Random network Coding on the iPhone: Fact or Fiction?

The authors begin by describing the general design principal of network coding. That is the transmission of random linear combinations of blocks. They also recognize that this technique can improve unicast performance when multiple paths are used simultaneously. They mention the computational complexity of such calculations and the energy consumption that is required and how this may limit potential practical use of such techniques. The authors promise a real world implementation on the Apple iPhone, and iTouch devices. They also promise an in depth investigation of the difficulties encountered including processor and hardware limitations, as well as their tuning efforts. They define three metrics: coding performance, energy consumption, and CPU usage.

Not bad for an abstract. I feel like they got a lot of work done already. However, I am not an expert in network coding and I will need to see a clear definition of network coding performance. Also they heavily implied that multiple paths would be used. I expect that the testbed must include at least four devices so that there will be a minimum of two paths. Any less would not allow them to convince me that their network coding implementation really increases unicast performance. That being said, I suppose they could also go the route of saying that the proof of network coding increasing unicast performance has already been proven by previous works and that they simply prove that they can achieve improved network coding performance over their testbed. I recall a paper from Elizabeth's class, "XORs in the air" that covered this topic. Anyways, good abstract, I look forward to the introduction.

The authors begin the introduction with a few references. They cite the XORs in th air paper as proof that network coding can improve unicast end-to-end throughput, as long as there are multiple paths that can be used simultaneously. So at this point, I am convinced that all they need to do is show that they can achieve "improved network coding performance", however, this still needs to be clearly defined.

The authors continue on to say the "no real-world implementation" has been done. Then they spend the next few paragraphs qualifying this statement by citing three previous "real world" implementations of network coding. The citations fall into two classes. First, those that use PC hardware: *Chachulski*, where they used PC's with wireless network cards, and *Kattie*, where they used PC's with Zigbee software radios. Secondly and most similar to this work is *Pederson*, where they implemented an XOR encoding scheme over Nokia mobile phones. This work takes advantage of the idea that XOR-only encoding can increase unicast throughput when multiple intersecting flows exist.

Now the authors spend some time justifying the use of random network coding over XOR-only network coding. They specifically list three: more flexibility, allowing coding over symbols, and the ability to use multiple paths. I am not sure what "more flexibility" means, and I am not clear why the use of symbols is better (perhaps if I was more expert in this field I would know that one), and I assume that by multiple paths they mean that intersections are not required. This alone is a big enough advantage. They continue on to cite literature examples to prove their case. Finally, they state that random network coding is more computationally expensive than XOR-only because it involves random linear combinations on the Galois Field.

For further motivation they offer the *Avalanche* P2P system which uses random network encoding.

Minor literary criticism: It seems like the first sentence of the third paragraph belongs as part of the second paragraph where they were talking about motivations for using random network encoding. Anyways, I am thoroughly motivated at this point and think that the work is significant.

The balance of this paragraph seems like a pitch to give phones public IP's so that they won't be treated as “second class citizens”. I have no idea where this came from. It hardly seems relevant to the paper and breaks the flow. If the authors feel that mobile phones must have a public IP in order to use random network encoding then they should simply state that and not start talking about classes of citizenship ;^)

In the next paragraph the authors offer a justification for the iPhone platform. I didn't think that the iPhone needed justification given it's popularity but lets see what they have for us. First, they offer the advanced technology of the iPhone platform. In fact they state that it is the “most” state-of-the-art hardware platform for multimedia technologies because it has an ARMv6 processor, has support for WiFi, EDGE, and 3G connections, and an excellent software development platform. Hmmm, this sounds a little like apple snobbery to me. (my apologies to the authors ;^) I am not a hardware expert but I do know that there are several competing phones that are very capable e.g., Motorola's Droid, and especially the latest Nokia which has an entire linux kernel. Secondly, the ARM processor is widely used in mobile devices (This is very true) and finally that the iPhone has already been used for streaming multimedia from the web.

I feel that this paragraph was unnecessary and debatable. The authors should have used more care in writing it and acknowledged that their reviewers might not all be from amongst the ranks of the Apple faithful ;^)

The authors then talk about the difficulty level of implementation on an iPhone. I'm sure this is true, but I am not so certain that their optimization techniques will be useful on other platforms. They state the array of tradeoffs that they consider in their work. These are network coding configuration vs. CPU usage, energy consumption rates, battery life.

