# Report on "Using Threads in Interactive Systems: A Case Study"

Franklin Chen

January 28, 1999

**Abstract**

The point of the Hauser paper was to analyze, as an example of large, real-life, thread-based interactive systems, two systems developed at Xerox, called Cedar and GVX. The authors wished to examine and classify common features and uses of threads, focusing on their role in program structure (not multiprocessing), and they particularly wanted to consider how the need for interactive usability, e.g., in window systems, posed challenges to both programmers and implementors of a thread library.

## 1 Dynamic classification

The authors used benchmarks covering some aspects of real-life operations (e.g., compile, make, format document, scroll window) to gather statistics about the dynamic behavior of the threads used in the two systems. They arrived at a classification based on lifetime:

**Eternal** Run forever, waiting on condition variable. Does almost no real work itself.

**Worker** Perform some computation.

**Transient** Has short life; forked off by long-lived thread.

## 2 Static classification

A different classification was also extrapolated from the data, using static information about thread usage, gleaned from examining the source code:

**Work deferrer** The most common use of forking; the purpose is to reduce latency when no return value is required.

**Pump** A component of a pipeline, transforming input to output, e.g., between a bounded buffer and an external device.

**Slack process** A kind of pump; adds latency (by merging and replacing data) in order to reduce total work.

**Sleeper** Repeatedly waits for an event and runs something.

**One-shot** Special case of sleeper, in which the thread goes away after just one wake and run.

**Deadlock avoider** Spawned off to avoid violating lock order constraints without having to manually perform an unwinding.

**Task rejuvenation** New thread to recover from a bad state, e.g., uncaught exception, stack overflow. This is a controversial use because it can mask design flaws that led to the bad state in the first place.

**Serializer** Consists of a queue and a thread processing serially from the queue.

**Concurrency exploiter** For multiple processors.

**Encapsulated fork** A code modularity pattern; not really an independent thread paradigm.

## 3 Issues

An examination of the systems revealed issues on both the programmer and implementor ends involving pitfalls and trade-offs involving simplicity/concurrency and actual efficiency.

## 3.1 Thread use

Sleepers, one-shots, pumps without critical timing constraints, and work deferrers were easy to use effectively, as they require little interaction between threads.

One example of a hard problem is the use of threads with X. Getting good performance for feedback for keyboard activity and mouse motion requires batching communication from an X client to an X server. A slack process is used to accumulate and merge requests from a buffer; an imaging thread puts paint requests on the queue for a buffer thread and issues NOTIFY. The hard part is ensuring that the buffer thread not act on a single paint request (otherwise there is no merging). The solution was to add to the thread library a *YieldButNotToMe* primitive and rely on the fact that most of the time, the image thread is favored as the next thread to be scheduled.

Also discussed was a new client library, Xl, designed with multithreading in mind; this was contrasted with thread-safe Xlib. Xl had advantages over thread-safe Xlib because by using a new serializing thread with condition variable, it avoided output flushes, since a client reading thread can block indefinitely, and the batching made the server connection asynchronous.

The authors also noted common usage mistakes that resulted in incorrectness or performance bugs: using IF instead of WHILE; timeout hacks to compensate for missing NOTIFY; timeouts in general with inappropriate values.

## 3.2 Thread implementation

Various implementation hacks were required to achieve satisfactory performance for the systems implemented.

### 3.2.1 Spurious lock conflicts

NOTIFY had to be changed to prevent the problem of one thread being notified and another thread awakened, with the notifier still holding a lock.

### 3.2.2 Priorities

Stable priority inversion was a problem. The hack was to have cycles donated from a blocked thread to a blocker thread for each per-monitor metalock.

### 3.2.3 Time-slice quantum

The X server *YieldButNotToMe* solution depended on the fact that the quantum is clocking the sending of X requests, and hence the interactive performance depends on the quantum. This kind of dependence of high-level performance on low-level implementation choices can cause problems.