using built-in hardware accelerator for cryptographic operations on SPARC servers. In particular, we demonstrate the advantage of hardware cryptographic instructions invoked via AES and SHA intrinsics, implemented in the Java Virtual Machine (JVM), over the more traditional Java Native Interface (JNI) calls. For the purpose of our study, we modified the SPECweb2005 benchmark by adding modern banking requirements, and created a new workload which we call the End-to-End Java Security (EEJS) workload. Using the workload, we compare different Java Cryptographic Service Providers (CSPs) and arrive at the conclusion that hardware cryptography has significant performance advantage for Java applications. With the EEJS workload, we also identify several enhancements applicable to the Java Secure Socket Extension (JSSE).

General Terms

Performance, Security.

Keywords

Java Security; Java Cryptography Performance; SPARC Processors; JVM Intrinsics; RSA; AES; SHA; JSSE; SPECweb2005.

1. INTRODUCTION

There were a few recent customer cases where the performance of security operations on the SPARC servers was deemed critical to customer applications. These cases raised the necessity of an in-house tool to model customer application scenarios and identify performance issues and optimization opportunities. This external "push" also aligned well with the internal "pull" to evaluate the new way of doing cryptographic acceleration since SPARC T4, which uses instructions rather than coprocessors. The two reasons combined have motivated the need to study the cryptography performance from an end-to-end perspective.[1]

Before delving into the end-to-end performance study, we briefly describe the history of SPARC cryptography acceleration. The Oracle SPARC line of processors have a history of supporting cryptographic operations at hardware level starting with the first Niagara Processor [1]. Its many generations consistently work towards adding support for more cryptographic algorithms and improving the performance of cryptographic operations. A key motivation for this effort is that certain cryptographic computations are inefficient if done in software, requiring many instructions when using a conventional instruction set. The earlier cryptographic acceleration (e.g., the implementations in UltraSPARC T2, T2 Plus and T3 [2][3]) is mostly delivered via a coprocessor mechanism, which requires device drivers or system calls to use these new capabilities, but also adds software overhead. In later generations (SPARC T4, T5 and onwards), non-privileged instructions are implemented, which avoid the overhead of traps into the kernel. With SPARC T4, new algorithms are also developed to replace old, more vulnerable algorithms as well as to address new security and usability requirements.[2]

The new hardware cryptographic instructions have been evaluated via numerous micro benchmarks. However, when they are used in a real-world application setting, sometimes the advantage is not immediate. This could be due to the fact that the software (security middleware) has not used the hardware cryptography in an effective way, or the fact that there is a bottleneck in the software. We need a tool to identify such software-hardware integration issues. Towards this end, we have devised the End-to-End Java Security (EEJS) workload. It is based on SPECweb2005 Banking Workload [4][5],[3] with enhancements to incorporate

ICPE'16, March 12–18, 2016, Delft, Netherlands.
© 2016 ACM. ISBN 978-1-4503-4080-9/16/03…$15.00.
DOI: http://dx.doi.org/10.1145/2851553.2851577

[1] All work by Pallab Bhattacharya for this article was performed prior to his employment at Facebook.

[2] The newer processors, SPARC T5, M6 and M7, are similar to SPARC T4 in terms of cryptographic processing.

[3] The SPECweb2005 benchmark has been retired by SPEC.

modern banking requirements. [4] We have learned these requirements from customer engagements.

A key area of our performance study is related to the Java Native Interface (JNI) calls when executing the cryptographic instructions. JNI calls involve copying of data and flushing of register windows, making them expensive and less suitable for small messages. A new approach, called *intrinsics*, eliminates the JNI overhead. The EEJS workload is an effective tool to compare the implementation based on intrinsics and the more traditional implementation that leverages JNI.

This paper makes the following contributions. First, we demonstrate that the best performance on SPARC servers comes from using the hardware cryptographic instructions. Second, with Java Development Kit (JDK) 8u40 and later, accessing the hardware cryptography via intrinsics shows advantage over going through the JNI. Third, we have identified enhancements in the Java Secure Socket Extension (JSSE). The effects of the performance improvements are quantified in terms of response time, number of simultaneous user sessions, and intensity of user activities such as page views, just to name a few.

For the remainder of the paper, in Section 2 we describe EEJS benchmark in detail. In Section 3 we cover the experiment setup and the cryptographic service providers offered in JDK on Solaris operating system. In Section 4, we present and evaluate the performance data. This is followed by Section 5 where we describe the JSSE enhancements along with their contributions to performance improvement. Finally we conclude the paper by summarizing our results in Section 6.

2. BACKGROUND

In this section, we first discuss the SPECweb2005 Banking workload and then introduce the EEJS enhancements.

2.1 SPECweb2005

SPECweb2005 Banking workload was developed after conducting extensive research in the financial sector, specifically focusing on the Web transaction types, the business logic, the payload size and the cipher-suite specification that was prevalent at the time. From the study, RSA1024_RC4_MD5/SSLv3 was most popular *circa* 2005, thus it was chosen for SPECweb2005 as the cipher used for securing communication between the clients and the banking application.