This was a good introduction, the motivation for the work is solid and I think it is significant. However, I think that some of the space in the final paragraphs was wasted and could have been put to better use. I agree that the iPhone is an excellent platform, however, like most Apple products it is also very proprietary which is a solid drawback.

In section two the authors present a concise introduction to random network coding. This section is very welcome, at least for me, because I am unfamiliar with the details of the technique. Data to be coded is first divided into *n* blocks of equal byte size, then a coefficient for each block is independently and randomly chosen. It then produces a coded block of *k* bytes. A receiving node decodes as soon as it has *n* linear independent coded blocks. It does this by forming an *n x n* coefficient matrix using the coefficients in each coded block. It then recovers the data in the original blocks using: b = C^-1 * x (the product of the inverse of C and x). The inverse of C is computed using Gaussian elimination which is of complexity *n*^3 and requires that matrix C be of full rank. Optimization of the multiplication operation here is possible here because addition in Galois Field is equivalent to an XOR operation. The authors present a C coded function that accomplishes this multiplication optimization.

In the next paragraph the authors present the challenges and solutions of their design. Aha! Here is the acknowledgment of other devices that use the same processor ;^) Of course the droid and the Nokia are here as well as many other devices. This does a lot to assuage the effects of the apple snobbery as well as to make the case that the techniques presented here are applicable to other devices ;^) The authors are looking for feasibility, and maximum optimization of performance. However, I still haven't seen a precise definition of network coding performance. Also, why is it maximum? Maximum

compared to what? To what is possible on the iPhone platform? Maximum amongst all the devices? This will make the difference between a weak accept and a strong accept for me.

The authors go on to describe their implementation of table based multiplication which requires multiple access to the lookup tables. In their previous work they used a loop based approach which is more costly than the traditional approach (8 iterations) but easier to parallelize. Current smartphone platforms are single core. The authors will investigate the benefits of parallelization on the current single core architecture.

In Section 2.1 the authors begin an evaluation of table-based coding techniques by describing some of the difficulties they encountered with the iPhone SDK (incidentally this would not have been a problem on the new Nokia SDK ;^) and then describe their baseline for performance measurements. They use 128 blocks of 4096 bytes which gives a segment size of 512KB (about 5.33 seconds). I agree that this is a reasonable buffering delay. Preliminary experiments show an encoding rate of 16.4 KB/s and decoding of 60 KB/s. This disproportionate rates are due a memory access pattern generated by the authors design that did not work well with the ARMv6 cache architecture. The authors changed their algorithm from column-by column to row-by-row and this improved the encoding rate to 66.7 KB/s on par with the decoding rate. This shows that without optimization the ARMv6 will not be able to keep up with a 96 KB/s (768 Kbs) multimedia stream.

Okay, I think I see where their baseline performance measurement is coming from. They are going to compare their optimizations to this and show that they are better. I find this reasonable, however, I will be even more impressed if these optimizations are not iPhone specific and can be used on any ARMv6 device.

In Section 2.2 the authors look into using SIngle Multiple Data (SIMD) vector instructions, however, they discovered that the ARMv6 used in the iPhone that they used has a ARM1176JZF-S processor which they used has a limited SIMD implementation which only allows operations on the half word boundry and 32-bit registers. They were expecting 128 bit registers. The authors implemented a version of their multiplication algorithm using the smaller registers. The code is quite compact and when tested with 128 blocks of 4096 bytes each (the baseline ;-) encoding of 86.6 KB/s and decoding of 81.9 KB/s (1.3 & 1.37 percent improvement). Still short of 96 KB/s but getting closer ;^)

In Section 3, the authors investigate the use of the *Thumb* vs. *ARM* instruction sets. The *Thumb* instruction set is a limited 16 bit instruction set space that implements a subset of the full 32 bit *ARM* instruction set. It is the default because it generally produces code sizes 35% smaller than the 32 bit *ARM* instruction set. However, it does not allow predicated instructions (which are tagged for conditional execution) instead requiring two instructions to accomplish the task. Also it does not allow the use of the *barrel shifter* which shifts data from the registers on its way to the ALU. When tested against the baseline setting the authors found an increase of 50% and 26% to 100.4 KB/s encoding and 75.8 KB/s decoding. However, when combined with the loop based tuning they achieve 89% and 93% to 163.8 KB/s and 157.8 KB/s for encoding and decoding. This improvement was from reducing the code size from 17 to 10 in the loop by using the *ARM* instruction set as well as from and because the loop based implementation makes heavy use of logical and shift based operations and thus benefited from the use of the *barrel shifter*. Note that the language in the last two sentences is a little inconsistent. The authors say, “This improvement is essentially due...”, talking about the reduced code base and then, “The loop-based implementation has achieved a more substantial gain...”. Nice improvement though ;^)

