Tenor: Making Coding Practical from Servers to Smartphones

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

ABSTRACT

It has been theoretically shown that performing coding in networked systems, including Reed-Solomon codes, fountain codes, and random network coding, has a clear advantage with respect to simplifying the design of protocols. These coding techniques can be deployed on a wide range of networked nodes, from servers in the "cloud" to smartphone devices. However, large-scale real-world deployment of systems using coding is still rare, mainly due to the computational complexity of coding algorithms. This is especially a concern on both extremes: in high-bandwidth servers where coding may not be able to saturate the uplink bandwidth, and in smartphone devices where hardware limitations prevail.

In this paper, we present *Tenor*, a comprehensive toolkit to make coding practical across a wide range of networked nodes, from servers to smartphones. We strive to push the performance of our cross-platform coding toolkit to the limits allowed by off-the-shelf hardware. To show the practicality of the Tenor toolkit in real-world network applications, it has been used to build coded on-demand media streaming systems from a GPU-based server to up to 3000 emulated nodes, and to iPhone devices with actual playback.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*

General Terms

Algorithms, Performance, Experimentation.

1. INTRODUCTION

It has been well recognized that innovative coding techniques, such as fountain codes [8] and network coding [1], has the theoretical potential to improve network performance. As examples, Byers *et al.* [2] have clearly illustrated the benefits of fountain codes in bulk content distribution systems. Ho *et al.* [5] proposed random network coding with random linear codes, in which a node in a network topology transmits a linear combination of incoming packets to its outgoing links. The coding coefficients in such a linear combination is chosen randomly over a finite field. Wu *et al.* [3] and Ghantsidis *et al.* [4] have further shown that random network coding is beneficial in bulk content distribution systems. In general, when coding is applied, the system benefits from its superior resilience to node departures and packet losses. With network coding and fountain codes, multiple servers can simultaneously serve a single receiver with a substantially simplified design of block reconciliation protocols.

However, large-scale real-world deployment of systems and applications using any of the coding techniques is still rarely seen. We believe that the main hurdle consists of the computational complexity of the encoding and decoding processes, a price that has to be paid to gain access to coding advantages. Theoretically, the high computational complexity of random linear codes is well known, and is used to motivate the application of more efficient codes, such as traditional Reed-Solomon (RS) codes and, more recently, fountain codes. While fountain codes are much less computationally intensive as compared to random linear codes, they suffer from a number of drawbacks: (1) coded blocks cannot be recoded without complete decoding, which defeats the original intent of network coding; (2) depending on the code used, there exists an overhead (about 5% with 10,000 blocks, and over 40% with 100 blocks), which decreases the efficiency of using bandwidth; and (3) the decoding process cannot be progressively performed while receiving coded blocks, which may lead to bursty CPU usage when the final blocks are decoded. Alternatively, while Reed-Solomon (RS) codes may also be used to reduce coding complexity, its smaller coded message space makes it difficult for multiple independent encoders to code a shared data source, to be sent to a single receiver.

From the perspective of computational capabilities of off-the-shelf hardware, we have recently witnessed a slew of state-of-the-art hardware advances. Graphics Processing Units (GPUs) have evolved to programmable general-purpose *throughput computers*. One NVIDIA GTX 280, for example, attains a peak performance of 1080 GFLOPS. On the extreme of mobile smartphone devices, the ARM Cortex-A8 core, being used in the iPhone 4, includes a full Single-Instruction, Multiple-Data (SIMD) implementation called NEON. Moore's Law dictates that these examples will only become more capable in computational performance.

Have we reached an era when coding techniques can be performed in networked systems and applications without serious concerns of their computational complexities? To answer this question, we believe that the development of new systems and protocols with coding should be motivated and promoted by a *toolkit* with coding implementations that take full advantage of a complete range of off-the-shelf computing hardware. Components of this toolkit should not be just reference implementations: instead, they should attain the highest possible performance with hand-tuned assembly level optimization, tailored to specific hardware platforms, such as GPUs and smartphone devices.

MM'10, October 25–29, 2010, Firenze, Italy.

In this paper, we describe *Tenor*, a comprehensive toolkit to make coding practical across a wide range of hardware platforms. Tenor supports random linear coding, fountain codes (LT codes), and Reed-Solomon codes in CPUs (single-core and multi-core), GPUs (single and multiple), and recent ARM-based mobile devices. Tenor is cross-platform with support on Linux, Windows, Mac OS X, and iPhone OS, and supports both 32-bit and 64-bit implementations. Tenor can be readily used as a black box without any knowledge of its implementation details.

In order to validate the effectiveness of the Tenor toolkit, we have built a coding-based on-demand media streaming system with a GPU-based 8-core Intel server, thousands of emulated clients, and a small number of actual iPhone family devices. Our experiences with this system, presented in this paper, offer an excellent illustration of *Tenor* components in action, and their benefits in rapid system development. With *Tenor*, it is trivial to switch from one coding technique to another, scale up to thousands of clients, and deliver actual video to be played back even on the latest iPhone 3GS.

Throughout this project, which completes our efforts in a 3-year period, we are convinced with our experiences that, with optimized implementations in *Tenor*, off-the-shelf hardware is sufficiently sophisticated to bear the computational load of coding tasks. On high-bandwidth servers, with the help of GPUs, we are able to saturate two Gigabit Ethernet interfaces with coding performed in real time. On mobile devices, the latest ARMv7 architecture is sufficient to support random linear coding with a realistic range of settings.

2. HARDWARE SUPPORT FOR CODING

In *Tenor*, we exploit the existing parallelism in each of the coding techniques to take full advantage of different hardware platforms. We first present a brief overview of the hardware features available on each platform.

Modern off-the-shelf CPUs: In our target coding applications, blocks of data from a few hundred bytes to several kilobytes of length are combined together in tight loops. Naturally, processing longer chunks of data at once, beyond the native width of the processor's Arithmetic Logic Units (ALU), can speed up the operation. SIMD instruction sets offer substantial assistance in Tenor, as they allow parallel operations to be performed on multiple data. As commodity processors have migrated to multi-core architectures, our other parallelization venue is to utilize multiple processing cores.

Graphics Processing Units: Modern GPUs have gradually evolved from specialized engines operating on fixed pixels and vertex data types, into programmable parallel processors with enormous computing power [7]. NVIDIA's *Tesla* is the most popular GPU architecture that enables high-performance parallel computing through the CUDA programming model and development tools. The GPU dedicates its die area to a higher number, albeit simpler, processing cores. Hundreds of such cores result in a level of parallel computing power exceeding multi-core CPU-based systems. Further, with wider and faster memory interfaces, the GPU has a much higher memory bandwidth at its disposal than the CPU.

Mobile devices: ARM processors are the most widely deployed processors for mobile devices. Among them, the ARMv6 architecture is used in a wide variety of mobile devices, including the iPhone 3G, but only features a plain 32-bit ALU. Recently, however, the ARMv7 architecture, found in Cortex-A8 cores employed in devices such as the iPhone 3GS or iPhone 4, have featured full SIMD support and opened up new opportunities.

3. TENOR: MAKING CODING PRACTICAL

Tenor includes high-performance implementations of a number of coding techniques: *random linear codes (RLC)*, *fountain (LT) codes*, and *Reed-Solomon (RS) codes* in CPUs (single and multi core(s) for both x86 and IBM POWER families), GPUs (single and multiple), and mobile/embedded devices based on ARMv6 and ARMv7 architectures. Tenor is cross-platform with support on Linux, Windows, Mac OS X, and iPhone OS, and supports both 32-bit and 64-bit platforms, where applicable. The toolkit includes 23K lines of C++ code.

