

Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread

Minyoung Sung, Soyoung Kim, Sangsoo Park,
Naehyuck Chang, and Heonshik Shin
School of Computer Science & Engineering
Seoul National University, Seoul 151-742, Korea
minyoung@cslab.snu.ac.kr

Abstract

Despite the portability and platform-independence of Java programs, their performance depends on the threading mechanisms of the host operating system. In this paper, we measure the performance of Java threads for two different multi-threading implementations, Linux Thread and Green Thread, using PersonalJava (TM) on a Linux-based platform. The experimental results show the relative strengths and weaknesses of the two threading mechanisms with respect to synchronization overhead, I/O efficiency, and thread control.

Keywords: Java threads, Performance evaluation, Embedded system

1 Introduction

Advances in computer technologies have made it economic to adopt Java as the execution platform for embedded devices such as set-top boxes, PDAs, and Internet TVs. Recently, Sun Microsystems released PersonalJava [8] for such embedded applications and is now applying it to various devices. Java provides several benefits, including platform independence, strong type checking, dynamic class loading, and extensive library support. In particular, its inherent support for multi-threading makes

it easy to design a sophisticated application that requires complicated inter-thread synchronization.

Despite the interoperability and platform independence of Java programs, their performance depends on the way in which a Java VM (Virtual Machine) maps Java threads to system threads [5]. Experimental results show that execution times vary by up to 48%, depending on the multi-threading mechanism. Therefore, for the design of embedded Java applications, it is essential to understand how the timing characteristics of Java threads depend on the multi-threading mechanism.

In this paper, we evaluate the performance of Java threads for two different multi-threading implementations: Linux Thread and Green Thread [9]. For this purpose, we have set up a PersonalJava runtime environment on our experimental Linux-based platform, and investigated the performance of Java threads with respect to context switching time, performance of thread control, synchronization overhead, and I/O efficiency.

2 Implementation of Java threads

Within Java VM, a Java thread abstracts a single flow of control and is represented by the class `java.lang.Thread`. Table 1 summarizes several important methods of the `Thread` class [4]. A Java thread can be realized either as a user-level thread or as a kernel-level thread depending on the Java VM implementation. Our PersonalJava run-time system for Linux offers two

Table 1: Summary of Java thread methods.

Methods	Description
<code>start()</code>	Activates the thread. This call makes the thread start execution from the beginning of its <code>run()</code> method
<code>stop()</code>	Terminates the thread
<code>join()</code>	Causes the calling thread to wait for completion of the called thread
<code>suspend()</code>	Suspends the called thread
<code>resume()</code>	Resumes the suspended thread
<code>yield()</code>	Causes the current thread to yield control to another thread
<code>sleep()</code>	Causes the current thread not to be scheduled for the specified length of time

different multi-threading mechanisms: Green Thread and Linux Thread. Green Thread is a user-level thread package which is included in the PersonalJava distribution. With Green Thread, all the Java threads are mapped to a single system task. Since threads are managed within a task, thread management can be performed with relatively little overhead. This contrasts with Linux Thread that relies heavily on system calls to implement the methods. Linux Thread allows kernel-level multi-threading and provides the thread interface for the POSIX 1003.1c specification [3]. With Linux Thread, each Java thread is associated with a separate kernel-level thread, or a task.

Using either multi-threading mechanism, it is easy to port the Java thread on to the experimental platform. With Green Thread, it is only necessary to modify some of the C macros related to context manipulation, so that they fit the register file of target processor. With Linux Thread, the port is straightforward except for the `suspend()` and `resume()` methods, which require special treatment since the POSIX thread specification does not support suspending or resuming threads other than the calling thread. To get around this problem, we chose an unused Linux signal and employed it for notification of the suspend and resume events, thus making the appropriate threads suspend or resume voluntarily.

Table 2: Specification of the experimental system.

CPU	Motorola MPC 860 50Mhz
Main memory	64 MB SDRAM
Auxiliary memory	8 MB flash memory
Cache memory	4 KB for instruction and 4 KB for data caches
Communication	1 serial port and 1 Ethernet I/F
Display	640x480 color LCD
OS	Linux version 2.2.13
Java VM	PersonalJava version 1.1.3

3 Performance results

We set up a PersonalJava run-time environment on a PowerPC-based Linux system, having chosen Linux as the operating system since it is widely employed for embedded applications and its readily available source code enables a detailed analysis. Table 2 illustrates the specification of the experimental system. Our Linux system is provided with a serial port and an Ethernet interface. The serial port is used to communicate with the shell and to launch Java VM on the platform. The Ethernet interface is directly connected to a desktop PC so that the Linux system mounts its root file system from the PC via NFS. This facilitates the collection of experimental data.