The overall structure of the SPECweb2005 Banking workload is illustrated in Figure 1. On the left hand side of the figure, the drivers simulate the browsers or mobile apps used by the banking customers. The drivers implement load generators that inject load into the client application, which in turn sends HTTPS requests to and receives HTTPS responses from the server [4].

[4] It should be noted that results from our modified benchmark are not compliant with SPECweb2005 run rules and cannot be compared with published results. However, the use of SPECweb2005 benchmark conforms to SPEC Fair Use Rule for research use [6].

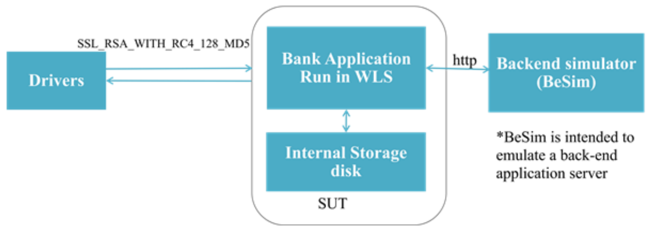

SSL_RSA_WITH_RC4_128_MD5
Drivers
Bank Application Run in WLS
http
Backend simulator (BeSim)
Internal Storage disk
SUT
*BeSim is intended to emulate a back-end application server

Figure 1 SPECweb2005 Banking Workload

The server is a bank application running on a hardware platform referred to as System Under Test (SUT). The application accesses static content such as check images that are hosted by the internal storage disk on the SUT. The server works with a backend simulator (BeSim) to serve the client requests. BeSim is intended to emulate a back-end application server that the bank application must communicate with to retrieve specific information (customer data, for example) needed to complete a transaction request from the customer. The communication between bank application and BeSim is over TCP and uses the HTTP protocol.

There are two quality-of-service (QoS) parameters, TIME_GOOD and TIME_TOLERABLE, which are used to define the performance metrics of the benchmark. Specifically, the performance metric SIMULTANEOUS_USERS is the maximum number of user connections that can be supported such that 1) TIME_GOOD QoS requirement can be met by at least 95% of the page requests and 2) TIME_TOLERABLE QoS requirement can be met by at least 99% of the page requests. Here, based on end-user experience [4], TIME_GOOD is set at 2 seconds and TIME_TOLERABLE is set at 4 seconds.

2.2 EEJS

The overall structure of the EEJS workload is shown in Figure 2. We had a few goals in mind when we developed the workload. First, we eliminated the heavy disk usage caused by the large document root required in the original SPECweb2005 benchmark. In our use case, we would like to focus on the security processing requirements, not to be bogged down by the disk IO performance on the SUT. There are a few ways to alleviate the disk IO bottleneck such as using faster storage (e.g., SSDs) or storage devices front ended with large amount of DRAM for caching. We opt for a software solution of implementing a small document root with a finite set of check images. Secondly, as SSLv3 has been deprecated and RC4 and MD5 are now considered insecure, we have modified the workload to use secure SSL ciphers (TLS_RSA_WITH_AES_128_CBC_SHA256 and TLS_RSA_WITH_AES_128_GCM_SHA256, for example). Additionally, we have changed the protocol between the bank application and BeSim to be HTTPS, reflecting what we have learned from bank customers. This is accomplished by replacing the BeSim client (as part of the bank application) with the Apache HTTP client **[7]**. Finally, to simulate realistic use cases, each request from the bank application to BeSim makes a new HTTPS connection.

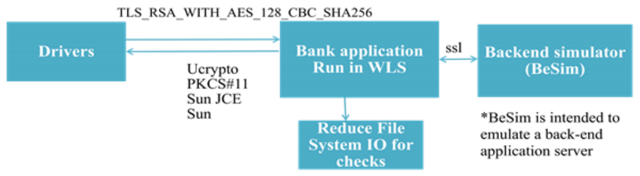

TLS_RSA_WITH_AES_128_CBC_SHA256
Drivers
Ucrypto
PKCS#11
Sun JCE
Sun
Bank application Run in WLS
ssl
Backend simulator (BeSim)
Reduce File System IO for checks
*BeSim is intended to emulate a back-end application server

Figure 2 EEJS Workload

EEJS inherits the QoS parameters from SPECweb2005, namely, TIME_GOOD (2 seconds) and TIME_TOLERABLE (4 seconds). Instead of reporting the SIMULTANEOUS_USERS performance metric, we use the number of simultaneous user sessions to control the load driven by the SUT. For example, fixing the load at N simultaneous user sessions, we measure the average response time, the percentage of responses within the TIME_GOOD limit, and the percentage of responses within the TIME_TOLERABLE limit.

A key characteristic of the workload is that there are significant SSL handshake operations. From our experiments, more than 30% of the requests involve full handshake. Therefore, this workload stresses the cryptographic processing capability of the SUT, which serves quite well our need to address recent customer cases referred to in Section 1.

From our experience, EEJS has proven to be an effective tool. In a series of experiments described in Section 4, it is used to demonstrate the performance advantage of hardware cryptography acceleration over conventional SPARC instructions. Furthermore, we compare different ways of accessing hardware cryptography, via the different Cryptographic Service Providers (CSPs). EEJS also helped identify a number of software enhancements, which can be applied to improve JSSE. These enhancements will be described in Section 5.