Tenor can be readily used as a black box without any knowledge of its implementation details. By exposing simple interfaces, Tenor allows seamless use of coding techniques in applications. An algorithm just needs to be set up with coding parameters before it proceeds with encoding or decoding processes. The rest of the coding process is handled transparently by Tenor. This allows applications to experiment with different coding techniques with minimal changes. In addition to our heavy use of hardware acceleration and optimization of individual coding schemes, extra system-level measures have been taken in Tenor to improve performance.

In Tenor, we assume that the source content to be disseminated, *e.g.,* a video file, is divided to a series of segments. Each segment $\mathbf{b}$ is divided into n source blocks $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$, where each block b_i has k bytes, the block size. To encode a new coded block x_j of k bytes, a *code* $\mathbf{C_j} = [c_{j1}, c_{j2}, \cdots, c_{jn}]$, consisting of n coefficients, is chosen and employed to combine the source blocks "somehow" into x_j. The decoding process processes the coded blocks as they are received from the network. The original source segment can be fully rebuilt when a sufficient number of coded blocks, depending on the actual coding technique, are successfully decoded. The process is then repeated for later segments.

The number of blocks per segment n and the block size k are coding parameters that depend on the application, the properties of the coding technique, the computing power of hardware, and network resources. Most practical coding settings are supported by *Tenor*. In particular, we target applications that require high coding rates such as content distribution and multimedia streaming. For each technique, a baseline reference implementation without acceleration, and various high-performance versions are provided in *Tenor*.

Fig. 1 shows the common interface of all three coding techniques in its most simplified form. The segment size n, block size k, and additional parameters specific to the coding technique are passed to Tenor to configure the *codec* object. Additional parameters include the initial seed for the pseudo-random number generator, parameters of the Robust Soliton degree distribution for LT codes, and number of threads to be used. The `encode` method accepts a pointer to the original segment, a pointer to the destination of the new coded block, and returns an identifier for the particular code $\mathbf{C_j}$ chosen for this coded block. The identifier can be a random seed for RLC and LT, or a row identifier of the generator matrix for RS coding. Similarly, the `decode` method accepts a code identifier, along with the coded block itself. It returns success when the whole segment is successfully decoded after receiving enough contributing blocks. The `reset` method resets the internal states of the codec to prepare it for coding a new segment. To facilitate various deployment scenarios, other variations of some methods exist as well.

We now present our detailed design for each coding technique.

3.1 Random Linear Network Coding

With random linear codes, a code $\mathbf{C_j}$ is a set of randomly chosen coding coefficients, each of one-byte length, in GF(2^8) Galois Field. For generating a coded block x_j, $\mathbf{C_j}$ is employed through a linear combination shown in Eq. 1. A node that receives n linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ can decode the original segment successfully. It first forms a $n \times n$ coefficient matrix $\mathbf{C}$ with each row corresponding to the coefficients of one coded block. It

```
// configures the codec in Tenor
void configure(int n, int k, [other parameters]);
// generates a coded block
int encode(byte *source_seg, byte *coded_blk);
// decodes a coded block
bool decode(int code_id, byte *coded_blk);
// resets the codec in Tenor
void reset();
```

Figure 1: *Tenor:* the Application Programming Interface.

then recovers the original blocks **b** through Eq. 2. A random seed is often used to identify each coefficient row. Through Gauss-Jordan elimination, the decoding process can occur progressively as coded blocks arrive.

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

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

Similar to the traditional multiplication of large numbers, logarithm and exponential tables have been widely used for fast GF multiplication. Such table-based multiplication, however, requires multiple accesses to the lookup tables, and constitutes the main performance bottleneck in random network coding. To accelerate this costly operation, In our previous work [9], the use of a loop-based approach in Rijndael's finite field has been explored. Although the basic loop-based multiplication takes longer to perform, it lends itself better to a parallel implementation by taking advantage of SIMD vector instructions. On multi-core systems, multithreading further improved the performance, up to linear speedup, by partitioning the coding workload. In addition, our previous work [10] explored network coding implementations based on a single GPU, with both loop-based and table-based schemes. By taking advantage of hundreds of GPU cores, coding rates up to 279 MB/s can be achieved at a typical $n = 128$ setting, far beyond the computation bandwidth required to saturate a Gigabit Ethernet interface on streaming servers.

In *Tenor*, we first bring all implementations of random linear network coding under the same roof through a common interface, and then proceed to include the following new features in our repository of RLC implementations.

3.1.1 Recoding

Recoding is a unique feature of RLC that differentiates it from LT and RS codes. With recoding, a receiving node can generate a new coded block from its received blocks even before the segment is fully decoded. A recoded block is formed by linear combinations of both the coefficient rows and data payloads of the received blocks, even when they are partially decoded. A recoded block, however, requires its full coefficient row be sent along with the data payload, since a random seed can no longer represent the new code $\mathbf{C_j}$ of the recoded block. Our implementation of RLC now fully supports the *recoding* process. Further, the decoding process can now decode any combination of incoming blocks in both forms: seed-embedded messages directly received through the source node(s), or coefficient-embedded messages received from the neighboring peers. In Sec. 4.4, we will show how recoding helps a P2P live streaming system to serve peers with coded content faster, such that they can meet their playback deadlines.

3.1.2 GPU-based network coding on streaming servers

Thus far in the literature, experimental results on GPU-based network coding have only reflected the raw asymptotic coding rates by generating thousands of coded blocks at once, in scenarios that coding dominates the system overhead, such as the process of transferring data to and from the GPUs. In Tenor, we have gone the "extra mile" to address unique challenges in practical live and Video-on-Demand (VoD) streaming systems.

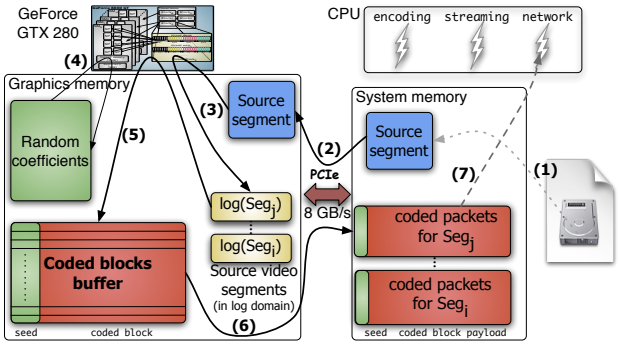

Figure 2: Steps of VoD GPU-based encoding.

VoD streaming systems: In live video streaming systems with GPU-based network coding, each source video segment is coded for many clients, such that thousands of coded blocks can be generated simultaneously. In contrast, VoD streaming systems pose additional challenges, due to the fact that clients request a diversely different distribution of video streams in general. Although a VoD system can employ network coding through *offline* encoding to generate the coded blocks and store them on non-volatile storage, *on-the-fly* encoding has a number of benefits. *First*, it provides "virtually unlimited" number of coded blocks while a pre-stored network coded content only stores a limited number of coded blocks for each source segment. *Second,* an off-line system requires the extra pre-coding step whenever new content is added to the VoD system. *Third,* the pre-coding has to be performed separately for individual servers in the system because replicating a single set of coded content to multiple servers will lead to linearly dependent, *i.e.,* redundant, content. *Finally,* limited disk access bandwidth in an off-line system can be a major bottleneck.

In order to best describe on-the-fly GPU-based RLC encoding in a VoD streaming server, we consider a $(n = 128, k = 4096)$ configuration, where $c = 128$ coded blocks are to be generated for each client at the GPU side and then delivered to the system memory. When a client node requests c coded blocks of a video segment from the VoD server, the following steps, shown in Fig. 2, ensue: (1) loading the source video segment from disk to the main memory; (2) transferring the entire segment, $n \times k$ bytes, to the GPU memory over the PCIe bus; (3) preprocessing the segment by transforming it to the `log` domain [10]; (4) generating c rows of random coefficients; (5) encoding the blocks to generate c coded blocks; (6) retrieving the coded blocks, with a total of $c \times k$ bytes, from the GPU memory to the system memory; (7) packetizing the coded blocks and streaming them out to the client. If steps (2) through (6) are performed serially, the overall encoding process slows down by 27% to 209 MB/s.

