Random Network Coding on the iPhone: Fact or Fiction?

Hassan Shojania, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

ABSTRACT

In multi-hop wireless networks, random network coding represents the general design principle of transmitting random linear combinations of blocks in the same "batch" to downstream relays or receivers. It has been recognized that random network coding in multi-hop wireless networks may improve unicast throughput in scenarios when multiple paths are simultaneously utilized between the source and the destination. However, the computational complexity of random network coding, and its energy consumption implications, may potentially limit its applicability and practicality in mobile devices. In this paper, we present our real-world implementation of random network coding on the Apple iPhone and iPod Touch mobile platforms, and offer an in-depth investigation with respect to the difficulties towards such an implementation, the limitations of the ARM processor and the hardware platform, as well as our hand-tuning efforts to maximize coding performance on the iPhone platform. With our implementation deployed on both the iPhone 3G and the second-generation iPod Touch, we report its coding performance, energy consumption rates, as well as CPU usage with multimedia streaming.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*

General Terms

Algorithms, Performance, Experimentation.

1. INTRODUCTION

First introduced by Ahlswede *et al.* [1] in information theory, *network coding* has received significant research attention in recent years in the networking community. In multi-hop IEEE 802.11-based wireless networks, Katti *et al.* [2] has shown that, random network coding is able to significantly improve end-to-end throughput of unicast sessions, provided that multiple paths between the source and the destination are used simultaneously.

Unfortunately, to date, there has been no real-world implementations of random network coding on real-world mobile devices. The closest efforts towards this objective are reflected in the following works in the literature: *First,* in Chachulski *et al.* [2], each node in a wireless network that performs random network coding is a PC equipped with a IEEE 802.11 (WiFi) wireless card. *Second,* Katti *et al.* [3] have also performed random network coding on off-the-shelf Intel PCs, each equipped with Universal Software Radio Peripherals and Zigbee software radios.

Third and most recently, Pedersen *et al.* [4] have implemented XOR-only network coding in Nokia N95 mobile devices, since XOR-only network coding may improve throughput when paths of multiple unicast flows intersect in multi-hop wireless networks, originated by Katti *et al.*'s earlier results on COPE [5]. However, random network coding is able to offer substantially more flexibility, allowing coding over symbols and the use of multiple paths. It has been the coding mechanism of choice in recent wireless networking literature, including MORE (Chachulski *et al.* [2]), and symbol-level network coding (Katti *et al.* [5]). Random network coding is much more computationally intensive than simply performing XORs on incoming packets, and involves random linear combinations on the Galois Field (GF).

Another motivation to justify the use of random network coding on mobile devices comes from advances of using random network coding in peer-to-peer (P2P) applications, such as Avalanche [6]. It is customary for mobile Internet devices to connect to the Internet using a variety of technologies: WiFi, EDGE, and 3G. When connected, they are allocated an IP address, potentially a public IP, and appear precisely as a peer in P2P streaming applications. Assuming that network coding is deployed in these applications, it is preferable not to treat mobile peers as "second-class citizens".

This paper presents our experiences with our real-world implementation of random network coding on the iPhone platform, including the second-generation iPod Touch, and the iPhone 3G. We have chosen the iPhone platform due to the following justifications: (1) It offers the most state-of-the-art hardware platform for multimedia applications, with an ARMv6 architecture core, support for WiFi, EDGE and 3G connections, as well as an excellent software development platform; (2) The ARMv6 core has been widely in use in other mobile devices, increasing the applicability of our implementation; (3) The iPhone platform has already been widely used in the real world for streaming multimedia applications from the Internet, especially from YouTube. Our objective in this paper is to present an in-depth evaluation of what can be feasibly achieved at the time of this writing on the best possible mobile devices in the market.

It is non-trivial to develop for the iPhone platform, especially an application as computationally intensive as random network coding. In this paper, we report our difficulties with the hardware platform, as we explore all possible avenues — including hand-tuning optimizations tailored for the ARMv6 core — to maximize coding performance. With

NOSSDAV'09, June 3–5, 2009, Williamsburg, Virginia, USA.

our implementation optimized for the iPhone, we have also evaluated its performance extensively. We have discovered that random network coding is feasible and manageable on the iPhone platform, in the context of streaming applications currently used (*e.g.,* at typical media streaming rates). We believe that the use of random network coding involves an array of tradeoffs: decisions must be made concerning the network coding configuration, CPU usage overhead, energy consumption rates and battery life, as well as limitations to participate fully in peer-to-peer streaming applications by contributing upload bandwidth. Future hardware platforms for mobile devices are expected to offer a better tradeoff and a smaller footprint incurred by network coding.

