Apple Confidential -- Need to Know

# V0 Performance Strategy
# February 23, 1993
# Philip Koch

## SUMMARY

We've now implemented most of the major pieces of V0, and have it *working*. The next major phase of software development is to accelerate and tune the system to make it *fast*. The D4 build incorporates most of the accelerations that we originally identified as being important, but preliminary data suggests -- to nobody's surprise -- that this first set is not enough to achieve our performance goals. We need to identify and port more code. We've learned a great deal from what we've done so far, and have a plan in place which we're confident will allow us to reach the program's performance goals. But its now clear that performance work will have to continue throughout the Alpha cycle, and we've identified a significant issue: Our current quality resources, although adequate to test the originally identified accelerations, are definitely not adequate to write the tools and test the additional parts of the system we now know we'll have to port. Its going to be much easier to *port* than to *test*. This is a problem.

## GOALS AND BASELINE DATA

Our performance goals are twofold:

1. For emulated applications, the goal is to achieve perceived performance at least as good as equally priced 68k-based Macintosh CPUs available at the time of introduction. Since we believe 25MHz 68040s will be selling in PDM's price range in January 94, this means we're measuring ourselves against the Quadra 700.

2. For native applications, the implicit (but hitherto unstated) goal of the program is to achieve performance "significantly" better for 601 applications compared to existing 68K and Intel applications. Marketing is still working on defining what "significant" means for future PowerPC customers.

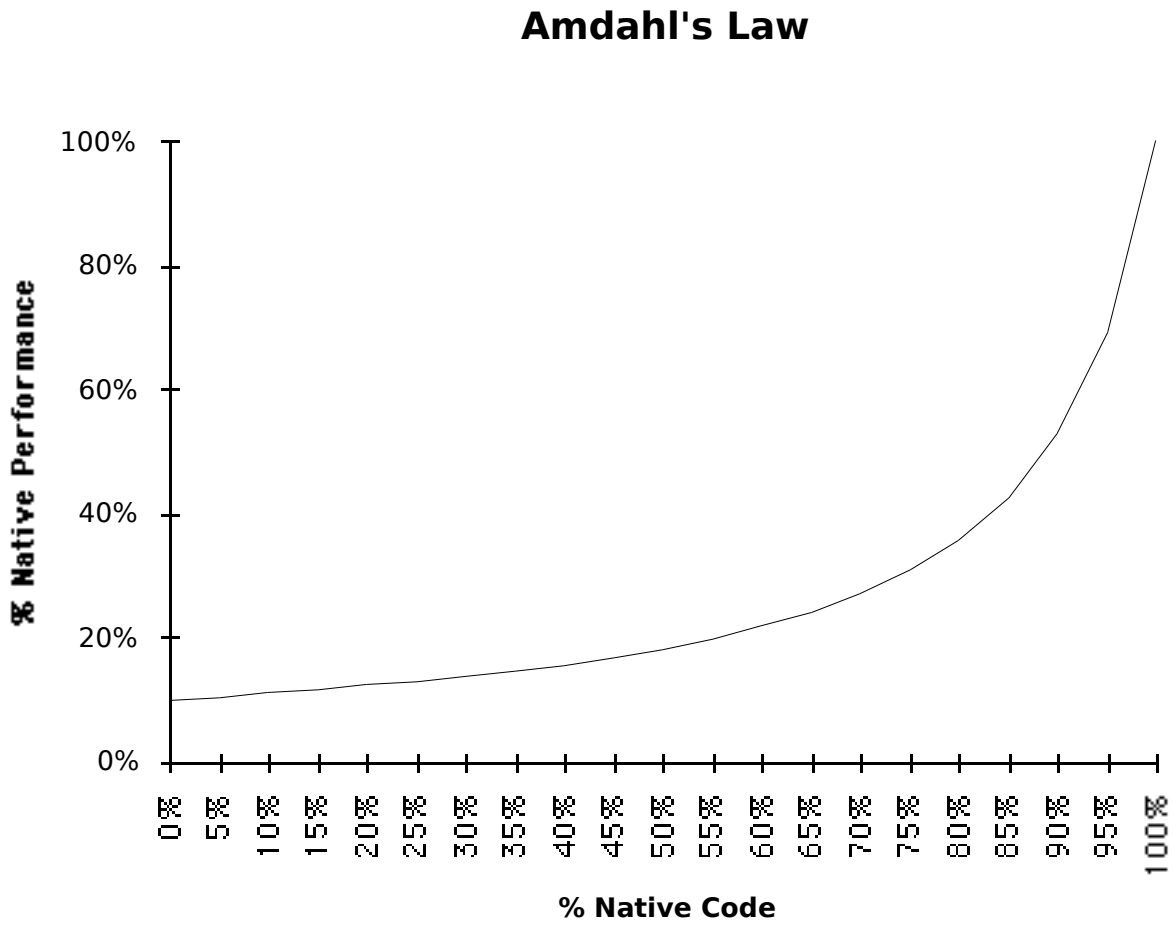
The metrics that we use to define our goals and to measure our progress, both for emulated and native applications, are based on user scenarios critical for PowerPC targeted markets. Dave Matzer and Marketing have defined for each of these applications a VU script which reflects the way average customers are using them. These scripts have been done to reflect both "user perceived performance" and "processor intensive" operations. Sample output from one of the scripts, comparing the speed on a IIci, a Quadra 700, and on PDM is attached.