In Tenor, we have used a more recent feature of CUDA devices that allows PCIe bus transactions proceed in parallel, in both directions, with GPU kernel executions. This effectively implies that better performance can be gained by breaking each encoding task to three stages and executing each in a "pipelined fashion" working on three successive tasks in parallel. We use CUDA's *stream* construct to manage such concurrency, by assigning the *pipeline stages* to streams, according to the following: *Stream 0* performs step (2) and (3) for the new task. *Stream 1* performs steps (4) and (5) to generate coded blocks for a task that already has its source segment moved into the GPU memory in the previous time slot. *Stream 2* performs step (6) to retrieve the coded blocks generated in the previous slot.

With such pipelined processing, the overall encoding rate is substantially improved to 262 MB/s, only 1% below the raw encoding

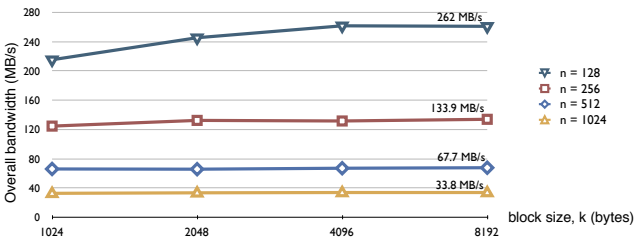

Figure 3: GPU-based encoding performance with pipelined processing in VoD streaming servers.

rate of 265 MB/s with $c = 128$. Fig. 3 shows the encoding performance for the VoD system of Fig. 2 across practical block sizes. Pipelined processing manages to mask most overhead of a practical VoD server deployment.

Live streaming systems: In a live video streaming setup, the number of generated coded blocks c for a video segment is much higher than the VoD case, since there are many more clients sharing a video channel. The overhead of transferring source segments to the GPU memory, as well as retrieving coded blocks back to the system memory, can be efficiently masked by using the same pipelined processing in VoD systems. This implies that the effective encoding performance will be still as high as raw encoding results with GPUs [10]. As an example, serving 5 clients by encoding only $c = 600$ blocks is sufficient to attain an effective coding rate of 278 MB/s at the $(n = 128, k = 4096)$ setting.

3.1.3 Network coding with multiple GPUs

Because the encoding process in random network coding is a highly parallel problem, in Tenor, we have included a custom-tailored implementation to exploit extra GPU power from multiple GPUs concurrently, in order to achieve even higher performance. Fig. 4 shows the raw encoding performance for a system running a GTX 280 and a GTX 260 in parallel. Since CUDA requires each GPU be managed by separate CPU threads, we use separate threads for each GPU, each managing a portion of the workload. As our GTX 260 GPU achieves only 0.66 of the performance of GTX 280 encoding, due to its lower number of GPU cores and lower frequency, this performance difference is taken into account when the workload is partitioned. The performance results effectively reflect the performance of individual GPUs added together. Now even at difficult settings such as $n = 1024$, an encoding rate of 58 MB/s can be achieved.

3.1.4 Network coding on the iPhone 3GS

With the absence of SIMD in the ARMv6 architecture in previous-generation smartphone devices, we had to resort to a number of fine-grained hand-tuning optimizations to achieve acceptable RLC coding rates in our previous work [11]. In comparison, the *iPhone 3GS* and *Palm Pre* smartphones, released recently, have both used a 600 MHz ARM Cortex-A8 as the application processor. These ARMv7-based architectures have implemented the NEON SIMD instruction set, which is similar to SSE2 and AltiVec with full support for 128-bit registers and 16 parallel byte operations.

With our first RLC implementation on the iPhone 3GS, we have observed a coding performance improvement of around 3 to 3.9 times over the iPod Touch, utilizing the ARMv6 core without NEON support. While it is encouraging, an inspection of the machine code reveals that the compiler does not generate the most efficient code. By hand-tuning portions of GF-multiplication through inline assembly, our new implementation improves the performance by another 46%, up to 5.7 times advantage over the iPod Touch, as shown in Fig. 5. The coding rate drops when the working set increases beyond the 256 KB L2 cache. As the iPhone 3GS is running at 600

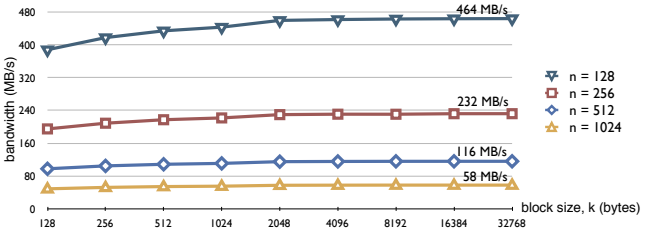

Figure 4: Multi-GPU encoding with GTX 280 and GTX 260.

MHz, only 13% faster than the iPod Touch's 533 MHz, such a performance improvement is directly due to the addition of NEON instructions and the new L2 cache. Comparing the accelerated loop-based coding to the legacy table-based implementation reveals a 3.5 to 6 advantage, consistent with our results on desktop CPUs [9].

We are pleasantly surprised by the performance of the ARM Cortex-A8 architecture: with a *single* SIMD unit at 600 MHz, it achieves nearly 1 MB/s at $n = 128$, while an Intel Xeon with *three* SIMD units at 2.8 GHz, achieves 9.4 MB/s. Such high coding rates open up new opportunities for coding-based streaming applications on smartphone devices. Decoding a high quality video stream at 768 kbps will increase the CPU usage by no more than 5% and 10% for $n = 64$ and $n = 128$, respectively.

3.2 LT Codes

To code a new coded block x_j, the LT encoder randomly chooses a degree d_j from a degree distribution $\mu(d)$. Then, d_j blocks are chosen from the n source blocks and combined together by xor-ing them [8]. In practice, the overhead is around 5% of the original n blocks when n is in the order of 10000. However, the overhead increases as n decreases. As a result, LT codes are usually configured with a higher n than RLC.

Fountain codes are *rateless* in the sense that the number of coded blocks that can be generated from the source segment is potentially unlimited, in sharp contrast to RS codes. The main advantage of LT codes is their low complexity. Even with n as high as thousands of blocks, only a few tens of blocks are xor-ed together, on average, for each coded block, *e.g.,* 17 at $n = 10240$ and 12 at $n = 1024$. In contrast, RLC requires n linear combinations in $GF(2^8)$. A *robust soliton degree distribution* guarantees a sufficient number of *degree-one* coded blocks, coded blocks with $x_j = b_i$, to "kickstart" the decoding process.

3.2.1 LT codes on the CPU

Implementing the LT encoder is quite straightforward. A random seed can be used to convey the selected code, which identifies the code degree and the subsequent selected blocks. The decoder, however, is much more complex as the decoding process is performed through maintaining a sparse graph.

