Dynamic Load Adaptation in Internet Real-Time Applications

Kevin Fall
kfall@ee.lbl.gov

Network Research Group
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Dimacs Workshop
November, 1996

Motivation

- Factors affecting real-time application performance:
  - coding
  - network:
    - reliability/loss
    - bandwidth
    - delay/jitter
  - OS + hardware
- General Approaches
  - resource pre-allocation/reservation
  - **dynamic adaptation**

Dynamic Adaptation

- Why?
  - common OS and network lacks reservation support
  - premium rates for guaranteed traffic
- Approach
  - Delay: dynamic playout point (*vat*)
  - Reliability: scalable reliable multicast (*wb*)
  - Bandwidth: congestion control (*tcp*)

Load Adaptation

- Problem: network or OS lose data when overloaded
  - Internet Routers: FIFO queues overflow during congestion
  - Local OS: network buffer overflow during processing times
- RLM: dynamically reduce load on network
  - quality "layers" at source
  - (de)subscribe as a function of packet loss at receiver
- EDLS: dynamically reduce load on local OS
  - estimate processing time
  - self-schedule application accordingly (reduces loss)

An Example Application: *vic*

- *vic* video conferencing application
- focus here on playback processing

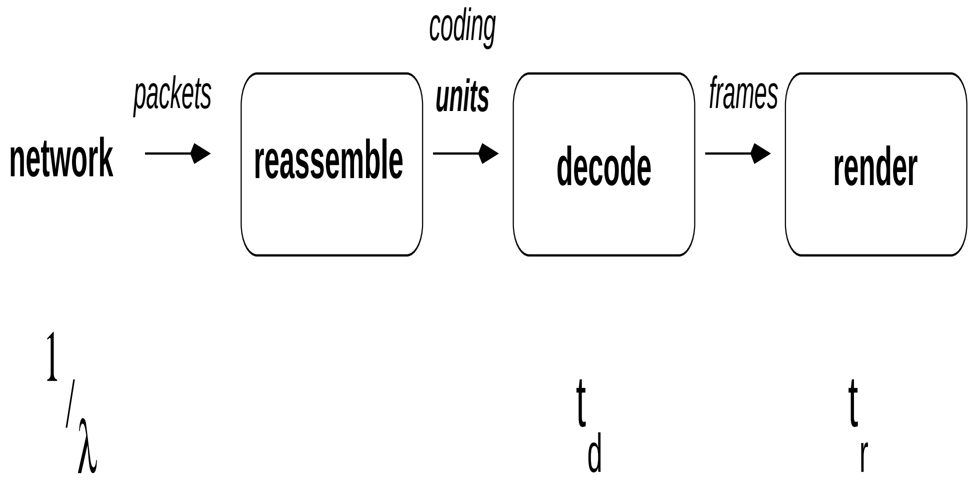

Figure 1: Vic Per-Source Playback Pipeline

- Bursty packet loss (in network buffer) occurs when:

$$1/\lambda \leq (t_d + t_r)$$

- Problem for slower processors
- Significant negative impact on perceived video quality

Outline of Algorithms

- EDLS: Early Discard Load Shedding
  - application is assigned maximum processing time for all sources
  - each processing pipeline is given an "allocation"
  - "allocation" enforced by token bucket filter
  - periodic reallocation across all sources
  - provides smoothing of drops across time (reduces bursty behavior)

Top-Level Algorithm

- Inputs: U CPU utilization (over some fixed interval), flows $F_1, \ldots, F_n$, execution interval T
- Operation: allocate TBF rate approximately fairly across flows
- Determining $F_1, \ldots, F_n$:
  - new source indicated by at least 2 in-seq RTP packets
  - source removed if inactive 8 secs (or more)
- Approximate *max-min* fairness, but is computationally simpler
- Current Experiments: $T = 5, U = 0.70$

Top-Level Algorithm (contd)

```
U: CPU duty cycle over 1 sec interval
n: number of flows (sources)

fair = U / n
remaining = U
for each source j
        d = estimate decode time of largest frame
        r = estimate sum of render times
        allocation[j] = MIN(d+r, fair)
        remaining = remaining - allocation[j]

fair = remaining / (# unsatisfied sources)
for each unsatisfied source k
        allocation[k] = allocation[k] + fair
```