We believe a 50MHz 601 is roughly 3x faster than a 25MHz 68040 for integer benchmarks such as ISpec, and that at 66MHz the 601 should be about 4x faster. The emulator runs at about 10% of full native speed, so that unaccelerated emulation on a 66MHz PDM should run about 40% as fast as a Quadra 700, or a little faster than a IIci. Thus we need to port substantial parts of the toolbox native in order to meet our goals.

The following graph shows the effect of Amdahl's Law on performance, measured as a function of the total percent of executed code which is running native. Thus the graph shows that at 0%

native we achieve only 10% of full native speed because that's how fast the emulator is. When 100% of the code is native, we naturally will run at 100% of native speed. But as the graph shows, the relationship in between these two extremes is distinctly

nonlinear. At first, lots of code can be running native but the performance improvement will be minimal because of the dominant effect of the remaining emulated code, which is running 10x slower. But as the percent native rises, the slope of the graph also rises: the noticeable improvement comes near the end, and it occurs relatively suddenly and dramatically. Amdahl's Law is a simple but inescapable mathematical fact: we need to have most of the code being executed native in order to see any speedup.




There are several important points to make about the Amdahl graph:

1. The x-axis is a measure of code being executed, not a static measure of the percent of the code ported native. The 90/10 rule, which many studies have shown applies to Macintosh, tells us that by porting the most frequently used portions of the toolbox, we can achieve substantially more than *n*% native execution even though only *n*% of the code base has been ported. This is the good news.

2. Unfortunately, the graph ignores the effects of the mixed-mode overhead, which is the cost of the transition between native and emulated execution modes. The roundtrip mixed-mode transition cost is roughly 50 emulated, or 500 native, instructions. Although for long-path routines the cost of this transition is insignificant, if a short subroutine is in the wrong mode the mixed-mode overhead dominates and can cause up to a 50000% slowdown (native code calling a one-instruction emulated subroutine)! So our actual performance, when %native is greater than 0% but less than 100%, will

always be less than Amdahl's Law predicts. This is the bad news, though we have several strategies than can minimize the mixed-mode penalty (see below.)

3. Ignoring the mixed-mode overhead for now, we see that a 66MHz PDM (4x Quadra performance running 100% native) will have to run at 25% of full native performance to achieve parity with the Quadra 700. The graph tells us that to achieve this, about 70% of the code being executed will have to be native. Thus emulated apps that spend less than 70% of their time in the toolbox cannot run as fast overall as on a Quadra.

## STRATEGY

So how do we move far enough to the right on the Amdahl curve to achieve our performance goals? Our strategy capitalizes on three basic insights:

1. The 90/10 rule. We win big by porting those few routines that account for the bulk of the execution time, such as BlockMove and the various QuickDraw inner loops. The PEG profiling data, which we have reconfirmed numerous times with other studies, shows that Mac traces are indeed subject to this phenomena. The 90/10 rule suggests the *domains* where we can profitably focus our limited porting resources.

2. Call-chain completion. Its not enough to just port the major loops in the selected domains. We're finding that in practice, nearly all of them make many calls to other, short subroutines. Because these subroutines are very short, they don't show up in the original profiles we used to select the domains. But because of the mixed-mode overhead, once the main loop is native the 50000% effect quickly makes those short subroutines very important. Thanks to Jim Gochee for crystalizing our thoughts here. We need to identify and port the *entire call-chain* in the important paths through the ported domains. Jim has written several tools to help us analyze these paths and identify the routines that need to be ported.