Finally, in Section 2.4 the authors hand tuned the compiler output for efficiency. The tuning was comprised of things like changing instructions to 16-by-32 bit instructions and hand tuning the code to ensure integrity through the rest of the contribution. Explicit use of the *barrel shifter* and reordering the instruction based on timing to prevent pipeline stalls. This improved encoding to 181.3 KB/s and decoding to 175.3 Kb/s. The final hand tuning makes use of a SIMD instruction uadd8 to replace two instructions (compiled into a single machine instruction) and shifts more data at a time. This provided a 4% improvement bringing the total to 188 KB/s encoding and 182.3 KB/s decoding.

The first three improvements seem like they could work generally on ARMv6, however, the last one with it's reordering of instructions according to register latency is certainly specific to the ARM1176JZF processor and therefore is only good for this generation of processors.

At the end of the day this section has defined some very clever software optimizations and produced some substantial coding performance benefits. Some of these optimizations may not be transferable (or even necessary/desirable on later generation processors but this is the case for any kind of optimizations at this level. The authors goal was to determine the feasibility of using random network coding on the current generation of processors and they have achieved this goal.

In Section three the authors evaluate the performance of hand tuned random network coding implementation on the iPhone 3G and 2nd generation iPod iTouch devices. They evaluate against metrics such as coding bandwidth, CPU usage, and energy consumption using fully dense coding matrices with non-zero coefficients.

When investigating coding performance the authors used a range of 128 bytes to 16 KB per block with 64, 128, and 256 blocks. They measured both the encoding and decoding bandwidth using both the table based and loop based approach detailed in the previous sections. The measurements are produced by taking the total bytes of all the generated coded/decoded blocks with an (*n. k)* coding setup over 1 second. The authors state that the graphs show encoding achieves it's peak performance across all settings. I am not sure what they mean by this. However, I do notice that for a large number (256) of blocks both the approaches fail to achieve the 96 KB/s feasibility threshold. The authors note that a decline in performance occurs with block sizes beyond $k = 256$ at $n = 64$ and $k = 128$ at $n = 128$. They theorize that this decline is due to the L1 cache size in the ARM1176JZF processor being limited to 16K. However, they were unable to find the specification in the public domain documentation of the ARM1176JZF of the iPhone platform. (This is another huge drawback of working with Apple hardware. Everything is undocumented in public domain ;^) However, they were able to use an undocumented interface to measure the L1 cache size and determine that it is indeed 16KB. They calculate that the in order to achieve their top rate of 415 KB/s the number of executed instructions is 470.2 Mega Instructions Per Second (MIPS). This represents about 88% of the capability of the ARM1176JZF indicating that the key bottleneck here is the processor speed. It simply can't go any faster. This is indeed impressive, they have squeezed almost every drop of performance out of the processor.

In Section 3.2 the authors the coding performance of the iPhone vs. the iTouch. Both devices use the ARM1176JZF processor however the iTouch is clocked at 412 Mhz with a 103 Mhz bus while the iPhone is clocked at 533 Mhz with a 133 Mhz bus (about 29% faster). This graph shows that the coding performance difference is roughly equivalent to the processing speed difference.

In Section 3.2 the authors prepare for their feasibility experiment. The four (or more) device testbed that I had envisioned earlier does not exist, although the authors seem like they are interested in moving

in that direction ;^) They will investigate the feasibility of using random network coding from the perspective of CPU usage, and energy consumption. The authors will vary the network coding settings looking for a *feasibile* setting. However, I still do not know what *feasible* means. They will need to define this.

In order to get a baseline performance measurement the authors profiled the performance of receiving and playback of 3 video streams with different properties. They used WiFi connectivity and the iTouch because this combination provides the best performance. They used the *Instuments* application in the iPhone IDE to do the monitoring. They found that the *YouTube* process is only active when the user is interacting with the GUI. The *mediaserverd* process handles the encoding and decoding and the *springboard* process which manages the matrix of applications consumes about 2% of the CPU. In addition there is a *DTMobileIS* process which collects measurements and consumes about 6% of the CPU. The table shows that only a small portion of the CPU resources are used in decoding and playback. This implies that the more complex operations are handed of to a GPU leaving a lot of available CPU for random network coding ;^) Also the ingress rates are a little higher than the video rate. This is because of streaming protocol overhead.