The remainder of this paper is organized as follows. Sec. 2 presents our experiences implementing random network coding on the ARM architecture core, which the iPhone platform uses. Sec. 3 evaluates our implementation on both the iPhone 3G and the second-generation iPod Touch. Sec. 4 concludes the paper.

2. RANDOM NETWORK CODING ON THE IPHONE PLATFORM

We first present a concise introduction to the operations performed in random network coding. In random network coding, data to be disseminated is divided into n *blocks* $[b_1, b_2, \ldots, b_n]^T$, where each block b_i has a fixed number of bytes k (referred to as the block size). To code a new coded block x_j, a node first independently and randomly chooses a set of coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jn}]$ in GF(2^8), one for each received block (or each original block on the data source). It then produces one coded block x_j of k bytes:

$$x_j = \sum_{i=1}^{n} c_{ji} \cdot b_i \tag{1}$$

A node decodes as soon as it has received n linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$. It first forms a $n \times n$ coefficient matrix $\mathbf{C}$, using the coefficients in each coded block x_j. Each row in $\mathbf{C}$ corresponds to the coefficients of one coded block. It then recovers the original blocks $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$ as:

$$\mathbf{b} = \mathbf{C}^{-1}\mathbf{x} \tag{2}$$

In this equation, the inverse of $\mathbf{C}$ needs to be computed using Gaussian elimination, with a complexity of O(n^3). The inversion of $\mathbf{C}$ is only possible when its rows are linearly independent, *i.e.,* $\mathbf{C}$ is full rank.

GF(2^8) operations are routinely used in random network coding within tight loops. Since addition in GF(2^8) is simply an XOR operation, it is important to optimize the implementation of multiplication on GF(2^8). A baseline implementation is performed using logarithm and exponential tables, similar to the traditional multiplication of large numbers. Fig. 1 shows a C function that multiplies two bytes, where `log` and `exp` reflect GF(2^8) logarithmic and exponential tables. Such a baseline implementation requires three memory reads and one addition for each multiplication.

```
byte table_gf_multiply(byte x, byte y)
{
    if (x == 0 || y == 0)
       return 0;
    return exp[log[x] + log[y]];
}
```

Figure 1: Table-based multiplication in GF(2^8).

We are now ready to present challenges and solutions involved in the design of our implementation of random network coding for the ARMv6 architecture, used in the iPhone platform. We have selected the ARMv6 architecture in this paper since it is used in a wide variety of other mobile devices — prominent examples include the Nokia N95, Nokia N800, Microsoft Zune, Motorola Razr2 V9, HTC TyTN II, and the Android-based HTC Magic. Our objectives are to first study the *feasibility* of using the ARMv6 architecture to perform network coding, and if feasibility is not an issue, to maximize the performance of our implementation.

Table-based multiplications in GF(2^8), shown in Fig. 1, require multiple accesses to the lookup tables, and constitute one of the important performance bottlenecks in random network coding. To accelerate this costly operation, in our previous work [7], we are the first to explore the use of a loop-based approach in Rijndael's finite field, rather than using traditional `log/exp` tables. Although the basic loop-based multiplication takes longer to perform (up to 8 iterations), it lends itself better to a parallel implementation in Intel CPUs by taking advantage of SSE2 SIMD (single-instruction, multiple data) vector instructions.

Before smartphone hardware platforms, such as the iPhone, gain access to multicore processors (such as the proposed ARM Cortex-A9), our only option of parallelizing GF multiplication is to explore parallel loop-based multiplication on a single processing core, using either SIMD instructions or an equivalent mechanism. Is it at all possible to achieve such parallel multiplication on the ARMv6 architecture core? At first glance, it is a daunting challenge since the ARMv6 core is designed for embedded devices, with a much simpler, non-superscalar architecture, plain 32-bit registers and arithmetic units, and runs at much lower frequencies.

2.1 Evaluating table-based network coding

As a starting point, we first attempted to implement random network coding based on table-based GF multiplication on the iPhone. It turned out to be a much more challenging venture than we predicted. The iPhone development SDK is based on the Objective-C language, and iPhone applications are required to be in GUI form, using the `UIKit` library.