First, the code associated with the received coded block is retrieved through the random seed, *i.e.,* retrieving the degree and block indices, and kept in a list associated with the coded block x_j. If an original block contributing to this coded block is already decoded fully, we partially decode x_j by removing its dependency on that original block. Another category of lists tracks each original block b_i to quickly figure out which coded blocks it has contributed to. *Second*, whenever an existing coded block is fully decoded, its associated original block is applied to all other coded blocks that depend on it, so that their decoding can further progress. The lists are heavily accessed to maintain and reduce the decoding graph. At $(n = 10240, k = 1024)$ as our benchmark, encoding and decoding rates are 32.9 MB/s and 6.4 MB/s, respectively, with such baseline implementation. Noting that block operations involved in both en-

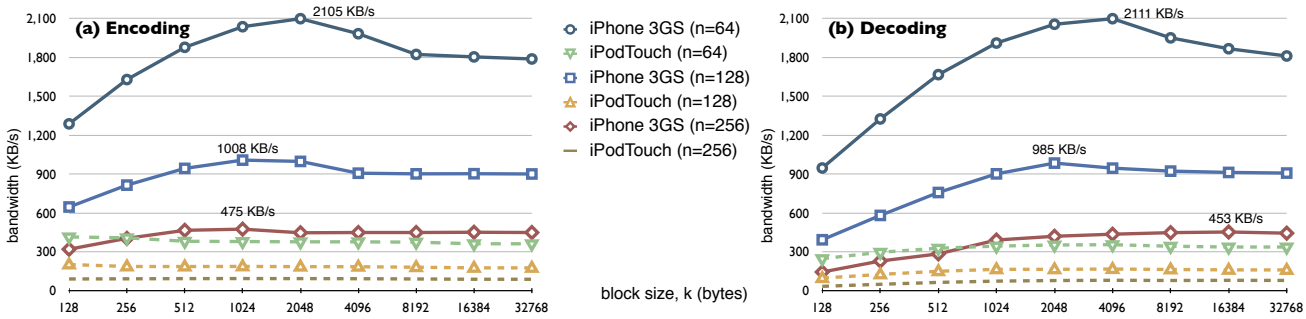

Figure 5: Random linear coding performance of the new iPhone 3GS versus the 2nd generation iPod Touch.

coding and decoding are equal, the performance difference is directly related to the maintenance of the decoding graph.

Our first step of acceleration uses SIMD instructions for block operations, along with other code optimizations. This improves the encoding and decoding rates to 165.5 MB/s and 9.3 MB/s, respectively, which represents a substantial improvement over our baseline encoding, but still a slow decoding process. In the second step, suspecting that dynamic lists associated with the sparse graph slow down the decoding process, we have resorted to coarsely allocated tables to manage the graph through bit masks and arrays. These tables have roughly 19 MB of memory footprint for the same n = 10240 setup and involve the search of a sequence of bits, accelerated with specialized instructions. The performance improvement is dramatic, and results in a decoding rate of 144.1 MB/s, much closer to the encoding performance.

Tailored to high performance coding servers, we implement a multi-threaded encoding process by launching one thread per core. Partitioning each coded block, however, brings little benefit, especially at smaller block sizes. The encoding rate of an 8-threaded implementation increases only by 13% to 186.7 MB/s. Alternatively, the performance can be improved further by fully coding each block in one of the threads, increasing the rate to 447 MB/s, a speedup of 2.7.

In Fig. 6-(a), *CPU-accel* and *CPU-th8* graphs respectively present the accelerated and 8-threaded accelerated encoding rates across a range of block sizes. As the block size k increases, the encoding rate increases initially but then drops across the board. This decrease occurs when the working set becomes too large to fit the 6 MB L2 cache available for each pair of cores in our system. The working set is dominated by the segment size, $n \times k$. In Fig. 6-(b), *CPU-accel* and *CPU-opt* respectively present the accelerated and graph-optimized decoding rates. The advantage of better graph maintenance is obvious, in particular, at smaller blocks.

3.2.2 LT codes on the GPU

An efficient port of LT codes to the GPU turns out to be challenging. GPU threads perform well when all threads of each *thread block* follow exactly the same execution path. The randomness of the degree distribution, however, causes the encoding of different blocks to take highly variable times, degrading the overall performance.

For encoding, we have first designed a joint CPU-GPU scheme that uses the CPU to generate the codes. In other words, CPU uses the GPU as an accelerator for performing the block operations. However, this only achieves an encoding rate of 172.8 MB/s at our benchmark, a minor improvement over the CPU-based rate of 165.5 MB/s. Roughly half of the coded blocks end up with a code degree of 2 but some reach degrees as high as 9335. This leads to a huge imbalance between the workload of the GPU cores, especially when a coded block of a higher degree is processed towards the end of the encoding task, leaving many other GPU cores idle. This imbalance can be mitigated by "sorting," such that GPU threads start with the more time-consuming coded blocks, *i.e.,* of higher degrees. This leads to a significant speedup and increases the encoding rate to 590 MB/s.

In the next phase, we also migrate the code generation process to the GPU. This involves non-intrusive modifications to make it suitable for GPU-based coding. Multiple GPU kernels are called sequentially to perform different stages of encoding. We omit a detailed discussion for the sake of brevity. Among them, code generation is the most time-consuming part, taking 58% of the execution time. The block operations and the sorting stages take 39% and 2% of the execution time. Our fully GPU-based encoding now achieves 1697 MB/s at our $(n = 10240, k = 1024)$ benchmark. Fig. 6-(a) shows the encoding results with rates up to several GB/s.

For decoding, we implement a joint CPU-GPU scheme. The CPU first decodes the code graph and generates a command stream to instruct the GPU about how the received coded blocks should be decoded. At small block sizes, the performance results are not too interesting due to a lack of sufficient parallelism in block operations. For $k \geq 4096$, as the parallelism increases, GPU-based decoding starts to defeat the CPU as shown in the upper chart of Fig. 6-(b).

One particular benefit of GPU-based decoding can be observed in content delivery over high-capacity links with large segment sizes. With LT, almost all of the decoding computation load occurs towards the very last leg, causing long period of full CPU usage, *e.g.,* 171 ms for a segment of 40 MB coded at (n=10240, k=4098). This, if not properly addressed, can lead to undesirable side effects, *e.g.,* the loss of incoming data over a lossy channel. Alternatively, GPU-based decoding has a 29 ms CPU footprint, a 83% reduction.

3.2.3 LT codes on the iPhone 3GS

In the final phase of our optimized LT coding implementation in Tenor, we implement LT-based encoding and decoding on the ARM Cortex-A8 core, taking advantage of its NEON SIMD instructions for block xor operations. The achieved results are shown in Fig. 6. At n = 10240, we are not able to evaluate coding rates for $k \geq 8192$, due to the limited 256 MB memory on the iPhone 3GS.

The low computation overhead of LT codes can be clearly observed in this example. At a fixed segment size of 2 MB, one could choose RLC at (n = 64, k = 32 KB) or LT codes at (n = 1024, k = 2 KB). At these settings, iPhone 3GS achieves a RLC decoding rate of 1.8 MB/s, while LT achieves 31 MB/s, a 17 times advantage. Of course, a 10% network overhead of LT codes at n = 1024, as well as its lack of the recoding capability, can be prohibitive factors in some setups.

3.3 Reed-Solomon Codes

Unlike the typical use of RS codes for *error correction*, our implementations of RS codes are mainly aimed for the same streaming and content distribution applications as RLC and LT codes. RS codes, similar to RLC, generate coded blocks by linearly combining the source blocks in Galois Field. In RLC, the coefficient codes

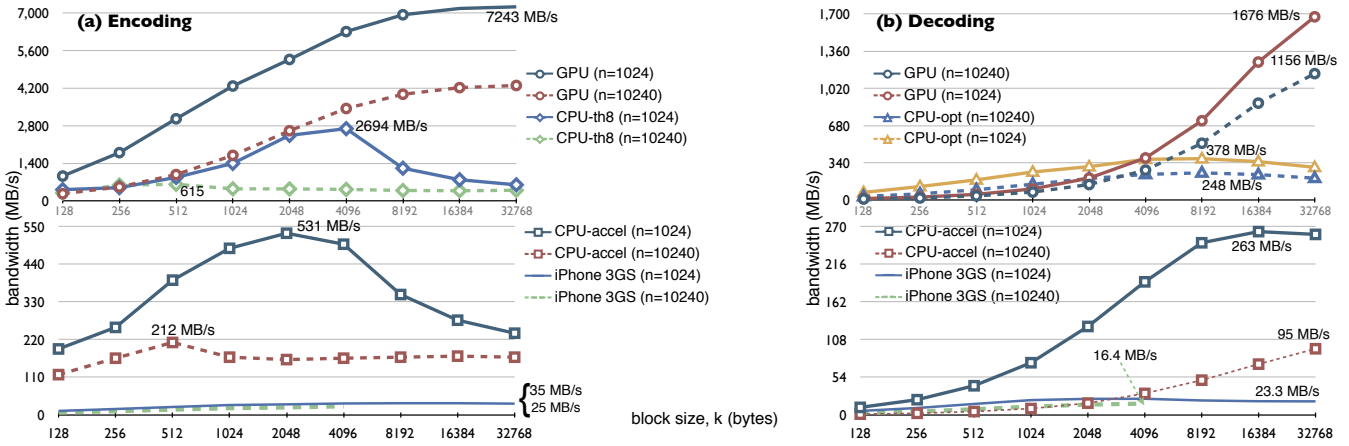

Figure 6: Coding bandwidth of accelerated fountain (LT) codes on different platforms.

