Performance on the iPAQ

The purpose of this file is to document our research on how to get acceptable performance using the iPAQ. We do not (yet) have benchmarks; the information in this email is based on rough measurements.

Major Performance Penalties

The network is very slow. This seems to be the main bottleneck when building distributed applications using the iPAQ. Using UDP instead of TCP to transmit video gives an order of magnitude increase in frame-rate. UDP is not suitable for audio because too many frames are dropped, yielding unacceptable audio quality. I suspect that compression/decompression beads will be necessary for any finished product. Wavelet-based compression algorithms may not be usable because the iPAQ does not have native support for floating-point arithmetic.

The arm does not an FPU, so floating point operations are prohibitively slow. This penalty can be circumvented by using fixed-point math, or, rather, finding libraries that use fixed-point math. For example, with use xaudiomp3 on the iPAQ, instead of the reference MP3 decoder implementation found in mpegaudiodecoder. The iPAQ does not have enough CPU to decode an MP3 using the reference implementation.

The scheduler plays a large role in performance. We removed a context switch in HttpClient[Response] by having it send on its member path entry, rather than on a reference path entry. Doing so improved audio quality from intolerably choppy, to having minor break-ups every few seconds in the beginning of a stream and very few break-ups after the first minute of playing (once enough data had been buffered). Avoid path breaks whenever possible. A mixed thread kernel may help, but we currently do not have an arm implementation of schedulerservices.

Surprisingly, video performance decreases as the total number of object files that Strings loads increases. When we last installed the Strings demo on the iPAQ, the Fight Club frame rate was so bad that we had to roll back to the previous installation. We removed rules, eliminated extraneous socket listen threads, and rolled back many object files, all to no avail. The only thing that made a difference was removing nonessential object files from the package. We didn't have time to look into this, but we don't think performance degrades with the amount of code linked into the executable. It's more likely that we fragment the memory, in the process of loading object files.

Minor Performance Penalties

The memory manager may be a factor. I have not benchmarked the memory manager, but the tests that are memory intensive (such as base64test) seem to take disproportionately long. I suspect that the Familiar distribution has memory manager that is optimized for size, not speed. In any event, we need benchmarks to see how much impact the memory manager has.

Non-Penalties

Drawing to the screen (accessing video ram) is not a bottleneck. In retail builds, the rgbdisplaytest can draw frames that are 120x96 at 192 fps.

Measurement Tools

There aren't many off-the-shelf metric tools for the Familiar distribution. The only one that I know of is "loadmeter", which is a graphical app that gives real-time metrics on CPU-usage, disk usage, and memory

What you should expect

The purpose of this section is to document what the iPAQ can do. If you cannot get the performance listed here, you're doing something wrong.

We achieved peak video performance by transmitting successive frames of 100 x 55 RGB bitmaps over a raw UDP socket. The video looked pretty good (~12 fps) and was definitely synchronized. If UDP is dropping lots of packets, you can insert the framedrop[drop] on the sending side. In theory, this gives more consistent performance by allowing us to systemically drop packets, rather than letting the network chaotically drop packets. This should be verified with benchmarks. You can scale the resulting BMP on the iPAQ to half-screen no penalty. Scaling it to full-screen is not noticeable on the CPU, but the frame rate becomes erratic.

We achieved peak audio performance by mpeg encoding the audio on the sender's side to minimize bandwidth consumption. The iPAQ can decode MP3s easily using xaudiomp3. It cannot decode MP3s at all using mpegaudiodecoder. When sending to a single iPAQ, the audio breaks up a bit at first, but then plays fine after the first 10 seconds. When synchronizing between the iPAQ and another machine, the audio breaks up considerably in the first five seconds, has a few chops for the next minute, and plays fine after that.

We had some audio quality problems when using the blade mp3 encoder. It seemed to introduce faint, squeaky echoes for some songs (most noticeable in songs with heavy distortion).

What we could not do

The purpose of this section is to document the things we were not able to do with the iPAQ. If you can do any of these, you should document how you managed to do so. If you can not do one of these, don't waste time tweaking parameters. We will likely need a major architectural change do it.

- We were not able to send video to the iPAQ and another machine with acceptable quality.
- We were not able to transmit both audio and video to the iPAQ.
- We were not able to transmit raw PCM to the iPAQ.
- We were not able to synchronize audio between the iPAQ and two other machines with acceptable audio quality.