To evaluate its performance, we use ($n = 128, k = 4096$) (128 blocks of 4096 bytes each) as our "base" network coding configuration. If such a configuration is used in a P2P media streaming application with a 768 Kbps (96 KB/s) streaming rate (representing high-quality videos), it leads to a media segment size of 512 KB or 5.33 seconds. This represents an acceptable buffering delay. Our first measurements show an encoding rate of 16.4 KB/s and decoding rate of 60 KB/s with 128 blocks of 4 KB each, on our second-generation iPod Touch. The low coding rates may not be as surprising as the observation that encoding performs only 27% of decoding, since decoding is more computationally complex.

It turns out that this anomaly is due to the memory access pattern, and most likely related to specifics of cache performance in the ARMv6 architecture. Our original encoding process generated n coded blocks in a column-by-column fashion, *i.e.,* generating a coded byte for all coded blocks. The encoding rate improved to 66.7 KB/s after revising the encoding process to a row-by-row pattern.

It has now become apparent that, at a decoding rate of only 60 KB/s, the iPhone will not be able to decode a network coded 96 KB/s stream, even at 100% CPU usage.

```
byte loop_gf_multiply_word(byte factor, word data)
{
    word PrimPolyMask, result = 0;                          (1)
    while (factor != 0) {                                   (2)
        if ((factor & 1) != 0)                              (3)
            result = result ^ data;                         (4)
        // creating the irreducible poly mask
        PrimPolyMask = data & 0x80808080;                   (5)
        PrimPolyMask = PrimPolyMask >> 7;                   (6)
        PrimPolyMask = PrimPolyMask*0x1d;                   (7)

        // clear top-bit of bytes before shift
        data = data & 0x7f7f7f7f;                           (8)
        data = data << 1;                                   (9)
        data = data ^ PrimPolyMask;                         (10)
        factor = factor >> 1;                               (11)
    }
    return result;                                          (12)
}
```

Figure 2: Loop-based GF(2^8) word multiplication for a 32-bit processor.

```
byte loop_gf_multiply_word_armv6(byte factor, word data)
{
    word PrimPolyMask, result = 0;                          (1)
    while (factor != 0) {                                   (2)
        PrimPolyMask = PrimPolyMask >> 1;                   (3)
        PrimPolyMask = data & 0x40404040;                   (4)
        if ((factor & 1) != 0)                              (5)
            result = result ^ data;                         (6)
        // creating the irreducible poly mask
        PrimPolyMask = smulw(PrimPolyMask, 0x1d << 10);     (7)

        // clear top-bit of bytes before shift
        data = data << 1;                                   (8)
        data = data & 0xfefefeff;                           (9)
        factor = factor >> 1;                               (10)
        data = data ^ PrimPolyMask;                         (11)
    }
    return result;                                          (12)
}
```

Figure 3: Hand-tuned loop-based multiplication for ARMv6.

2.2 Revisiting loop-based network coding

Is it feasible to implement random network coding using loop-based multiplication on the ARMv6 architecture? The current iPhone and iPod Touch series use the ARM1176JZF-S processor, which belongs to the ARM11 family, and is based on the ARMv6 core with a set of features that include SIMD support [8]. Our first impression was that such SIMD support is of the NEON SIMD type. The ARM NEON technology is quite similar to SSE2 and AltiVec SIMD technologies found in x86 and PowerPC families, and comes with support for 128-bit registers and 16 parallel byte operations [9]. Having access to such SIMD support could have been of tremendous help to a parallel loop-based implementation of GF multiplication in network coding. Much to our dismay, we discovered that ARM1176's support of SIMD is not a full-fledged SIMD of the NEON type. Rather, it is a limited set of parallel instructions on byte or half-word length granularities of 32-bit registers [10].

We now propose a scheme, inspired by our work in [11], to take advantage of simple ARMv6 32-bit processors, even with no SIMD support, to perform loop-based GF multiplications in parallel. The scheme uses a series of shifts and logical operations to perform a loop-based *byte-by-word* GF-multiplication, through parallel byte-length arithmetic and test operations on 32-bit registers. The basic skeleton of our scheme is shown in Fig. 2.

With such a byte-by-word loop-based implementation of random network coding on the iPhone, at the same ($n = 128, k = 4096$) setting, we achieve encoding and decoding rates of 86.6 KB/s and 81.9 KB/s, respectively. This reflects a speedup of 1.3 and 1.37 over the table-based implementation. However, we are still far short of participating in a high-quality streaming session of 96 KB/s.