- Estimators use exponential smoothing filters ($gain = 0.10$)
  - General Form: $E_{n+1} = (1 - g)E_n + gm$
- (can predict d+r times using least-squared linear fit)

Per-Source Algorithm

- Inputs: allocation for source k, A_k given by previous algorithm
- Operation: enforce a rate-limiter, discard frames when over-rate
- *Idea:* periodic (smoother) discards result in improved quality
- Execute the following:

```
state = ACCEPT PACKETS
for each packet received
        if packet is marked as frame boundary
                increase bucket
                if bucket >= high watermark
                        state = ACCEPT PACKETS
                        give to decoder + renderer
                        reduce bucket by d+r elapsed time
                else if bucket < low watermark
                        state = !ACCEPT PACKETS
                        drop
        else if (state == ACCEPT PACKETS)
                give to decoder
        else
                drop
```

Per-Source Algorithm (contd)

- Observations:
  - Packet drop/accept levels controlled with hysteresis
  - Packets comprising dropped frame are all dropped together
  - Dropped packets are never decoded
  - Enforces TBF for source k with parameters (b, r) where:
    - b (bucket size) = 1.0
    - r (fill rate in duty cycle units) = `allocation[`k`]`
  - Bucket is increased by $(T_n - T_{n-1})r$ on each complete frame
  - Bucket is reduced by actual measured D+R times
  - Updates are normalized to 1-second duty cycles

Results: Packet Loss in Application

- Load Shedding application loses *no* packets in OS queue
- Contrast with unmodified (Vic 2.8):

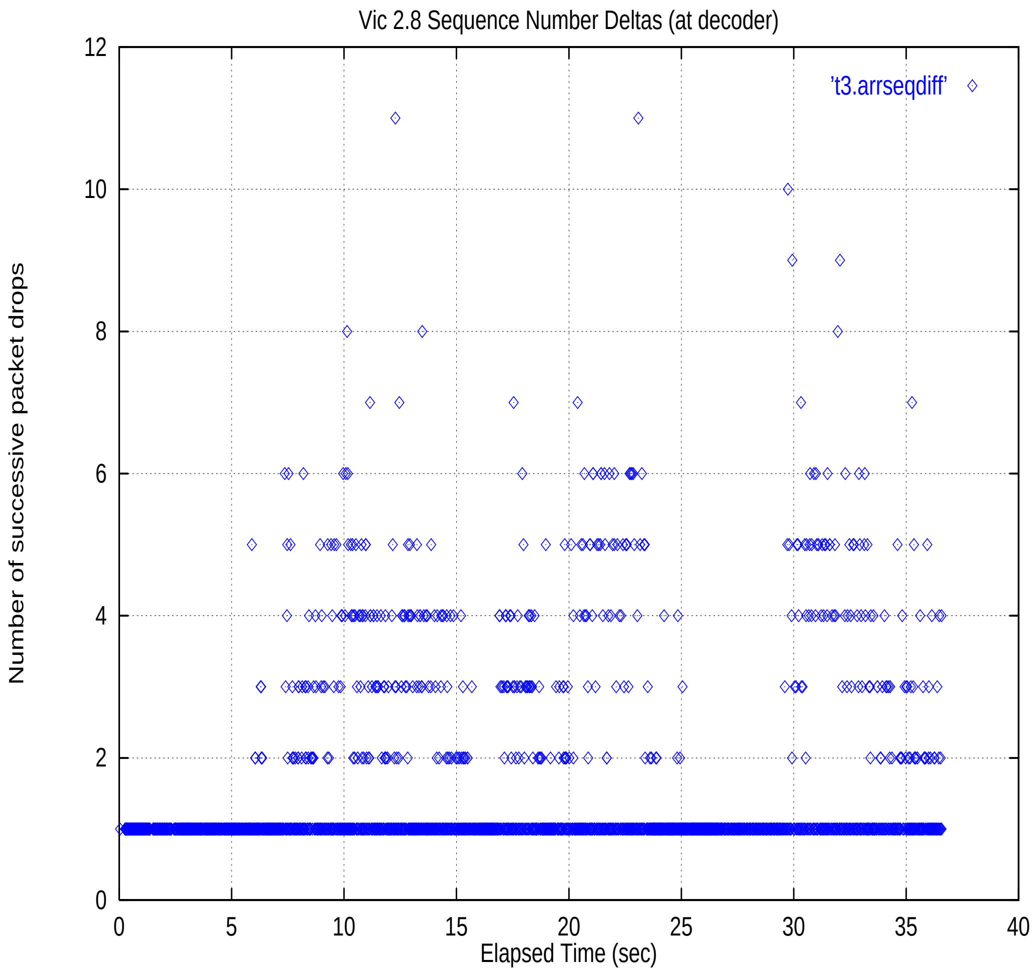

Figure 2: Vic 2.8 Packet Arrival Sequence

Drops

- Source Material
  - 3049 total packets ($\approx$36 sec of video)
  - 836 frames coded with "intra-H.261"
- Packet Drops

