Making Coding Practical:
From Servers to Smartphones

by

Hassan Shojania

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Making Coding Practical:

From Servers to Smartphones

Hassan Shojania

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2010

The fundamental insight of use of coding in computer networks is that information to be transmitted from the source in a session can be inferred, or decoded, by the intended receivers, and does not have to be transmitted verbatim. Several coding techniques have gained popularity over the recent years. Among them is random network coding with random linear codes, in which a node in a network topology transmits a linear combination of incoming, or source, packets to its outgoing links. Theoretically, the high computational complexity of random linear codes (RLC) is well known, and is used to motivate the application of more efficient codes, such as the traditional Reed-Solomon (RS) codes and, more recently, fountain codes (LT codes). Factors like computational complexity, network overhead, and deployment flexibility can make one coding schemes more attractive for one application than the others. While there is no one-fit-all coding solution, random linear coding is very flexible, well known to be able to achieve optimal flow rates in multicast sessions, and universally adopted in all proposed protocols using network coding. However, its practicality has been questioned, due to its high computational complexity. Unfortunately, to date, there has been no commercial real-world systems reported in the literature that take

advantage of the power of network coding.

This research represents the first attempt towards a high-performance design and implementation of network coding. The objective of this work is to explore the computational limits of network coding in off-the-shelf modern processors, and to provide a solid reference implementation to facilitate commercial deployment of network coding. We promote the development of new coding-based systems and protocols through a comprehensive toolkit with coding implementations that are not just reference implementations. Instead, they have attained high-performance and flexibility to find widespread adoption.

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

In order to validate the effectiveness of the Tenor toolkit, we build coding-based on-demand media streaming systems with GPU-based servers, thousands of clients emulated on a cluster of computers, and a small number of actual iPhone devices. To facilitate deployment of such large experiments, we develop *Blizzard*, a high-performance framework, with the main goals of: 1) emulating hundreds of client/peer applications on each physical node; 2) facilitating scalable servers that can efficiently communicate with thousands of clients. Our experiences offer an 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 mobile devices.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The fundamental insight of use of coding in computer networks is that information to be transmitted from the source in a session can be inferred, or decoded, by the intended receivers, and does not have to be transmitted verbatim. Towards this goal, several coding techniques have gained popularity over the recent years. *Have we reached an era when coding techniques can be performed in networked systems and applications without serious concerns of their practical complexities?* Answering this question is the main contribution of this dissertation.

This chapter discusses motivations of this work, introduces background of the research area, presents major features and contributions of the work, and outlines the rest of the dissertation.

1.1 Motivation

It has been well recognized that innovative coding techniques, such as fountain codes and network coding, has the "theoretical potential" to improve "network performance". As examples, Byers *et al.* [21, 20] have clearly illustrated the benefits of fountain codes,

also known as LT codes [53], in bulk content distribution systems. Ho *et al.* [39] 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. It is a well known result that network coding may achieve better network throughput in certain multicast topologies. The essence of network coding is to allow coding at intermediate nodes throughout the network topology between the source and the receivers, in multiple unicast or multicast sessions.

With coding, beside increased throughput, the system's "resilience" to node departures and packet losses significantly improves. Also, multiple servers can simultaneously serve a single receiver with a substantially "simplified design of block reconciliation protocols". These properties are ideal for content distribution, both as media streaming or bulk data.

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 serve thousands of clients, and in "smartphone" devices where hardware limitations prevail.

In wireless networks, recent works have implemented XOR-only network coding in Nokia N95 mobile devices [60]. However, random network coding is able to offer substantially more flexibility, allowing coding over symbols and the use of multiple paths. 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 [34]. 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".

Application of coding is not limited to traditional content delivery on best-effort Internet, such as video streaming in [74, 11] or bulk data in [32, 33]. One could use coding techniques for increased resilience and build new application layer protocols over UDP, *e.g.,* with rate control schemes such as TFRC [29], even for single-sender single-receiver communication. More recently, distribution of real-time multimedia over IPTV networks, with dedicated IP backbones, has been gaining momentum [28]. For high-quality delivery of broadcast channels in IPTV networks, fast channel-change time and error recovery are some of the major challenges and essential quality-of-service (QoS) features [19, 18]. Rather than the traditional loss repair based on forward error correction (FEC) considered in [19, 18], more sophisticated coding techniques such as network coding for error resilience can be considered. Furthermore, peer-assisted error recovery (PAR) for IPTV broadcast channels proposed in [51], and peer-assisted IPTV video-on-demand systems proposed in [24] can take advantage of similar coding-based delivery techniques we consider in this research.

As we discuss in Sec.1.2 and the rest of this dissertation, random linear coding is more flexible than the other techniques of our interest and can be deployed in a wider range of application scenarios. It has been repeatedly shown on paper that network coding can lead to more robust protocols with less overhead [30], and better utilization of the available bandwidth at any time. However, to date, there has been no commercial real-world systems reported in the literature that take advantage of the power of

network coding. We believe that the main cause of this observation — and the main disadvantage of network coding — is the high computational complexity of random network coding with random linear codes, especially as the number of blocks to be coded scales up. Since random linear codes are universally adopted in all proposed practical protocols using network coding, we believe that it is important to design and implement random linear codes such that its real-world coding performance is maximized, on modern off-the-shelf hardware platforms.

This research has tried to promote the development of new coding-based systems and protocols through a comprehensive toolkit with coding implementations that are not just reference implementations. Instead, we have attained high-performance and flexibility to find widespread usage. This is particularly important for streaming servers that need to sustain the bandwidth required for hundreds of directly connected peers in a peer-assisted Video-on-Demand (VoD) streaming system, and to saturate the line speed of its Gigabit Ethernet interface.

Have we reached an era when coding techniques can be performed in networked systems and applications without serious concerns of their practical complexities? This is the main question that we seek answer for in the course of this dissertation. We look into the computational complexity of these coding techniques in depth and examine a wide range of off-the-shelf hardware/software platforms for such coding purposes.

1.2 Related Work

First introduced by Ahlswede *et al.* [13] in information theory, *network coding* has received significant research attention in the networking community. The pioneering work by Ahlswede *et al.* [13] and Koetter *et al.* [46] have proved that the cut-set capac-

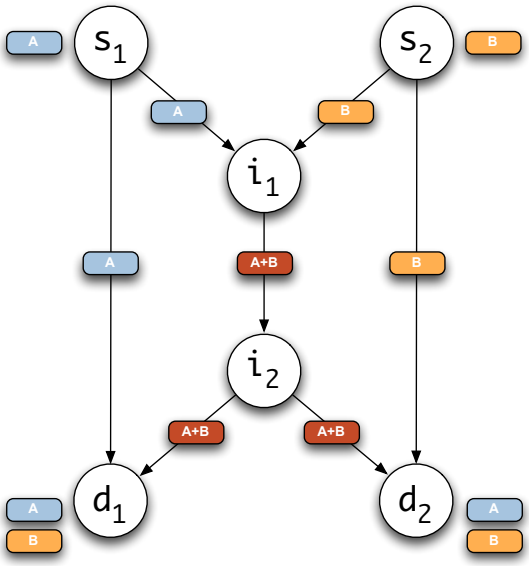


Figure 1.1: Butterfly network showing the benefits of network coding. Reproduced from [73].

ity bounds of unicast flows from the source to each of the receivers can be achieved in a multicast session with network coding in directed networks. The fundamental insight of network coding is that information to be transmitted from the source in a session can be *inferred*, or decoded, by the intended receivers, and does not have to be transmitted verbatim.

The main advantage of network coding hinges upon the *coding* capabilities of intermediate nodes, in addition to forwarding and replicating incoming messages. An example of network coding benefits is shown in Fig. 1.1. In this well-known *butterfly network*, the network is shown as a directed graph with arrows indicating the direction of data flow on the link between the nodes. Let us assume all links have equal capacity and each source node s_1 and s_2 have a piece of data, equal in size named A and B respectively, which needs to be delivered to the destination nodes d_1 and d_2 in the shortest possible time. It is obvious that in absence of coding, the intermediate

nodes, i_1 and i_2, have to use routing to deliver A and B to the destination nodes. The downside is that the link between i_1 and i_2 becomes a bottleneck and has to deliver A and B sequentially. This results in an overall network delivery time equal to two back-to-back messages. By sending a combined form of A and B, *e.g.*, $A + B$, from i_1 to i_2, no longer routing and back-to-back messages are necessary. As an example, d_1 destination node will infer message B by decoding its incoming messages A and $A + B$. Similar scenarios can exist even with a single source node.

Li *et al.* [50] has further proved that linear coding, in which a node in a network topology transmits a linear combination of incoming packets to its outgoing links, suffices in achieving the maximum rate. These results are significant in the sense that, with network coding, the cut-set capacity bounds of unicast flows from the source to each of the receivers can be achieved in a multicast session. In other words, network coding helps to alleviate competition among flows at the bottleneck, thus improving session throughput in general.

To practically implement the paradigm of network coding, one needs to address the challenges of computing *coding coefficients* to be used by each of the intermediate nodes in the session, so that the coded messages at the receivers are guaranteed to be decoded. This process is usually referred to as *code assignment*. Although deterministic code assignment algorithms have been proposed and shown to be polynomial time algorithms (*e.g.*, [62]), they require extensive exchanges of control messages, which may not be feasible in dynamic peer-to-peer networks. As an alternative, Ho *et al.* [39] has been the first to propose the concept of *randomized network coding*. With randomized network coding using random linear codes, an intermediate node transmits on each outgoing link a linear combination of incoming messages, specified by independently and randomly chosen *code coefficients* over some finite field.

Intuitively, it is promising to apply principles of network coding in large-scale con-

tent distribution and media streaming systems. Since the landmark paper on randomized network coding by Ho *et al.*, there has been a gradual shift in research focus in the area of network coding, from purely theoretical studies to more practical studies on applying network coding in a practical setting. Such a shift of focus has been marked by Wu *et al.* [25], in which the authors have concluded that randomized network coding can be designed to be "robust to random packet loss, delay, as well as any changes in network topology and capacity." The highly visible *Avalanche* project by Microsoft Research [34] has further proposed that randomized network coding can be used for bulk content distribution, in competition with *BitTorrent* [10], one of the most practical P2P content distribution protocols. The follow-up work in Avalanche has sought to demonstrate the feasibility of network coding with a real-world implementation in C# [32, 33]. The work has concluded that "network coding incurs little overhead, both in terms of CPU and I/O, and it results in smooth and fast downloads" in its operating settings.

While existing work (such as the Avalanche project [34, 32]) has shown the effectiveness of network coding in P2P content distribution protocols, pessimistic results also cast doubts on how much network coding may improve BitTorrent, because of its excessive computational overhead. For example, a dedicated server with a 3.6 GHz Xeon processor can only achieve coding rates of around 500 KB/second for 256 blocks. This is apparently not sufficient to saturate the upload bandwidth of high-bandwidth peers, such as those with dedicated connections of more than 100 Mbps.

Theoretically, the computational complexity of random linear codes (RLC) has been well known: it has been a driving force towards the development of more efficient codes in content distribution applications, including traditional Reed-solomon (RS) codes, *fountain codes* [53], and more recently, *chunked codes* [56], however, all of them come with their own drawbacks. While fountain codes are much less compu-

tationally intensive as compared to random linear codes, they suffer from their own drawbacks: (1) Coded blocks cannot be recoded without complete decoding, which defeats the original nature of network coding; (2) there exists some bandwidth overhead (about 5% with 10,000 blocks, and over 50% with 100 blocks); and (3) the decoding process cannot be progressively performed while receiving coded blocks, which leads to very bursty CPU usage when the final blocks are decoded. Alternatively, while Reed-Solomon (RS) codes may also be used to reduce coding complexity, it also suffers from the lack of progressive decoding, and its significantly smaller coded message space makes it difficult for multiple independent encoders to code a shared data source, to be sent to a single receiver.

In multi-hop IEEE 802.11-based wireless networks, Katti *et al.* [23] 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. To date, however, there has been no real-world implementations of random network coding on real-world mobile devices, interconnected via 802.11-based wireless links. The closest efforts towards this objective are reflected in the following works in the literature: *First,* in MORE by Chachulski *et al.* [23], each node in a wireless network that performs random network coding is a PC equipped with a IEEE 802.11 (WiFi) wireless card. There, benchmarking of coding performance results have been performed on Intel Celeron 800 MHz PCs. *Second,* Katti *et al.* [42] have also performed random network coding on off-the-shelf Intel PCs, each equipped with Universal Software Radio Peripherals and Zigbee software radios. Random network coding is performed in software on off-the-shelf Intel PC platforms. *Third* and most recently, Pedersen *et al.* [60] 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 [43]. However, random network coding is able to offer substantially more flexibility, allowing coding over symbols and the use of multiple paths. 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).

To summarize, 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. On the other hand, while there is no doubt that more efficient codes exist, they may not be suitable for randomized network coding in a practical setting[1]. In contrast, random linear codes are simple, effective, and can be recoded without affecting the guarantee to decode.

1.3 Contributions

Our main goal is "making network coding practical across devices and platforms". We explore the computational limits of network coding in off-the-shelf modern processors, and provide a solid reference implementation to facilitate commercial deployment of network coding. Factors like computational complexity, network overhead, and deployment flexibility can make one coding schemes more attractive for one application than the others. While there is no one-fit-all coding solution, random linear coding is very flexible, well known to be able to achieve optimal flow rates in multicast sessions, and universally adopted in all proposed protocols using network coding. As a result, random linear codes is of our primary interest compared to other coding

[1]Sec. 7.6 will further discuss the pros and cons of these techniques in practical settings.

techniques.

We believe that our work on a high-performance implementation of random linear codes may help the community to realize the full potential of randomized network coding in a real-world setting. Further, the techniques explored in our high-performance network coding implementation have been also employed to perform other linear coding schemes operating in the Galois Field, such as Reed-Solomon codes. To complete the picture, we will also add high-performance implementations of fountain codes.

Our key contributions can be summarized in three fronts:

1. We develop *Tenor*, a comprehensive toolkit to make coding practical across a wide range of networked nodes, from servers to smartphones. This toolkit represents the first attempt towards high-performance designs and implementations of:

    - a SIMD-accelerated multi-threaded network coding, that takes advantage of both multi-core CPUs and SIMD vector instructions on modern processors.
    - random network coding on the GPUs with various optimization schemes targeting high-performance streaming servers with coding capacities in multi-Gigabits/sec range.
    - random network coding on the mobile devices, *e.g.,* iPhone platform, such that the device can participate in typical video streaming sessions in client-server or P2P setups.
    - Reed-Solomon and LT codes for CPUs, GPUs and mobile devices.

    The final work includes high-performance implementations of a number of

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

By providing highly efficient designs and implementations of random linear codes, Reed-Solomon and LT codes on a wide range of hardware and software platforms, Tenor has strived to push the performance of cross-platform coding toolkit to the limits allowed by off-the-shelf hardware. With achieving coding rates ranging from few MB/s for smartphones, to hundreds of MB/s for general-purpose CPUs, and up to thousands of MB/s for GPU-based servers in typical coding settings required for streaming and content distribution applications, Tenor can be deployed as the common *coding facility* across "all nodes" of client-server and peer-to-peer systems.

2. To facilitate deployment of large experiments with thousands of emulated clients and real devices, we develop Blizzard, of roughly 17K lines of C++ code, a high-performance network framework with the main goals of: 1) emulating hundreds of client/peer applications on each physical node; 2) facilitating scalable servers that can efficiently communicate with thousands of emulated and real clients.

3. To show the practicality of the Tenor toolkit in network applications, Tenor has been used to build coded peer-to-peer (P2P) and also video-on-demand (VoD) media streaming prototypes from multiple GPU-based servers to up to thousands of emulated nodes on a cluster of computers, and to desktop computers

and iPhone devices with "actual video playback". Our experiences offer an 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 mobile devices.

Throughout this work, 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 multiple Gigabit Ethernet interfaces with coding performed in real time. On mobile devices, the latest ARM Cortex-A8 architecture is sufficient to support random linear coding with a realistic range of parameter settings.

We hope this work helps both the academics and practitioners to take advantage of the full potential of coding in real-world settings for delivery of bulk data and streaming content[2].

1.4 Outline Of the Dissertation

In the remainder of this dissertation, a through examination of of our work is given.

Chapter 2 presents the first attempt towards a high-performance implementation of network coding [64]. We first propose to implement progressive decoding with Gauss-Jordan elimination, such that blocks can be decoded as they are received. We then employ hardware acceleration with SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors, respectively. We then use a careful threading design

[2]So far, UUSee Inc. [12], a leading peer-to-peer streaming content provider in mainland China, has employed network coding based on the work in Chapter 2 for its P2P VoD and live streaming systems. See Sec. 2.5 for more details.

to take advantage of multi-core and symmetric multiprocessor (SMP) systems.

With GPU computing gaining momentum as a result of increased hardware capabilities and improved programmability, our work in Chapter 3 shows the first GPU-based attempt, with a design involving thousands of lightweight threads, for boosting network coding performance significantly [67]. Chapter 4 continues on that front with another step forward, and presents a new array of GPU-based algorithms that improve network coding performance such that over 3000 peers can be served at high-quality video rates with a single GTX 280 GPU [65].

Chapter 5 shifts towards mobile devices and presents the first real-world implementation of random network coding on mobile devices, such as Apple iPhone and iPod Touch mobile platforms [66]. It offers 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.

Chapter 6 presents *Blizzard*, our third generation network framework in the *iQua Lab* [9], which builds on the strengths of *Crystal* [70]. Blizzard specifically targets higher scalability as both an emulation framework, and a high-performance server.

Chapter 7 presents *Tenor*, our comprehensive toolkit to make coding practical across a wide range of networked nodes, from servers to smartphones, and also across a number of coding techniques [63]. 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, Chapter 8 builds coded on-demand and live media streaming systems from a GPU-based server to up to 3000 emulated nodes, and to iPhone devices with actual playback. To facilitate deployment of such large experiments, we take advantage of Blizzard framework.

Chapter 9 concludes the dissertations and discusses the future directions for con-

tinuing this research.

Chapter 2

Parallelized Progressive Network Coding With Hardware Acceleration

On paper, it has been repeatedly shown that network coding can lead to more robust protocols with less overhead [30], and better utilization of the available bandwidth at any time. Further, network coding is capable of providing better quality of service (QoS) because improved session throughput, whether in unicast or multicast scenario, and resilience against peer failure are both important QoS parameters. Unfortunately, to date, there has been no commercial applications or protocols that take advantage of the power of network coding. We believe that the main cause of this observation — and the main disadvantage of network coding — is the high computational complexity of *random linear codes* [50], especially as the number of blocks to code scales up. Since random linear codes are universally adopted in all practical network coding proposals, we believe that it is crucially important to design and implement random linear codes such that its real-world coding performance is maximized, on modern off-the-shelf processors. In addition, a high-performance implementation of network coding is critical to determine whether network coding can offer any advantages over

BitTorrent-like P2P protocols, given its high computational complexity.