2.3 Thumb versus ARM instruction sets

The ARMv6 architecture supports both *Thumb* and *ARM* instruction sets. The *Thumb* instruction set is specifically designed to reduce code density for memory-constrained embedded systems by encoding a subset of the 32-bit *ARM* instructions into a 16-bit instruction set space. The iPhone development platform generates Thumb instructions by default, since it typically reduces code sizes by about 35%. However, we discovered that Thumb instructions has a number of drawbacks in the context of network coding. First, a *predicated instruction* (tagged for conditional execution) is not allowed, leading to two instructions instead of one in the ARM instruction set. More importantly, a Thumb instruction cannot use both the barrel shifter and the ALU unit, unlike an ARM instruction. A barrel shifter can shift incoming data from the register file on its way to the ALU. This is a unique feature of ARM cores that cannot be used with Thumb instructions.

Noting such differences, we have compiled our implementation for the ARM instruction set, and repeated our previous experiments at the ($n = 128, k = 4096$) setting. Table-based coding now improves by 50% and 26% to 100.4 KB/s and 75.8 KB/s for encoding and decoding, respectively. Our loop-based coding improves by 89% and 93% to 163.8 KB/s and 157.8 KB/s for encoding and decoding, respectively. This improvement is essentially due to a reduction in the number of executed instructions. The number of machine instructions executed in each iteration of the GF-multiplication loop is reduced from 17 to 10, by using ARM instead of Thumb instructions. The loop-based implementation has achieved a more substantial gain, due to its heavy use of logical and shift operations, which can be combined into concurrent barrel shift and ALU operations.

2.4 Hand-tuned optimizations

To improve the coding performance further, we attempted to optimize our GF-multiplication by using specific features from the ARMv6 core. Our focus is the multiplication at statement (7) in Fig. 2. By examining the ARM assembly generated by the compiler, we have observed that statements (6) and (7) are combined and performed with only 3 machine instructions that perform a series of shifting and arithmetic operations, and that take advantage of barrel shifts. The full 32-bit multiplication was avoided by the compiler, due to its 2-cycle throughput and an extra 2-cycle output latency.

We note that our multiplication in statement (7) is not a full 32-bit multiplication, and a byte-by-word multiplication would be sufficient. The closest alternative in the ARM instruction set is `SMULW`, a 16-by-32 bit multiplication with a single-cycle throughput. To take advantage of `SMULW`, however, we will have to hand-tune the remaining code to ensure integrity of our computation (`SMULW` operates only on signed values and returns the upper 32-bit of the result). The hand-tuned optimization of loop-based multiplication is shown in Fig. 3. In addition, we explicitly take advantage of the barrel shifter before an ALU operation, as indicated in statement pairs (3)-(4) and (8)-(9). Finally, the calculation steps are reordered based on the timing of individual instructions, in order to minimize pipeline stalls due to the latency of register writes. With our hand-tuned implementation, the

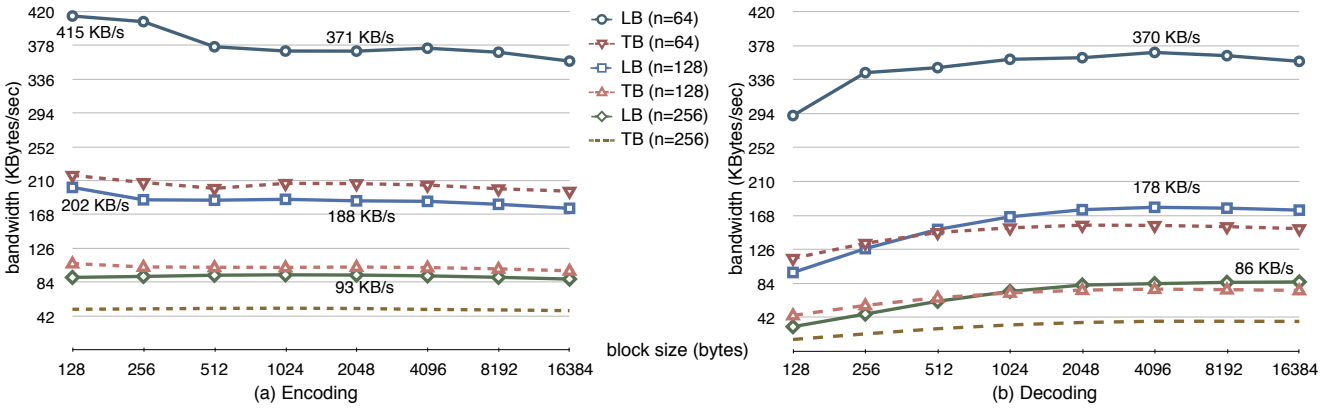

Figure 4: Coding bandwidth of loop-based (LB) and table-based (TB) for network (a) encoding; and (b) decoding processes on the 2ndGen iPod Touch.

number of ARM instructions executed in each iteration is reduced from 10 to 8. This effectively improves the coding performance of our base setting by 11%, to 181.3 KB/s and 175.3 KB/s for encoding and decoding, respectively.