3. EXPERIMENT SETUP

Here we will first introduce CSPs, what they are and how they are configured. Then we will describe the hardware and software setups.

3.1 Cryptographic Service Providers

Cryptography implementations in the JDK are provided via several different CSPs. For example, with SPARC servers there are OracleUcrypto, SunPKCS11, SUN, SunJSSE, SunJCE and SunRsaSign providers [8]. Each CSP provides a package or set of packages that supply concrete implementation of a subset of the JDK Security API security features. Each JDK installation has one or more providers installed and configured by default [8] . These built-in providers and their usage priorities are listed in a configuration file named *java.security.* [5]

OracleUcrypto provider is a Solaris specific CSP that leverages the Solaris Ucrypto library to offload and delegate cryptographic operations to hardware supported by Oracle SPARC T4 and later processor based on-core cryptographic instructions [9]. Among other ciphers, RSA [10], AES [11] and SHA [12] ciphers are available in the OracleUcrypto provider. SunPKCS11 provider is supported on Solaris (SPARC and x86) and Linux (x86), in both 32-bit and 64-bit Java processes [13]; it supports algorithms including RSA, AES and SHA. The SunJCE and SUN providers are "pure Java" CSPs in the sense that the cryptographic algorithms are implemented entirely in the JDK API classes without relying on any native libraries.

For SPARC T4 and later Oracle SPARC servers, we have embedded assembly instruction level implementation of AES and SHA ciphers in the Java side code generation logic of the JVM. These assembly code implementations leverage appropriate hardware cryptography instructions. Such JVM embedded assembly implementations are also termed as JVM Intrinsics. A key advantage of intrinsics is that they avoid the JNI overhead associated with accessing hardware cryptography instructions via native libraries such as the Solaris Ucrypto library. AES intrinsics are available since JDK 8u20, via the SunJCE provider. SHA intrinsics are available since JDK 8u40, via the SUN provider.

EEJS has been used to evaluate the performance gain of the intrinsics, in comparison with OracleUcrypto and SunPKCS11 providers. The comparisons will be described in Subsection 4.1.

3.2 Hardware and Software Setups

Refer to Figure 2 for the following description of our experiment setup. The SUT is a recently announced SPARC T7-1 server, running Solaris 11 Update 3 Build 27. The T7-1 server has 32 cores and 480 GB of DRAM. On the SUT, we run the bank application with 4 cores of the server. Oracle WebLogic application server (WLS) running on the SUT is configured to use the following JVM flags: -Xms16g -Xmx16g -Xmn8g -XX:+PrintGCTimeStamps -XX:+PrintGCDetails. We use WLS version 12.2.1 in our experiments.

We have used JDK 8u40 and 8u60 for the Java runtime. As mentioned before, both AES and SHA intrinsics are available since JDK 8u40 for SPARC.

We deploy the bank application on WLS, which is also responsible for catering application-specific static content such as check images from the internal storage disk. The workload drivers are run on two Oracle X2-2 servers. Each X2-2 has two Intel Westmere CPUs and 48GB of DRAM. BeSim is run on an Oracle X3-2 server, which has two Intel SandyBridge CPUs and 128GB of DRAM. All the systems, i.e., the hosts running the drivers, WLS and BeSim, are connected by a 10Gb Ethernet private network.

For our experiments, we focus on the comparison between OracleUcrypto, SunPKCS11 and intrinsics. With JDK 8 on SPARC servers, OracleUcrypto is the default first priority CSP. We can edit the java.security file to change CSP priorities from OracleUcrypto to SunPKCS11. To exercise AES intrinsics, we need to disable AES in ucrypto-solaris.cfg or sunpkcs11-solaris.cfg, depending on the chosen provider. In this way, the AES intrinsics implemented for SunJCE CSP will be used. Similarly, to use SHA intrinsics implemented for SUN CSP, one needs to disable SHA in ucrypto-solaris.cfg or sunpkcs11-solaris.cfg. One can certainly use both AES and SHA intrinsics together, with appropriate changes in the ucrypto-solaris.cfg or sunpkcs11-solaris.cfg configuration files.

For the connection between the driver and the bank application, cipher suite TLS_RSA_WITH_AES_128_CBC_SHA256 is used. This cipher suite uses RSA for key exchange. Note that RSA consists of key encryption/decryption, as well as key generation. With the default OracleUcrypto provider, SunPKCS11 is used for key generation. When SunPKCS11 is disabled in file java.security, key generation falls through to the next available provider which, in our case, is SunJCE.

4. PERFORMANCE EVALUATION WITH EEJS

In this section, we provide examples of how EEJS is used for performance evaluation. The examples here include comparing different CSPs and comparing hardware cryptography acceleration with software implementation of cryptographic algorithms.

[5] For a SPARC server running Solaris OS, the path to the file is $JAVA_HOME/jre/lib/security/java.security.