To our knowledge, this work represents the first attempt towards a high-performance implementation of network coding. We first propose to implement progressive decoding with Gauss-Jordan elimination, such that blocks can be decoded progressively as they are received. We then employ hardware acceleration with SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors, respectively. We finally use a careful threading design to take advantage of symmetric multiprocessor (SMP) systems and multi-core processors. The objective of this work is to explore the computational limits of network coding in off-the-shelf modern processors, and to provide a solid reference implementation to facilitate commercial deployment of network coding. Our high-performance implementation is packaged as a C++ class library, and runs in Linux, Mac OS X and Windows, in Intel, AMD and IBM PowerPC processor families. On a Dual dual-core PowerPC G5 2.5 GHz server, the coding bandwidth of our implementation is able to reach 43 MB/s with 64 blocks of 32 KB each.

The remainder of this chapter is organized as follows. Sec. 2.1 overviews Galois Field and random linear codes. Sec. 2.2 presents our design of using Gauss-Jordan elimination to achieve progressive decoding. Sec. 2.3 presents our SIMD and threaded-based parallelization schemes for maximizing the coding performance of random linear codes, on modern off-the-shelf processors. Sec. 2.4 evaluates our high-performance implementation and Sec. 2.5 summarizes the chapter.

2.1 Random Linear Codes

This section covers the basis of random linear codes by first overviewing the Galois Field.

2.1.1 Operations in finite fields

Random linear codes operate in *finite fields* [61] rather than the regular rational numbers field. The finite fields are closed under multiplicative inverse and their finite size make them appealing for practical implementation. Because the multiplicative group of a finite field is *cyclic*, the arithmetic operations in finite fields of moderate sizes can be implemented through lookup tables [61].

Further, it can be proved that the size of finite fields, also known as Galois Fields (GF), must be a power of a prime and every power of a prime is a size of some finite field [72, 61]. This means every finite field has an alphabet size of p^m for a prime number p and integer m and represented by GF(p^m). Prime 2 is of particular importance due to its practical implications for digital computers with commonly used configurations such as GF(2^8) and GF(2^{16}).

In random linear codes, a finite-field of GF(2^8) is widely used which naturally results in operations on byte-length data symbols. Addition and subtraction of x and y elements in GF(2^8) are simply equivalent to $xXORy$. Unlike regular integer multiplication which results in data-width expansion, multiplication of a one byte symbol in GF(2^8) by another symbol results in a one-byte entity. A common fast approach for multiplication in GF takes advantage of its cyclic property and employs lookup tables.

Logarithm and exponential tables, similar to the traditional multiplication of large numbers, are widely-used for fast GF multiplication on general-purpose computers [69]. Fig. 2.1 shows a C++ function to multiply using three table references where $\log$ and $\exp$ reflect GF(2^8) logarithmic and exponential tables. The basic $\log$ and $\exp$ tables have 256 entries each.[3] However, the $\exp$ table's index of $\log(x) + \log(y)$ can

[3]To be exact, 255 entries of each table is used in practice as $\log(0)$ is not defined and $\exp(255) = \exp(0)$.

vary between $[0, 508]$ because $0 \leq \log(z) \leq 254$. Although a 256-entry table still can be used for the exponential table but that would require the index to be calculated as $(log[x] + log[y])\%255$ which obviously requires extra calculation. As a result, most implementations use a 512-entry lookup table for the exponential table to avoid the extra adjustment of the index. Such a baseline implementation requires three memory reads and one addition for each multiplication.

```
byte gf256::baseline_gf_multiply(byte x, byte y)
{
    if (x == 0 || y == 0)
       return 0;
    return exp[log[x] + log[y]];
}
```

Figure 2.1: Table-based multiplication in GF(2^8): our baseline implementation.

2.1.2 Random linear encoding and decoding

With random linear codes, data to be disseminated is divided into n *blocks* $[b_1, b_2, \ldots, b_n]$, 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 in network coding, a network node first independently and randomly chooses a set of coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jn}]$ in GF(2^8) Galois Field [61], one for each received block (or each original block on the data source). It then produces one coded block x_j of k bytes:

$$x_j = \sum_{i=1}^{n} c_{ji} \cdot b_i \tag{2.1}$$

Since each coded block is a linear combination of the original blocks, it can be uniquely identified by the set of coefficients that appeared in the linear combination.

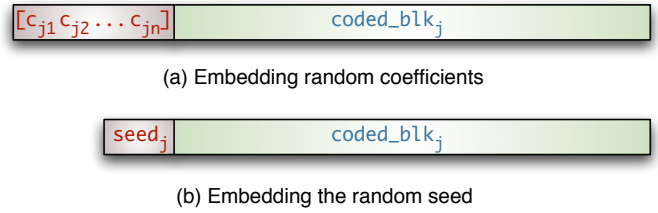


Figure 2.2: Format of the coded message, msg_j.

Generation of each coded block x_j requires n row operations, each consisting of k multiplications in GF(2^8), and subsequent additions of the resulting rows.

In general, sending out a coded block also requires sending the full coefficients row $[c_{j1}, c_{j2}, \cdots, c_{jn}]$. This can be done by forming the message msg_j through putting together the random coefficients $[c_{j1}c_{j2}...c_{jn}]$ and the coded block together as shown in Figure 2.2-(a). By selecting the block size k several times larger than n, the overhead of transmitting the embedded coefficients, of n bytes, over the network can be reduced. The full coefficients are required especially when *full random coding* is in use, *i.e.,* the intermediate nodes *recode* their incoming coded blocks (this mode will be discussed further in Sec. 7.3.1). However, when a coded block is received directly from the server, *i.e.,* without recoding at the middle nodes, a *random seed* can represent the n random coefficients. As shown in Figure 2.2-(b), the seed of the pseudo random generator, typically of 4 bytes, is embedded into the message reducing the overhead of identifying the coefficient row.

A peer 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 $n \times n$ matrix $\mathbf{C}$, using the coefficients of each block b_i. 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:

$$\mathbf{b} = \mathbf{C}^{-1}\mathbf{x} \tag{2.2}$$

In this equation, it first needs to compute the inverse of $\mathbf{C}$, using Gaussian elimination. It then needs to multiply $\mathbf{C}^{-1}$ and $\mathbf{x}^T$, which takes $n^2 \cdot k$ multiplications of two bytes in GF(2^8). The inversion of $\mathbf{C}$ is only possible when its rows are linearly independent, *i.e.,* $\mathbf{C}$ is full rank. Obviously, decoding is more complex than the encoding process as it also requires inverting the coefficients matrix, $\mathbf{C}$.

GF(2^8) operations are routinely used in random linear codes within tight loops. Since addition in GF(2^8) is simply an XOR operation [61], it is important to optimize the implementation of multiplication on GF(2^8). The classic fast approach for GF-multiplication, as described in the previous section, takes three memory reads and one addition for each multiplication.

2.2 Progressive Decoding

We note that a network node does not have to wait for all n linearly independent coded blocks before decoding a segment. In fact, it can start to decode as soon as the first coded block is received, and then *progressively* decodes each of the new coded blocks, as they are received over the network. In this process, the decoding time overlaps with the time required to receive the original block, and thus hidden from the tally of overhead caused by encoding and decoding times. This is an attractive feature that is uniquely available with random linear codes using dense code matrices. Although progressive decoding of later messages becomes increasingly more complex, the decoding complexity is much better balanced than fountain codes, where the bulk of the decoding process is performed after receiving the final coded blocks [53].

We use *Gauss-Jordan elimination* [57] to implement such a progressive decoding process, rather than the more traditional Gaussian elimination. Gauss-Jordan elimination is a variant of Gaussian elimination, that transforms a matrix to its *reduced row-echelon*

form (RREF), in which each row contains only zeros until the first nonzero element, which must be 1. The benefit of the reduced row-echelon form is that, once the matrix is reduced to an identity matrix, the result vector on the right of the equation constitutes the solution, without any additional needs of decoding. Since we operate in GF(2^8), the usual numerical instability caused by Gauss-Jordan elimination does not affect our decoding process.

As each new coded block x_j is received, its coefficients (carried within x_j) are added to the coefficient matrix C. A pass of Gauss-Jordan elimination is performed on this matrix, with identical operations performed on the coded blocks, such that they become partially decoded blocks x'_j. After all n coded blocks are received, these partially decoded blocks become the original blocks. In addition, if a network node receives a coded block that is linearly dependent with existing blocks that have been received already, the Gauss-Jordan elimination process will lead to a row of all zeros, in which case this coded block can be immediately discarded, and there are no explicit linear dependence checks required.

Our implementation of the progressive decoding process is shown in Fig. 2.3, at the moment that the third coded message has been progressively decoded. The first three rows have already gone through iterations of Gauss-Jordan elimination and are now in RREF.

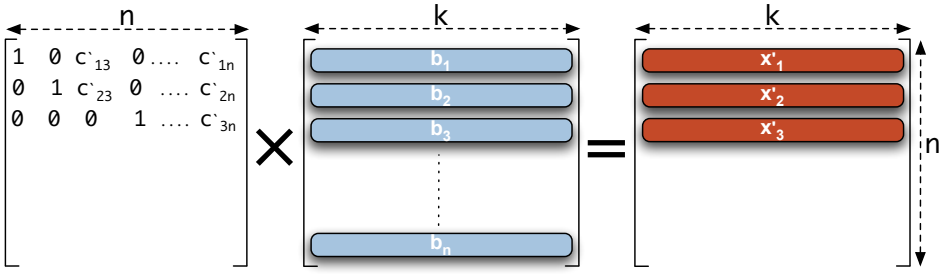


Figure 2.3: State of Gauss-Jordan elimination in progressive decoding.

Our baseline implementation of progressive decoding is summarized in Fig. 2.4, marked by the percentage of execution times in each stage for a typical ($n = 256$, $k = 1024$) experiment over 100 runs. The stages of **A** and **E** are the most time-consuming portions of Gauss-Jordan elimination, as they loop through all existing matrix rows involving up to $n - 1$ row operations each. Stage **D** takes at most one full row operation. The overall decoding complexity is n^2 row operations.

The overall encoding complexity is n^2 row operations as well, since each encode operation requires n row operations, leading up to n^2 for generating n coded blocks. However, the n^2 row operations of the decode process is performed on both coefficient rows of dimension n and coded block rows of dimension k. At encoding, the row operation are performed only on the original blocks of dimension k. As a result, decoding is generally more computationally complex than encoding.

Stage A	Reduce leading coefficients in the new coefficient row to 0.	**[50.05%]**
Stage B	Find the leading non-zero coefficient in the new coefficient row.	**[0.05%]**
Stage C	Check for linear independence with existing coefficient rows.	**[0.00001%]**
Stage D	Reduce the leading non-zero entry of the new row to 1, such that the result is in REF.	**[0.38%]**
Stage E	Reduce the coefficient matrix to the reduced row-echelon form (RREF).	**[49.5%]**

Figure 2.4: Progressive decoding: our baseline implementation.

2.3 Parallelized Network Coding with Hardware Acceleration

Random linear coding suffers from two major performance bottlenecks. First, multiplication in GF(2^8) is a costly operation. Second, the multiplication and addition operations are performed in tight loops over rows of coefficients and coded blocks, each of n and k bytes respectively. Each row operation is performed through a series of byte-length GF(2^8) operations because GF(2^8) multiplication is not easily scalable to a higher granularity than the byte level. The following experiment reflects the importance of addressing these performance bottlenecks.

A sample encode and decode using our baseline implementation of random linear coding takes 9.89 and 11.91 seconds, respectively, for $n = 256$ blocks of $k = 1024$ bytes, on an iMac with 1.83 GHz Intel Core Duo processor. If the table-based multiplication in Fig. 2.1 is replaced with a simple $x + y$ addition, just for measurement purpose, the execution time of encode and decode will be reduced to 7.12 and 9.53 seconds, a significant reduction of 39% and 25%, respectively. In the second test, we assume the same $x + y$ addition is now performed in parallel for every 16 neighboring elements, *i.e.*, row operations in 16-byte granularity. The same encode and decode operations now execute in only 0.191 and 0.242 seconds respectively which is an extra 37 and 39 times reduction in execution time!

Noting the above observations, we attempt to address both bottlenecks through hardware acceleration and parallelization, which complement each other to radically improve the coding performance.

2.3.1 Hardware acceleration with SIMD instruction sets

One way to increase the granularity of row operations is to perform GF(2^8) multiplication in wider chunks without necessarily going to GF(2^{16}) domain. All row multiplications of random linear coding require multiplying a single one-byte factor into all byte elements of a row, whether a coefficients row or a data block row. One can multiply a factor by two bytes of a row at once by building logarithm and exponential tables of $64K$ entries (to address 2^{16} elements). Similarly, the chunk size can be increased to three bytes by employing tables of $16M$ entries and so on. Obviously the tables grow quickly and become difficult to hold in memory. Also, the cache misses increase quickly and offset any gain achieved through widening the multiplication domain.

As an alternative, we propose to revisit the basics by performing the multiplication on-the-fly using a loop-based approach in Rijndael's finite field [72][61], rather than using traditional `log/exp` tables. Although the basic loop-based multiplication takes longer to perform, it lends itself better to a parallel implementation that takes advantage of vector instructions in order to operate on wider chunks of elements from a matrix row at the same time. The loop-based equivalent of the table-based multiplication in Fig. 2.1 is shown in Fig. 2.5, which resembles a regular hand multiplication by looking into lower bit of x and adding y at each iteration. At each iteration, y is shifted to the left to reflect moving to the next bit of x. However, loop-based GF(2^8) multiplication also requires a division by an *irreducible polynomial* [61] (governed by the cyclic nature of the finite field) at the end of multiplication. This division at the end can be emulated by subtracting the irreducible polynomial whenever the shift of y is about to overflow. In our implementation, the irreducible polynomial of $x^8 + x^4 + x^3 + x^2 + 1$ is used. Note that subtraction and addition are both equivalent to the *xor* operation.

```
byte gf256::loop_gf_multiply(byte x, byte y)
{
   byte result = 0;
   bool overflowing;
   while (x != 0) {
       if ((x & 1) != 0)
          result = result ^ y;
       overflowing = y & 0x80;
       y = y << 1;
         // irreducible poly:  x^8+x^4+x^3+x^2+1
       if (overflowing == true)
          y = y ^ 0x1d;
       x = x >> 1;
   }
   return result;
}
```

Figure 2.5: Loop-based multiplication in GF(2^8).

The loop takes at most 8 iterations to complete.

With such a loop-based implementation of multiplication, we are ready to take advantage of SIMD (single-instruction, multiple data) instruction sets, which are available on all modern commodity processors, including Intel, AMD, and IBM PowerPC families. These SIMD instruction sets allow a single operation — such as floating point/integer arithmetic and logical operations — be performed on multiple data in a parallel fashion. IBM's implementation of such instruction set is known as AltiVec and supported on all POWER family of processors. Intel's x86 vector instruction set is called SSE (Streaming SIMD Extensions) which has matured since its SSE2 variant introduced in the Pentium 4 family. AMD has also added SSE2 support since its Opteron

and Athlon64 products. Both AltiVec and SSE2 employ 128-bit (16 bytes wide) registers and allow parallel integer operations on each register as 16 byte-long (or 8 short, 4 regular, 2 long) integers [31, 40].

Let us now observe how SIMD instructions may improve the performance of row operations in random linear coding. The encoder performs a series of row operations solely on the incoming (or original) blocks, b_j. In the decoder, the bulk of Gauss-Jordan elimination also requires series of row operations on both coefficient and coded block rows as suggested by stages **A** and **E** in Fig. 2.4, which collectively consume more than 99% of the execution time. In each row operation, a single byte factor is multiplied by a full row and the result is *xor*-ed to another row. Noting the nature of such row operations, the loop-based multiplication in Fig. 2.5 opens up an opportunity for a vector implementation of the row operation, by GF(2^8)-multiplying the factor into 16 adjacent elements of a row, and then *xor*-ing the 16-byte result into another row at once. As a result, 16 elements of a row is now processed with one execution of the loop-based multiplication[4]. Further SIMD-based optimizations were also applied to the other stages in Gauss-Jordan elimination.

A challenging implementation detail worth noting is related to the alignment of memory allocations in the accelerated implementation. For performance reasons, many SSE2 and AltiVec instructions either require or prefer to have their memory arguments 16-byte aligned. Mac OS X guarantees the heap memory allocations to be 16-byte aligned. On Linux and Windows, we had to use special OS-specific memory allocation APIs for this purpose.

With our new accelerated implementation of random linear coding, we repeat the

[4]Applying the irreducible polynomial to individual elements of a 16-byte chunk has to be conditioned on the value of each element. We omit implementation details due to space constraints, but it can be handled via a few SIMD instructions.

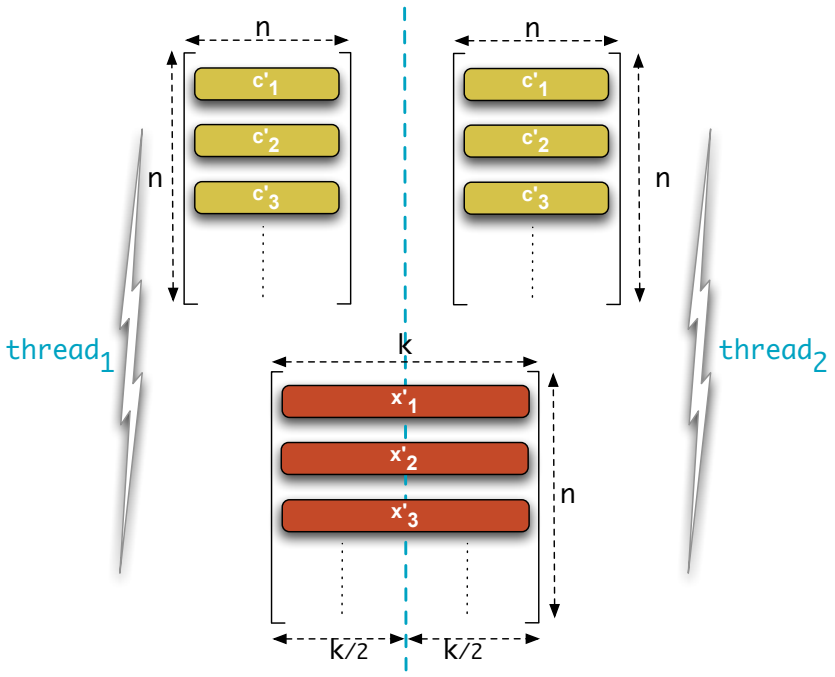


Figure 2.6: Parallelized decoding: partitioning the decoding of coded blocks.

same test scenario described earlier. The accelerated implementation takes 1.78 and 2.20 seconds for encoding and decoding processes, respectively, reflecting speedups of 556% and 541% over the baseline table-based GF(2^8)-multiplication! The speedup is less than the ideal 1600%, due to the obvious usual overhead preventing a linear speedup. Such a speedup via the use of SIMD instruction sets would not be possible without switching to the loop-based multiplication. So far, our work leads to a fully accelerated and cross-platform implementation of randomized network coding, on Intel, AMD and PowerPC processors, and across Windows, Linux and Mac OS X systems.