are randomly generated, but in RS codes, each $\mathbf{C}_j$ is pre-determined based on a structured *generator matrix*. With a source segment divided into n blocks, an RS code with a (m, n) setting can generate up to m unique coded blocks. Our interface naturally includes a `set_generator` method to pass a generator matrix to the codec. The `encode` method returns a row index of the generator. At decoding, the index identifies $\mathbf{C}_j$ so the code matrix $\mathbf{C}$ is rebuilt after receiving n coded blocks. After calculating $\mathbf{C}^{-1}$, the source segment is recovered.

Any $m \times n$ matrix can be a valid generator matrix provided that any n rows of it can form a full rank $n \times n$ matrix. There are well-known generators that support a wide range of (m, n) settings. To accommodate a general setup, our implementation in Tenor does not make assumptions about any specific generator matrix, *i.e.,* any valid generator is supported. A drawback is that we can not take advantage of the structure to come up with more efficient calculations of the inverse of the sub-generator matrix. For example, a direct inverse of the Vandermonde matrix in [6] takes $4.5n^2$ operations compared to n^3 with Gaussian elimination. On the other hand, fast computation of $\mathbf{C}^{-1}$ does not improve performance much, because it comprises a small fraction of the decoding process in our typical settings, *e.g.,* only 3.1% with $(n = 128, k = 4096)$.

High performance implementations of RS coding in $GF(2^8)$ require only minor changes from our RLC implementation, also operating in $GF(2^8)$. Compared to RLC, the main advantage of RS codes is the guaranteed decode-ability as all coded blocks are guaranteed to be linearly independent. However, the structure constraints the space of coded blocks, as compared to the very large code space in RLC. For our target applications, especially when *error resilience* is a concern (*e.g.*, with lossy transmissions over UDP), a larger set of codes are normally needed beyond the maximum 255 codes provided by $GF(2^8)$-based RS codes. As a result, our main focus here is to implement high performance RS coding in $GF(2^{16})$, which allows the code space m to grow to as large as $2^{16} - 1$. We use the Vandermonde matrix as the generator in our benchmarks. Sparser generators will result in higher performance.

3.3.1 RS codes on the CPU

Similar to RLC, GF multiplication is the bottleneck of the coding process in $GF(2^{16})$. A traditional table-based multiplication in $GF(2^{16})$ requires a `log` table of 2^{16} and a `exp` table of 2^{17} entries, each taking two bytes. Unlike $GF(2^8)$ `log/exp` tables, these tables are so large that easily overflows the L1 cache. With table-based GF-multiplication in $GF(2^{16})$ doing even worse than $GF(2^8)$, our accelerated implementation follows the same loop-based principle of our RLC implementation, by performing a coefficient by 16-byte GF-multiplication in $GF(2^{16})$, where each coefficient is two bytes. Our SIMD implementation now treats each 16-byte sequence of data loaded in its 128-bit registers as 8 short integers. We observe a speedup of between 1.5 to 4 over our baseline implementation, which is more modest than the speedup of 3 to 6 observed in the case of RLC [9]. This is obviously due to the fact that loops now have up to 16 iterations instead of 8. A threaded implementation is also provided.

The accelerated and threaded coding results are shown as *CPU-accel* and *CPU-th8* graphs in Fig. 7.

3.3.2 RS codes on the GPU

Because of the large size of `log/exp` tables, they can not fit in the 16 KB *shared memory* available per Streaming Multiprocessor of the GPU. As a result, an optimized table-based scheme as in [10] is no longer possible. Instead, we follow the loop-based approach but operate in $GF(2^{16})$. Since each GPU core has a 32-bit ALU, we encode a word-length portion of a coded block by each GPU thread. Each word is treated as two short integers in $GF(2^{16})$.

The encoding performance is shown in Fig. 7-(a). The GPU graphs reflect consistent encoding performance across block sizes. It can be observed that the GTX 280 defeats CPU based RS encoding by a substantial margin. At small block sizes, however, the GPU-based decoding performance, shown in Fig. 7-(b), is worse than the threaded CPU-based implementation. This is due to the lack of parallelism as each coded block is decoded one by one, restricting the number of parallel threads to be launched. Once the block size increases, more threads can work in parallel, and the GPU starts to outperform threaded CPU-based decoding.

3.3.3 RS codes on the iPhone 3GS

Our RS coding implementation in Tenor and its accelerated GF-multiplication in $GF(2^{16})$ are similar to our CPU-based RS implementation, and employ NEON SIMD instructions instead. The encoding and decoding graphs are shown in the bottom charts of Fig. 7. Without much surprise, the coding results reflect approximately half of what have been achieved in random network coding with the same number of blocks.

4. PERFORMANCE EVALUATION

Having *Tenor* and its high performance implementations of coding techniques at our disposal, we have developed coding-based VoD and P2P applications in real-life setups that deliver and playback real video. Our applications are deployed in a LAN with realistic setups,

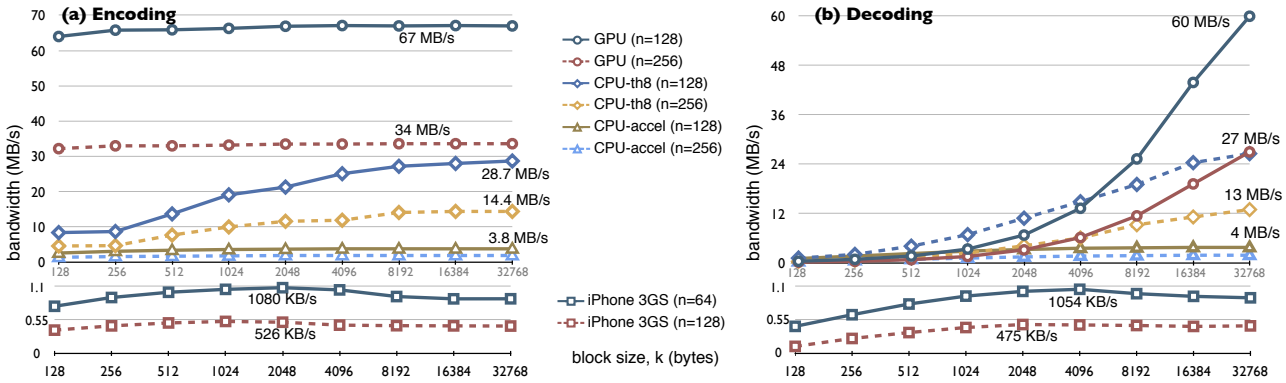

Figure 7: Coding bandwidth of accelerated RS codes in $GF(2^{16})$ on different platforms.

and are pushed to their performance limits. Our goal is not to build the most sophisticated protocols. For example, we use TCP-based delivery in our LAN deployments rather than UDP. Instead, our focus is on the performance aspects of the system with the use of coding techniques.