We started out with the exact setup described in the previous section. However, we soon realized that there were a few software bottlenecks that had to be alleviated before we could move to performance comparisons of cryptography operations. The bottlenecks stemmed from the large number of connections between the bank application and BeSim (the SSL connections on the right hand side of Figure 2). To work around these bottlenecks we temporarily reverted back to plain-text HTTP connections, as were used in SPECweb2005. The data reported in this section is based on this workaround. In Section **5** we will describe how to address the discovered software bottlenecks. Identifying and addressing these bottlenecks is a major contribution of the EEJS workload analysis work.

4.1 Comparison of CSPs

We first compare the efficiency of different cryptographic service providers. Towards this end, we control the number of simultaneous user sessions and compare 1) the average response time, 2) the number of successful requests, 3) the percentage of requests that meet TIME_GOOD threshold, called *good requests,* and 4) CPU utilization including system and user times. Here, a request is considered successful if the response for it is fulfilled within the TIME_TOLERABLE interval.

Table 1: The 5 CSP Configurations Used in the Performance Comparison

CSP Configuration	A	B	C	D	E
RSA Encryption and Decryption	SunPKCS11	OracleUcrypto	SunPKCS11	OracleUcrypto	OracleUcrypto
RSA Key Generation	SunPKCS11	SunPKCS11	SunPKCS11	SunPKCS11	SunJCE
AES	SunPKCS11	OracleUcrypto	SunJCE with AES intrinsics	SunJCE with AES intrinsics	SunJCE with AES intrinsics
SHA	SunPKCS11	OracleUcrypto	SUN with SHA intrinsics	SUN with SHA intrinsics	SUN with SHA intrinsics

Table 1 shows five different configurations (A, B, C, D and E) that have been evaluated here. Under each configuration are the CSPs used for the different cryptographic operations involved. For example, with Configuration A, SunPKCS11 is used for encrypting and decrypting keys as part of RSA; so is RSA key generation, AES and SHA. By contrast, with Configuration E, OracleUcrypto is used for encrypting and decrypting keys in the RSA, while SunJCE is used for key generation, and intrinsics within SunJCE and SUN are used for AES and SHA respectively. As mentioned in Section 3.1, both AES and SHA intrinsics are assembly-level implementations of the underlying methods and are embedded in the JVM code generation logic. JVM emits this assembly code while executing appropriate intrinsified methods from the JDK API. The other configurations B, C and D are self-explanatory. Note that, Configuration B is the current out-of-box default configuration for JDK8 on SPARC servers.

The results shown in Figure 3 are from the set of experiments with 1,000 simultaneous user sessions. For each configuration, we run the load with 3 minutes ramp-up, 5 minutes warm-up, 10 minutes steady-state and 3 minutes ramp-down phases. The performance metrics are collected during the steady-state period of the run. Figure 3 shows the measured performance metrics, expressed in terms of the relative ratio to the default configuration (Configuration B).

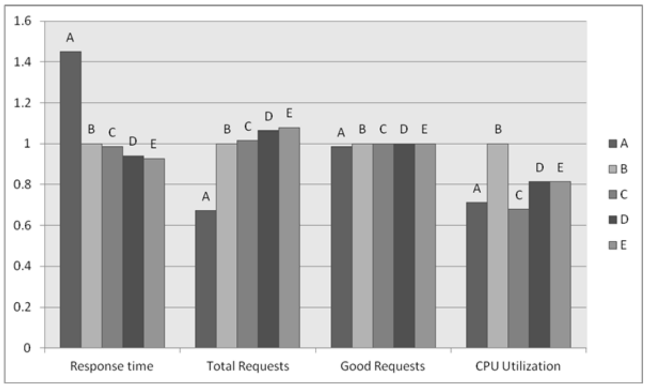


Figure 3: Performance Comparison between CSP Configurations

In the bar chart of Figure 3, lower is better for 'Response Time' and 'CPU Utilization' metrics and higher is better for 'Total Requests' and 'Good Requests' metrics.

Note that the configurations are chosen mainly to explore: 1) the performance advantages of current default configuration over the more traditional PKCS11 provider (Configuration A) and 2) the effect of using AES and SHA intrinsics. We observe that Configuration E provides the lowest average response time and highest total requests. It also has better QoS in the sense that it has higher percentage of good requests.[6] Note that the performance of Configuration E is achieved with better CPU efficiency; it uses less CPU than Configuration B.

From Figure 3, we see that Configuration B outperforms Configuration A. This is mainly because OracleUcrypto CSP's usage of Solaris Ucrypto library accesses unprivileged cryptography instructions directly from user space, which should be compared with PKCS11 that is based on system calls. By contrast, Configuration D outperforms Configuration B because AES and SHA intrinsics avoid the JNI overhead that is inherent in the OracleUcrypto provider. Configuration C stands in the middle between Configurations B and D; its use of intrinsics makes it more efficient in terms of AES and SHA, while its use of PKCS11 for RSA makes it less efficient in comparison to using Ucrypto library for RSA.

The performance contrast between Configurations D and E is an interesting one. They differ only in the CSP used for key generation. Configuration D uses SunPKCS11, while Configuration E uses SunJCE. When SunJCE is used for key generation, SHA intrinsics are used in a critical step (SecureRandom) of the key generation, which leads to better performance than using SunPKCS11 for key generation. JDK bug report JDK-8044659 [7] has the description of the key generation process in this case.