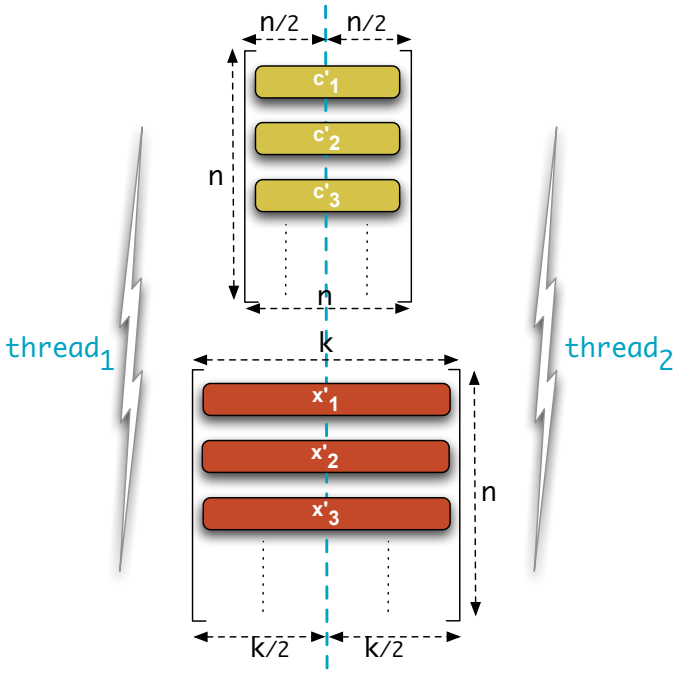


Figure 2.7: Parallelized decoding: partitioning both coefficients and coded blocks.

2.3.2 Parallelized network coding

Since modern commodity processors are routinely multi-core processors, we naturally wish to further improve our accelerated implementation by increasing the level of parallelization, such that all processing cores may be fully utilized. A multi-threaded implementation would naturally take advantage of additional processors, and its performance may benefit significantly through workload partitioning. That acknowledged, parallelized network coding with more than one thread per processor may actually affect performance negatively due to threading overhead, as random linear coding is computationally intensive (CPU bound), without I/O intervals in between.

We first recall that the encoding process generates a new linear combination of incoming blocks, weighted according to random coefficients. This calculation can be partitioned among several threads, each working on a partition of all original blocks

to generate the corresponding partition of the coded block. All threads start with the same sequence of random coefficients $[c_{j1}c_{j2}...c_{jn}]$, either through already prepared coefficients, or generating the coefficients on their own from a shared seed.

Next, the decoding process can similarly divide each coded block into partitions and assign each, of width k/cpu_count, to a different thread. Every thread retrieves the coefficient sequence on its own and maintains the full coefficient matrix of width n. As shown in Fig. 2.6 for two processors, each thread operates on its private copy of the coefficient matrix, and partitions the coded block without any need to communicate with other threads. Such partitioning also improves cache performance.

Ideally, all threads start their new task at the same time and finish around the same time, since they process equal amounts of data. However, the encoding or decoding process is not complete until all threads have completed their tasks. This implies that one of the threads should serve as the coordinating thread, which synchronizes the task assignment and collection to and from other worker threads.

With only coded blocks partitioned, the achieved speedup of the decoding process is limited to the parallel portion of the overall task (which is equal to $k/(k+n)$ since each row operation is performed on both coefficient and coded block rows). If the block size k is much larger than number of blocks n, such a partitioning scheme performs close to perfection. But in a typical real-world scenario of $n = 256$ and $k = 1024$, the coefficient row operations cost 20% of the total row operations.

To further improve the speedup, we propose to extend parallelized decoding to include the coefficient matrix. Fig. 2.7 shows an ideal task partitioning for two threads. There exist a few challenges towards this goal, however. At stage **A** of Gauss-Jordan elimination in Fig. 2.4, all threads need to have knowledge of the full row of coefficients associated with the last received coded block, *i.e.*, partial knowledge is not sufficient. Searching for the first non-zero element at stage **B** is a more problematic

issue, requiring each thread to pass the result of its local search to the coordinating thread, and wait for a response on the global result. Stage **E** needs to retrieve the coefficient of all previous rows that are right above the first non-zero element of the current row. Obviously, such coefficients can belong to any other partition and is not locally owned.

To solve the issues of stages **A** and **E**, each thread needs to keep some redundant data, and to access a global list updated by the coordinating thread. Unfortunately, stage **B** requires explicit synchronization between threads. To address this challenge and to reduce cache coherency updates, we have carefully designed an appropriate synchronization scheme that assigns each thread its own cache-aligned data structure. Each thread's local structure is set by the thread, and read only by the coordinating thread. The coordinating thread's response is set through a similar structure, and read by all other threads.

2.4 Performance Evaluation

In cross-platform C++, we have implemented and fine-tuned our parallelized progressive network coding with hardware acceleration. Our implementation is packaged as a library that can be statically or dynamically linked, runs well in Windows, Linux and Mac OS X, and on Intel, AMD and PowerPC processors. Our original objective of implementing a high-performance network coding engine is to explore its computational limits in modern processors. In this section, we evaluate the performance of our implementation with respect to its coding bandwidth in megabytes per second, as well as the speedup when compared to our baseline implementation. Both implementations operate in GF(2^8). Most our experimental results reflect the average of 100 runs.

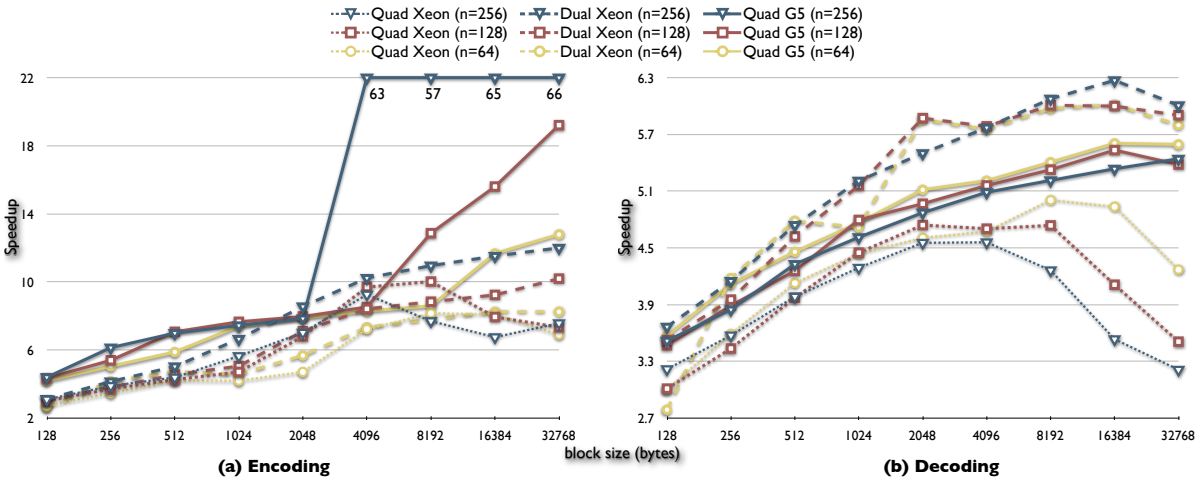


Figure 2.8: Speedup of single-threaded SIMD acceleration for **(a)** encoding and **(b)** decoding processes, over the baseline implementation.

We evaluate our implementation in three hardware platforms: **(1)** a Quad CPU Intel Pentium 4 Xeon 2.8 GHz server (16 GB RAM, 512 KB L2 cache on each CPU and 2 MB shared L3 cache, Linux kernel 2.6.17); **(2)** a Dual CPU Intel Pentium 4 Xeon 3.6 GHz server (2 GB RAM, 2 MB L2 cache on each CPU, Linux kernel 2.6.13); and **(3)** a Dual dual-core Power Mac G5 server with two PowerPC G5 2.5 GHz dual-core processors (4 GB RAM, 1 MB L2 cache on each CPU, Mac OS X 10.4.8). The Intel servers use SSE2 SIMD acceleration, while the Quad Power Mac G5 uses AltiVec.

2.4.1 Hardware acceleration with SIMD instruction sets

We first evaluate our single-threaded implementation of network coding, accelerated with SIMD instruction sets. We evaluate 128 bytes to 32 KB per block, with $64, 128,$ and 256 blocks. When compared to our baseline implementation without acceleration, the speedups of accelerated encoding and decoding are shown in Fig. 2.8. The coding bandwidth, in terms of MB per second, is shown in Fig. 2.9.

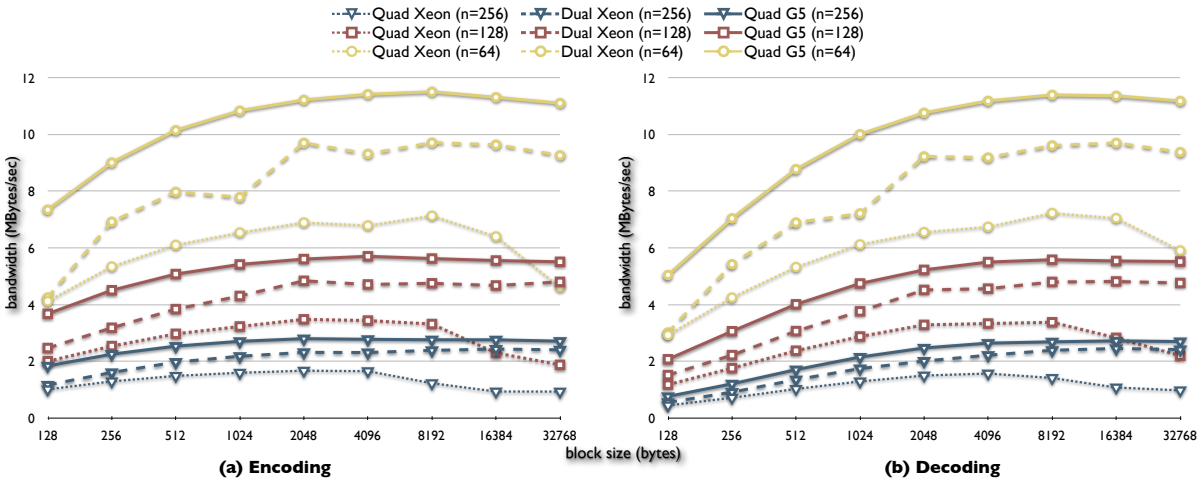


Figure 2.9: Coding bandwidth of single-threaded SIMD accelerated **(a)** encoding and **(b)** decoding.

We have observed from Fig. 2.8 that the encoding process achieves much higher speedups than the decoding process, especially at larger block sizes. If we refer to the actual coding bandwidth in Fig. 2.9, however, we may observe that though the encoding bandwidth is up to 2 MB/s higher than decoding with smaller block sizes, the gap becomes much closer as the block size increases. In addition, the coding bandwidth of both encoding and decoding eventually saturate around similar block sizes of 2–8 KB. The decoding bandwidth graph resembles a delayed form of the encoding graph. In what follows, we attempt to decipher the phenomenon that we have observed.

Although both encoding and decoding processes require n^2 row operations, there exist some differences. The encoding process for each coded block always requires n row operations, performed in a single loop. Its n^2 row operations are performed only on the incoming blocks of width k. In contrast, progressive decoding on coded blocks that are received earlier requires less row operations that on those received later, and accesses a smaller number of rows. Overall, decoding requires n^2 row operations on the coded blocks of width k, plus coefficient rows of width n. In general, encoding is

more "regular" than decoding.

We believe that the more impressive encoding speedup is due to the decreasing performance of the baseline encoding implementation, caused by the L2 cache. As the block size increases, the caching performance of memory operations is penalized, leading to lower coding performance. The accelerated encoding, however, continues to maintain its gain by compensating for the decreasing performance of caching. With respect to decoding, the extra decoding workload on the coefficient matrix becomes less significant as $n/(n+k)$ decreases. Further, the decreasing performance of caching is of lesser importance for decoding, since the decoding data set is initially small, and grows gradually as new coded blocks are received. If the cache is not sufficiently large to store the entire data set, such a gradual increase leads to less cache thrashing than encoding, which traverses the entire data set right from the very first coded block.

Not surprisingly, at a fixed block size, both encoding and decoding achieve a higher coding bandwidth with a smaller number of blocks, since they both require n^2 row operations. The dual-CPU Intel server consistently performs better than the quad-CPU Intel server in this experiment, due to its larger L2 cache and higher CPU clock speed. However, the PowerPC G5 system attains a high-performance margin over the Intel servers, apparently due to its higher performance SIMD implementation. It is impressive to observe that both encoding and decoding bandwidth approach 10 MB/s on the dual-CPU Intel server and even surpass 11.4 MB/s on the G5 system, with 64 blocks. The effect of the 512 KB L2 cache of the Quad-CPU Intel server is clearly visible in Fig. 2.9-(a). The encoding bandwidth declines when the working set overflows the cache capacity after approximately ($k = 2$ KB, $n = 256$), ($k = 4$ KB, $n = 128$), and ($k = 8$ KB, $n = 64$) points.

At a fixed number of blocks, we have also observed that the coding bandwidth increases until a saturation point as higher block sizes are used, a typical behavior

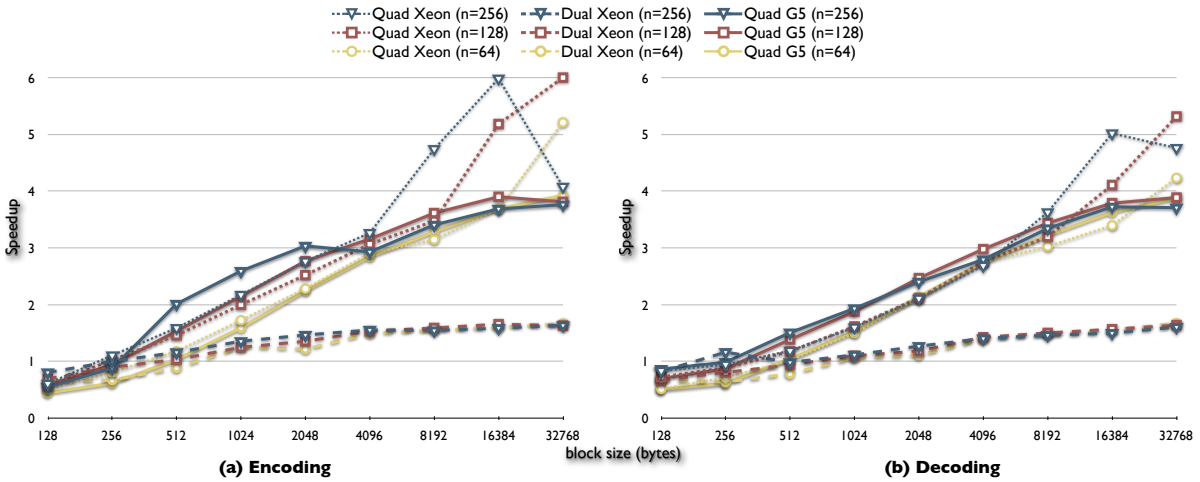


Figure 2.10: Speedup results of multi-threaded SIMD acceleration for **(a)** encode and **(b)** decode over the single-threaded accelerated scheme of Fig. 2.8.

often observed in parallelized computation in response to the increase of problem sizes. This is mainly due to hardware factors such as better performance of the memory fetcher, instruction cache and branch predictor units. The coding bandwidth is affected as the working set no longer fits in the cache and becomes memory-bound.

2.4.2 Parallelized network coding

We now repeat the same experiments to evaluate parallelized network coding with one thread per CPU. We first start with partitioning the decoding of coded blocks only (Fig. 2.6). In Fig. 2.10, we present the speedup of using multi-threaded parallelization over a single thread with SIMD acceleration. In all test cases, the Quad-CPU Intel server outperforms the dual-CPU Intel server, showing the obvious advantage of multi-threading. An interesting observation is the super-linear speedup on the Quad-CPU Intel server. This is a classic example of how the aggregate cache of multiple processors can improve the performance of a memory-intensive computation task by

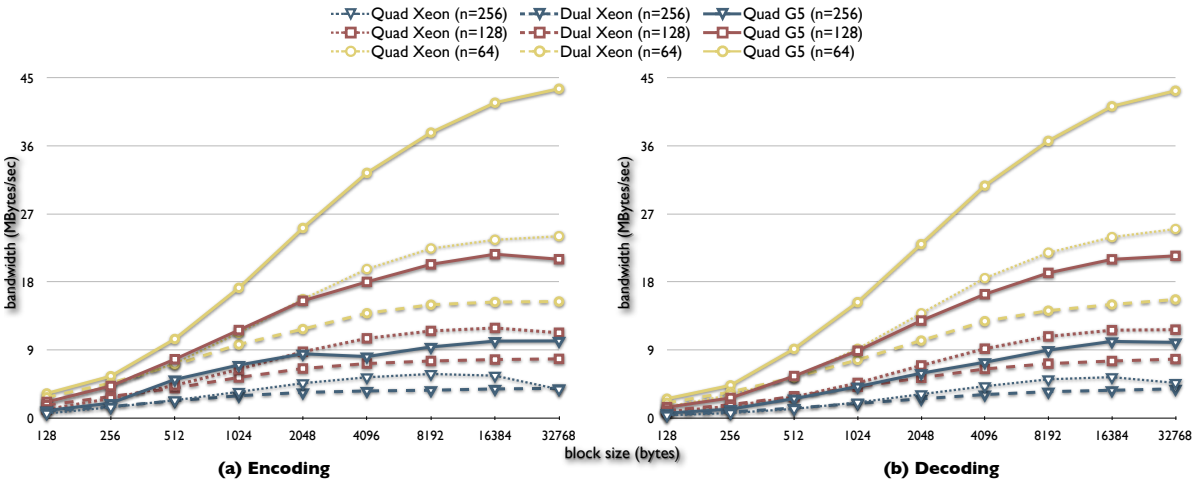


Figure 2.11: Coding bandwidth performance of multi-threaded SIMD acceleration for **(a)** encode and **(b)** decode processes.

partitioning the per-processor working data set.

Fig. 2.11 shows the coding bandwidth. When encoding 256 blocks, we observe from Fig. 2.10 that the Quad-Intel server achieves a speedup of 6, which is almost 4 times higher than the speedup of the Dual-Intel server. However, Fig. 2.11 shows that the Quad-Intel server is only 1.5 times faster than the dual-Intel server in terms of the absolute coding bandwidth. This implies that a dramatic speedup of multi-threading does not translate to equally significant bandwidth gain. This reflects the importance of analyzing speedup and coding bandwidth together at all times.