For accurate measurement, we use the CPM (Communication Processor Module) timer provided by the timer chip [6]. In our experiments, it was configured to provide a resolution of 0.1 microsecond.

Context switching overhead (`yield()` method)

The basic policy for scheduling Java threads is the priority-based round-robin policy [2]. The thread with the highest priority is always scheduled ahead of any other lower-priority threads, and threads of the same priority are scheduled in a round-robin order. Green Thread strictly follows this policy. With Linux Thread, this policy can be realized by setting the scheduling policy to `SCHED_RR`, as specified in the POSIX real-time options¹.

¹Contrary to our expectations, the task scheduler in Linux version 2.2.13 just ignores the `SCHED_RR` option; so we patched the scheduler appropriately to support real-time scheduling.

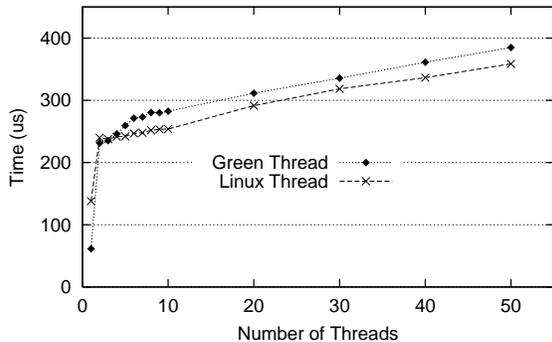


Figure 1: Context switching time using `yield()` method.

Fig. 1 shows the overhead for context switching of Java threads. The context switching time is measured by creating multiple Java threads of equal priority and making them call the `yield()` method repeatedly. As shown in the figure, Linux Thread is relatively efficient in context switching. The context switching time of Green Thread is longer by up to 11%, except when there are only a few threads. Green Thread uses the `setjmp()` and `longjmp()` functions for storing and loading thread context. It spends most of the switching time in these functions when there are only two or three threads. As the number of threads increases, however, the overhead for thread management becomes substantial. Note that in case of a single thread, Green Thread shows excellent performance, because there is no need for context manipulation. This contrasts with Linux Thread, which invokes the `sched_yield()` system call regardless of the number of threads.

Performance of thread control

Table 3 summarizes the average time taken for several important methods of thread control. We see that Green Thread shows better performance than Linux Thread. For the `start()` method, in particular, Green Thread is 7.87 times faster than Linux Thread. A Java thread is created when the new construct is applied to a subclass of `Thread`. Then, the Java VM creates a new `Thread` object and initializes the detailed internal representation of a Java thread. The creation time in Table 3 refers to the time needed for these processes. Once created, the Java thread

Table 3: Time taken for several important thread controls.

	Green Thread	Linux Thread
creation	1661.14 μ s (0.77)	2157.02 μ s (1.30)
<code>start()</code>	721.07 μ s (0.13)	5674.23 μ s (7.87)
<code>stop()</code>	654.48 μ s (0.63)	1033.92 μ s (1.58)
<code>join()</code>	217.26 μ s (0.77)	282.04 μ s (1.30)
<code>suspend()</code>	685.6 μ s (0.84)	813.1 μ s (1.19)
<code>resume()</code>	771.0 μ s (0.93)	826.5 μ s (1.07)

is activated when some other thread calls the `start()` method of the `Thread` object. The activated thread then starts executing from the beginning of its `run` method. The time taken by the `start()` method is measured from the time when a thread calls the `start()` method to the time when the new thread starts execution of the `run()` method. With Green Thread, thread activation is straightforward and incurs relatively little overhead. With Linux Thread, however, Java VM must create a corresponding execution entity, i.e., a task within the kernel. After creating a task, the Java VM waits for the newly created task to inform it that self-initialization is complete. This additional overhead explains the increased activation time of Linux Thread. The latency for thread activation is one of the most important performance metrics for embedded applications [7]. Therefore, to achieve a tolerable response with Linux thread, it is strongly recommended that a thread is in an activated state at initialization, rather than creating it on demand

Synchronization (locking overhead)

In Java, a method or a code block may be declared `synchronized`. This means that an object will be locked while the corresponding method or the code block executes. Since the Java language explicitly supports multithreading, synchronization over objects occurs frequently in order to make class libraries thread-safe. Therefore, synchronization is considered to be a critical performance factor with Java VM [1]. Fig. 2 shows the time taken to lock an unlocked object (`sync`) and to lock an object already locked by the calling thread (`nested sync`). Green Thread shows excellent synchronization performance: its lock operation is two to four times faster than that of Linux Thread. As described later in this section, this per-

