

# The $\mu$ KVM Benchmark Result and Analysis

## 1 Functionality Benchmark

To verify that the user-level functionality of the J2SDK is preserved in the  $\mu$ KVM, we benchmarked on the io, networking and thread due to their implementation completeness. We used the mauve test suite for this functionality tests. The test result in Table 1 shows that our implementation meets the mauve specification as well as the J2SDK implementation

The difference in the total number of tests performed on the J2SDK and the  $\mu$ KVM prototype results from tests grouping: one test failure may cause skipping of the rest of tests in the same group. All the failed tests on our packages and those on peer Java packages are either the same or simply different on exception types. For example, writing to a closed socket in Java throws an `IOException`, while the  $\mu$ KVM throws a null pointer exception because the connection associated with that socket has been disconnected and becomes null.

Package	J2SDK			$\mu$ KVM		
	Failed	Succeeded	Total	Failed	Succeeded	Total
File IO	9	648	657	9	648	657
Networking	9	365	374	8	376	384
Thread Class	0	85	85	0	85	85
total	18	1098	1116	17	1109	1126

**Table 1.** Results of mauve tests

## 2 Performance Benchmark

The performance benchmark tests were run with security switched off. We will profile security overhead by turning the security on in the next set of benchmark.

All test programs were performed on Sun Blade 50 running Solaris 2.5 with 650MHz processor and 256Mb memory. Comparisons between the J2SDK and the  $\mu$ KVM were reflected in percentages of their differences (Diff), which were calculated as  $\text{Diff} = (\mu\text{KVM} - \text{J2SDK})/\text{J2SDK} * 100\%$ . The memory overhead was profiled by using the `-Xrunhprof:heaps=sites` option of the Java interpreter.

### 2.1 Performance of File Operations

The test program simply opened 500 or 1000 files and recorded the total time spent on the operations. From Table 2 we can see that the  $\mu$ KVM introduces a very small overhead in file opening. Part of this overhead came from the fact that the  $\mu$ KVM had to spend extra time in loading io library since it was no longer preloaded as system classes. The memory overhead was due to mapping connections into delegations between two java connector objects, which led to the constructions of more objects. In any case, the overhead is reasonably small that would barely impact the performance of a real application.

As shown in Table 3, different block sizes were used to read/write the 5M data. Benchmark results indicate that the difference between  $\mu$ KVM and J2SDK on file read/write is not statistically significant.

File Number	File Open Time (ms)			Memory (byte)		
	J2SDK	$\mu$ KVM	Dif(%)	J2SDK	$\mu$ KVM	Dif(%)
500	395	407	2.98	2,385,752	2,458,072	3.03
1000	847	875	3.33	2,408,112	2,496,888	3.69

Table 2. File Open Benchmark

BlockSize (byte)	Read (ms)			Write (ms)		
	J2SDK	$\mu$ KVM	Dif(%)	J2SDK	$\mu$ KVM	Dif(%)
64	877.5	874.4	-0.35	1,136.3	1,125.3	-0.97
128	466.1	469.4	0.71	688.1	688.2	0.01
256	264.8	266.0	0.45	488.8	484.5	-0.88
512	158.5	159.2	0.44	322.9	318.6	-1.33
1024	104.4	103.9	-0.48	234.0	230.7	-1.41

Table 3. File Read/Write Benchmark

### 2.2 Network Performance Benchmark

A typical TCP/IP network application, consisting of a client and a server, was set up in a Local Area Network. We measured the time the client spent on

Message Size x Num	Memory (byte)			Transfer Time Per Packet(ms)		
	J2SDK	$\mu$ KVM	Diff(%)	J2SDK	$\mu$ KVM	Diff(%)
64 x 16384	10,778,808	10,791,640	0.12	1.34	1.36	1.77
128 x 8192	9,724,792	9,736,928	0.12	2.65	2.71	2.36
256 x 4096	9,215,144	9,222,768	0.08	5.42	5.49	1.38
512 x 2048	8,959,760	8,979,464	0.22	10.59	10.83	2.24
1024 x 1024	8,804,448	8,857,112	0.60	21.42	21.20	-1.01
2048 x 512	8,742,032	8,794,616	0.60	43.24	43.77	1.23
4096 x 256	8,715,408	8,767,992	0.60	86.51	88.14	1.89
8192 x 128	8,724,672	8,756,664	0.37	171.71	175.68	2.31
16384 x 64	8,744,664	8,757,432	0.15	335.18	346.47	3.37