The Quad-G5 server achieves impressive coding rates by reaching near linear speedup at large block sizes. However, the bulk of these high coding rates stems from the original performance gain through SIMD acceleration shown in Fig. 2.9. We have also observed that, the encoding bandwidth of the Quad-Intel server at 256 blocks peaks at $k = 8$ KB and decreases afterward. This is exactly the point that the overall data set grows to 2 MB, and overflows the 4 processors' aggregate cache ($4 \cdot 512$ KB L2 cache). The decrease is sharper for $k = 32$ KB as the encoding data set grows to 8 MB,

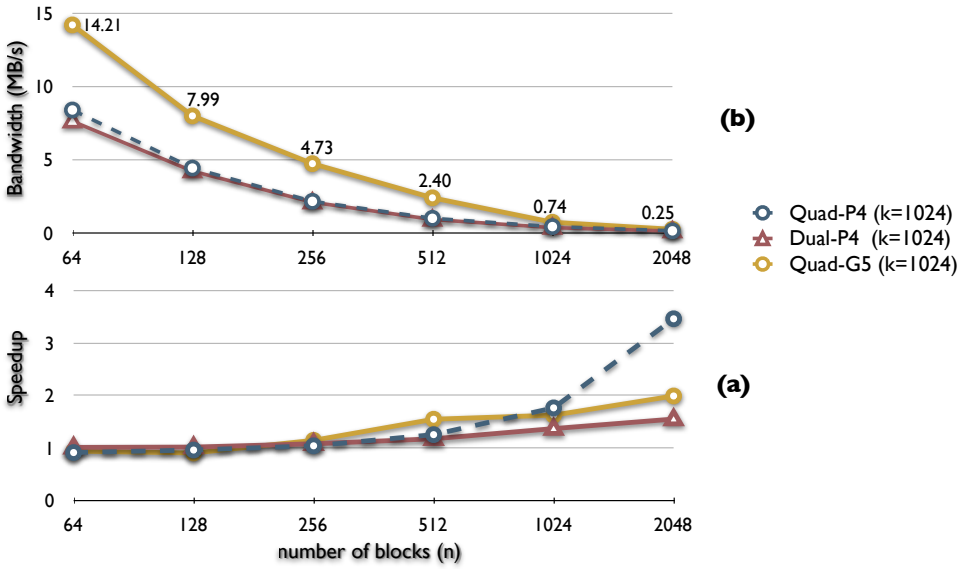


Figure 2.12: Decoding with full partitioning for $k = 1024$: **(a)** speedup over decoding with partial partitioning; **(b)** decoding bandwidth.

which surpasses even the 2 MB L3 cache. Note that decoding is more tolerant of the increased block size, due to its gradually increasing working sets.

Finally, we study the advantages of full partitioning (Fig. 2.7), by also partitioning the decoding of the coefficient matrix. Although the decoding of the coefficient matrix is no longer performed redundantly by all threads, the extra synchronization required leads to additional overhead. Fig. 2.12 shows its resulting speedup over partial partitioning along with decoding bandwidth. Unlike previous experiments, the block size is now fixed at $k = 1024$ bytes, and we increase the number of blocks. This is designed to emphasize the advantage of applying threading to the coefficient matrix when $n/(n+k)$ does not diminish quickly with increasing k. By increasing $n/(n+k)$, we obviously expect to see higher gains due to the partitioning of coefficient rows with width n. At $n = 2048$ and $k = 1024$ bytes, for example, the coefficient matrix of 4 MB will become larger than the coded block matrix of 2 MB. As a result, its partition-

ing improves the cache performance besides improving the parallelism and obviously would lead to a high speedup.

The experiment with 512 blocks is more interesting, because the data set completely fits into the cache and the achieved speedup is solely due to full partitioning. At ($n = 512$, $k = 1024$), each processor of the Quad-Intel server will operate on 128-byte coefficient rows and 256-byte coded block rows. This reduces the aggregate row size to $128 + 256$ bytes from $512 + 256$ bytes of partial partitioning, effectively reducing the aggregate row into half. Of course, we only gain a speedup of 1.26, rather than the ideal 2, because of the extra threading overhead and synchronization of full partitioning.

Since a larger number of blocks dramatically affects the coding bandwidth, it is natural to use the smallest number of blocks possible in real-world network coding. This implies that in most real-world applications based on network coding, parallelized network coding with full partitioning may not gain more than roughly 10% to 15% of coding bandwidth over partial partitioning. Nevertheless, we have still observed that encoding and decoding bandwidth reaches 20.3 and 19.2 MB/s at 128 blocks of 8 KB each with our Quad PowerPC G5 server, and 43.5 and 43.3 MB/s at 64 blocks of 32 KB each. If we use 16 blocks of 32 KB each, they are even able to reach 155.8 and 156.2 MB/s![5] As another way to show our results, Fig. 2.13 conveniently illustrates the coding bandwidth for 128 blocks of 4 KB each across different hardware and OS platforms.

No matter how one sees them, these represent impressive coding performance, thanks to our parallelized and accelerated implementation of network coding.

[5]To establish a context, I/O bandwidth of SATA disk drives is usually between 30 and 60 MB/s.

Platform setup					Coding rate (MB/s)	
System	OS	SIMD type	# of threads	L2 Cache	Encoding	Decoding
Quad PowerPC G5 2.5 GHz	Mac OS X	AltiVec	4	1 MB	18.01	16.38
Quad P4 Xeon 2.8 GHz	Linux	SSE2	4	512 KB	10.54	9.17
Dual Opteron (AMD) 2.4 GHz	Linux	SSE2	2	1 MB	9.56	8.75
Dual P4 Xeon 3.6 GHz	Linux	SSE2	2	2 MB	7.21	6.50
iMac Intel Core Duo 1.83 GHz	Mac OS X	SSE2	2	2 MB (shared)	5.81	5.60
Intel Core Duo 1.66 GHz	Windows XP	SSE2	2	2 MB (shared)	4.62	4.43

Figure 2.13: Platform comparison of coding performance at ($n = 128$, $k = 4096$).

2.5 Summary

This chapter represented the first attempt towards a high-performance implementation of randomized network coding[6]. The objective of this research was to explore the computational limits of random linear coding in modern processors. We proposed to use Gauss-Jordan elimination to perform progressive decoding, such that the decoding time may overlap with the time required for data transmission. Our implementation is now complete with hardware acceleration with SIMD instruction sets available on modern commodity processors, as well as multi-threading to take advantage of symmetric multiprocessors to parallelize computation tasks. With a wide variety

[6]So far, UUSee Inc. [12], one of the leading peer-to-peer streaming content providers in mainland China, has employed a CPU-based multi-threaded, SIMD accelerated, network coding scheme for offline encoding of video content, mostly encoded to high quality streams around 500 Kbps, based on the work presented in this chapter and [64] for its video-on-demand (VoD) system. The clients of this VoD system employ network coding based on SIMD accelerated encoding/decoding presented in this chapter. A simpler setup of network coding is used in their live streaming system. UUSee has a user base in order of millions of peers.

of working sets in our coding tests, significant speedup and coding bandwidth have been achieved. We are now confident to claim that, as long as we code fewer than 128 blocks, the computational complexity of randomized network coding may not become a performance bottleneck even on dedicated servers with more than 100 Mbps connections. In peer-to-peer applications with typical DSL bandwidth, we believe the CPU usage is minimal. The task of coding more than 128 blocks still remains to be an interesting challenge.

Chapter 3

GPU-accelerated Many-core Network Coding

Chapter 2 has shown a SIMD-accelerated multi-threaded implementation of network coding, that takes advantage of both multi-core CPUs and SIMD vector instructions on modern processors [64]. This chapter represents another substantial step forward, and presents the first attempt in the literature to maximize the performance of network coding by taking advantage of not only multi-core CPUs, but also potentially hundreds of computing cores in commodity off-the-shelf many-core *Graphics Processing Units (GPUs)*. Modern NVIDIA GPUs, for example, are designed with hundreds of specialized *cores*, each with much less complexity than a CPU core, but nevertheless supports operations required for general-purpose high-performance computing. We are motivated by the natural curiosity of whether or not GPUs are able to help improve the performance of network coding. Such an interest is further stimulated by the relative low cost of mainstream GPUs as compared to multi-core CPUs. As an example, the NVIDIA GeForce 8800 GT retails for approximately $1/20$ of the cost of a dual Quad-core Intel CPU setup.

Based on the NVIDIA CUDA framework, we present *Nuclei,* our design and implementation of GPU-based many-core network coding, across platforms including Windows, Linux and Mac OS X. With *Nuclei*, we have made the following new observations on the feasibility of GPU-based network coding. *First*, although the GeForce 8800 GT, featuring 112 cores, is less capable than the 8-core Intel Xeon server with respect to its raw computing power, the GPU is designed to better hide memory latency, a critical issue that affects the performance of network coding, particularly in the typical coding range for streaming servers. *Second*, by completely detaching the encoding process from the CPU and hand it over to the GPU, the CPU cores on dedicated streaming servers are set free to perform other CPU-intensive tasks that GPUs are not able to perform. *Third*, due to the specific GPU design that schedules its threads in hardware, the performance of GPU-based network coding is not affected by competing threads and background tasks. In contrast, variations of up to 9% are observed with CPU-based network coding, due to background threads. *Finally*, GPUs are equally capable of both encoding and decoding. Though decoding is much more challenging to be performed on the GPU with less capable performance than the 8-core Intel Xeon server at small block sizes, the GPU decoding performance improves substantially as block sizes become larger, and is on par with 8-core CPUs at a block size of 16 KB.

Nuclei has achieved stellar performance results, making it feasible to saturate Gigabit Ethernet interface by combining 8-core Intel Xeon CPUs and a mainstream NVIDIA GeForce 8800 GT GPU. With respect to encoding, for example, the GeForce 8800 GT by itself is able to achieve an encoding rate of 84 MB/second with 128 blocks, outperforming all eight CPU cores combined. A combined CPU-GPU encoding scenario achieves coding rates of up to 116 MB/second for a variety of coding settings, which is sufficient to saturate the Gigabit Ethernet interface.

This chapter is organized as follows. Sec. 3.1 provides an overview of GPU computing. Sec. 3.2 presents our design towards many-core GPU-based network coding. Sec. 3.3 evaluates our GPU-based network coding implementation. Finally, Sec. 3.4 summarizes the chapter.

3.1 Overview of GPU Computing

Modern GPUs have gradually evolved from specialized engines operating on fixed pixels and vertex data types, into programmable parallel processors with enormous computing power [52]. NVIDIA's Tesla GPU architecture, introduced in November 2006 and now employed in a wide range of professional and consumer GPU products, is the first such GPU architecture that enables high-performance parallel computing applications, written in the C language using the Compute Unified Device Architecture (CUDA) programming model and development tools [58], and is now considered to be the "most ubiquitous" supercomputing platform [52].

Our performance evaluation in this work uses the mainstream NVIDIA GeForce 8800 GT GPU with 112 cores, which is supported by the CUDA platform. Our cross-platform implementation in *Nuclei*, however, can be used on any CUDA-supported GPU, ranging from the high-end GeForce GTX 280 with 240 computing cores, to the mobile GeForce 8600M GT, with 32 cores.

The Tesla architecture and CUDA

The Tesla GPU architecture is based on a scalable array of *Scalar Processors (SPs)*. These SPs are grouped into groups of 8, called *Streaming Multiprocessors (SMs)*, and share a number of resources, including *registers* and *shared memory* [26]. On the 8800 GT, for example, there are 112 SPs grouped into 14 SMs, as shown in Fig. 3.1. Threads

too are grouped into *thread blocks*, which allow threads of each block to share data and synchronize among themselves. Each thread block is assigned to a SM and runs on its SPs. After a thread block terminates, new blocks are launched on the vacated SM. The GPU process finally finishes when all thread blocks terminate.

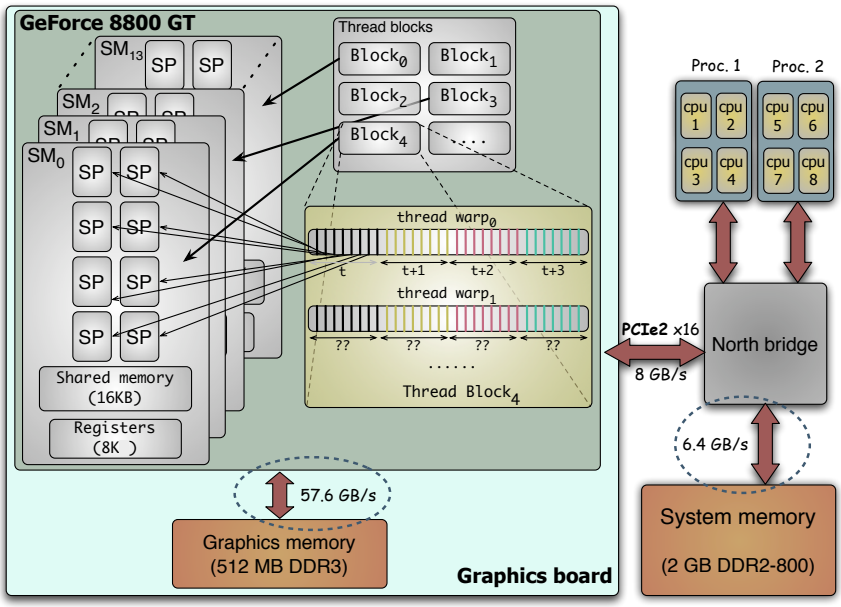


Figure 3.1: The architecture of our Mac Pro testbed with the NVIDIA GeForce 8800 GT GPU. The Mac Pro comes with two 2.8 GHz Quad-Core Intel Xeon processors, 2 GB memory, and 12 MB cache per processor.

Each SM manages and executes concurrent threads (of one or more thread blocks) in hardware with zero scheduling overhead. Every 32 threads of a thread block are grouped together in a *thread warp*, running in a synchronized manner with each other, executing the same instruction in a SIMT (Single Instruction, Multiple Thread) fashion. Since there are no more than 8 SPs in a SM, the 32 threads of a thread warp are scheduled eight at a time in 4 cycles. As the example in Fig. 3.1 shows, the first 8-thread group of warp_0 of the 4^{th} thread block is currently executed at cycle t. Each of

the next 8-thread group of warp_0 will execute in the next 3 cycles. However, threads of warp_1 are not ready to be scheduled, *e.g.*, because of a pending memory access.

Since each SP is deeply pipelined, read-after-write dependencies can occur often. Similarly, a memory access causes a thread warp to stall for the next few hundred cycles till the data arrives. The beauty of the Tesla architecture comes from the fact that, as soon as it detects a thread of a thread warp can not proceed, it can switch to another ready warp with zero scheduling overhead [26]. As a result, it is essential to have as many thread as possible so the SPs can execute from other thread warps till the stalled one can be rescheduled. The large thread count, together with the support for many outstanding load requests, helps to hide memory load latency [58].

Comparing to the CPU, the GPU dedicates its die area to a higher number of processing cores. SPs are designed to be simple in-order engines without branch prediction, register renaming, multiple pipelines and many other more advanced but standard features found on modern CPUs. The good news is that, with wider and faster memory interfaces, the GPU has a much higher memory bandwidth at its disposal than the CPU. For example, the DDR3 memory found in the 8800 GT has a peak performance of 57.6 GB/s, compared to a meager 6.4 GB/s of our Mac Pro server based on dual Quad-core 2.8 GHz Intel Xeon CPUs, with its DDR2-800 memory. Further, with a 16-lane bidirectional PCI Express 2.0 interface bus between GPU and CPU, a 8 GB/s bidirectional data rate gives us abundant bandwidth to move data around between the GPU and system memory, with almost no cost for the size of data we cope with to perform network coding.

A CUDA-enabled program, written in C, consists of the *host code* that runs on the CPU, and a *device code* that runs on the GPU [38]. When the CPU invokes a GPU *kernel* (the device code analogous to an application process), the CUDA driver programs the GPU to launch a large number of threads. The host code is compiled by a regular

C/C++ compiler. The device code is compiled by NVIDIA's own compiler, generating an assembly language output in a format called Parallel Thread eXecution (PTX). PTX is a "virtual" language, and is not specific to a particular GPU product. However, most of its instructions have machine instruction equivalents in the current CUDA GPUs. Either the PTX intermediate representation or, if the target GPU is known at compile time, the final generated code called *cubin* are embedded in the final application executable. When an application executes, the CUDA runtime generates the machine code, if needed, and loads it to the GPU through the device driver.

As GPUs only recently became fully programmable devices, the CUDA programming tools are experimental in nature [38]. As a result, developing and debugging CUDA device code are far from straightforward, particularly when complex situations such as race conditions are inevitably involved.

3.2 *Nuclei*: GPU-accelerated Network Coding

In this section, we present challenges and solutions involved in the design of *Nuclei*, our implementation of GPU-accelerated network coding.

3.2.1 Random network coding: Performance bottleneck

Random network coding suffers from a major performance bottleneck. Byte-length multiplication in GF(2^8) is a costly operation due to three memory accesses when $\log/\exp$ tables are used, and it is performed in tight loops over rows of coefficients and coded blocks, each of n and k bytes, respectively. To address this bottleneck, we proposed in Chapter 2 to revisit the basics by performing the multiplication on-the-fly using a loop-based approach in Rijndael's finite field, rather than using traditional

$\log/\exp$ tables [64]. Although loop-based multiplication takes longer to perform (up to 8 iterations), it lends itself better to a parallel implementation that takes advantage of vector instructions in order to operate on wider chunks of elements from a matrix row at the same time. The loop-based equivalent of the table-based multiplication is shown again in Fig. 3.2 with statement line numbers for further reference.

```
byte loop_gf_multiply_byte(byte factor, byte data)
{
   byte result = 0;                                    (1)
   bool overflowing;                                   (2)
   while (factor != 0) {                               (3)
       if ((factor & 1) != 0)                          (4)
          result = result ^ data;                      (5)
       overflowing = data & 0x80;                      (6)
       data = data << 1;                               (7)
       // irreducible polynomial:  x^8+x^4+x^3+x^2+1
       if (overflowing == true)                        (8)
          data = data ^ 0x1d;                          (9)
       factor = factor >> 1;                           (10)
   }
   return result;                                      (11)
}
```

Figure 3.2: Loop-based byte-length multiplication in GF(2^8).

With such a loop-based approach, we were able to take advantage of SIMD (Single Instruction, Multiple Data) vector instruction sets to perform GF(2^8) multiplication on 16 byte-long units of each row, rather than single-byte units [64]. Rather than using vector instructions provided by the CPU, is it possible to use the GPU and still achieve accelerated multiplication using such a loop-based approach? At first glance, it is a daunting challenge due to the lack of SIMD instructions and wide execution units on the GPU.