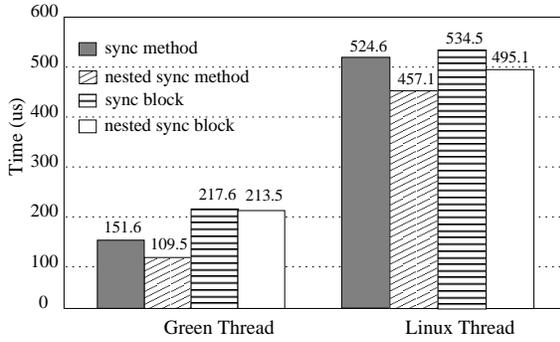


Figure 2: Lock overhead

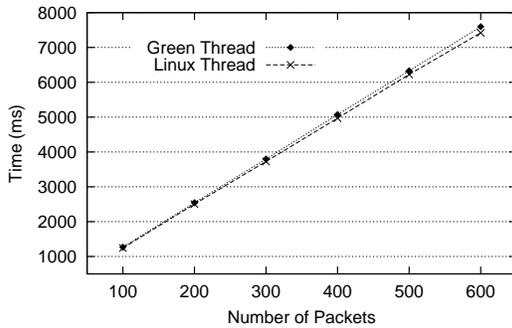


Figure 3: Network I/O performance

formance gain by Green Thread leads to a reduced overall execution time. Within Java VM, synchronization is performed by entering and exiting a monitor. The striking difference in the synchronization overhead lies in the way that a monitor is implemented. Monitors within Linux Thread are built upon the mutex facility. Although mutex uses the `testandset` instruction for efficiency, the lock overhead is higher than that of Green Thread, in which the process of entering a monitor simply corresponds to incrementing a counter in the user code.

I/O performance

One of the most serious problems with user-level multi-threading is that any thread may block all the threads belonging to the same system task by making a blocking

Table 4: Execution time for several Benchmarks.

Benchmarks	Green Thread	Linux Thread
HelloWorldApp	5.085 s	6.211 s
CaffeinMark	31.929 s	33.652 s
JavaPerformance	42.769 s	63.204 s
Yield	38.066 s	41.752 s
Ping	39.345 s	41.157 s

system call. Green Thread avoids this situation by the use of nonblocking I/O and asynchronous I/O facilities. Fig. 3 shows the round-trip latencies for 4096 byte-sized `DataGramPackets` as the number of packets increases. Due to the implementation complexity of I/O operations, Green Thread shows slightly increased latency. Green Thread has the limitation that it can not be used for I/O devices that do not support asynchronous or nonblocking I/O facilities.

Execution time for several benchmarks

Table 4 presents the execution time for several well-known Java Benchmarks. The execution times vary by up to 48%, depending on the multi-threading mechanism. Green Thread showed better performance throughout, mainly because its synchronization technique is more efficient than that of Linux Thread.

4 Conclusion

The performance of Java programs depends on the multi-threading mechanisms of the host operating system. In this paper, we presented a comparative analysis of embedded Java thread for two multi-threading mechanisms: Green Thread and Linux Thread. By experiment, we investigated the performance of these two Java threading mechanisms in terms of synchronization overhead, I/O efficiency, and the performance of thread control. The experimental results show that Green Thread significantly outperforms Linux Thread in thread control and synchronization. In particular, Green Thread is 7.87 times faster for thread activation and 4.17 times faster for method synchronization. In contrast, Linux Thread shows slightly better performance in I/O and context switching operations.

The results presented in this paper provide insights which we believe will assist in the design of multi-threading implementations of Java in embedded environments.

References

- [1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, Thin Locks: Featherweight Synchronization for Java, Proc. ACM Conf. on Programming Language Design and Implementation, 1998, pp.258–268.
- [2] D. J. Berg, Java Threads : a White paper, Sun Microsystems Computer Corporation, 1996
- [3] D. R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.
- [4] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison Wesley, 1996.
- [5] Y. Gu, B. S. Lee, and W. Cai, Evaluation of Java Thread Performance on Two Different Multithreaded Kernels, Operating Systems Review 33 (1) (1999) 34–46.
- [6] Motorola Inc., MPC860 PowerQUICC User’s Manual, 1998.
- [7] S. Oikawa and H. Tokuda, Efficient Timing Management for User-Level Real-Time Threads, Proc. IEEE Real-Time Technology and Applications Symposium, 1995, pp.27–32.
- [8] Sun Microsystems Inc., PersonalJava (TM) Technology White Paper, SunSoft, 1998.
- [9] Sun Microsystems Inc., Java on Solaris 2.6: A White Paper, Sunsoft, 1997.