Self-Tuning Synchronization Mechanisms in Network Operating Systems

Yuval Hershko * Daniel Segal† Hadas Shachnai ‡

Department of Computer Science
The Technion, Haifa 32000, Israel

1 Introduction

A most challenging direction in the development of operating systems towards the next millennium, is the usage of *self-tuning* modules, at various levels of the system. Self-tuning can come into play in many aspects of system management: in the broadest sense it implies that computation, as well as storage of data and allocation of resources can be tuned, by monitoring the system behavior throughout its operation.

In this work we discuss the design and implementation of self-tuning modules: such modules (which may be, e.g. a scheduler, a file system manager or any system call), should be able to adjust to changes in the workload or the type of applications/users in the system. This means, that at different periods of time these modules may use different algorithms for implementing their main functions. The goal is to employ self-tuning for improving the performance of the system. Our two main performance measures are system *throughput*, and the *average response time*.

We focus on self-tuning synchronization mechanisms. In a network operating system various processes may try to access simultaneously shared data, stored at a file server. Mutual exclusion is achieved by using hardware synchronization primitives. One common approach in synchronization is *blocking*, that is, at any time only one of the contending processes can access the shared data, and the other processes are blocked, waiting for their turn. A process can access the shared data only after it has acquired a lock. This approach is sometimes called *pessimistic*, as it assumes that atomicity may be violated at any time (e.g., by an interrupt), therefore a lock on the shared data guards against this potential violation, whenever the atomic operation is executed. The *Test&Set (TS)* primitive is typically used as the lower-level operation supporting blocking mechanisms [4]. *Wait-free* mechanisms allow each of the contending processes to access the shared data, and when several processes try to update the data simultaneously, only one process succeeds, and the other processes have to repeat their update operation. Thus, no process is ever blocked while trying to access the shared data. Indeed, wait-free mechanisms rely on the *optimistic* approach, that atomic sequences are rarely interrupted, and thus, the overhead of implementing a global lock can be avoided. The *Compare & Swap (CS)* used on Motorola 68000 and the *Load_Linked* and *Store_Cond (LL/SC)* incorporated into the MIPS II architecture, Digital's Alpha and IBM's PowerPC (see in [5]) are wait-free synchronization mechanisms.

Previous work study the settings for which these two types of mechanisms are most suitable. In particular, it was shown, that blocking mechanisms are preferable in use, when either (*i*) processes perform long sequences of operations on the shared data, or (*ii*) most of the accesses result in changes in the data (due to write operations). On the other hand, when each process performs only a short sequence of operations, system throughput increases when wait-free mechanisms are used (see, e.g. [1, 2]).

Building on these findings, we study the performance of a system having a dual-mode synchronization module: at certain times this module uses a blocking mechanism, and at other times – a wait-free mechanism. We propose two rules for switching between these modes: the Timeout rule and the Queue-length rule. Our simulation study shows, that sensitive self-tuning (as implemented by the Queue-length rule) can substantially increase the throughput, with a corresponding decrease in the average response time; both rules can be easily implemented on modern machines, which typically support blocking primitives, as well as wait-free primitives.

2 Model and Synchronization Mechanisms

Consider a computer network, in which shared data is stored at a file server. Any client can access this data (for read/update operations) via the network, by sending a request to the server to provide this type of service. We call each of these requests a *transaction*. A client connects via the network to the server, and then sends a

*corwin@tx.technion.ac.il.
†segald@techst02.technion.ac.il
‡hadas@cs.technion.ac.il.

set of transactions sequentially: a new transaction will be sent once the previous one has been completed by the server.

A transaction is a sequence of operations. The length of the transaction is given by the size of this sequence; when a transaction does not include any write operation, it is called a *read (R)* transaction, otherwise it is a *write (W)* transaction.

Indeed, if at some time the server executes several W transactions concurrently, the result may be invalid. Thus, we require that the execution of each transaction will be *atomic* (that is, a transaction will either be executed to successful completion, or not started).