3.2.2 Loop-based GF(2^8) multiplication on the GPU

Unlike modern CPU cores with 128-bit registers and execution units, current CUDA-enabled GPUs have plain 32-bit registers and execution units. Nevertheless, they have more than a hundred processing cores (*e.g.*, 112 SPs for the 8800 GT) that run in parallel.

To take full advantage of the available 32-bit arithmetic units in the GPU, we rewrite byte-length multiplication as `loop_gf_multiply_word` to multiply a one-byte coefficient with a 4-byte word. This is not a straightforward task, however. Both Intel SSE2 and PowerPC AltiVec SIMD instructions include special instructions that allow arithmetic and comparison operations on individual bytes of 16-byte registers in parallel. With no similar instruction available on the GPU, we have to emulate such byte-long operations on 4-byte words. The main issues involved are the following: **(1)** To left-shift individual bytes of `data` word without affecting the neighboring bytes (statement (7) in Fig. 3.2); **(2)** To determine the "overflow" status by examining the top bit of individual bytes of the 32-bit data word (statement (6) in Fig. 3.2); **(3)** To apply the irreducible polynomial byte `0x1d` to each byte of the data word based on the "overflow" states (statements (8) and (9) in Fig. 3.2). We use a series of `if/else` conditions, byte extractions, bitwise operations and bit shifts to address these issues. Even with our best effort to minimize the code complexity, the compiled code results in no fewer than 16 *cubin* instructions on the 8800 GT.

To evaluate the computational performance of our loop-based GF-multiplication, we have designed a benchmark that entails an encoding process that produces $c = 7168$ coded blocks of size 2048 bytes each, where each coded block is a linear combination of 1024 blocks, *i.e.*, $(n = 1024, k = 2048)$. To detach the benchmark from the memory performance of the GPU, we create input coefficients and rows of data

on the fly, rather than loading them from graphics memory. To evaluate the worst-case scenario, we consistently use `0xff` as coefficients to make sure that the loop always iterates for the maximum 8 times. Our benchmark is implemented by launching 7168 GPU threads and assigning the computation of each coded block to its own thread. Since a core in the GPU has a 32-bit execution unit, each GPU thread executes `loop_gf_multiply_word` for $n \cdot k/4$ times, and the entire benchmark executes the function for a total of $(c \cdot n \cdot k/4)$ times.

The execution of the benchmark takes 2509 ms (milliseconds) to complete, which shows a much better performance than byte-length GF-multiplication, which takes 7186 ms to complete.

3.2.3 Further optimizations of word-length GF-multiplication

In a CUDA kernel, all 32 threads of a *thread warp* execute the same instruction in a synchronized manner. When reaching an `if/else` condition, even if only a single thread takes one path against the others, the overall warp execution time will be delayed as if *both paths* are executed. When generating the word-length polynomial mask pattern in our `loop_gf_multiply_word` implementation, with each byte having `0x1d` if individual bytes of the `data` word is about to overflow, we resorted to a series of conditions. It is apparent that, to further improve performance, we need to avoid these conditions as much as possible.