4.1 An Emulation Framework That Scales

To push the performance of our GPU-based VoD server to its limits, we wish to conduct a large-scale experiment with thousands of clients emulated on a cluster of Linux-based compute nodes. To facilitate deployment of such large experiments, we have implemented an emulation framework that is designed to scale to thousands of clients in the server cluster. With our emulation framework, we are able to emulate hundreds of clients on each physical compute node. Our scalable emulation framework is designed to support multiple connections with independent message queues, to run a large number of emulated clients at different ports of a physical node, to start and stop clients at arbitrary times, and to maintain P2P network topologies among clients. Separate threads are used for emulated clients, so that they can run in parallel and behave independently.

To implement our emulation framework so that it is scalable to thousands of clients in a cluster of only 20 compute nodes, we take full advantage of the `epoll` interface in Linux, specifically intended for scalable networking with asynchronous I/O. As expected, `epoll` turns out to be much more efficient than traditional synchronous networking with the `select` system call. In addition, we have developed our emulation framework so that it is cross-platform, with support for the `kqueue` interface in Mac OS X and iPhone OS as well, which allows us to detect the status of sockets in a similar fashion as `epoll` in Linux. As a result, our emulation framework involves over 17 thousand lines of C++ code, and runs on Linux, Mac OS X, as well as iPhone OS.

4.2 Large-Scale VoD Streaming with Coding

Our setup for a large-scale Video-On-Demand (VoD) client-server experiment, which incorporates coding as a part of its streaming protocol, is shown in Fig. 8. The server node is a Mac Pro (running Linux on two Quad-core 2.8 GHz Xeon processors, 8 GB of memory, two Gigabit Ethernet interfaces, and a GTX 280 GPU with 1 GB of graphics memory), connected to a 20-node server cluster — each running Linux on a Quad-core 3.0 GHz Xeon processor — through a Gigabit Ethernet switch.

In the server, we employ our highly efficient GPU-based coding schemes, which were specifically designed for streaming servers, such that coding rates very close to the raw rates that can be achieved despite thousands of unique clients served in a VoD application. On clients, we employ accelerated CPU-based coding implementations in the decoding process, using our emulation framework to emulate up to hundreds of clients on each physical compute node.

This VoD setup resembles a YouTube streaming server with clients requesting different video contents and the server providing them over TCP. However, the content is transmitted in coded form, *i.e.,* coded blocks generated from video segments of each content. The clients rebuild each segment of the content by decoding the received coded blocks. When a segment is fully decoded, it will be sent to the higher layer video decoder (*e.g.,* H.264) for presentation. The presentation step is skipped for our emulated clients.

An important benefit of coding is that multiple servers can simultaneously serve a receiver with simplified reconciliation protocols. To put the coding capability to good use, we use two servers, emulated as two server applications on our physical server, that seamlessly serve each client. As our GPU-based encoding schemes can generate coded content far beyond the capacity of a Gigabit network interface, we use both Gigabit network interfaces that are available on our server.

We use a *single* disk with a simple setup. Because optimizing disk read access is not our primary objective, we do not wish the disk-to-memory load when serving video content to affect the performance of our experiments. As a result, we cap the number of unique videos in our experiments to 200. Each client, up to 3000 in our experiments, selects one of the 200 videos at random. At a streaming rate of 96 KB/s, this leads to a disk-to-memory transfer rate of about 20 MB/s with the rest of the reads served from the disk cache (SATA disks typically have a raw I/O rate of 30 to 70 MB/s).

4.2.1 *The streaming protocol*

When a client starts, it requests a video from both servers. We use two server applications, both running on our physical server, and each listening to a different port of the *bonded interface*, *i.e.,* the logical aggregate of both interfaces. This effectively emulates a YouTube-like scenario, with one server emulating multiple servers. In an Internet deployment, each server can be placed at a different geographic location. As a server's link bandwidth varies over time, *e.g.,* due to an increased load or a network bottleneck, other servers

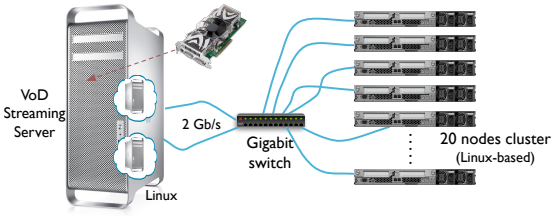

Figure 8: Large-scale Video-on-Demand (VoD) experiments with a server cluster.

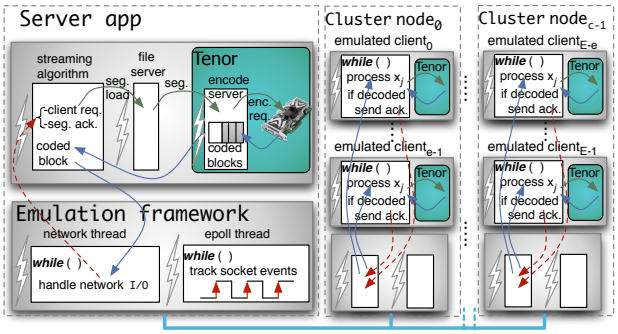

Figure 9: A coded VoD streaming server with its emulated clients.

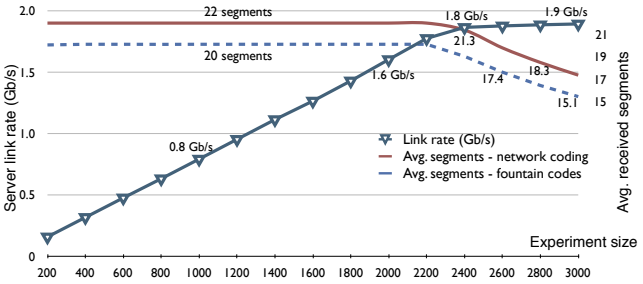

Figure 10: Aggregate link rates from two Gigabit interfaces, and the average number of delivered segments to each client.

can make up for it. In contrast, with coding, the need of reconciling missing blocks becomes unnecessary, simplifying the design of the streaming protocol on servers. As coding removes the uniqueness of individual blocks, any coded block, regardless of its origin, contributes equally to the decoding process. Receiving a sufficient number of coded blocks suffices to recover an original segment.

After an initial handshake between a client and the servers, the servers *push* the coded blocks, generated from each video segment, to the client. The server applications manage their clients through various thread queues. After the initial request from a client for the first segment of a video, it is queued to a dedicated thread, *file server*. After the segment is loaded to the main memory, the thread will be delegated to Tenor's *encoding server*, which performs GPU-based encoding of the loaded segment. The encoding process is executed in several stages in a pipeline fashion described in Sec. 3.1.2.

After coded blocks are retrieved from the GPU, they are buffered in the system memory and gradually pushed to the client according to our streaming protocol. As the streaming process for a client reaches a certain threshold in the active segment, the server generates a new request for loading from the disk, and subsequently performs GPU-based encoding on the following video segment. To ensure smooth transition to the next segment, coded blocks are double-buffered in the system memory: one buffer for the *current segment* which is currently being streamed, and another buffer for receiving coded blocks in the *future segment* from the GPU. The streaming algorithm, both on the server and client sides, relies on our emulation framework for its messaging subsystem and the low-level network operations. Fig. 9 depicts a very high-level view of the components of a server application connecting to a few cluster nodes, each emulating a few clients.

Each coded block is sent within a message that also carries a small header, of 16 bytes, to identify the segment, the server, and the code, *i.e.,* a random seed, associated with the coded block. At the clients, coded blocks are decoded progressively as they arrive. After each segment is successfully decoded by the client, an acknowledgment message is sent to both servers so that they can move on to the following segment. We use segments of 512 KB in length, which corresponds to 5.33 seconds of media at a streaming rate of 96 KB/s. The segment size implies that the initial buffering delay is 5.33 seconds (one segment in length).

We use two different coding techniques in our experiments, so that their performance may be compared.