app	packets processed	percent processed	CPU utilization	cost ratio
Vic 2.8	1922	63	95	0.66
LS-Vic	1343	44	70	0.63

- (partial) Frame Drops

app	packets processed	percent processed	CPU utilization	cost ratio
Vic 2.8	579	70	95	0.72
LS-Vic	452	54	70	0.77

Results: Blocks per Frame

- More blocks decoded per displayed frame
- fewer intra-frame artifacts at cost of lower frame rate

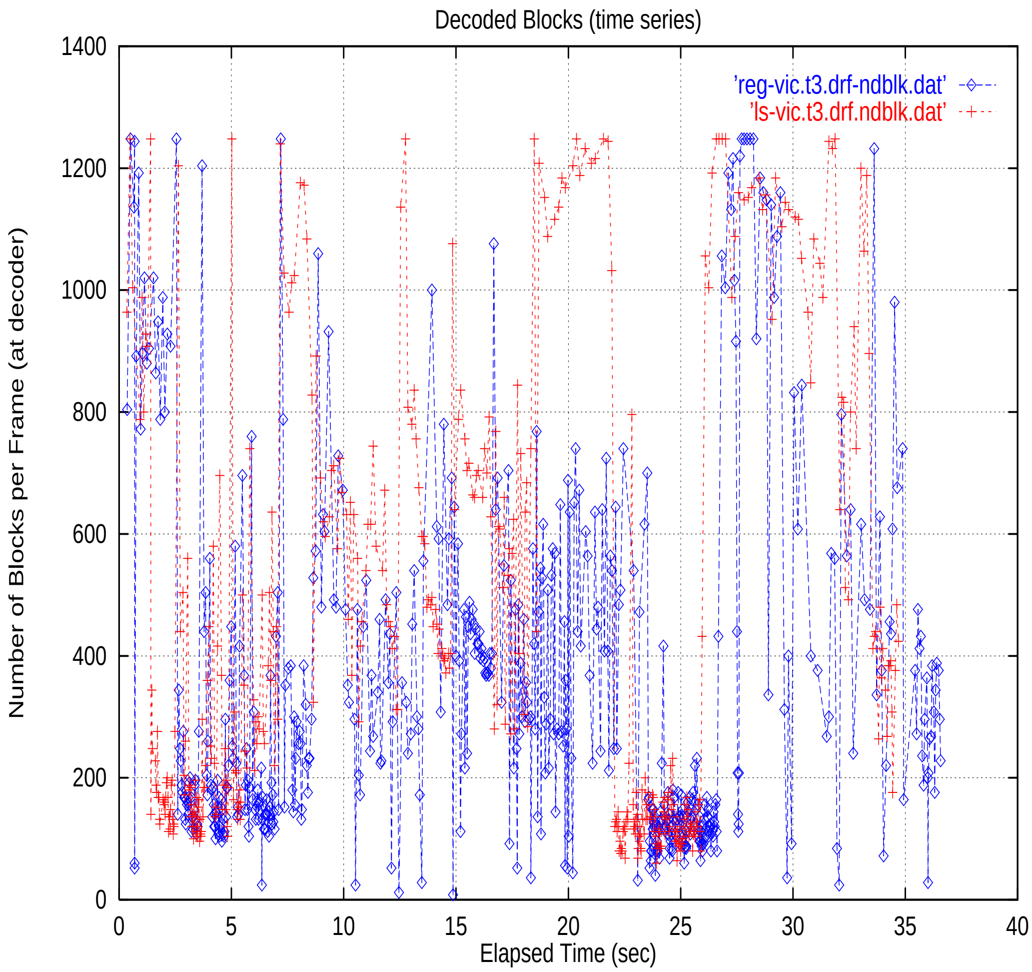

Figure 3: Block Sizes per Frame Series