3. Dual implementations. OK, so we must port some short subroutines too, just because they're called from the 90/10 domains in critical places. This introduces another problem: when emulated apps call those same short subroutines directly (ie, from emulated code, rather than from native code), the mixed-mode penalty again occurs. Calling short subroutines that require a mixed-mode transition is always to be avoided. Unfortunately, many of the short routines we know we need to port for call-chain completion are *also* called a lot from emulated code. The solution is to have *two* implementations, one native and one emulated. See below.

Here's the strategy we'll keep iterating until we reach our goals (or we run out of time):

a. For both native and emulated applications, we'll use the Marketing user scenarios to identify workloads that we want to run fast. Our resources and time are both limited, so its very important to be selective and concentrate our attention to maximize the impact of our efforts. Most applications used as a metrics are part of Evangelism's "InsideTrack" program and have been identified as being critical for the PowerPC targeted customers. Futhermore these scripts have been defined to reflect the way average people are using these applications.

b. Profile the targeted workloads, using ASP and MacsTime, to identify the target domains

where the bulk of the execution time is concentrated.  Hopefully, the 90/10 rule applies.

c. Using Jim Gochee and Marianne Hsiung's call chain analysis tools, identify those subroutines in the target domains that need to be ported. We need to get almost all of them, or we'll slip to the left on the Amdahl curve and performance will suffer.

d. Port. We have several options here, see below.

e. Measure results and iterate until done. For example, rerun the VU scripts. If we've met our goals, or run out of time, we're done. Else go back to step (a) and start over again. This isn't an infinite loop because eventually 100% of the code will be ported.

We need to educate developers, both internal and external, about what we are porting and thus what they can expect will run fast and what will not. We expect this to be an interactive process, as developers educate us (for example, in Kitchens) about what they need to be fast.

## PORTING OPTIONS

Once we identify routines that need to be ported, there are several options:

1. Rewrite in C if necessary, then recompile. This is our first choice, especially if the code is in C already.

2. Use FlashPort. Although not a long-range solution, FlashPort could be a very valuable way to quickly port code with minimum risk of introducing algorithmic bugs. We're tracking FlashPort's progress carefully; Brian Topping, Kristin Webster, and John Mitchell have made several trips to New Jersey, and we try out each release. D4 contains a FlashPorted Resource Manager. At present we remain skeptical of FlashPort's utility, but we are ready to use it if feasible.

3. Use MicroAPL. Although Brian Topping did an early evaluation of their assembly language translator before it was finished, as yet we have no real experience with actual use. We hope to free up an engineer in Brian Heaney's team to try it out soon.

No matter how a routine is ported, we have another orthogonal choice: how to package the ported implementation. Once again, there are three alternatives:

1. Native-only. Long-path-length routines, such as the QuickDraw blit loops, need only be native because even if they are called from emulated code, the mixed-mode penalty can be amortized over the long execution time.

2. Fat traps. Recall that the mixed-mode penalty is about 50 emulated instructions; for routines with paths not much longer than this, porting them will actually be slower. As described above, we will need to port short routines nonetheless, if they are called from other native code, such as native apps or the main target-domains such as QuickDraw. We have extended mixed-mode to include the notion of a *fat trap*, containing two implementations of the same code. With a fat trap, mixed mode can choose to execute the version in the same mode as its caller, thus avoiding the majority of the mixed-mode penalty. Fat traps preserve full SetTrapAddress semantics, although they don't entirely avoid mixed mode overhead. We plan to support fat traps by Alpha.

3. Independent dual implementations. In some cases, especially very short traps that are frequently called, even the minimal mixed-mode overhead imposed by fat traps may be too great. In this case, we can install the native implementation of the trap directly in

the native glue library, or in a separate DLL, so that it can be called directly by native code. This entirely avoids mixed mode and thus incurs no penalty whatsoever. However, this solution does not preserve trap patching semantics. Scott Boyd is investigating the implications of breaking SetTrapAddress; we believe there are some routines that essentially are never patched, for which the risk may be justified. In other cases it will be necessary to use fat traps.

## PROGRESS TO DATE

D4 contains the following accelerations. These comprise most of the obvious candidates originally identified by the PEG ASP analysis.