**Table 4.** Network Benchmark (a)

Packet Number	Memory (byte)			Transfer Time Per Packet(ms)		
	J2SDK	$\mu$ KVM	Diff(%)	J2SDK	$\mu$ KVM	Diff(%)
64	2,672,696	2,857,020	6.90	22.82	22.62	-0.88
128	3,229,464	3,418,824	5.86	20.24	20.81	2.84
256	4,001,921	4,016,608	0.37	21.27	21.53	1.20
512	5,601,536	5,613,800	0.22	21.32	21.40	0.39
1024	8,804,448	8,857,112	0.60	21.42	21.20	-1.01
2048	15,229,616	15,242,432	0.08	21.57	21.99	1.99
4096	28,084,752	29,097,080	0.04	21.59	22.10	2.36

**Table 5.** Network Benchmark (b) Packet size = 1024 bytes

sending packets to the server with two benchmarks. In the first benchmark, the total size of data sent to the server was fixed to be 1M bytes with packet size ranging from 64 to 16384 bytes. In the second benchmark, the packet size was fixed to 1024 bytes while the packet number varied from 64 to 4096. Data in Table 5 show that the  $\mu$ KVM scales as well as the J2SDK.

### 2.3 Thread Benchmark

We measured the time used for creating a thread and starting the thread. In worst case, the  $\mu$ KVM slowed down 14.29% in creation time and 13.64% in starting time. The main source of the overhead came from the fact the  $\mu$ KVM prototype was built by adding new classes and code to the original JVM and J2SDK for comparison reason, which cause the loss of performance in the  $\mu$ KVM. Further cpu profiling with the java option `-Xrunhprof:cpu=time` identified that the extra overhead in  $\mu$ KVM can be removed if we implemented the  $\mu$ KVM ground up with only connector-based threading model.

## 3 Security Overhead Benchmark

The benchmarks above focused on raw performance and therefore excluded the additional overhead incurred from access control checks. We now assess the

Thread Number	Memory (bytes)			Creation Time(ms)			Duration Time (ms)		
	J2SDK	$\mu$ KVM	Diff(%)	J2SDK	$\mu$ KVM	Diff(%)	J2SDK	$\mu$ KVM	Diff(%)
10	2,370,520	2,402,968	1.37	1.17	1.33	14.29	11.00	12.38	12.63
30	2,378,175	2,420,843	1.79	1.00	1.00	0.00	12.77	14.24	12.36
60	2,387,044	2,444,158	2.39	1.00	1.00	0.00	25.54	29.03	13.64
100	2,396,531	2,473,476	3.21	1.00	1.03	3.70	65.34	73.61	12.67

**Table 6.** Thread Benchmark

J2SDK vs.  $\mu$ KVM security overhead. Benchmark programs used in this section were the same as the ones in 2.1 but with security enabled.

Table 7 shows the overhead resulting from running the security architecture on the J2SDK and the  $\mu$ KVM respectively. On the J2SDK, performance suffered greatly. The speed for opening 500 or 1000 files slowed down by 136.45% and 77.33% respectively. Security also introduced overhead to the  $\mu$ KVM, but much less severe than it did to Java, i.e. 68.5% and 39.89% in the same two test cases.

Memory consumption increased on both the J2SDK and the  $\mu$ KVM in the security overhead benchmarks, as shown in Table 8. E.g., when 1000 files were opened, Java experienced 43.28% memory increase while  $\mu$ KVM got 35.95%.

Table 9 is the head-to-head comparison between the security overheads of the J2SDK and the  $\mu$ KVM. It clearly shows the  $\mu$ KVM outperformed the J2SDK in both speed and memory. Precisely, the  $\mu$ KVM was 26.66% faster than Java when 500 files were opened and 18.51% faster when the file number increased to 1000. As for memory consumption, the  $\mu$ KVM consumed only 0.01% more memory in the 500 case while 1.63% less in the 1000 case.

File Number	J2SDK File Open Time (ms)			$\mu$ KVM File Open Time (ms)		
	-Security	+Security	Diff(%)	-Security	+Security	Diff(%)
500	395	934	136.45	407	686	68.55
1000	847	1,502	77.33	875	1,224	39.89

**Table 7.** Security Overhead on File Open

File Number	J2SDK Memory (byte)			$\mu$ KVM Memory (byte)		
	-Security	+Security	Diff(%)	-Security	+Security	Diff(%)
500	2,385,752	2,968,824	24.44	2,458,072	2,969,224	20.79
1000	2,408,112	3,450,360	43.28	2,496,888	3,394,200	35.94

**Table 8.** Security Overhead on Memory

File Number	File Open Time (ms)			Memory (byte)		
	J2SDK	$\mu$ KVM	Diff(%)	J2SDK	$\mu$ KVM	Diff(%)
500	934	686	-26.66	2,968,825	2,969,224	0.01
1000	1,502	1,224	-18.51	3,450,360	3,394,200	-1.63

**Table 9.** Security Overhead Comparison