As shown in statements (5), (6), and (7a) of Fig. 3.3, we can generate the polynomial mask pattern by shifting the overflow state of bits to the first bit of each byte and then multiply the entire word by `0x1d`. With this improvement, the number of *cubin* instructions decreases from 16 to 14, and our benchmark now takes 2373 ms, a 5.7% improvement of performance.

```
word 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 polynomial mask
        PrimPolyMask = data & 0x80808080;                   (5)
        PrimPolyMask = PrimPolyMask >> 7;                   (6)

  (7a)  PrimPolyMask = PrimPolyMask * 0x1d;
        { PrimPolyMask = __mul24(PrimPolyMask, 0x1d);
  (7b)  { if (data &0x80000000)
          PrimPolyMask = PrimPolyMask + 0x1d000000;

        // 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 3.3: Loop-based GF(2^8) word-length multiplication for a CUDA-enabled GPU.

Since current CUDA-enabled GPUs do not have native 32-bit multiplication, the compiler translates the multiplication of statement (7a) to 4 instructions using 24-bit multiplication and shifting. However, we can rewrite statement (7a) as (7b) to take advantage of the 24-bit multiplication directly. Now the compiled code is reduced to 12 *cubin* instructions and the benchmark executes in 2067 ms.

At a final attempt of our optimization effort, we sidestep the compiler and optimize the PTX virtual assembly file directly. This approach successfully removes another *cubin* instruction, now down to 11 instructions, which improves the benchmark's

execution time to 1905 ms, a 32% improvement over our initial implementation. This is a very interesting result, since a "back-of-the-envelope" calculation suggests that the total computing power of all 112 cores on the 8800 GT is almost used to the best possible. The number of cycles taken by executing a single `loop_gf_multiply_word` can be derived as:

$$\begin{aligned}\text{Cycles}_{\text{GF}-\text{mul}} &= \text{total cycles / total computed words} \\ &= (\text{time} \cdot \text{cores} \cdot \text{freq})/(c \cdot n \cdot k/4) \\ &= (1.905 \cdot 112 \cdot 1.5\,\text{GHz})/(7168 \cdot 1024 \cdot 2048/4) \\ &= 85.16 \text{ cycles}\end{aligned}$$

Since our benchmark fixed the number of loop iterations to 8, each iteration takes 10.65 cycles. At first glance, it seems surprising that the result is less than the execution time required for 11 instructions. A closer examination reveals that the conditional addition in statement (7b) executes once every other time, resulting in an effective 10.5 cycles in an ideal execution.

Although both CUDA and PTX virtual instructions support 64-bit data types, a `GF-multiply` implementation for double-word `data` can not achieve better performance. This is due to the fact that the current CUDA-enabled GPUs have 32-bit integer engines, and 64-bit operations are emulated through a series of 32-bit operations.

3.2.4 CPU vs. GPU: Estimating the computing power

Having our highly optimized GPU-based implementation of `GF-multiply`, we are now ready to compare its performance against the SIMD-accelerated CPU-based implementation of Chapter 2. Our corresponding benchmark for the CPU, $(c = 7168, n = 1024, k = 2048)$, divides each source block of k bytes into 8 partitions, each processed by a dedicated thread, one per CPU core on our 8-core Mac Pro server. The benchmark

Table 3.1: CPU vs. GPU: Theoretical Computing Performance

	Mac Pro 8-core Intel	8800 GT
f: core freq. (GHz)	2.8	1.5
n: number of cores	8	112
w: multiply data width (bytes)	16	4
ic: instruction count/iteration	12	11
Per core superscalar factor	3	1
cc_1: cycle count/iteration	$12/3 = 4$	11
Est. throughput (cycles/inst.)	$7 \cdot 0.33 + 5 \cdot 0.5 = 0.40$	1
cc_2: cycle count/iteration	$12/(1/0.4) = 4.8$	11
cc_3: cycle count/iteration	5	10.5
$\text{Performance}_{\frac{\text{CPU}}{\text{GPU}}}$	$\frac{f_{\text{CPU}}}{f_{\text{GPU}}} \cdot \frac{n_{\text{CPU}}}{n_{\text{GPU}}} \cdot \frac{w_{\text{CPU}}}{w_{\text{GPU}}} \cdot \frac{cc_{\text{GPU}}}{cc_{\text{CPU}}}$	

finishes execution in 1672 ms, reflecting that the 8-core Mac Pro system outperforms the 112-core 8800 GT with a 13.9% margin.

As Table 3.1 shows, this is not a surprising result after comparing a number of performance metrics of the CPU with those of the GPU. In our comparison, we can observe that each GPU core is an in-order processing unit with a single Arithmetic Logic Unit (ALU), while each CPU core has three ALUs, each capable of processing a SSE2 instruction in any cycle [41]. This dramatically offsets the higher number of GPU cores, despite the fact that GPU cores run at almost half of the CPU speed. In our first estimate, we assume a coarse throughput advantage of 3 (equal to the superscalar factor) for CPU cores. The performance advantage of CPU-based over GPU-based implementations, $\text{Performance}_{\frac{\text{CPU}}{\text{GPU}}}$, can be computed through estimating the cycle count per iteration. We now progress through three alternative estimates of cycle count per iteration, from coarser to finer granularities. The first estimate cc_1 results in a performance advantage of 1.47, where the CPU is faster. Our second estimate, cc_2, takes a

closer examination into CPU instructions and comes up with an average throughput based on individual instructions, which results in a performance advantage of 1.22. Our final refinement considers the data dependency of instructions in the CPU-based implementation, and results in a performance advantage of 1.12, quite close to our measurement results.

This confirms our measurement results that the 8-core Intel Xeon system is expected to have a higher computing power compared to the 112-core 8800 GT. However, as we shall soon see in the next section, memory performance will substantially affect the performance of a CPU-based implementation.

3.2.5 Many-core network encoding on the GPU

The process of random network encoding essentially consists of a matrix multiplication in the GF domain, and can be considered as a *embarrassingly parallel* computation problem, where a parallel implementation is possible with little or no communication and synchronization among threads. Without considering memory access to source blocks and coefficients, the performance of network encoding is only limited by the hardware's computational power, since the encoding process of multiple coded blocks — and even different section of a coded block — can proceed in parallel by using a large number of threads.

We now complete the picture by considering memory access in a complete process of network encoding. As we shall see, achieving a high speedup can not be taken for granted and requires careful task partitioning.

Partitioning for many-core network encoding

Our synthetic benchmark does not consider memory access. In CUDA, GPUs can only access their graphics memory, so the coefficients and source blocks have to be transferred from the host to the graphics memory first. Similarly, encoding results residing in the graphics memory need to be transferred back to system memory. With 8 GB/s on the PCI Express 2.0 interface, transfer times to and from graphics memory are negligible.

We use the same $(c = 7168, n = 1024, k = 2048)$ setup for our new benchmark with memory access. To ensure a fair comparison with previous tests, we fill the coefficient matrix with all `0xff` to ensure a maximum load. The encoding benchmark on the GPU now takes 5306 ms, reflecting a substantial increase from 1905 ms, suggesting poor memory performance.

After attempting other alternatives, we have settled on a partitioning mechanism with a much finer granularity, with each GPU thread encoding only a "single word" of the coded block, rather than a full block. With careful assignments of words to threads of each warp, we can now take advantage of memory coalescing [26], so most memory accesses of a thread warp fall next to each other, significantly reducing the number of memory accesses by the memory controller on the GPU. With such fine-granularity partitioning, 512×7168 threads have been launched, much higher than the original 7168 threads. However, unlike CPU threads, GPU threads are very lightweight as GPUs are designed to switch to new threads seamlessly in hardware, in order to hide memory latency. Using this new partitioning scheme, our encoding process now takes 3016 ms, a 76% improvement over our original coarse partitioning.

Our next measure towards further optimization reads coefficients in 4-byte chunks, rather than byte by byte, and then caches them for use in the next four multiplications.

This reduces read requests for coefficients to $1/4$ of the original approach, and the execution time is reduced to 2098 ms, now over 2.5 times better than our original partitioning.

On the 8-core Mac Pro, the same network encoding benchmark takes 2250 ms, which implies that the GPU performance defeats 8-core CPUs when memory access is considered. reflecting the GPU's better ability to hide memory access latency. While the CPU-based implementation suffers 35% due to memory access, the GPU performance only degrades 10%. This clearly demonstrates the GPU's superior ability to hide memory access latency, due to seamless hardware switching across a large number of threads.

Finally, we use actual random coefficients instead of `0xff`. The CPU takes 2157 ms, while the GPU, being constrained by computation and not memory access, decreases significantly to 1751 ms, showing a 23% advantage over the CPU. All considered, the GPU performs 23% better than the 8-core CPU with memory access.

Mixed CPU-GPU network encoding

When even better performance is required, both CPU and GPU can perform network encoding in parallel. There are two main approaches for task partitioning between CPU and GPU, both having fundamentally the same computation load. Either one divides the working set by partitioning each source block into two (*e.g.*, half each), and assigns each partition to CPU and GPU; or one assigns the encoding tasks of some coded blocks entirely to the GPU, and the remainder to the CPU.

We choose to set up the same $(c = 7168, n = 1024, k = 2048)$ experiment based on the second approach. First, we divide a generation of 7168 coded blocks evenly between the 8-core CPU and GPU, 3584 coded blocks by each. The encoding process

takes 1094 ms, reflecting a speedup of 1.99 over a similar CPU only execution. Since the GPU has a better encoding performance, we may consider increasing the GPU's share of coded blocks. Assigning 54% of coded blocks to the GPU improves the encoding performance to 1000 ms, a 2.17 speedup.

Generating random coefficients in the GPU

So far, our GPU-based encoding implementation uses random coefficients generated by the CPU, and transferred to the graphics memory before the start of the encoding process. Although neither the process of generating random coefficients nor the transfer times take a long time to execute, migrating the task of generating random coefficients to the GPU makes the design simpler, as it fully detaches the CPU from the encoding process. In this case, the application using network coding simply requests the GPU for a number of coded blocks.

Our CPU-based pseudo-random number generator uses integer division and modulo operations, which are expensive operations on GPU cores. Instead, we use an alternative division algorithm that uses a series of shift, multiply and addition for division by a constant integer [35]. Our GPU-based random number generation runs hundreds of generators in parallel, each generating the random sequence for a coded block through a random seed. It takes no longer than 1.61 ms for 7168×1024 random coefficients, which is ten times faster than using the CPU.

3.2.6 Many-core network decoding on the GPU

The decoding process has a higher computational complexity than encoding, as Gauss-Jordan elimination involves n^2 row operations on coefficient rows of length n and coded blocks of length k. Compared to encoding, this leads to a reduced coding per-

formance in general. However, the more critical challenge is the smaller degree of parallelization in the decoding process. Gauss-Jordan elimination requires the decoding of each coded block to start only after the decoding of the previous coded blocks is finished. This implies that the decoding process, unlike the encoding process, lends itself to parallelization only *within* the decoding of the current coded block, and not *across* the decoding of a number of coded blocks.

Such a lesser degree of parallelization limits the performance gain of GPU-based decoding much more than the CPU-based implementation, since the GPU needs to run thousands of threads to be able to achieve its peak performance. In addition, threaded decoding of each coded block requires at least one synchronization point, which makes the decoding process a *coarse-grained* parallel program.

We have tested a number of GPU-based decoding schemes and their performance for the $(c = 1024, n = 1024, k = 2048)$ setup. Not surprisingly, our best-performing scheme only achieves 82% of the CPU-based decoding implementation at this setup, using GPU-based decoding with CPU assistance. We present the details of our schemes in the following sections.

GPU-based decoding with CPU assistance

In a progressive decoding application scenario, we receive each coded block along with its associated coefficients and decode it partially. After receiving and decoding the n-th coded block, the decoding process completes and all n source blocks are recovered if no linear dependence has been encountered. In our first scheme, for every new coded block, we partition the aggregate $n + k$ coefficients and data — *i.e.*, a row of the aggregate $[\mathbf{C}|\mathbf{x}]$ matrix from Eq. (2.2) — such that each 4-byte word of the aggregate data is assigned to a thread, leading to a total of $(n + k)/4$ threads.

Each thread reduces the leading coefficients of the new coded block through a number of linear combinations. However, it can not do further work as a global search for the first non-zero coefficient has now become necessary. Since CUDA's synchronization construct only works for threads within a *single thread block*, and not a global synchronization among all GPU threads, we are forced to perform this synchronization at the CPU side. This effectively breaks the decoding process into two GPU kernel processes. After finding the first non-zero coefficient at the CPU side, we launch another GPU kernel to perform the remainder of the decoding operations for the current block, with each GPU thread performing a series of linear combinations for a 4-byte column of the aggregate $[\mathbf{C}|\mathbf{x}]$.

Although this scheme perfectly divides each aggregate row among threads, it suffers from launching an extra GPU kernel to perform synchronization at the CPU side. The decoding performance of 1024 coded blocks with a $(n = 1024, k = 2048)$ setup achieves 82% of the CPU-based performance (2484 ms against 2031 ms).

Full GPU-based decoding

In an attempt to avoid CPU-assisted synchronization and the extra GPU kernel call, we divide the data portion of the coded block among all thread blocks, but give each thread block its own private copy of the coefficient row. We can now use CUDA's synchronization construct within each thread block to perform the search for the first non-zero coefficient. However, we do not wish to consume an excessive amount of computing power on processing redundant coefficients, so we define a thread block to be as large as possible, employing only one thread block per each of the 14 SMs of the 8800 GT. This leads to each thread block effectively decoding $n + k/14$ bytes of aggregate data through $(n + \frac{k}{14})/4$ GPU threads, each thread working on a 4-byte

column. Unfortunately, even after applying a number of data caching optimizations in the per-SM shared memory, the GPU-only decoding is not able to perform better than 6073 ms, lagging far behind our CPU-assisted decoding approach.

So far, in both of our GPU-based and CPU-based schemes, we considered progressive decoding in a sense that each coded block is decoded individually, *i.e.*, right after receiving the block from network, ending up with a total of n GPU kernel or CPU calls. From a measurement point of view, this implies that we have to synchronize all execution threads, either CPU or GPU, right after the decoding of each new coded block finishes. For many practical application scenarios, *e.g.*, $(n = 128, k = 4096)$, the complete decoding of n coded blocks takes only around $10 - 20$ ms. This suggests that we can buffer the n coded blocks from the network, and then start decoding.

Such a decoding approach helps both CPU-based and GPU-based decoding. First, more of the internal structures (used across the decoding of several coded blocks) can remain in the cache. Second, the threads can be executed longer without being interrupted as individual coded blocks are decoded. With such an approach, the performance of the GPU-based implementation improves to 2175 ms, but our 8-threaded CPU-based scheme also performs better, now at 1688 ms, with GPU-based decoding achieving only 78% of the CPU performance.

3.3 Performance Evaluation

In this section, we evaluate the performance of *Nuclei*, our design and implementation of GPU-accelerated many-core network coding. We use fully dense coding matrices with non-zero coefficients in our evaluation. The performance will be even higher with sparser matrices.

3.3.1 Coding bandwidth

As we evaluate the performance of *Nuclei*, we have tested a range of 128 bytes to 16 KB per block, with 128, 256 and 512 blocks. When compared to our baseline SIMD-accelerated CPU-based implementation with 8 threads (one per CPU core), the coding bandwidth of *Nuclei*, in MB per second, is shown in Fig. 3.4. The encoding (decoding) bandwidth should be interpreted as the total number of bytes that are produced (or decoded) per second.

Fig. 3.4(a) shows that GPU encoding in *Nuclei* achieves its peak performance across almost all coding settings. We are able to make a number of observations from these results. As the number of blocks n doubles from 128 to 256 and again to 512, the encoding bandwidth halves first from 66.9 MB/s down to 33.8 MB/s, and again to 16.8 MB/s. This is due to the fact that generating a coded word requires n GF multiplications. For 128 blocks, the encoding of each word requires reading 128 words of source data and writing one word of coded data, in addition to the reading of coefficients. As such, our coding bandwidth of 66.9 MB/s results in a memory access rate of 10.8 GB/s, which is substantially far below the 57.6 GB/s theoretical limit.

These results have confirmed that our encoding performance is only limited by the computation limits of the 8800 GT. For a number of executed instructions to achieve an encoding bandwidth of 66.9 MB/s at 128 blocks, it is equivalent to an instruction rate of 151 GIPS (Giga instructions per second), which is 90% of the advertised theoretical limits of 504 GFLOPS (*i.e.*, 168 GIPS). This represents a surprisingly high-performance level, confirming that our partitioning scheme performed very well in hiding the memory latency. It also confirms that the GPU can execute concurrent threads *with zero scheduling overhead* in hardware as claimed, and can perfectly hide register read-after-write latencies when a sufficient number of parallel threads exists

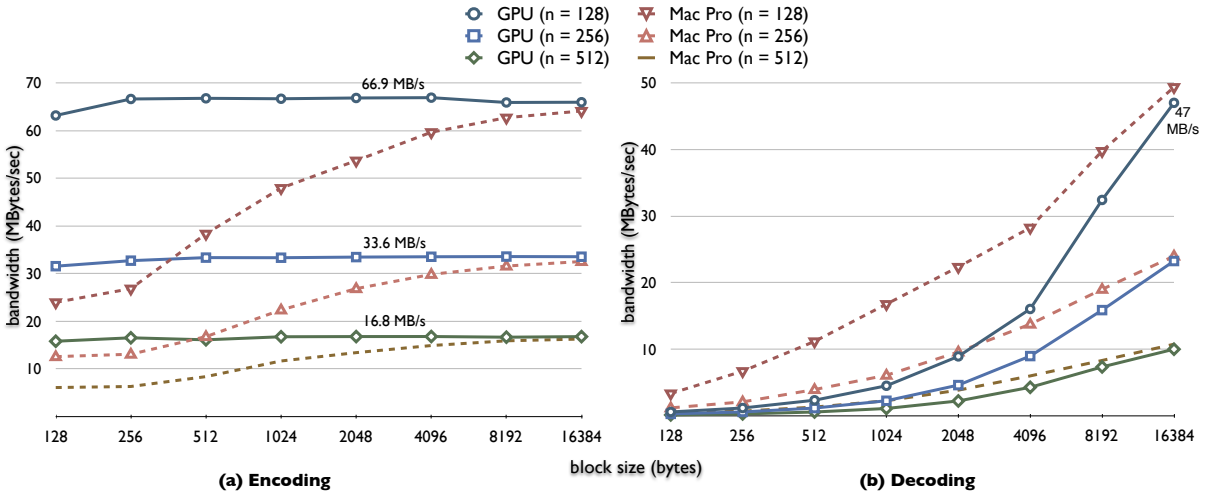


Figure 3.4: Coding bandwidth of GPU-based and CPU-based (8-threaded with SIMD acceleration) for the **(a)** encoding; and **(b)** decoding processes.

within each SM. Since the GPU is only limited by its computation power, its peak performance has been achieved across all (n, k) settings. In comparison, the CPU-based encoding performance follows the same trend as discussed in Chapter 2, but at higher coding rates due to more CPU cores. As the block size increases, the CPU cache performance has improved, leading to better memory performance and higher encoding rates.

With respect to decoding, the decoding performance shown in Fig. 3.4(b) is generally lower with both the GPU and the CPU, because each coded block has to be decoded serially. The CPU performs better than the GPU across the board, especially at smaller block sizes, since the GPU does not have sufficient data (small $k/14$) to launch a sufficient number of threads to achieve an acceptable performance gain, and the CPU's computation power is not affected by the block size, unlike its memory performance. On the GPU, as an example at $k = 128$, each of the 14 SMs decodes only $k/14 = 9.14$ bytes of data on average. At $n = 128$, the decoding time remains around 31 ms even if we increase the block size from 128 to 2048 bytes, because latencies of the

computation pipeline and memory accesses can not be hidden when there is so little useful computation to be performed. As k increases, though the CPU's performance has improved due to a higher k/n ratio and improved memory performance, the GPU has quickly caught up as soon as it has sufficient data to process and to launch a sufficient number of threads.

3.3.2 Network coding with both GPU and CPU

In order to evaluate the maximum achievable performance, we now explore the limits of encoding bandwidth when we employ both the CPU and GPU in parallel. We partition the encoding process by generating α portion of the required coded blocks by GPU, and the remainder of the coded blocks by CPU. Because the coding performance of GPU and CPU are not equal, and with the GPU outperforming CPU, we need to configure α optimally. Fig. 3.5 shows our results. A peak encoding bandwidth of 116.7 MB/s has been achieved at $n = 128$, which can offer sufficient packet payload to saturate the Gigabit Ethernet interface at servers.

3.3.3 Revisiting table-based network coding

We have mainly focused on loop-based GF-multiplication on the GPU so far in this chapter. A table-based implementation creates the $\log/\exp$ tables once at the host side and transfers them to the graphics memory. To achieve higher performance as these tables are heavily accessed, the GPU threads can be better designed by first loading the tables from the graphics memory into the *shared memory*, of each SM, effectively using it as a *managed L1 cache*. Afterwards, each GPU thread performs byte-by-word GF-multiplication, similar to our loop-based approach. Still, our experiments have shown that such a fine-tuned design performs 30% worse than our loop-based approach.

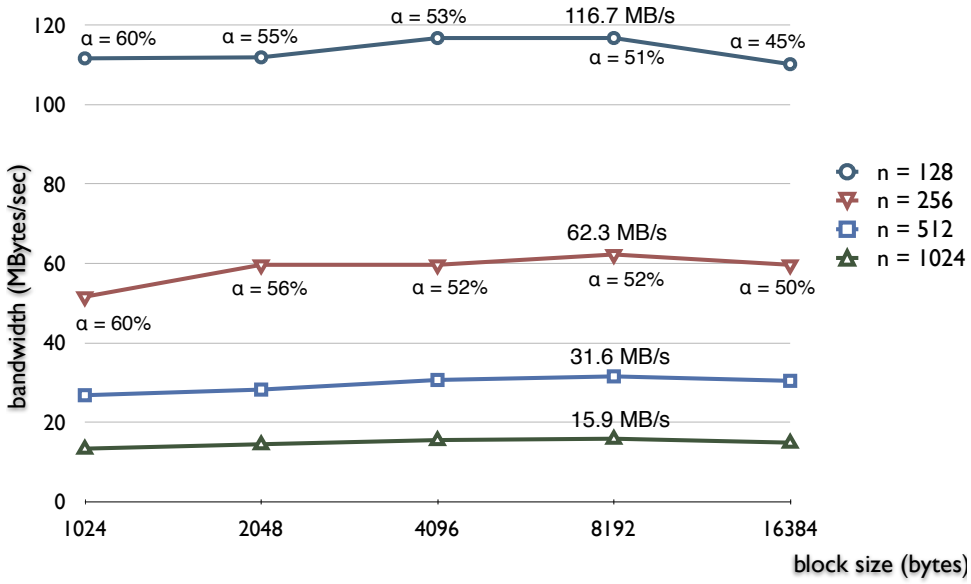


Figure 3.5: Encoding bandwidth with the 8-core CPU and the 112-core 8800 GT GPU combined.

It is, however, possible to optimize table-based GF-multiplication further, which will be explored in details in Chapter 4. The basic idea is the following. In our partitioning, each GPU thread, which calculates a word-length worth of a coded block, accesses a full coefficient row of $\mathbf{C}$ and a full column of original blocks $\mathbf{b}$. However, many other threads use the same row and column in their own coding processes that leads to many redundant conversions to the $\log$ domain. By first preprocessing $\mathbf{C}$ and $\mathbf{b}$ and transferring them fully to the $\log$ domain, the number of table accesses can be effectively reduced by $2/3$. Our new optimized table-based encoding can improve the performance by 25% over the loop-based approach, as demonstrated in Fig. 3.6.

Finally, we have also discovered that the performance of optimized table-based GF-multiplication can be further improved by manipulating the tables to more optimally exploit the GPU hardware of SP cores, and by improving the access pattern to the shared memory. With these performance improvements, we are able to achieve

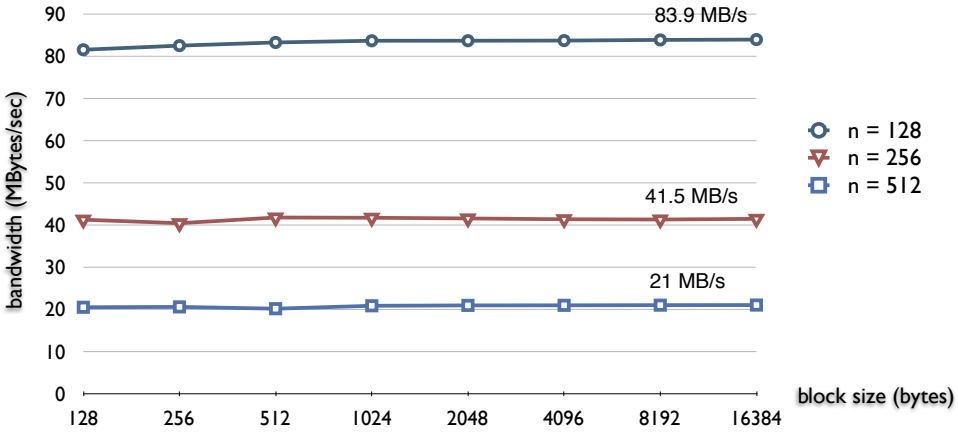


Figure 3.6: Encoding bandwidth of the optimized table-based approach on the 8800 GT GPU.

encoding rates up to 294 MB/second with $n = 128$ blocks, in our experiments on the high-end NVIDIA GeForce GTX 280 GPU. At this performance level, there is no need to involve the CPU, *e.g.*, in a combined encoding scheme with the GPU, to satisfy most real-world performance needs.