The server can switch between two types of synchronization mechanisms: a *blocking* mechanism and a *wait-free* mechanism. Incoming requests are sorted by length: *short (S)* transactions will be executed using the *wait-free* mechanism; *long (L)* transactions will be executed using the *blocking* mechanism. The effect of the R/W ratio will be explored, by varying the R/W ratio.

2.1 Rules for Switching Modes

We consider two rules for switching between synchronization mechanisms (*modes*): (i) Timeout – predefine a time period for each mode (e.g. t_b for *blocking* mode and t_f for *wait-free*). The server will alternate between these modes, running the blocking mode for t_b seconds, and then switching for t_f seconds to the wait-free mode. (ii) Queue-Length – maintain a queue of transactions waiting to be executed, for each mode; the server switches to a different mode, when the number of transactions waiting for execution in this mode exceeds a predetermined maximum.

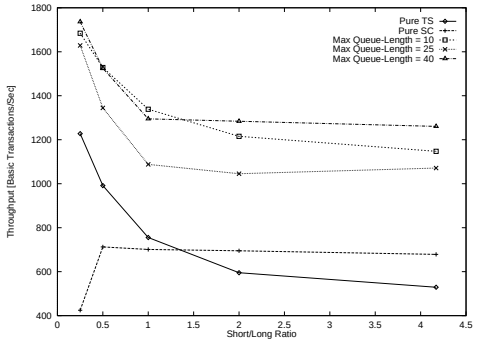

Figure 1: Throughput vs. S/L Ratio under the Queue-Length Rule

3 Simulation Results

We simulated a system having one transaction server and fifty clients. We used two Sun Ultra machines, running Solaris 2.6 and connected on an Ethernet: the server ran on one machine, and the clients on the other; we used as communication protocol the TCP/IP.

For each switching rule we measured the throughput of the server and the average response time. Our results are given by reference to a *basic transaction*, whose length is normalized to one. The wait-free mode was simulated using the LL/SC mechanism; for the blocking mode we used the TS.

We varied in the experiments either the R/W ratio or the S/L ratio. When we varied the R/W (S/L) ratio, the S/L (R/W) ratio remained fixed and equal to 1. The threshold value, for separating between short and long transactions, was chosen to be 50.

The Timeout rule was simulated using a timeout period of 0.2 sec. for the LL/SC mode; the period for the TS mode was changed. We found that switching by an insensitive rule, such as the timeout, typically does not improve upon the basic mechanisms: it outperforms the basic mechanisms only in rare and special cases, while in many cases it does even worse than the basic mechanisms.

In Figure 1 we plotted the throughput vs. S/L ratio, under the Queue-length rule. We note, that Queue-length guarantees a full-spectrum performance-gain of $70-85$ percent, compared to the basic mechanisms. Also, by using maximum queue lengths of 10 and 40 we increase the throughput (compared, e.g. to the length 25): with these values, a large number of transactions has to accumulate in a queue, before the server switches to the corresponding mode. This reduces the total number of switches. Moreover, any switch results in a service of more transactions.

3.1 Conclusions

We compared the Timeout rule, that switches between modes at predetermined points of time, to the Queue-length rule, which sensitively tunes the synchronization mode used by the server. Switching between modes by Timeout can result in some improvement over a single synchronization mode. In contrast, the adaptive characteristics of Queue-length rule enable to handle a variety of inputs in real time, while maintaining the throughput high. Thus, when comparing the two rules (with respect to our two performance measures), Queue-length is a sure winner.

References

[1] B. N. Bershad. "Practical Considerations for Non-Blocking Concurrent Objects", *Proceedings of the Distributed Computing Systems Conference (DCS)*, May 1993.

[2] M. Herlihy, J. Eliot and B. Moss, a "Transactional Memory: Architectural Support for Lock-Free Data Structures". *Proceedings of ISCA '93.*

[3] A. S. Tannenbaum, *Distributed Operating Systems.* Prentice-Hall Int., Editing, 1995.

[4] M. Milenkovic. *Operating Systems : Concepts and Design.* McGraw-Hill Inc, pages 104–112, 1992.

[5] R. L. Sites. *Alpha Architecture Reference Manual.* Digital Press, Maynard, MA, 1992. MIPS Computer Company. The MIPS RISC architecture.