First, we use random linear network coding with a setting of $(n = 128, k = 4096)$. At such a setting, a single GTX 280 GPU is able to serve at 261 MB/s in an actual VoD setup, sufficient for over 2600 clients, each at a 96 KB/s streaming rate. In the GPU encoder, we generate a slightly higher number of coded blocks for each segment — 130 rather than 128 blocks — to accommodate the rare case that a client needs additional blocks after encountering a linear dependent block. As long as a total of 128 linearly independent coded blocks are received from both servers combined, a segment can be successfully decoded. Each client uses the SSE2-accelerated RLC codec provided by *Tenor*. On each physical compute node, the computational load of the decoding process is manageable for a few hundred of clients.

Second, we use LT codes in our servers to serve VoD clients. Even though LT codes enjoy lower computational complexity as compared to RLC, it incurs additional network overhead, which amounts to 17% at $n = 1024$. One needs to use a larger number of blocks in each segment to decrease such overhead, *e.g.,* $n = 4096$ lowers the overhead to 9%, and $n = 10240$ to 5%. However, a larger n is not suitable for a 512 KB segment: at $(n = 1024, k = 512)$, a block size of 512 bytes will incur a 56-byte header, including both message and TCP/IP headers. This is why LT codes are more suitable for distribution of large files, rather than streaming video with much smaller segments. On the other hand, a large LT codes space is not needed for our experiment because we do not have loss of coded blocks, and we have limited number of servers. In our experimental scenario, we have managed to incur a reasonable amount of overhead of 11.7% by going with *fixed* LT codes in the $(n = 128, k = 4096)$ setting, the same as RLC so that fair comparisons can be made.

4.2.2 *Experimental results*

We have performed a number of VoD streaming experiments with both RLC and LT codes. In our experiments, we vary the number of clients in the system from 200 to 3000, *i.e.,* emulating 10 to 150 clients on each of the 20 compute nodes in our server cluster, and evaluate the streaming performance. In each experiment, all clients start their streaming sessions within a 5-second interval. Each session lasts 120 seconds.

We first wish to investigate the aggregate link rate from both servers serving emulated clients over the bonded network interface, across different experiment sizes. Since both coding techniques operate at the same setting, their link rates are very close, and only one is shown in Fig. 10. However, the average number of fully decoded segments varies between the coding techniques as shown in Fig. 10. With RLC, each client's 96 KB/s streaming rate is fully utilized and all clients receive $120/5.33 = 22.5$ worth of segments streamed to them, and manage to fully decode 22 segments. Of course, this is the case as long as the link capacity from the server is not saturated. When the experiment size grows beyond 2200 clients, the total number of coded blocks sent from the server can not increase beyond the 2 Gb/s link capacity of two Gigabit Ethernet interfaces combined. As the number of clients increases, the total number of delivered segments stays around the same (around 51200 for RLC), and the average number per client decreases. At a peak aggregate link rate of nearly 1.9 Gbps, we are quite close to the 2 Gbps link capacity. Because both servers share the same link, each provides approximately half of each client's streaming rate. The GPU never

reaches its encoding limits as the link capacity is saturated by streaming to over 2200 clients.

In contrast, when LT codes are used in the servers, the average number of fully decoded segments is only 20 at best, rather than 22.5. This reduction is a direct consequence of the network overhead, as 11.7% more coded blocks than the ideal $n = 128$ are required to decode each segment. As a result, the streaming rate of 96 KB/s effectively delivers no more than 85.7 KB/s worth of video streams with LT codes. With RLC, however, the entire 96 KB/s streaming rate is almost perfectly utilized. To be exact, linear dependent blocks only scarcely occur in RLC, leading to very little overhead: no more than 0.0033% of the received coded blocks are linearly dependent.

Fig. 11-(a) shows the main advantage of LT codes compared to random network coding, with respect to the average CPU usage in compute nodes that emulate the clients. As the number of emulated clients increases with larger experiments, the CPU usage (dominated by the decoding load) increases almost linearly. This continues till it saturates beyond the 2200-client experiment, as the aggregate link rate from the servers has become the bottleneck. LT codes have shown a 6-times advantage compared to random linear codes. Such a clear advantage could be a critical factor when coding techniques are being chosen, especially in embedded systems with low-end processors used.

Fig. 11-(b) reflect the percentage of *redundant blocks* received by the clients. Redundant blocks involve the on-the-fly coded blocks that belong to the previous segment, yet received after the segment is successful decoded. They essentially waste network bandwidth, so we wish to keep them low. The number of redundant blocks increases from the 2200-client experiment, as the saturation of line capacity leads to delayed delivery of acknowledgments from clients to servers. In normal experiments without saturating server capacities, however, the number of redundant blocks is virtually zero. This is due to two factors. *First, Tenor* supports a "fast decoding mode" in its decoding implementations, such that an acknowledgment can be sent to servers before the payload of the last block is decoded. *Second,* the very low link delay between our server and compute nodes — typically only a few milliseconds — has contributed to timely acknowledgments in normal circumstances when server capacities have not yet been reached.

4.3 On-Demand Streaming to Smartphones

In our next set of experiments, we use an iPhone 3GS smartphone in a similar setup as Sec. 4.2.1. In the same VoD setup as Fig. 8, we add a wireless router to the LAN. Now our iPhone 3GS, like other clients, initiates a streaming session with both server applications, but through a WiFi connection. The streaming session runs for 120 seconds at 96 KB/s. We proceed with experiments using both random linear codes and LT codes. Our focus is on the performance of using *Tenor* to perform coding on the iPhone 3GS smartphone.

VoD streaming to the iPhone 3GS using RLC. We use random linear codes in the $(n = 128, k = 4096)$ setting as before, and have observed that 21 segments have been successfully decoded by the iPhone 3GS device, rather than 22 segments received by other clients, running on compute nodes in the server cluster. With the iPhone 3GS, the average inter-segment arrival is 5.53 sec compared to the ideal 5.33 sec. Further investigations have shown that this phenomenon is due to longer latencies for acknowledgments to travel over a wireless link from the iPhone 3GS device, as compared to over the Gigabit Ethernet from typical emulated clients in the server cluster. Delayed acknowledgments have delayed the servers when they need to move on to the next segment. When an acknowledgment is being transmitted back to the servers, a few extra coded blocks will be sent by servers and discarded later.

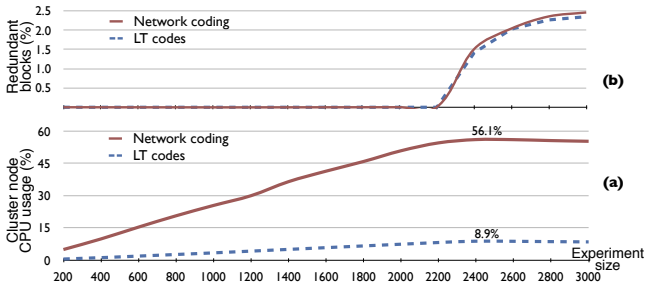

Figure 11: CPU usage and redundant blocks.

One way to overcome longer wireless latencies in our setup is to assume *implicit acknowledgments*, instead of explicit ones, when a server has sent its share of coded blocks. The client now sends negative acknowledgments (NACKs) to the server when it discovers a linearly dependent block, or when it observes that a server is falling behind its expected share of the streaming rate, *e.g.*, due to the saturation of server link capacities. Not surprisingly, the NACK-based scheme compensates for longer wireless latencies and all 22 segments are successfully received at the iPhone 3GS.

The overall CPU usage in the iPhone 3GS is around 16%. About 10% of this is due to network coding, owing to our optimized implementation of RLC in *Tenor*, and the remainder is due to our streaming protocol.

VoD streaming to the iPhone 3GS using LT codes. As we observed in our previous experiments in Sec. 4.2.2 using LT codes to stream videos to emulated clients in the server cluster, our fixed LT codes incur a network overhead of around 11.7% for $n = 128$. This results in the delivery of only 19 segments to the iPhone 3GS. The LT overhead can be compensated by increasing the streaming rate to 108 KB/s so the full 22 segments are received, *i.e.,* a network streaming rate of 108 KB/s effectively delivers 96 KB/s worth of video content with LT codes.