This result certainly does not imply that the loop-based encoding on GPU should be written off altogether. The next generations of CUDA GPUs will likely increase their integer arithmetic units to 64 bits which potentially can double the performance of loop-based GF-multiplication. In contrast, it is less likely to observe a substantial performance improvement of the shared memory on GPU SMs in the future. Further, our optimized loop-based approach in Fig. 3.3 can be applied to similar processor cores as GPU SPs, especially with 32-bit execution units and little or no SIMD support, *i.e.*, the mainstream ARM v6 family used in smartphones[7].

[7]In fact, we evaluate such loop-based scheme in Chapter 5.

3.3.4 Feasibility of using network coding on streaming servers

As we have just shown, the performance of *Nuclei* makes it feasible for it to be deployed in high-performance streaming servers using network coding, with hundreds of clients served concurrently. As an example, consider the scenario of using a media segment size of 512 KB, with 128 blocks of 4 KB each, corresponding to a $(n = 128, k = 4096)$ setting. With a streaming rate of 768 Kbps that is typical for high quality video streams, each segment contains content that lasts 5.33 seconds, which is an acceptable buffering delay on the client side. With *Nuclei* operating at this setting, the coding bandwidth is sufficiently high to serve up to 870 clients with the mainstream 8800 GT GPU alone. In addition, when a media segment is ready to be encoded and served, it can be transferred and stored in the graphics memory on the GPU. Even with the modest memory capacity of 512 MB on the 8800 GT, hundreds of such segments can be accommodated.

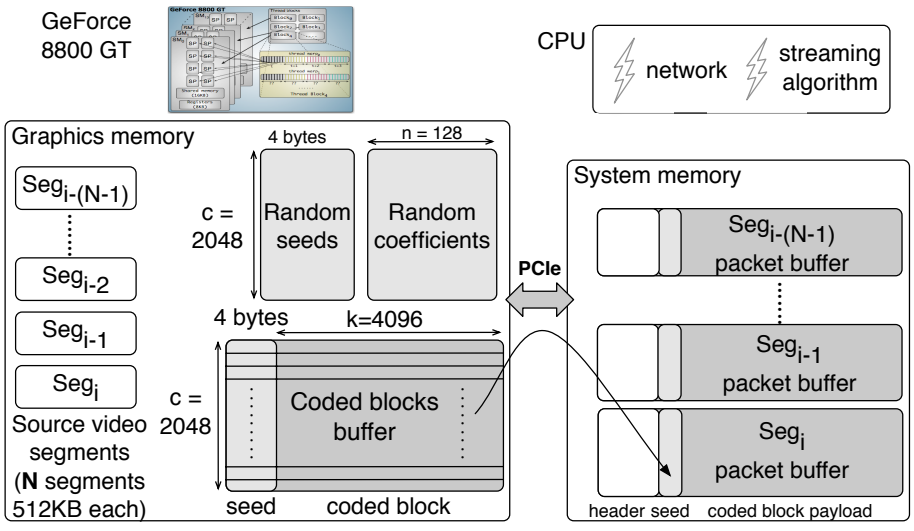


Figure 3.7: GPU-accelerated network coding in a dedicated streaming server.

The layout of segments and buffers in both graphics and system memory are shown

in Fig. 3.7. The GPU keeps two other buffers to manage random seeds and their associated random coefficients. Coded blocks are directly loaded from the graphics memory as they are produced. The only involvement of the server CPU in network coding is to transfer new source segments to the GPU for encoding, to request a number of coded blocks from a particular segment in the GPU, and when they are ready, transfer them back to its buffers in the system memory. As such, the CPUs are relieved to perform other CPU-intensive tasks, such as media encoding.

3.4 Summary

This chapter presented *Nuclei*, a high-performance design and implementation of many-core network coding using GPUs. *Nuclei* is able to achieve 90% of the advertised theoretical limits (504 GFLOPS) of the NVIDIA GeForce 8800 GT, across a wide range of network coding configurations. We have provided an in-depth comparison of GPU-accelerated network coding against a CPU-based implementation. We have shown that many-core GPUs can be deployed as an attractive alternative solution to multi-core servers, by offering a higher encoding performance level at a much lower cost. Furthermore, multi-core CPUs and many-core GPUs can be used to perform network coding simultaneously, which achieves a level of coding performance sufficient to saturate a Gigabit Ethernet interface and to serve in media streaming servers. With a much better memory performance than CPUs and a rapidly increasing number of cores, GPUs are very promising to bring network coding to reality.

Chapter 4

Extreme Network Coding on the GPU

Chapter 2 has shown an accelerated multi-threaded implementation of network coding, that takes advantage of both multiple CPU cores with aggressive multi-threading, and SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors as well. Further, the work presented on *Nuclei* in chapter 3, the first network coding implementation on *Graphics Processing Units (GPU)* in the literature [67], achieved encoding rates of up to 114 MB/second by combining 8-core Intel Xeon CPUs and a mainstream NVIDIA GeForce 8800 GT GPU.

Though in chapter 3 we have achieved a level of performance that has not been previously reported, this chapter represents another substantial step forward, and presents a new array of optimization techniques and algorithms to further improve the performance of GPU-based network coding by a significant margin beyond our previous work. With our new algorithms, A single NVIDIA GeForce GTX 280 performing network coding at 128 blocks achieves encoding rates up to 294 MB/s and decoding rates up to 254 MB/s, far beyond the computation bandwidth required to saturate a Gigabit Ethernet interface on streaming servers. At such high rates, we argue that GPU alone is sufficient to deploy network coding on streaming servers setting

the CPU cores free for other CPU-intensive tasks.

We have made the following new observations. *First*, the NVIDIA GeForce GTX 280, featuring 240 cores, far outperforms our CPU-based implementation on a Dual Quad-core 2.8 GHz Intel Xeon server (8-core Mac Pro). *Second*, unlike the CPU, the GPU can benefit from a novel and highly optimized table-based encoding technique that outperforms the loop-based encoding technique employed in Chapter 3 by a factor of 2.2. *Third*, by parallel decoding of multiple segments, the performance of GPU-based network decoding can be improved by a factor of 2.7 to 27.6, across a range of practical configurations.

With our new results, we argue that a single GPU can achieve a sufficiently high level of performance to satisfy the needs of most application scenarios. Further, for network coding applications, the price/performance ratio of GPUs is far superior to multi-core servers. For the exceptionally demanding applications, multiple GPUs can be employed in parallel. By setting free the CPUs from computation-intensive network coding, a far more reliable system with a better performance guarantee can be deployed, which is especially beneficial to media streaming servers.

The remainder of this chapter is organized as follows. Sec. 4.1 presents the basis of parallel coding on GPUs along with network coding performance results for the GTX 280. Sec. 4.2 presents and evaluates a variety of novel techniques to maximize the real-world network coding performance with GPUs. Finally, Sec. 4.3 summarizes the chapter.

4.1 Network Coding with the GTX 280

In this section, we present the basis of parallel network coding on GPUs, its challenges, and our solutions. Then we evaluate GPU-based network coding on the GTX 280, and

compare the results against the best performing results reported in our previous work in Chapter 3.

The NVIDIA GeForce GTX 280 GPU, though retails at mainstream GPU pricing, is supported by the CUDA programming platform. Our performance evaluation of GPU-based network coding in this chapter is based on the GTX 280 GPU. Compared to 8800 GT used in Chapter 3, GTX 280 has more than twice as many computing cores, 240 versus 112. GTX 280 also has doubled its register count to 16384 as shown in Fig. 4.1. It also has added support for double-precision floating point. However, it has only one double-precision floating-point (DPFP) unit in each Simultaneous Multiprocessor (SM). On the other hand, each Scalar Processor (SP) is also capable of single-precision floating-point beside integer arithmetic.

Our design and implementation, however, can be used on any existing and future GPU that supports the CUDA programming platform. To facilitate our subsequent discussions in this chapter, we refer the reader to [26] and also Sec. 3.1 for an overview of the Tesla GPU architecture and CUDA.

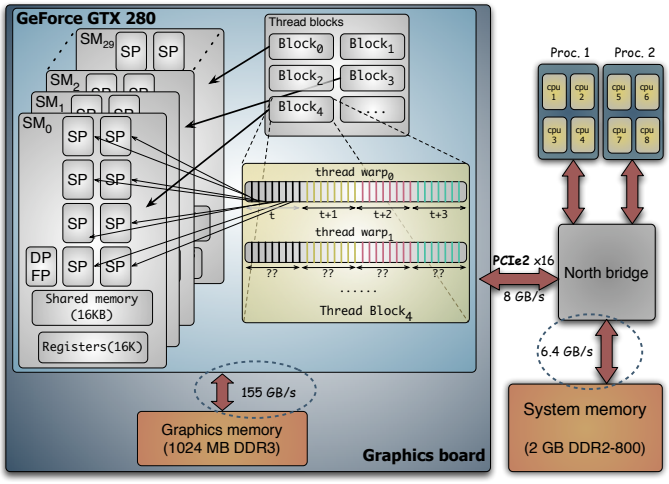


Figure 4.1: The architecture of our Mac Pro testbed with the NVIDIA GeForce GTX 280 GPU.

4.1.1 Performance bottlenecks in network coding

Random network coding suffers from two major performance bottlenecks. First, a table-based multiplication in GF(2^8) is a costly operation. Second, the multiplication and addition operations are performed in tight loops over rows of coefficients and coded blocks, each of n and k bytes, respectively. Each row operation is performed through a series of byte-length GF(2^8) operations, since table-based GF(2^8) multiplication is not easily scalable to a higher granularity than the byte level.

To address such a performance bottleneck, we first proposed in Chapter 2 to revisit the basics by performing the multiplication on-the-fly using 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 that takes advantage of vector instructions in order to operate on wider chunks of elements from a matrix row at the same time. The loop-based equivalent of the table-based multiplication in Fig. 2.1 resembles a regular hand multiplication (see Fig. 3.3) by looking into the lower bit of x and adding y at each iteration (addition is equivalent to *XOR* in GF(2^8)).

With such a loop-based approach, we took advantage of SSE2 and AltiVec SIMD (single-instruction, multiple data) vector instruction sets on Intel and IBM PowerPC processor families to perform GF(2^8) multiplication on 16 byte-long units of each row, rather than single-byte units [64]. In Chapter 3, we followed the same principle for GPUs but faced two major challenges. First, unlike modern CPU cores with 128-bit registers and execution units, current CUDA-enabled GPUs have plain 32-bit registers and arithmetic units. As a result, we can only go as far as single byte by 4-byte word GF-multiplication. Second, GPU cores lack the sophisticated SIMD instructions that allow test and manipulation of individual bytes of a word. This leads to longer and

less efficient code.

Our previous work in Chapter 3 showed that it is possible to use the GPU and still achieve accelerated GF-multiplication using a loop-based approach, despite the lack of CPU-like vector instructions. A number of optimization techniques, including hand-optimization of the PTX assembly code, were used for efficiently employing the 112 cores of 8800 GT. It was reaffirmed that the GPU's advantage over CPUs is their ability to schedule thousands of lightweight threads with almost zero overhead in hardware, to hide stalls in the processing cores due to data dependency and memory access.

4.1.2 Task partitioning on the GTX 280

Although current CUDA-enabled GPUs lack the wide registers and arithmetic-logic units (ALUs) of the modern CPUs, they have many processing cores (*e.g.*, 240 for the GTX 280), which run in parallel and can achieve the same functionality as wide execution units. In the following sections, we briefly overview our parallelization scheme of network encoding and decoding. These partitioning schemes are essentially the same as the ones used for 8800 GT in Chapter 3 but with some extra fine-tuning for the GTX 280. A good understanding of these schemes is necessary when we present our new techniques in Sec. 4.2.

Partitioning parallel network encoding

The process of random network encoding essentially consists of a matrix multiplication in the GF domain, as described in Sec. 2.1, with single-byte coefficients multiplied in each row of source blocks, and the results *XOR*-ed together to form a coded block. A parallel implementation of network encoding falls in the category of what is known

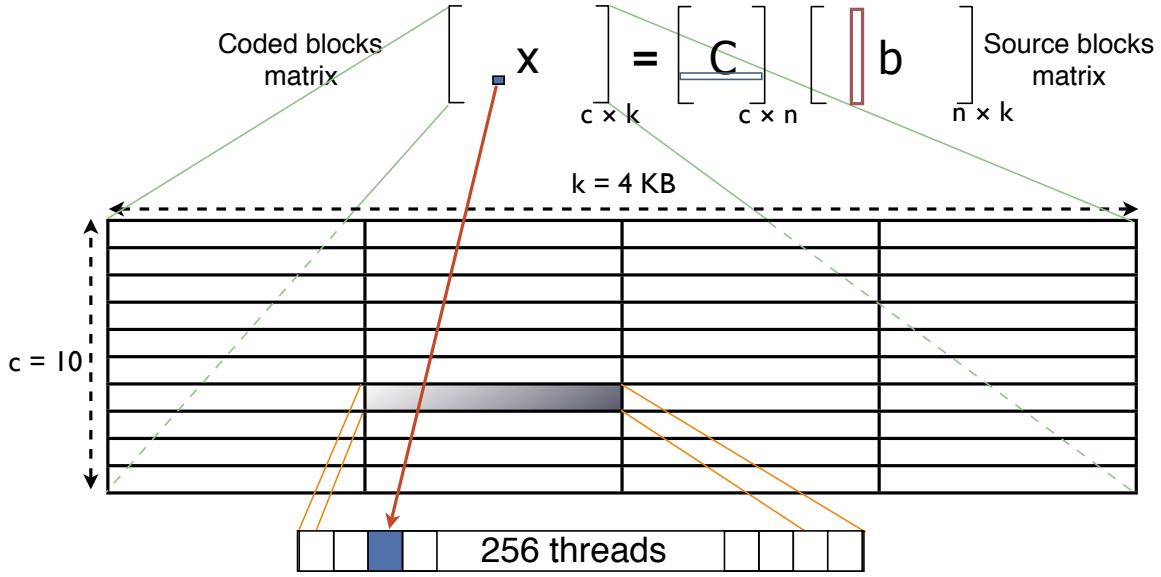


Figure 4.2: 10,000 GPU threads performing parallel network encoding for 10 coded blocks of each 4 KB.

as *embarrassingly parallel* problems, where a parallel implementation is possible with little communication and synchronization among parallel threads. Ignoring memory access to source blocks and coefficients, the performance of loop-based network encoding is only limited by the computational power of the hardware, since the encoding process of multiple coded blocks — and even different section of a coded block — can proceed in parallel by employing a large number of threads.

Fig. 4.2 shows how generation of 10 coded blocks, each of $k = 4$ KB, is assigned to GPU threads. Each element of the grid reflects a *thread block* consisting of 256 threads. Each thread of a thread block generates a 4-byte word of the coded block by performing the encoding operation according to Eq. (2.1) on n words from the source blocks using loop-based GF-multiplication. Effectively, each thread block generates 1 KB worth of coded data, *i.e.*, $10,000$ threads are used to generate 10 coded blocks.

With careful assignments of words to threads of each *thread warp*, we take advan-

tage of **(1)** the memory broadcast feature [26] of the GTX 280 for coefficients load, **(2)** the memory coalescing [26] for loading source and storing coded blocks. Since most memory accesses of a thread warp, except coefficient reads, fall next to each other, such partitioning significantly reduces the number of accesses to the GPU memory.

Partitioning parallel network decoding

The decoding process has a higher computational complexity than encoding, as Gauss-Jordan elimination involves n^2 row operations on coefficient rows of length n and coded blocks of length k. However, the more critical issue is the smaller degree of parallelization inherent in the decoding process. Gauss-Jordan elimination requires the decoding of each coded block to start only after the decoding of the previous coded blocks is finished. This implies that the decoding process, unlike encoding, lends itself to parallelization only "within" the decoding of the current coded block, and not "across" the decoding of a number of coded blocks.

Such a lower degree of parallelism limits the performance gain of GPU-based decoding much more than the CPU-based implementation, since the GPU needs to run "thousands of threads" to be able to achieve its peak performance. In addition, threaded decoding of each coded block requires a synchronization point, for searching the first non-zero coefficient, which makes the decoding process a *coarse-grained* parallel program [67].

This synchronization point is a major obstacle in deep parallelization of decoding. CUDA's synchronization construct only works "within" threads of a thread block and does not support global synchronization among all GPU threads. To work around the required global synchronization, we divide the data portion of the coded block among all thread blocks, but give each thread block its own "private copy" of the coefficient

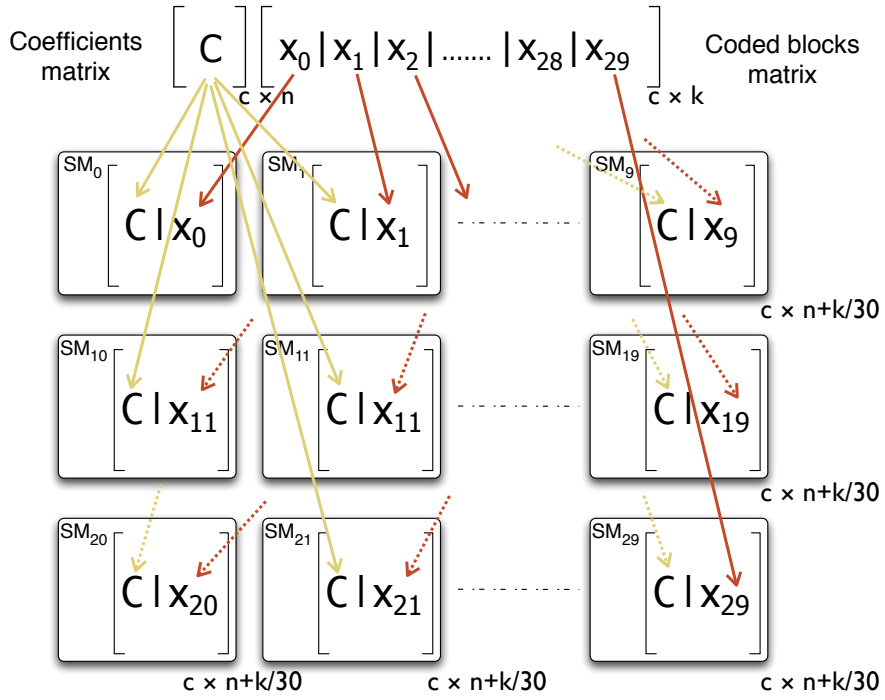


Figure 4.3: Parallel network decoding on 30 SMs (Stream Multiprocessors [26]) of GTX 280 with duplicate coefficient matrix and partitioned coded blocks matrix.

row. We now can use CUDA's per thread block synchronization to perform the search for the first non-zero coefficient in each thread block. However, we do not wish to consume too much of our computing power on processing redundant coefficients, so we define a thread block to be as large as possible, employing only one thread block per each of the 30 SMs of the GTX 280. This effectively leads to decoding $n + k/30$ bytes of aggregate data by each thread block implying $(n + \frac{k}{30})/4$ GPU threads, each working on a 4-byte column.

This partitioning scheme is shown in Fig. 4.3, where SP_i of the GTX 280 performs the decoding on the aggregate $[\mathbf{C}|\mathbf{x}_i]$ matrix where $\mathbf{x}_i$ is the i-th partition of the coded block matrix $\mathbf{x}$. As we shall show in the next section, such parallelized decoding is still not able to fully take advantage of the GTX 280's computation power. We will revisit

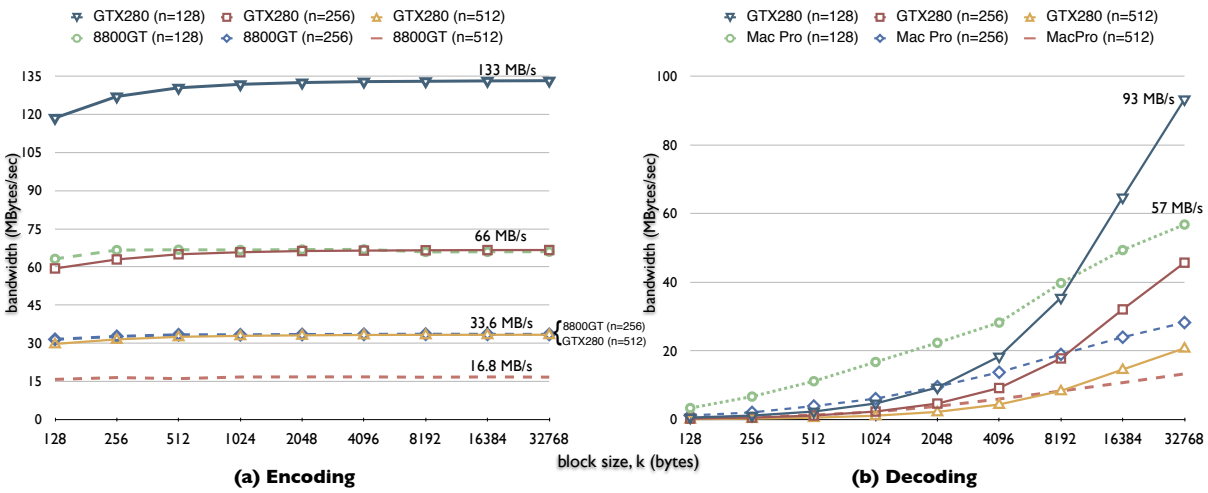


Figure 4.4: Coding bandwidth of GPU-based and CPU-based (8-threaded with SIMD acceleration) for network **(a)** encoding; and **(b)** decoding processes.

parallelized decoding in Sec. 4.2 with a new approach.

4.1.3 Evaluating the GTX 280

Now we evaluate the GTX 280 running our loop-based implementation of network coding on the GPU. GTX 280 boasts 240 processing cores compared to 112 in the GeForce 8800 GT, but it runs at a slightly lower core frequency of 1458 MHz against 1500 MHz. With almost twice the amount of computing power of the 8800 GT, we can expect better coding performance over the 8800 GT. However, the main questions are that whether the coding bandwidth can scale up linearly, and whether computation and memory can interleave efficiently so that memory latency can be hidden. We use fully dense coding matrices as Chapter 3 with non-zero coefficients in all of our evaluations in this work unless explicitly mentioned otherwise. The performance will be even higher with sparser matrices.

To evaluate the GTX 280, we use the testbeds in Chapter 3 as our benchmarks,

namely a 8800 GT GPU and a 8-core Intel Xeon 2.8 GHz Mac Pro server (SIMD accelerated with 8 threads, one thread per CPU core). To have a readable graph, however, we only use the 8800 GT as the encoding benchmark and the Mac Pro as the decoding benchmark. Although the 8800 GT and the Mac Pro essentially achieve similar coding rates at $k = 16$ KB, 8800 GT consistently achieves better encoding performance, especially at smaller block sizes [67]. For decoding, the case is reversed and CPU-based decoding on the Mac Pro system defeats 8800 GT hands down (refer to Chapter 3 for a full performance comparison between the two).

We have tested a range of network coding configurations with 128 bytes to 32 KB per block, with 128, 256 and 512 blocks. The coding bandwidth of the GTX 280, in terms of MB per second, is shown in Fig. 4.4. Note that the encoding (decoding) bandwidth should be read as the total bytes of generated coded blocks (decoded source blocks) within one second with a network coding setup of (n, k).

Fig. 4.4(a) shows that encoding in GTX 280 achieves a rate almost twice of 8800 GT, a linear speedup, across all coding settings. This is not surprising as the encoding process is an *embarrassingly parallel* operation, and the fact that GTX 280's memory bandwidth is more than double of 8800 GT's (155 GB/s vs. 57.6 GB/s). As a result, the computation power of GPU still remains the performance bottleneck of our loop-based encoding scheme.

At a 133 MB/s encoding rate for the $n = 128$ setting, 4463 million GF-multiplications are executed every second. At an average 7 iterations per GF-multiplication in a random test benchmark as in Chapter 3 and each iteration taking an average of 10.5 instructions, the instruction rate will be 329 GIPS (Giga instructions per second). Our GTX 280's theoretical limit of 1080 GFLOPS translates to 360 GIPS. As a result, GF-multiplications alone (not considering the overhead associated with loop traversal, GPU kernel launch, etc.) effectively achieves 91% of the advertised computing power

of GTX 280. This confirms that our encoding task partitioning scheme managed to hide memory latency very well. Similar calculations put the memory access rate at 20.9 GB/s (each word of the generated coded data requires $5 \times n + 4$ bytes of read and write), which is substantially lower than the theoretical memory limit of 155 GB/s.

With respect to decoding, as discussed in Sec. 4.1.2 and observed in Fig. 4.4(b), the decoding performance is generally lower for both GPU and CPU-based schemes because coded blocks are decoded serially. The GTX 280 performs better than 8800 GT in Chapter 3, by defeating the Mac Pro for blocks of 8 KB and larger. However, the CPU still performs better than the GTX 280 at smaller block sizes, because the GPU does not have sufficient data (small $k/30$) to launch a sufficient number of threads accordingly. In contrast, CPU's more efficient microarchitecture performs a better job even at small block sizes. As k increases, the performance of both GPU and CPU increases partially due to a lower overhead associated with the decoding of the coefficient matrix, proportional to the n/k ratio. The lack of parallelism in the decoding process prevents the GTX 280 to fulfill its almost twice computing advantage over the 8800 GT. For example, at $n = 128$, the decoding rate of GTX 280 achieves virtually the same performance as the 8800 GT (in a graph not shown here) up to a block size of 1024 bytes. From blocks of 2 KB to 16 KB length, the GTX 280 achieves a modest 5% to 38% gain over 8800 GT.

4.2 Pushing the Envelope of Network Coding

We now propose a number of new schemes leading to significant performance improvements in GPU-accelerated network coding. A number of our proposed schemes also improve CPU-based coding.

4.2.1 Table-based parallel encoding

So far, we focused only on loop-based GF-multiplication. Would the table-based approach perform better with the GPU? We already know from Chapter 2 that this is not the case for CPU-based coding.

We now migrate the table-based GF-multiplication of Sec. 2.1 to the GPU. Exponential and logarithm tables are created on the CPU side once and then transferred to the GPU memory. Our experiment here follows the same partitioning scheme of Sec. 4.1.2. However, at each byte-by-word GF-multiplication, a similar table-lookup scheme as Fig. 2.1 will be invoked four times in a row.

Accessing $\log$ / $\exp$ tables from GPU memory results in very poor performance. In an improved version, we load up the tables from the graphics memory into the on-chip *shared memory* of each GPU's SM at the start of the encoding process. Because all threads of a thread block can access the shared memory seamlessly, we effectively use the shared memory as a "managed L1 cache" shared among the active threads of a thread block with far less access latency than a memory fetch from the graphics memory. However, each SM has only 16 KB of such on-chip memory so the thread blocks have to be sized carefully to let many threads share the same $\log$ / $\exp$ tables.

To improve the initial caching of the $\log$ / $\exp$ tables, each thread of a thread block loads a 4-byte portion of the table in an optimized fashion, ensuring memory coalescing in threads of each thread warp. After the tables are loaded, all threads proceed as a regular encoding thread we employed in Sec. 4.1.2. This method has led to a substantial improvement over our first table-based scheme. Unfortunately, in an experiment at $n = 128$, such a fine-tuned table-based scheme still performs 26% worse than our loop-based approach.

Table-based GF-multiplication for streaming servers

Before completely deserting table-based GF-multiplication, we have performed a more in-depth investigation into how network coding can be employed in a streaming server. As our performance results from Fig. 4.4 indicate, the encoding performance is so high that it can be deployed in high-performance live or VoD streaming servers to serve hundreds of downstream peers or clients. As an example, consider the scenario of using a media segment size of 512 KB, with 128 blocks of 4 KB each. With a streaming rate of 768 Kbps that is typical for high quality video streams, each segment contains content that lasts 5.33 seconds, which is an acceptable buffering delay on the client side.

Operating at this setting, the coding bandwidth of 133 MB/s is sufficiently high to saturate a Gigabit Ethernet interface and serve up to 1385 downstream peers. In addition, when a media segment is ready to be encoded and served, it can be transferred and stored in the graphics memory on the GPU. 1024 MB memory on the GTX 280 is able to easily accommodate hundreds of such segments. Then, per request from the downstream peers, many coded blocks will be generated from a GPU-resident segment. For example, serving so many peers in a live video stream requires generating at least $177,333$ coded blocks from every video segment.

Revisiting our baseline table-based GF-multiplication of Fig. 2.1 and taking into account the fact that thousands of coded blocks are generated from the source blocks of each video segment, we can come up with a more efficient use of table-based GF-multiplication as suggested by the following algorithm: **(1)** As soon as a new video segment becomes available and transferred to the graphics memory, it will be transformed to the GF logarithmic domain by transforming every byte of its content. **(2)** Similarly, as soon as a new coefficient matrix, filled with random numbers, is gen-

erated by the GPU, it too will be transformed to the $\log$ domain. **(3)** In the actual encoding process, we execute a simpler GF-multiplication operation based on Fig. 4.5. Multiplication by 0 is now detected by checking the inputs against `0xff` since $\log(0)$ is equal to `0xff` in Galois Field.