[6] The five configurations are virtually indistinguishable in terms of the percentage of good requests, with only Configuration A slightly lower.

[7] The notation JDK-*nnnnnnn* refers to a bug ID in the OpenJDK bug tracking system.

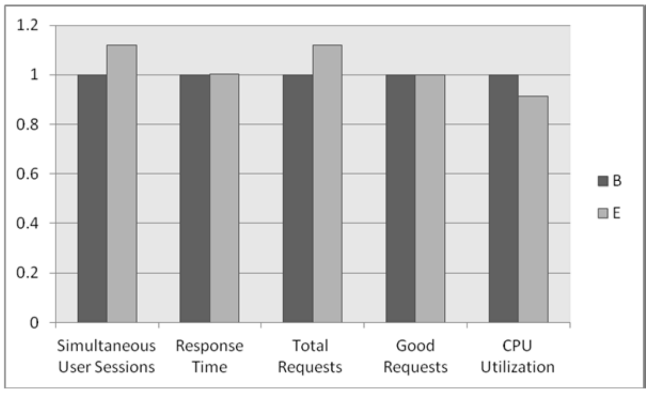


Figure 4: Capacity Comparison between Configurations B and E

With the results in hand, we proceed to verify that the CPU efficiency of Configuration E can be translated into higher number of user sessions. Figure 4 summarizes our results. Here, we increase the number of simultaneous user sessions for Configuration E until the QoS metrics match those for Configuration B. We can see that Configuration E can support 12% more simultaneous user sessions, while keeping the average response time on par with Configuration B. Moreover, Configuration E achieves this with lower CPU utilization.

From the results in this subsection, one can conclude that there is a combination that leads to best EEJS performance among the configurations tested. The combination consists of 1) OracleUcrypto for RSA encryption and decryption, 2) SunJCE for RSA key generation, 3) SunJCE for AES, and 4) SUN for SHA. Note that 2), 3) and 4) directly benefit from intrinsics. There is ongoing effort to make the best combination the default in future JDK releases.

4.2 Advantage of Cryptography Instructions

Configuration E can be further compared with a "pure software" configuration where we intentionally disable crypto intrinsics based hardware acceleration. To test a pure software configuration, we disable AES intrinsics from SunJCE, using "-XX:-UseAESIntrinsics" JVM command line flag. We also disable SHA intrinsics from the SUN provider using "-XX:-UseSHA1Intrinsics -XX:-UseSHA256Intrinsics -XX:-UseSHA512Intrinsics" JVM flags. In addition, we swap in SunJCE provider for RSA encryption and decryption. We add this "pure software" configuration to our configuration mix and call it Configuration F.

From Figure 5 we see that hardware cryptography instructions with Configuration E provide very significant performance advantage over the implementation using regular ISA ("pure software") instructions, as in Configuration F. Configuration F is not able to support the required QoS requirements, while CPU is close to 100%. By contrast, Configuration E has half the response time, utilizing only half of the CPU as that of Configuration F.

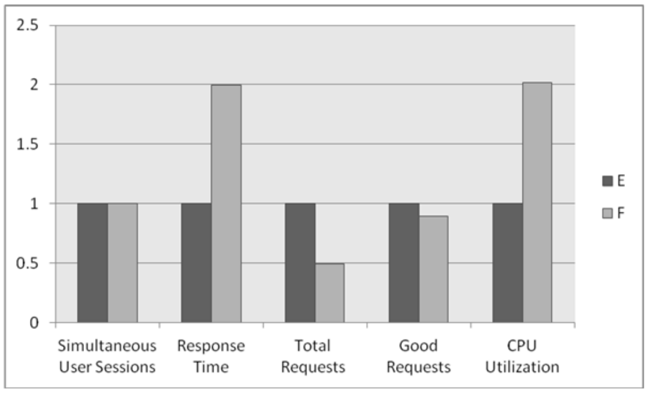


Figure 5: Comparing the implementation using cryptography instructions (Configuration E) with the software-only implementation (Configuration F)

5. PERFORMANCE OPTIMIZATIONS WITH EEJS

By analyzing performance characteristics of EEJS workload we have identified enhancements in JSSE. This section describes the associated performance bottlenecks and possible enhancements. Before describing the JSSE enhancements we provide a brief introduction of JSSE.

The Java Secure Socket Extension (JSSE) enables secure Internet communications. It provides a framework and an implementation for a Java version of the SSL and TLS protocols. It includes functionality for data encryption, server authentication, message integrity, and optional client authentication [14]. JSSE provides both an application programming interface (API) framework and an implementation of that API. The JSSE API supplements the core network and cryptographic services defined by the `java.security` and `java.net` packages by providing extended networking socket classes, trust managers, key managers, SSL contexts, and a socket factory framework for encapsulating socket creation behavior [14].