As our final optimization attempt, we take advantage of the only SIMD instruction that may potentially be helpful. The statement pair (8)-(9) was originally meant to shift-left individual bytes of 32-bit `data` without affecting the neighboring bytes. Although these two statements were already compiled to a single machine instruction, replacing them with a single `uadd8` SIMD instruction (effectively doubling individual bytes of `data`) leads to a minor improvement of around 4%. At this point, the encoding performance has improved to 188 KB/s, and decoding to 182.3 KB/s.

3. PERFORMANCE EVALUATION

We now proceed to evaluate the performance of our hand-tuned implementation of random network coding, on an iPhone 3G and a second-generation iPod Touch. The focus of our attention is on the coding bandwidth, CPU usage, and energy consumption, in the context of a realistic application scenario for media streaming.

Our evaluations use fully dense coding matrices with non-zero coefficients unless explicitly mentioned otherwise.

3.1 Coding performance on the iPod Touch

As we evaluate the coding performance on our second-generation iPod Touch, we have tested a range of 128 bytes to 16 KB per block, with 64, 128 and 256 blocks. Both encoding and decoding bandwidth of our table-based and optimized loop-based implementation are shown in Fig. 4. They are interpreted as the total bytes of generated coded blocks (or decoded source blocks) with a (n, k) coding setup within one second. Fig. 4(a) shows that encoding achieves its peak performance across almost all coding settings. It is not a surprise that Fig. 4(b) shows a lower decoding performance, since decoding needs to perform an $O(n^3)$ matrix inversion, in addition to matrix multiplication.

In encoding performance results, an interesting result is the reduction of encoding rates with block sizes beyond $k = 256$ at $n = 64$, and beyond $k = 128$ at $n = 128$. The reduction occurs for both table-based and loop-based implementations, with a sharper decline for the loop-based approach. This behavior leads to an interesting find. The encoding process has a rather fixed working set, which is $n \times k$, *i.e.,* the total size of a segment. The sharp decline happens when the segment size goes beyond 16 KB. Our hypothesis is that the L1 cache size on the iPod Touch is 16 KB, and cache misses have been the cause to such declines, when the source blocks no longer fit in the L1 cache. However, there exists very little public-domain literature documenting the ARM1176 specification of the iPhone platform; and without entering supervisor mode (only possible in the OS kernel), we are unable to read the ARM's status registers with specialized ARM instructions. We eventually managed to use an undocumented interface to the kernel to read such specifications. The L1 data cache is indeed 16 KB across both the iPhone and the iPod Touch, confirming our hypothesis.

The number of executed instructions to achieve an encoding bandwidth of 415 KB/s at $(n = 64, k = 128)$ is estimated to be around 470.2 MIPS (Mega instructions per second). This instruction rate is over 88% of the theoretical limits of 533 MIPS for our ARM1176 processor running at 533 MHz in the second-generation iPod Touch. This represents stellar performance, reflecting that the encoding performance of our loop-based implementation is mainly constrained by the computation limits of the ARMv6 core.

3.2 Coding performance: iPhone vs. iPod Touch

The first-generation iPhone, iPod Touch and the iPhone 3G have all used the same ARM1176 as its main processing core, but they are clocked at 412 MHz (with a 103 MHz bus), about 29% lower than the clock frequency of 533 MHz in the second-generation iPod Touch (with a 133 MHz bus). Fig. 5 compares the encoding performance of a second-generation iPod Touch and an iPhone 3G. The ratio in coding performance very closely reflects the ratio of processor frequencies.