- A-Trap dispatching (in the emulator, by Gary Davidian)
- SANE (emulator)
- BlockMove (emulator)
- QuickDraw. This work is being done by Tim Cotter with help from Steve Johnson, among others.
- DrawText, by Jim Gochee
- FMSwapFont, by Jeff Cobb
- the Resource Manager (using FlashPort), by Brian Topping

These implement three major "domains": Quickdraw, the text rendering mechanism, and the resource manager. We have not yet run the VU scripts against D4, which has just been built. But we believe, based on experience with several native apps such as Tim Nichol's "DiaTim" and Visual Sort, that we will not experience a major speedup from the above accelerations until we finish porting the entire call chains. Jim Gochee has made substantial progress analyzing call chains in the above domains, we have a prioritized list of the traps that need to be ported (as well as those that already have), and we've already done most of the work, which will be ready for the Alpha candidate build, if not D5. Because filling in the call chain mostly involves porting short routines, the work goes fairly quickly once one is familiar with the build and development environments.

Gary Davidian has a few ideas for further speedups in the emulator, for instance special casing common code bursts such as are emitted by compilers in procedure entry/exit sequences, but we don't expect these to achieve more than a few percent speedup in emulation. The big wins have to come in porting -- ie, we have to move to the right on the Amdahl curve, rather than hoping to raise the baseline.

Once the current domains are filled in, assuming we haven't reached our goals the strategy calls for looping back to step (a) to reevaluate profile data and identify one or more domains to work on next. In this case, we have a strong suspicion that the memory manager should be the next target (although this should be verified, and we need to be careful to think in terms of porting routines, not managers.) Brian Topping is already working with Jeff Crawford to begin porting Figment.

## OTHER WORK

This has been an OS-and-toolbox-centric analysis, but there is lots of work going on elsewhere

that effects V0 performance:

1. IBM: CPU speedups. It is likely that the final 601 parts will exceed the originally targeted 50MHz midpoint, which should make it possible to ship faster PDMs. We've

had prototypes in the lab running at up to 74MHz (?), though 66 seems more likely to be the midpoint of the yield curve.

2. MSD: PDM tuning. We have several knobs with which to tune PDM, including CPU speed, L2 cache, frame buffers, bus speeds, etc. Everything helps. Fortunately, hardware tuning is largely independent of software acceleration, so we can proceed in full parallel.

3. Somerset: Emulator/cache tuning. Rich Witek, Casey King, and Gary are investigating a redesign of the emulator's data structures, in order to improve its cache characteristics. This is particularly important on the 603, but its possible (though unlikely) that this investigation could lead to an improvement in performance on the 601 as well.

4. StarTrek: ported toolbox. We're working closely with the StarTrek team to be sure that we can leverage off each other's porting efforts, as well as to make sure that the APIs supported by these two projects differ as little as possible. Initially, StarTrek is likely to benefit more from our code (such as Quickdraw) than vice versa, but in the medium-to-long term the opposite should be true. We believe it is critical to cooperate with StarTrek, and maintain a single code base.

5. MSSW: "Go Native" program. Psychic TV does not include accelerating any of the extensions to the system. Several, such as QuickTime and GX, are natural candidates for acceleration. Indeed, they may be critical in some markets. The PowerPC program office is evangelizing the groups responsible for the extensions to accept the responsibility to "Go Native."

## ISSUES

Unfortunately, its even harder to schedule software performance than it is function. We're confident that we know how to move to the right on the curve, and cautiously optimistic about being able to move "far enough" before Beta. This optimism in based in part on the assumption that, as engineers roll off writing drivers and mixed-mode, etc, they can begin porting in earnest.

There may be a tradeoff between accelerating native applications and minimizing risk and impact on emulated apps. We expect that many traps will require porting when called from native code -- ie, native apps -- but could be left emulated if called from emulated code. Fat traps and "independent" implementations make it possible to serve both needs, but involve quality and schedule risks.

The major issue is that we believe that it will be much easier to port code, especially if we use FlashPort or output from StarTrek, than it is to test it. In many cases, even though porting won't introduce new functionality, test tools don't exist and will have to be written. There will be very real tradeoffs between performance, risk, and quality. We don't believe existing quality resources are anywhere near adequate to support the amount of porting it will take to meet our performance goals. Initial estimates indicate that we're short about eight quality engineers within Psychic TV. This is currently a top priority issue for Psychic TV.

Attached is an example of performance data generated by the VU scripts. This example shows execution times of the ClarisWorks script (smaller is better), using D4 software on a 66MHz EVT1. Times on a IIci and a Quadra 700 are also shown for comparison. Recall that the goal for emulated apps is Quadra performance.