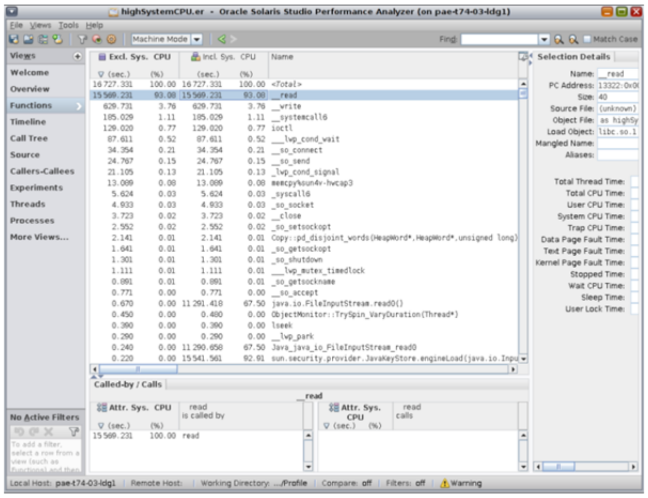
EEJS enables the performance study of related Java packages in an end-to-end application setting. It helps us identify a few enhancements that can help improve performance.

5.1 Buffered Reading of Trusted Certificates

The primary responsibility of the `TrustManager` is to determine whether the presented authentication credentials should be trusted. If the credentials are not trusted, the connection will be terminated [14]. To authenticate the remote identity of a secure socket peer, an `SSLContext` object will be initialized with one or more `TrustManager` objects. If a null `TrustManager` object is passed into the `SSLContext` initialization, a new `TrustManager` will be created.

Running EEJS with JDK 8u40, we noticed high system CPU time (around 70%) on the SUT, as well as high response time, when a null `TrustManager` was passed into the `SSLContext` initialization. During the initialization, a new `TrustManager` was created and initialized with a source of certificate authorities and related trust material, which was obtained by reading from the *cacerts*[8] file and loading the trusted certificates.

[8] $JAVA_HOME/jre/lib/security/cacerts


highSystemCPU.er - Oracle Solaris Studio Performance Analyzer (on pae-t74-03-ldg1)
File Views Tools Help
Machine Mode
Find:
Match Case
Views
Welcome
Overview
Functions
Timeline
Call Tree
Source
Callers-Callees
Experiments
Threads
Processes
More Views...
No Active Filters
Local Host: pae-t74-03-ldg1 | Remote Host: | Working Directory: .../Profile | Compare: off | Filters: off | Warning

```
Excl. Sys. CPU        Incl. Sys. CPU        Name
  (sec.)     (%)        (sec.)     (%)
16 727.331  100.00  16 727.331  100.00  <Total>
15 569.231   93.08  15 569.231   93.08  __read
   629.731    3.76     629.731    3.76  __write
   185.029    1.11     185.029    1.11  __systemcall6
   129.020    0.77     129.020    0.77  ioctl
    87.611    0.52      87.611    0.52  ___lwp_cond_wait
    34.354    0.21      34.354    0.21  __so_connect
    24.767    0.15      24.767    0.15  __so_send
    21.105    0.13      21.105    0.13  _lwp_cond_signal
    13.089    0.08      13.089    0.08  memcpy%sun4v-hwcap3
     5.624    0.03       5.624    0.03  _syscall6
     4.933    0.03       4.933    0.03  _so_socket
     3.723    0.02       3.723    0.02  __close
     2.552    0.02       2.552    0.02  _so_setsockopt
     2.141    0.01       2.141    0.01  Copy::pd_disjoint_words(HeapWord*,HeapWord*,unsigned long)
     1.641    0.01       1.641    0.01  _so_getsockopt
     1.301    0.01       1.301    0.01  _so_shutdown
     1.111    0.01       1.111    0.01  ___lwp_mutex_timedlock
     0.891    0.01       0.891    0.01  _so_getsockname
     0.771    0.00       0.771    0.00  __so_accept
     0.670    0.00  11 291.418   67.50  java.io.FileInputStream.read0()
     0.450    0.00       0.480    0.00  ObjectMonitor::TrySpin_VaryDuration(Thread*)
     0.390    0.00       0.390    0.00  lseek
     0.290    0.00       0.290    0.00  __lwp_park
     0.240    0.00  11 290.658   67.50  Java_java_io_FileInputStream_read0
     0.220    0.00  15 541.561   92.91  sun.security.provider.JavaKeyStore.engineLoad(java.io.Inpu

Called-by / Calls
                         __read
Attr. Sys. CPU         read            Attr. Sys. CPU     read
  (sec.)     (%)       is called by      (sec.)  (%)      calls
15 569.231  100.00     read

Selection Details
          Name: __read
    PC Address: 13322:0x0
          Size: 40
   Source File: (unknown)
   Object File: as highSy
   Load Object: libc.so.1
  Mangled Name:
       Aliases:

     Total Thread Time:
        Total CPU Time:
         User CPU Time:
       System CPU Time:
         Trap CPU Time:
  Data Page Fault Time:
  Text Page Fault Time:
Kernel Page Fault Time:
          Stopped Time:
         Wait CPU Time:
            Sleep Time:
        User Lock Time:
```

Figure 6: Functions ordered by exclusive system CPU time

Further analyzing the Oracle Studio Performance Analyzer [15] profiles as shown in Figure 6, we observed that 93% of the system CPU time was caused by the *__read()* system call, which, in turn, was called by the read() system call. To identify what was being read, we looked at the truss [16] output and observed that reading from the *cacerts* file was done on a single byte basis, as follows. The first four bytes of *cacerts* were the signature bytes, which were obtained by calling `readInt()` of the `DataInputStream`. Inside `readInt()`, it made four calls to `read()`. Similarly, the next four bytes representing the version of the *cacerts* file, as well as the subsequent four bytes representing the number of trusted certificate entries, were read one byte at a time. Furthermore, for each trusted certificate entry, the alias of the certificate, the certificate creation date, and the trusted certificate were also read on a single byte basis.