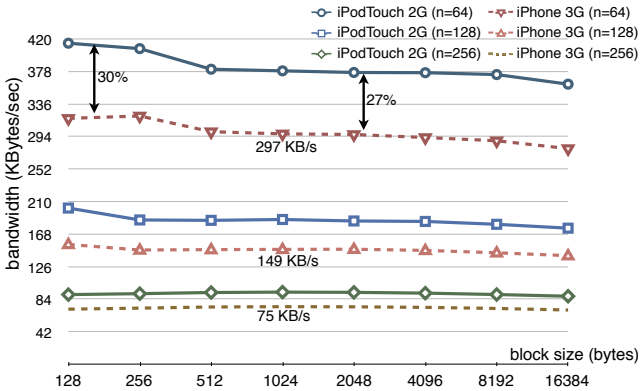

Figure 5: Coding bandwidth of loop-based network encoding: iPhone 3G vs. 2ndGen iPod Touch.

3.3 Is network coding feasible for media streaming on the iPhone?

We are now ready to investigate the following question: From the perspectives of CPU usage and energy consumption, is it feasible to incorporate random network coding as part of a P2P media streaming solution on the iPhone platform? Although we are not yet at the stage of deploying a complete and working P2P media streaming system on the iPhone, it would be interesting to study its feasibility before such a deployment. We would like to know if there exist realistic network coding settings that can work efficiently while the content of a media stream is received and played back on the iPhone.

3.3.1 Profiling YouTube playback on the iPhone

Before bringing network coding to the picture, we need to know more about properties of receiving and playing back content of a video stream on the iPhone platform. We use iPhone's `YouTube` player for this purpose, and try to profile the network bandwidth and CPU usage while watching a video stream over the WiFi connection of our iPod Touch (chosen over the iPhone 3G due to its higher clock frequency). We use WiFi connectivity over 3G as it is the common denominator of wireless interfaces across the iPhone family of devices. It also allows us to playback higher quality videos than using the 3G network.

To monitor the activity of processes during playback, we use the `Instruments` application included in the iPhone development environment (`Xcode`), running on a Mac desktop connected to our iPod Touch via a USB connection. We tested three video clips from YouTube, with various media properties. We monitor the CPU usage and network activity while each video clip is played. As it is not possible to save video clips on the iPhone platform, we eventually needed to determine their properties using our Mac desktop.

Table 1: YouTube contents & steaming experiments

Clip name	Size	Bitrate	Length min:sec	CPU usage	Ingress
`History of the Internet`	320x 180	300 Kbps (37.5 KB/s)	8:10	12.9%	61 KB/s
`Validation`	320x 240	400 Kbps (50 KB/s)	16:23	14.1%	101 KB/s
`Obama's Inauguration`	640x 360	620 Kbps (77.5 KB/s)	21:22	14.9%	112 KB/s

Table 1 shows the properties of each clip and our measurement results regarding the average ingress streaming rate and average CPU usage due to video decoding and playback of the clip. It turned that the `YouTube` process is only active for a short period of time, when the user interacts with the GUI to search and to launch a video clip. The decoding and playback are handled by the `mediaserverd` process. The `springboard` process, which manages the matrix of applications on the iPhone, consumes about 2% of the CPU. We noticed a separate `DTMobileIS` process continuously consuming around 6% of CPU cycles, which is the instrumentation service running on the device to collect measurements and to transfer the data via USB to the monitoring Mac desktop.

As noted in Table 1, media decoding and playback consume only a small portion of the CPU processing power, implying that more complex media decoding operations are all delegated to the POWERVR GPU. This is good news for us as it enables us to use the available CPU cycles for network coding. The average ingress rate is higher than the actual video rate because it reflects the raw incoming bytes, and also due to the overhead of the streaming protocol.

3.3.2 CPU usage of random network coding

At this point, we wish to design experiments with network coding that complement the playback of YouTube video clips. Without a fully integrated network coded streaming system, we have designed a simpler experimental setup that is still able to capture the CPU usage of network coding at the streaming rates relevant to each video clip. In our experiments, we have implemented an iPhone application that performs network decoding and encoding at rates corresponding to a realistic P2P media streaming scenario. Since the iPhone prohibits third-party applications running as a background process, we have no choice but to run our streaming experiments in a standalone manner, *i.e.,* without the `YouTube` application running concurrently.

We first need to determine the appropriate settings for network coding in our video clips. Our $(n = 128, k = 4096)$ benchmark leads to a segment size of 512 KB, suitable for higher quality streaming rates, such as 768 Kbps. For a 300 Kbps clip such as `History of the Internet`, a segment size of 512 KB would correspond to over 13 seconds worth of content, leading to a long initial buffering delay. We have selected a segment size of 256 KB, corresponding to initial buffering delays of 3–7 seconds in our three video clips. With this segment size, we intend to evaluate two network coding settings: $(n = 128, k = 2048)$ and $(n = 64, k = 4096)$.