Hmmm, all this is very nice but I still don't have a definition of *feasibility.* Clearly *feasible* would mean something less than the remaining CPU after the video receiving and playback functions are taken care of but what is it? Eighty percent? Ninety? Fifty? I think they should have defined this more clearly (or at least explicitly) and should do the same for energy consumption. Make a definition and then justify why it was chosen. For instance, feasible energy consumption means the application will not bleed the battery in less than 2 hours because this is how long an average movie plays. Whatever, just some number and a justification.

In Section 3.3.2 the authors define their experiment to determine if random network coding on the iPhone is *feasible* according to CPU usage. They built an application that performs network coding and decoding at rates corresponding to a realistic P2P media streaming scenario. I guess this means 768 Kbs (96KB/s). Also the iPhone does not allow third party applications to run in the background so it is not possible for the authors to run the actual *YouTube* application at the same time. The authors choose a segment size of 256 KB because this results in a buffering delay of 3-7 seconds with the 3 videos chosen for the experiment. They will evaluate this segment size with two network coding settings (*n=128, k=2048)* and (*n=64, k=4096)*. A deployed P2P network coded streaming system would need decode the incoming stream and encode a new stream for a small number of neighbors. The authors will conduct experiments with 1, 2, and 4 outgoing streams. However, based on their coding performance results the ARM1176JZF does not have the power to decode one stream while encoding four more at the same time. To remedy this the authors vary the density of the random network coding. They give a citation which shows that density can be reduced to as much as 10% of the original without increasing the risk of linear dependence among the coded blocks. This seems like it should have been mentioned at some earlier point in the paper. Why didn't they use this technique on the earlier experiments? It seems very effective. If it can increase efficiency by ~ %400 why not use it from the start? I guess I would have to look at the reference to find out but at this point I am willing to just believe them and trust that there is some reason for this. Anyways, they use a density of 1/*d* for *d* downstream nodes.

The results from this experiment are shown in a table next to an estimated CPU usage of R/BW_dec + R/BW_enc. The rates for *d*=1 (1 decoding stream and 1 downstream node) match the estimates very closely and (*n=64, k=4096)* has about half the CPU usage as (*n=128, k=2048)* as expected. However, as *d* increases the actual results beat the estimates. This is because as the matrices become sparcer the

efficiencies become greater. The authors also evaluate the CPU usage at d=0 reflecting a node that does not wish to, or does not have the battery capacity to serve other nodes. This setting provides an additional ~50% CPU performance advantage. The authors also present a graph of the CPU usage over the course of one of the experiments. This shows that other tasks are sometimes using the CPU during the experiment (sending the CPU to 100% usage) and that the random network coding usage exhibits a sawtooth pattern. This sawtooth pattern is due to varying complexity of the Gauss-Jordan elimination computation.

In Section 3.3.3 the authors investigate the energy consumption characteristics of their random network coding design. They were unable to do this with Apples IDE and had to use some *unofficial* header files to do so. Also because Apple does not allow third party apps to run in the background they had to *jailbreak* the device so that they could run the video and the network coding and the monitoring software at the same time. They tested at ($n=64, k=1024$). I would have liked to see some justification for this choice but there is none. They tested video playback without network coding, video playback with network decoding at 77.5 KB/s, and video playback with both encoding and decoding at 77.5 KB/s over a 31 minute period. The results show ~33% of the consumption is due to encoding and ~ 15% is due to decoding. However, the authors admit that this is only a first order approximation because the accuracy of the API is limited. The authors suggest that network coding with 64 blocks is suitable for the current generation of iPhone platform.

In section four the authors give their conclusions. They restate what they did in the paper. They also state that it is possible to take advantage of random network coding on the current generation of mobile devices. They recommend a setting of 64 blocks of 4096 bytes each. They have shown that decoding for rates of up to 620 Kbps will cause an increase of 22% CPU usage. Adding encoding for 4 downstream nodes will increase the additional CPU usage to 35%. The authors do note that in a real deployment that tradeoffs between CPU usage and energy consumption may lead to different design decisions. However, as hardware platforms improve network coding efficiencies will also improve making it more likely that network coding will be deployed. What they do not mention is that bandwidth may increase faster than hardware efficiency, making the added complexity of little value.