Performance is improved with a fix for JDK bug JDK-8129634 delivered by Java Engineering team. The fix wraps the `DataInputStream` representing the *cacerts* file in a `BufferedInputStream` object. Instead of retrieving single byte from *cacerts* file, the `BufferedInputStream` object buffers 8192 bytes at a time and the read from the `SSLContext` initialization is now from this buffer. With JDK 8u60 that includes this bug fix, system CPU time drops to around 20% (from the previous 70%). Reducing the number of reads thus makes a significant difference in system CPU utilization.

5.2 Consolidation of the Cacerts Keystore

A keystore is a database for storing key management related data. Information residing in a keystore can be grouped into two categories: key entries and trusted certificate entries [14]. With EEJS running, we observed from the truss output that the *cacerts* file was opened for read repetitively and frequently. The frequency of the file open operations coincided with the frequency of SSL handshakes. We analyzed the call stack and associated source code and made the following observation. A new thread was created for each handshake. Each new thread started a new `SSLContext` initialization. During the `SSLContext` initialization, each `TrustManager` instance read the *cacerts* file and created a `KeyStore` instance. When there were multiple threads, each establishing its own `SSLContext`, there were multiple `KeyStore` instances accessing the common *cacerts* file, causing synchronization and memory overheads.

To optimize the performance here, only one `KeyStore` instance needs to be created. In addition, the *cacerts* file should only be read when there is a modification in this file, instead of being opened and read every time. These changes are tracked by JDK Request for Enhancement (RFE) JDK-8129988.

Without the change, with JDK 8u60 the maximum number of simultaneous user sessions is 490 while maintaining the TIME_GOOD and TIME_TOLERABLE QoS requirements. With the prototype changes for JDK-8129988, the average response time and the number of page views for 490 simultaneous user sessions improve by 9.5%, while meeting the QoS requirements.

5.3 Elimination of Hot Locks

5.3.1 sun.security.ssl.CipherSuite$BulkCipher.isAvailable()

When a socket is created between WLS and BeSim server, WLS gets the default cipher suite list and iterates over the list to check the availability of individual ciphers. From the jstack output of WLS server threads, we noticed that a number of server threads are blocked in the `BulkCipher.isAvailable()` method. It is a synchronized method which checks the availability of a bulk cipher. It is called when a cipher suite list is requested at the initial phase of a handshake. Since each new connection between WLS and BeSim Server involves a full handshake, this method is frequently called.

Within the method, there is a cache, implemented with a hash map. The cache is used to map the bulk cipher to its availability status. However, with JDK 8u60, even though the cache exists, it is cleared every time the method tries to get the cipher suite list, defeating the purpose of the cache. By not clearing the hash map cache the average response time and the number of page views have improved by additional 3% for 490 simultaneous user sessions, and there is an additional 3% improvement on the percentage of TIME_GOOD responses. This change is tracked in JDK RFE JDK-8133070.

5.3.2 SecureRandom.nextBytes()

The `SecureRandom` class provides the functionality of a Random Number Generator (RNG). It differs from the `java.lang.Random` class in that it produces cryptographically strong random numbers. Random numbers are used throughout cryptography, such as for generating cryptographic keys, algorithmic parameters, and so on [8]. There are several different algorithms for `SecureRandom`: `PKCS11`, `NativePRNG`, `SHA1PRNG`, `NativePRNGBlocking` and `NativePRNGNonBlocking`. The PKCS11 algorithm is provided by the SunPKCS11 provider, while the others are provided by the SUN provider. If the entropy gathering device in java.security file is set to file /dev/urandom or file /dev/random, then NativePRNG is preferred to SHA1PRNG. Otherwise, SHA1PRNG is preferred [9]. In this study, NativePRNG is used to generate secure random bytes.

A synchronized method is used to generate the secure random bytes. This method employs a global shared buffer for storing the random numbers read from the /dev/random file. The shared buffer is implemented using a byte array. When a fixed size of secure random bytes is requested, the method first checks if there are any bytes remaining in the buffer. If there are, it reads the requested number of bytes from the buffer and does a 'xor' operation with the input random bytes.[9] If there are not enough bytes remaining in the buffer, the synchronized method replenishes the buffer by reading random numbers from /dev/random. As the buffer is a global shared variable, the 'xor' operation between the buffer and the input has to be synchronized, leading to hot locks. The locks can be alleviated by using a finer grained synchronization.