If a network coded P2P streaming system is deployed on the iPhone, it not only needs to decode the incoming stream, but also needs to encode and serve a small number of neighboring nodes with network coded blocks. We intend to run different experiments for 1, 2 and 4 downstream nodes. Unfortunately, based on our coding performance results, the ARMv6 core does not have the computation power to decode at the video bit rate R and encode at $4R$ (for four downstream nodes). To reduce the computation load, we vary the density of random network coding. Existing work [12] has demonstrated that the coding matrix can be as sparse as a $7 - 10\%$ density setting, without increasing the risk of linear dependence among coded blocks. Conservatively, we use a density of $1/d$ if d downstream nodes are served concurrently so the aggregate encoding rate will still be R.

Table 2 shows the average CPU usage in our experiments, with network coding performed at the corresponding streaming rates of the three test video clips. In addition to the actually measured CPU usage, we have also estimated the CPU usage through CPU usage$_{\text{Est}} = R/\text{BW}_{\text{enc}} + R/\text{BW}_{\text{dec}}$, where BW reflects the coding bandwidth from Fig. 4 at the related setting. As observed from the table, the measured CPU usage for $d = 1$ is very close to our estimates based on the above formula. Also, the CPU usage of $(n = 64, k = 4096)$ is almost half of $(n = 128, k = 2048)$'s across the board as expected. However, the CPU usage decrease for $d = 2$ and even further for $d = 4$ are surprising results. This turns out to be from improved decoding performance as the codes become sparser. The last column, $d = 0$, corresponds

Table 2: CPU usage of network coded streaming.

	$(n = 128, k = 2048)$				
Clip name	Est.	$d = 1$	$d = 2$	$d = 4$	$d = 0$
`Hist. of the Internet`	41%	42%	37%	35%	22%
`Validation`	55%	55%	49%	44%	31%
`Obama's Inauguration`	86%	85%	75%	69%	44%
	$(n = 64, k = 4096)$				
Clip name	Est.	$d = 1$	$d = 2$	$d = 4$	$d = 0$
`Hist. of the Internet`	21%	22%	18%	17%	12%
`Validation`	27%	28%	25%	22%	16%
`Obama's Inauguration`	43%	43%	38%	35%	22%

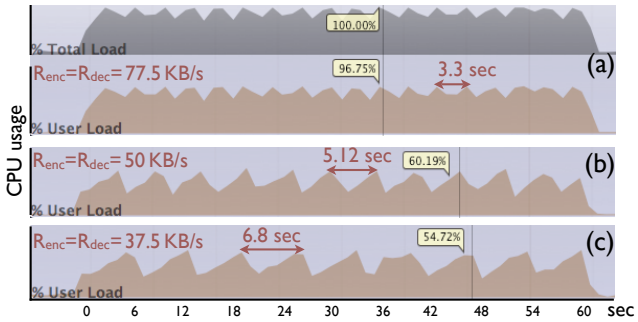

Figure 6: Instantaneous CPU usage for $d = 1$.

to decoding a stream at rate R without encoding, *e.g.,* not serving any neighboring node. The CPU usage in this setting is about 51% to 58% of the $d = 1$ setting, when encoding was concurrently present. This setting reflects a realistic mobile node which does not have incentives to serve others, or does not have the capability, *e.g.,* due to a low battery.

Our experiments on the instantaneous CPU usage have revealed other interesting discoveries. Fig. 6 shows the CPU usage over the course of our $(n = 128, k = 2048)$ experiments, at three different video bit rates. Fig. 6(a) shows both user-mode and the total system CPU usage to emphasize that other background tasks, *e.g.,* the instrumentation service on the device, and some system tasks eventually increase the CPU usage to 100% at some instances. In addition, we observed a sawtooth shape of the CPU usage. While encoding a number of blocks takes a constant time for each block, decoding a full segment, n blocks, with Gauss-Jordan elimination has varying complexity, from one to $2n - 1$ row operation(s) [7], causing such a sawtooth phenomenon.

3.3.3 Energy consumption with network coding

To test the consumption of energy with network coding, we use some unofficial header files to query the remaining capacity of the battery, as a percentage of the full capacity. We playback the `Inauguration` clip as the foreground application while our power metering utility is running as a console application launched through a remote session into a "jailbroken" iPhone over WiFi. We have tested three settings at $(n = 64, k = 4096)$: video playback without network coding, playback with network decoding at 77.5 KB/s, and playback with both encoding and decoding, each at 77.5 KB/s.