On the more positive side, the advantage of LT codes becomes obvious when we monitor its CPU usage on the iPhone 3GS: it is only 7%, and is significantly lower than the 16% usage with random linear codes. Less than 1% of this CPU usage is due to LT decoding, and the remainder due to our protocol.

Streaming video playback on the iPhone 3GS. As we receive and decode video segments, it is natural to expect that the video stream be played back on the device. We have implemented streaming video playback on different platforms. By running a local lightweight HTTP server within our client application, we are able to feed a video stream from the client to a standard media player, such as the QuickTime player, by using its HTTP progressive download and playback feature. The Quicktime media player will read the video stream from the local lightweight HTTP server on the same device, while the client process receives coded streams from the VoD servers. Similar playback scheme, albeit some modifications, is also deployed on the iPhone platform.

In our next experiment with random network coding at $(n = 128, k = 4096)$, the VoD servers stream real video clips encoded with H.264 at 768 kbps (96 KB/s). The CPU usage with playback is around 34% on the iPhone 3GS. Half (17%) of the CPU usage is due to the `mediaserverd` process, which handles video decoding and playback, and the remainder to our client application.

4.4 P2P Live Streaming with Coding

Peer-to-peer (P2P) applications are able to benefit from coding even more than client-server systems. In a P2P system, each peer may be served by multiple peers, but peers are free to join and leave the system. As a result, traditional block reconciliation schemes

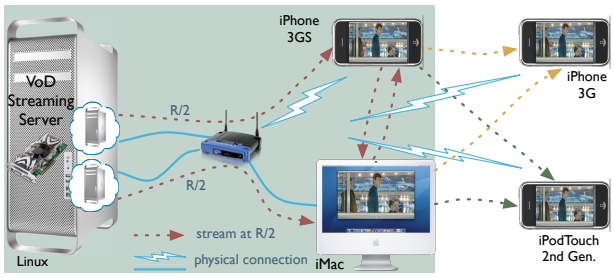

Figure 12: Live P2P streaming with random network coding.

among peers using explicit exchanges of availability bitmaps may be too cumbersome to deliver media segments in a timely fashion.

We are now ready to extend our client-server experiments to a peer-to-peer scenario, with each peer potentially serving others. To maximize contributions from peers rather than servers, our experiments are conducted with a P2P live streaming application, rather than Video-on-Demand streaming, so that all peers consume the same video stream. The servers send a video stream to a number of peers, who may further relay them to their neighbors. In our upcoming experiment, we use a few nodes in static topologies mainly to investigate the performance of our least capable nodes, *i.e.,* our iPhone family devices, in a peer-to-peer scenario. The P2P scenario is more taxing to smartphone devices, as a peer can now serve other peers, leading to a higher computational load when coding is performed.

In live streaming, all peers progress with the same pace playing back a video stream. A peer is able to serve its neighbors even before fully decoding the segment. However, due to the mesh topology of P2P systems, replicating the incoming coded blocks to the neighbors will lead to downstream peers receiving multiple copies of the blocks from different paths. Instead, a peer *recodes* its existing coded blocks in the segment. Recoding, a unique feature of random linear network coding, is used by our following experiments.

In our first experiment, we use the simple topology shown in the shaded area of Fig. 12, involving two peers, an iMac and an iPhone 3GS. Two servers are deployed in a similar setup as our earlier VoD experiments, but both servers stream the same file to emulate a live stream. Each server streams at half of the streaming bit rate, $R/2$. Each peer receives the rest of the content through the other peer, so that it effectively decodes a full stream at rate R and recodes at a rate of $R/2$. Both the iPhone 3GS and the iMac use the NEON/SSE2 accelerated recoding implementation in *Tenor*. In a RLC setting of $(n = 128, k = 4096)$, a 120-second streaming experiment can successfully deliver the expected 22 segments to each peer. The benefit of P2P can be observed by noting that servers now only stream at an aggregate rate of R, instead of $2R$ without P2P. The CPU usage on the iPhone 3GS increases to 23% from 17% in the VoD experiment, which is due to the extra recoding load of $R/2$. Each peer receives coded blocks from a server and a peer. The blocks originating from servers identify their $\mathbf{C}_j$ through a random seed, while recoded blocks coming from the peers carry all 128 coefficients.

In our second experiment, we have added an iPod Touch and iPhone 3G to our P2P topology, as shown in Fig. 12. As both use the older ARM1176 processor without NEON support, they are not powerful enough to serve other peers, in addition to decoding and video playback. We have no choice but to deploy them as sink nodes in the topology.

The iMac and iPhone 3GS nodes in the P2P setup of Fig. 12 now serve three downstream peers with recoded blocks, effectively decoding at R and recoding at $3 \cdot R/2$. Table 1 summarizes the number of successfully decoded segments along with the CPU usage of our client application running on each peer, excluding the usage due to playback which adds another 12% to 17%. $(n = 128, k = 4096)$ is too demanding for our least capable node, the iPhone 3G, such that it falls behind decoding its queued coded blocks and its playback runs into frequent pauses. As shown in the table, all devices have sufficient CPU power for a RLC setting of $(n = 64, k = 8192)$. Obviously, both a desktop computer, such as the iMac, and a recent handheld device, such as the iPhone 3GS, are in a good position to serve even more peers, while the older iPod Touch and iPhone 3G are better used to consume video streams, without serving others. Overall, we can conclude that with *Tenor*, P2P systems with coding support can be realistically deployed even on today's smartphone devices, although the computational capability of current-generation devices has to be considered when peer-to-peer topologies are formed.

Table 1: P2P experimental results.

Peer	down-streams	$(n = 64, k = 8KB)$ CPU usage	Segments	$(n = 128, k = 4KB)$ CPU usage	Segments
iMac	3	2.5%	22	4%	22
iPhone 3GS	3	20%	22	32%	22
iPod Touch	0	38%	22	73%	22
iPhone 3G	0	47%	22	100%	20

5. CONCLUSION

This paper presents *Tenor,* a turn-key solution that includes highly optimized and accelerated designs and implementations of random network coding, Reed-Solomon codes, and LT codes, across a wide range of hardware and operating system platforms. To evaluate *Tenor*, we have implemented both client/server and peer-to-peer streaming applications, involving two virtual servers using two Gigabit Ethernet interfaces in one physical server node, thousands of emulated clients in a 20-node server cluster, as well as actual smartphone devices. Our experiments have offered excellent illustrations of *Tenor* components in action. We are convinced that *Tenor* makes it feasible to rapidly develop practical systems using coding techniques that *Tenor* supports, in both wireless networks and the Internet.

6. REFERENCES

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans. on Info. Theory*, July 2000.

[2] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *Proc. of ACM SIGCOMM*, 2002.

[3] P. Chou, Y. Wu, and K. Jain. Practical Network Coding. In *Proc. of Allerton Conf. on Comm., Control, and Comp.*, 2003.

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

[5] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros. The Benefits of Coding over Routing in a Randomized Setting. In *Proc. of ISIT 2003*, June-July 2003.

[6] J. Li. The Efficient Implementation of Reed-Solomon High Rate Erasure Resilient Codes. In *Proc. of IEEE ICASSP*, 2005.

[7] Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. In *IEEE MICRO*, March-April 2008.

[8] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Efficient Erasure Correcting Codes. *IEEE Trans. Info. Theory*, 47(2):569–584, February 2001.

[9] H. Shojania and B. Li. Parallelized Network Coding With Hardware Acceleration. In *Proc. of IWQoS*, 2007.

[10] H. Shojania and B. Li. Pushing the Envelope: Extreme Network Coding on the GPU. In *Proc. of IEEE ICDCS*, 2009.

[11] H. Shojania and B. Li. Random Network Coding on the iPhone: Fact or Fiction? In *Proc. of ACM NOSSDAV*, 2009.