This change, which is tracked by JDK RFE JDK-8098581, helps improve the throughput of a secure random micro benchmark by more than three times. With this change, the average response time and the number of page views improve by additional 19% for the configuration with 490 simultaneous users.

5.3.3 Registration of CSP Services

We also noticed a hot lock when getting services from a CSP. By design, there is a map maintaining the services offered by each CSP, and the constructor for a CSP object should record the supported services in this map. When an application gets service from a particular CSP, it looks up the map to find the supported services. However, in our EEJS experiments, we found that with the exception of the PKCS11 provider, other CSPs do not register the services in their respective constructor. As a consequence, when an application looks up the map, it always returns a null map. When the application finds a null map, it falls back to check a legacy map for services. It first checks a `transient` variable to see if the legacy map has been modified and then get the services from the legacy map.

Ideally, a single map should suffice. When service map is null, falling back to legacy map incurs the overhead of exercising extra code path. In addition, as the method for getting services is synchronized, it causes performance regression when multiple threads are involved. This issue is currently tracked by JDK RFE JDK-8133906.

5.4 Performance Optimization Results

We measured the combined effect of all the changes mentioned in Section 5. With the officially released binaries of JDK 8u60, 490 simultaneous user sessions can be achieved while maintaining QoS requirements. With the changes described in this section, 800 simultaneous user sessions can be achieved. This is a 1.6 times improvement in terms of throughput. Table 2 shows these performance optimization results. If we restrict the load to 490 simultaneous user sessions, the aforementioned changes lead to an improvement of 34.6% in average response time, 34.6% improvement in page views, and 2.36 times improvement in login latency.

6. SUMMARY

In this paper, we describe the motivation of EEJS workload, its use in conducting performance evaluation, and the performance optimization results from it. Performance evaluations using JDK 8u40 running on SPARC servers lead to the conclusion that using JVM intrinsics for AES and SHA ciphers and using OracleUcrypto for RSA encryption and decryption provides the best performance among the configurations evaluated. There is ongoing work around out-of-the-box CSP configuration to provide the best combination of CSPs based on the underlying platform.

[9] The input random bytes were generated from an earlier operation invoking a secure random algorithm such as SHA1PRNG.

Table 2: Performance results comparison between JDK 8u60 and improvements from Section 5

	JDK 8u60	With the changes mentioned in Section 5.2, Section 5.3.1, and Section 5.3.2
Normalized Response time	1	0.742
Normalized total number of requests	1	1.345
Normalized login latency	1	0.423
Good	95.6%	99%
Tolerable	99.6%	99.8%
Fail	0.4%	0.2%

Since the EEJS workload is heavy on the SSL handshakes, it is also used to demonstrate the clear advantage of hardware cryptography acceleration. This is accomplished by comparing a configuration using hardware cryptography with one that uses conventional instruction set.

We have identified a number of JSSE enhancements by analyzing performance characteristics of EEJS workload. Several of these enhancements are in the process of being incorporated into future JDK releases.

7. ACKNOWLEDGMENTS

The authors would like to thank Oracle Java Security team and Solaris Security team for their support and encouragement, and our colleagues Jan-Lung Sung, Richard Smith and Andy Bowers for their valuable critiques to improve the paper. We would also like to thank the anonymous reviewers for their constructive feedback to help further improve the paper.

8. REFERENCES

[1] Kongetira, P. 2004. A 32-way Multithreaded SPARC Processor. In *Hot Chips 16*.

[2] Spracklen, L. 2009. Sun's 3rd generation on-chip UltraSPARC security accelerator. In *Hot Chips 21*.

[3] Shoaib Bin Altaf, M. and Wood, D.A. 2014. LogCA: A Performance Model for Hardware Accelerators. In *Computer Architecture Letters*. Volume: PP, Issue: 99 (Sep. 2014).

[4] SPECweb2005 Release 1.20 Benchmark Design Document. https://www.spec.org/web2005/docs/designdocument.html

[5] Trademark for the SPEC Benchmark. https://www.spec.org/spec/trademarks.html

[6] SPEC Fair Use Rule. Academic/research usage. http://www.spec.org/fairuse.html#Academic

[7] Apache HTTP Client. https://hc.apache.org/httpcomponents-client-ga/

[8] Java Cryptography Architecture (JCA) Reference Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

[9] Java Cryptography Architecture Oracle Providers Documentation for JDK 8. http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html

[10] Rivest, R.; Shamir, A.; Adleman, L., 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21 (2): 120–126.

[11] Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication

197. United States National Institute of Standards and Technology (NIST). October, 2012.

[12] FIPS 180-4: Secure Hash Standard. United States National Institute of Standards and Technology (NIST). August 2015.

[13] JDK 8 PKCS#11 Reference Guide. http://docs.oracle.com/javase/8/docs/technotes/guides/security/p11guide.html

[14] Java Secure Socket Extension (JSSE) Reference Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

[15] Oracle Solaris Studio Performance Analyzer. http://www.oracle.com/technetwork/server-storage/solarisstudio/features/performance-analyzer-2292312.html

[16] Man pages for truss. http://docs.oracle.com/cd/E23823_01/html/816-5165/truss-1.html