The decline of iPhone's battery energy reserves over a 31 minute period is shown in Fig. 7. The results reflect that about 33% of the reduction is due to network encoding and decoding, with decoding contributing to around 15% of the reduction. This result should be treated as a first-order estimate, since the accuracy of the API is not high.

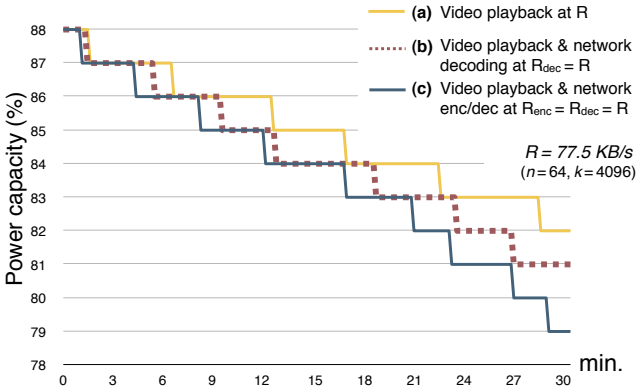

Figure 7: Energy consumption on the iPhone 3G.

For our discussions, we are able to conclude that, as a general guideline, network coding with 128 blocks results in excessive CPU usage, especially for video streams with high bit rates. Though the CPU usage does not linearly relate to energy consumption, it certainly is one of the leading causes. We believe that network coding with 64 blocks is more suitable to the current generation of the iPhone platform.

4. CONCLUSIONS

This paper presents the first real-world implementation of random network coding on smartphones, and in particular on the iPhone family of mobile devices. We provided a highly optimized network coding implementation for the ARMv6 architecture core which should be applicable, directly or with minor modifications, to the majority of mobile devices in the market. An in-depth analysis of coding bandwidth, CPU usage and energy consumption experiments were presented. The verdict is that it is possible to take advantage of network coding at realistic P2P video streaming rates on current-day mobile devices. With a coding setting of 64 blocks of 4096 bytes each, decoding even at high rates of 620 Kbps is possible with a 22% increase in CPU load. Generating encoded blocks for four neighboring nodes, beside decoding, for the same 620 Kbps rate is possible with a 35% increase in CPU load.

In a realistic P2P streaming scenario, however, the tradeoffs between CPU usage and power consumption might limit the use of network coding to low rates. There are a number of high-level system design decisions to make in order to justify the use of network coding. On the other hand, mobile devices are evolving rapidly and equipped with more advanced hardware in each new generation. Faster and more advanced processors would further assist to improve the performance of network coding. As we see it, as mobile hardware platforms advance in the future, the tradeoffs will be more aligned in favor of using network coding on mobile devices for real-world multimedia streaming applications.

5. REFERENCES

[1] R. Ahlswede *et al.* Network Information Flow. *IEEE Trans. on Information Theory*, 46, July 2000.

[2] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. Trading Structure for Randomness in Wireless Opportunistic Routing. In *Proc. of ACM SIGCOMM*, 2007.

[3] S. Katti *et al.* Symbol-level Network Coding for Wireless Mesh Networks. In *ACM SIGCOMM 2008.*

[4] M. Pedersen *et al.* Implementation and Performance Evaluation of Network Coding for Cooperative Mobile Devices. In *Proc. of IEEE ICC Workshops*, May 2008.

[5] S. Katti *et al.* XORs in The Air: Practical Wireless Network Coding. In *Proc. of ACM SIGCOMM*, 2006.

[6] C. Gkantsidis and P. Rodriguez. Network Coding for Large Scale Content Distribution. In *INFOCOM 2005.*

[7] H. Shojania and B. Li. Parallelized Network Coding With Hardware Acceleration. In *IEEE IWQoS 2007.*

[8] ARM Ltd. *ARM1176JZ(F)-S Processor Sheet.*

[9] ARM Ltd. *NEON Support in the RealView Compiler.*

[10] ARM Ltd. *ARM Architecture Reference Manual*, 2005.

[11] H. Shojania, B. Li, and X. Wang. Nuclei: GPU-accelerated Many-core Network Coding. In *INFOCOM 2009.*

[12] G. Ma, Y. Xu, M. Lin, and Y. Xuan. A Content Distribution System based on Sparse Linear Network Coding. In *Proc. of NetCod 2007.*