Block Sizes

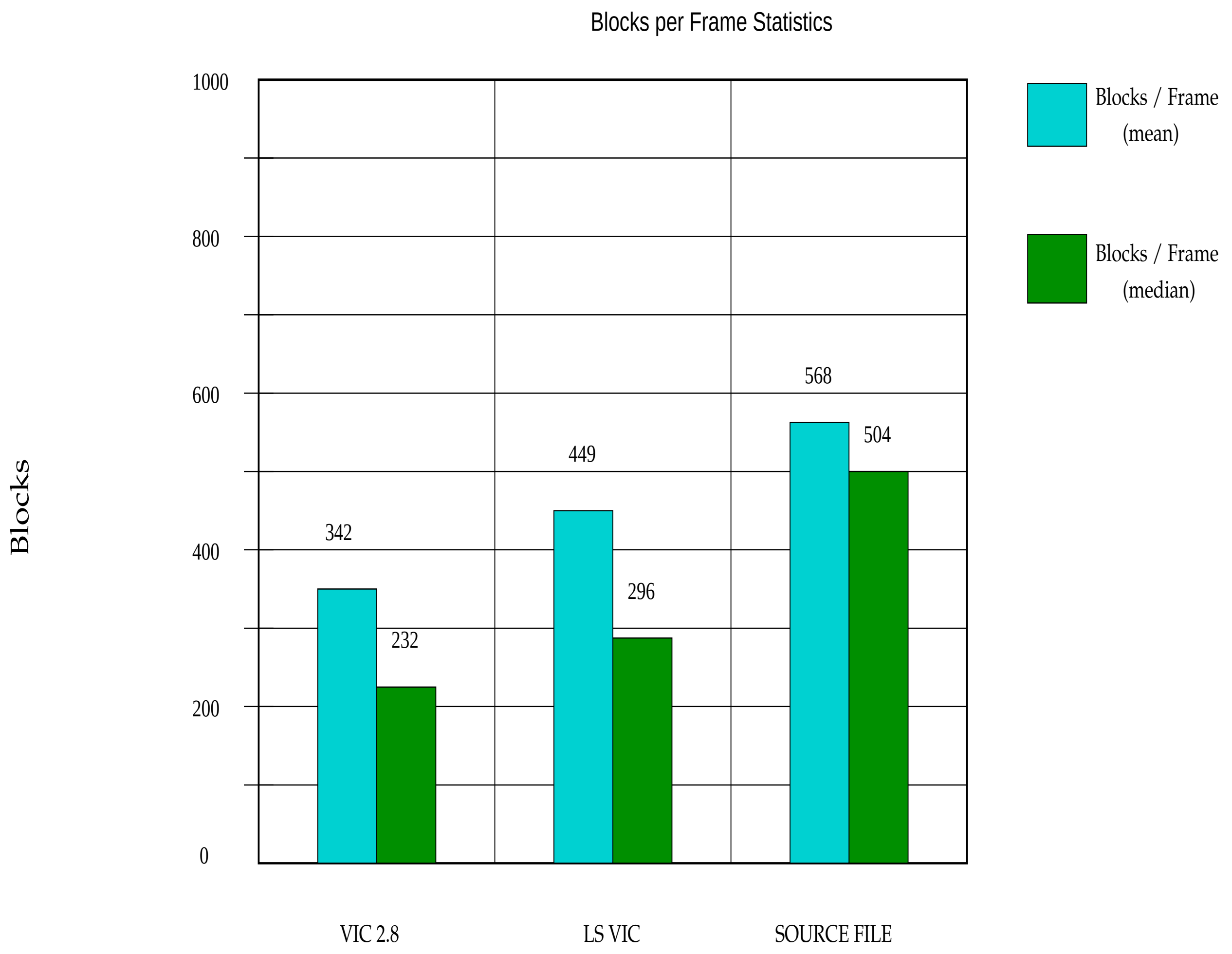

Figure 4: Block Statistics

Conclusions and Futures

- Conclusions
  - Load Shedding needed for slower processing environments
  - CPU time allocation capability aimed at cooperative applications
  - higher blocks/frame measure reduces intra-frame artifacts at cost of lower frame rate
- Futures
  - more measurements and tuning
  - combine with RLM
- Thanks: Steve McCanne, Van Jacobson