```
byte preprocessed_gf_multiply(byte log_x, log_y)
{
    if (log_x == 0xff || log_y == 0xff)
       return 0;
    return exp[log_x + log_y];
}
```

Figure 4.5: New table-based multiplication in GF(2^8) with inputs already in logarithmic domain.

Note that beside the improvement achieved through preprocessing the source blocks, *i.e.,* the video segment, the same preprocessing of the coefficient matrix is also a significant help. This is because every coded word generated by a thread of the encoding scheme of Fig. 2.1 fetches and multiplies a full coefficient row of n bytes, so a one-time preprocessing reduces the redundant transforms to the $\log$ domain in the original table-based approach by $k/4$ times.

Evaluating table-based parallel encoding

We now employ the optimized table-based scheme we presented in Sec. 4.2.1. We design a new partitioning scheme for all preprocessing and encoding stages and assign only a single thread block per each SM. This is intended to reduce the number of loads of $\log$ / $\exp$ tables because each thread block requires a full table at every kernel execution. Unlike CPU caches, CUDA's shared memory is "not persistent" across GPU kernel calls. The performance results are shown in Fig. 4.6, comparing the optimized

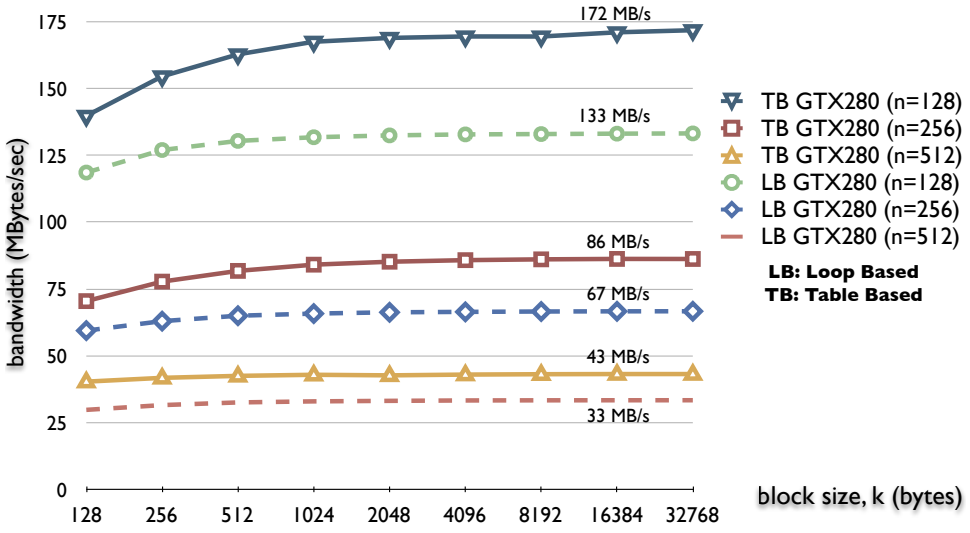


Figure 4.6: Parallel network encoding using the optimized table-based vs. loop-based scheme on GTX 280.

table-based scheme against the loop-based scheme both on GTX 280. As it is evident from the graph, the encoding performance has improved by at least 30% across all settings.

Although a rough estimate of executed instructions for this optimized table-based scheme (including the preprocessing overhead) amounts to around half of the loop-based GF-multiplication, the observed performance improvement is not proportional to the reduction in instruction count. This is mainly attributed to two factors. First, the bandwidth of shared memory is one access (up to 32-bit) per bank in every two cycles [26]. Second, hosting the $\log / \exp$ tables in the shared memory leads to many concurrent byte-length and random accesses to different sections of the tables. These accesses can not be coalesced. Further, bank conflicts will be an issue too. As a result, a 30% performance increase is still a significant gain. At such high encoding bandwidth, now more than 1844 downstream peers can be supported in the streaming server scenario presented in Sec. 4.2.1.

Further, our table-based improvement is not limited to the streaming server applications that generate many coded blocks for every source segment. In an experiment, we produced only n coded blocks for each segment of an array of segments, *e.g.*, a VoD scenario where each client requests a different video segment. The performance degraded only by 0.6% compared to the single-segment case, due to the extra preprocessing. This suggests that our algorithm in Sec. 4.2.1 can be applied to multiple source segments at once.

To be fair to the CPU-based scheme, we also deploy the same optimized table-based approach with preprocessing of the source blocks and coefficients matrix to the logarithmic domain. Not only CPU-based encoding fails to achieve any gain, its bandwidth drops up to 43% from the loop-based SIMD accelerated solution. Obviously, for a CPU-based solution, even the optimized form of the table-based scheme is still not a contender for the loop-based scheme.

This result certainly does not imply that the loop-based encoding on the GPU should be written off altogether. The next generations of CUDA GPUs will likely increase their integer arithmetic units to 64 bits, which potentially can double the performance of loop-based GF-multiplication. Further, our optimized loop-based approach can be applied to similar processor cores as GPU SPs, especially with 32-bit execution units and little or no SIMD support, *i.e.*, the mainstream ARM v6 family used in smartphones.

Pushing the envelope: further optimizations

Further improvement of coding performance is possible through a number of optimization schemes. *First*, for each byte by word multiplication, we combine four tests of `log_x == 0xff` in Fig. 4.5 into one because a single coefficient is multiplied into

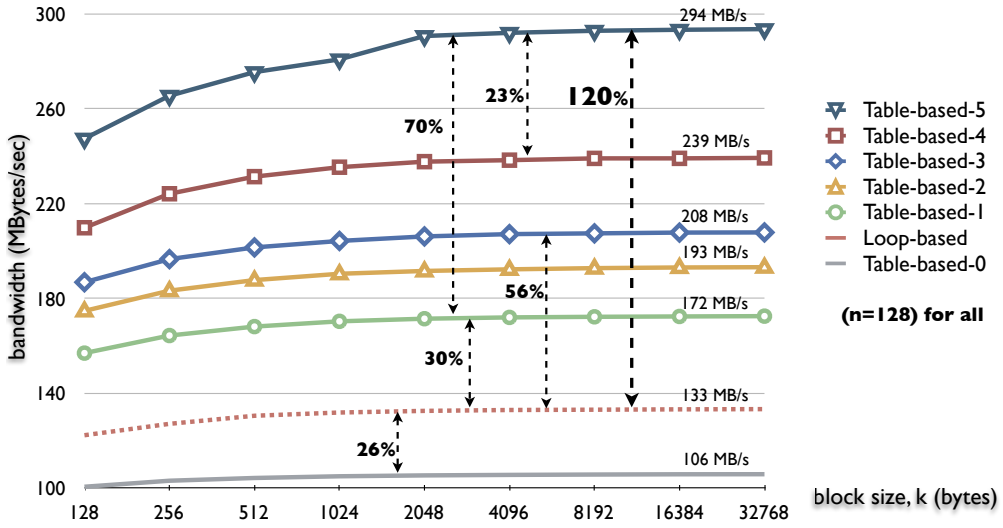


Figure 4.7: Encoding performance of various schemes for $n = 128$ on GTX 280.

the 4-byte word. *Second*, the tests against `0xff` for individual bytes of each word can be converted to tests against `0x00` by using a *new* log *table* such that a zero input is mapped to `0x00` instead of the original `0xff`. This leads to significant performance increase because the tests against zero can be automatically performed during a register load without the need for extra compare instructions. As a result, branching no longer happens as the compiler will use *predicated instructions* leading to even lower instruction count. Of course, the exp table also has to be adjusted to compensate for the new log table. The results of these two schemes are shown as `Table-based-2` and `Table-based-3` respectively in Fig. 4.7. `Table-based-1` reflects our new table-based scheme in Sec. 4.2.1, while `Table-based-0` shows the original results before any optimization.

For our *third* optimization effort, we evaluate the performance of *texture memory* for storing the exp table. We basically run the same algorithm except that we access the exp table resident in the texture memory (texture memory accesses are cached). According to the `Table-based-4` graph in Fig. 4.7, this leads to another 15% per-

formance improvement. Unfortunately there is very little public information on the structure of the texture cache (*e.g.*, cache line size and latency) and how concurrent accesses to texture caches are resolved. This improvement is mainly due to the locality of accesses to the exp table and also the smaller number of instructions needed for address calculation in texture memory accesses compared to shared memory's address calculation. Also, we suspect that the texture cache controller can combine multiple pending requests to a cache line even if initiated from different SMs (in GTX 280, every three SMs use a shared texture cache).

In table-based GF-multiplication, each thread of a thread warp accesses a different byte from the exp table, in general. Except our last scheme, which used the texture memory, we store the exp table in the shared memory. The shared memory has 16 banks, each with a 4-byte width, to serve 16 concurrent requests issued from threads of a half-warp every two cycles. Because these accesses are to random locations in the exp table, albeit with some locality, stalls due to bank conflicts are common. In average, around 3 conflicts happen within each 16 parallel requests, which need to be resolved through serially issued accesses. Our *fourth* and final optimization effort attempts to eliminate the bank conflicts by employing multiple copies of the exp table, so each thread accesses its private copy of the table. Ideally, we need 16 such exp tables which should easily fit in the shared memory of 16 KB size. However, banks are interleaved within the address space of the shared memory, so mapping each thread's access to its private exp table requires a number of extra instructions for address calculation. It turns out that this extra overhead defeats any gain achieved by conflict-free access.

To alleviate address mapping for referencing the table, we store each element as a word rather than a byte. However, with word-length elements, we can only fit 8 copies of exp tables in the share memory (each table has 512 elements). Although this scheme can not fully eliminate bank conflicts, it reduces the conflict probability

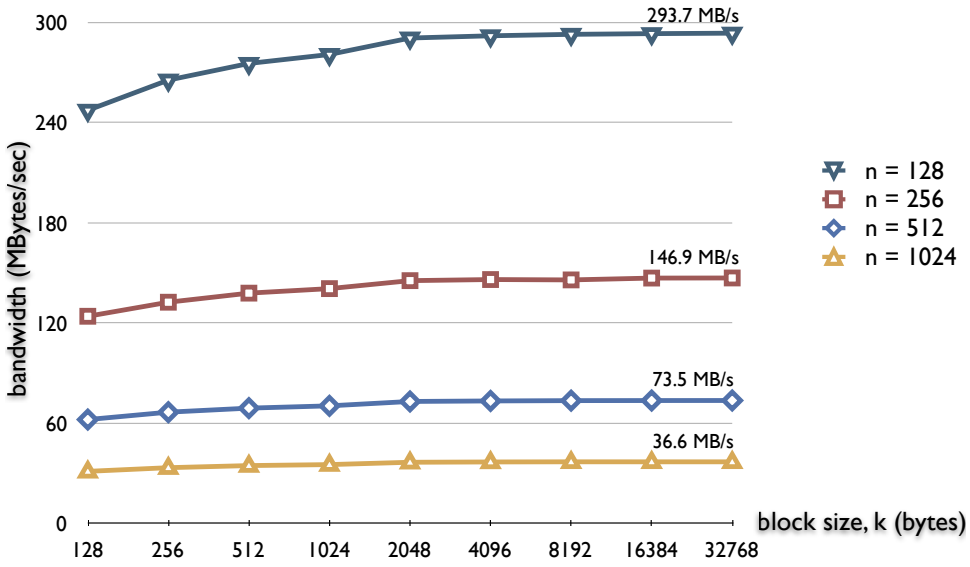


Figure 4.8: Highly optimized encoding on GTX 280.

significantly. Fitting eight tables does not turn out to be easy as the shared memory is also used for other essential tasks, *e.g.*, passing parameters to the GPU kernel. Also, we optimize address calculation to minimize the number of instructions. The result, shown in the `Table-based-5` graph, improves our previous scheme by another 23% up to 293.7 MB/s, which is is 2.2 times of our loop-based scheme. Such an encoding rate can serve more than 3050 peers, way up from 1385 peers, at the streaming setting we discussed in Sec. 4.2.1. Our estimates show that the encoding performance would be around 330 to 340 MB/s for a fully conflict-free deployment if the share memory size was at least 32 KB.

Fig. 4.8 summarizes our best encoding performances across various coding settings. Now even encoding at $n = 1024$ achieves rates in order of a few tens of MB/s. As a final note, memory accesses of the encoding process are almost perfectly hidden within the computations. A benchmark that generates dummy input data (source blocks and coefficients) on the fly, instead of accessing the graphics memory, performs

better by only 0.5%. This confirms that access to the graphics memory has almost no negative effects in the performance of our encoding algorithm.

4.2.2 Parallel multi-segment decoding

As pointed out in Sec. 4.1.3, due to the lack of parallelism in our decoding scenario, our GPU-based network decoding had difficulty to scale up well and to fully take advantage of the available GPU computing power. Coded blocks have to be decoded one by one till a segment is fully decoded. Only then the decoding of the next segment starts. However, this is not the only application scenario to use network decoding. Avalanche [34], which uses network coding in bulk content distribution, gathers a large number of coded blocks over a period of time and performs decoding offline. Even in peer-to-peer VoD applications where the downstream bandwidth varies significantly over time, a peer might receive multiple video segments at the same time to take advantage of the available bandwidth at the moment.

The degree of parallelism in the decoding process increases linearly with the number of available segments, since coded blocks from different segments can be processed in parallel. Then we will have more coded blocks to work on, *i.e.,* more GPU threads to launch, leading to better masking of memory latency and data dependency in GPU cores.

Let us revisit our task partitioning scheme in Fig. 4.3, and assume that there exist coded blocks from 30 different segments. Then an ideal solution will decode each segment fully by a dedicated SM so there will be no longer the need to duplicate the coefficient matrix $\mathbf{C}$ at every SM. Each SM works on a full coded block matrix $\mathbf{x}$. However, a new problem arises as the original thread assignment scheme will no longer work. We assigned one thread for the decoding of every 4-byte column of the aggre-

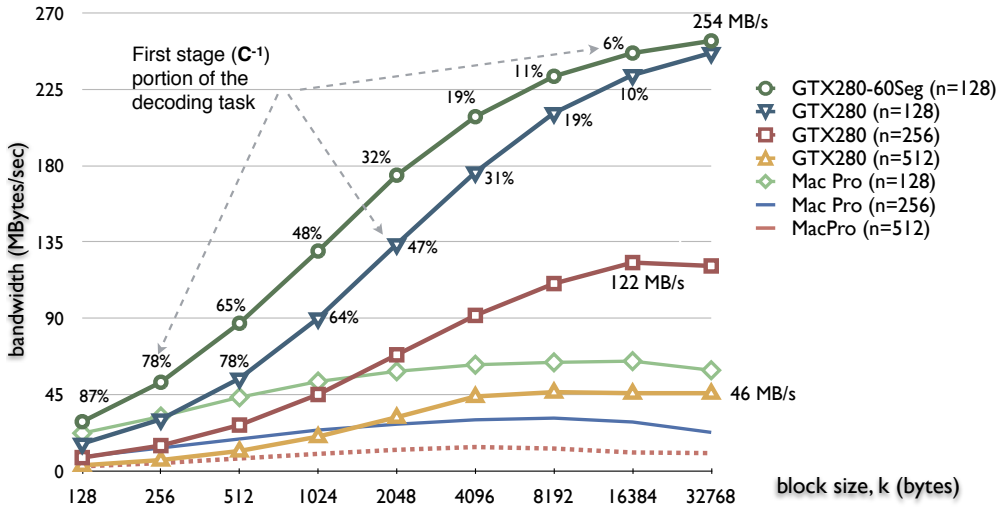


Figure 4.9: Parallel multi-segment decoding on GTX 280 and Mac Pro.

gate $[\mathbf{C}|\mathbf{x}]$. For example, at $(n = 128,\ k = 4096)$, 1056 thread was required, far beyond the limits of a single thread block. Using multiple thread blocks for a segment is not an option either, as it will cause synchronization problems we discussed in Sec. 4.1.2. Further, each thread would have to work on multiple columns, leading to less efficient code and many load and stores to memory.

As a solution, we propose to calculate $\mathbf{C}^{-1}$ first by doing Gauss-Jordan elimination for the aggregate matrix $[\mathbf{C}|\mathbf{I}]$. Then a regular multiplication in Galois Field, similar to the encoding process of Eq. 2.1, will restore the original segment. Although the GPU will not be fully used in the first stage of such a decoding process (due to the small coefficients matrix and the serial nature of row operations for matrix inversion), it will be fully utilized in the multiplication stage because of its high degree of parallelism.

After implementing the new scheme in CUDA, we have also developed a parallel segment decoding scheme for the CPU. For our 8-core Mac Pro system, we operate on 8 segments in parallel at a time, with each segment being processed by a CPU thread. The performance comparison is given in Fig. 4.9. As it is clear from the graph, GTX

280 outperforms the Mac Pro for all configurations with block sizes more than 256 bytes by a ratio between 1.3 and 5.3. By comparing Fig. 4.9 against Fig. 4.4, we notice that GPU-based multi-segment decoding achieves far better gains than the CPU-based multi-segment decoding when they are compared against their single-segment results. At $(n = 128, k = 16384)$, for example, the GTX 280 gains by a factor of 3.6, while the Mac Pro only gains by a factor of 1.3. As a result, multi-segment decoding should be the preferred scheme whenever the application scenario allows.

Another disadvantage of CPU-based decoding can be observed when the block size increases. The Mac Pro's decoding bandwidth starts dropping at block sizes of 8 KB for $n = 512$, at 16 KB for $n = 256$, and at 32 KB for $n = 128$. This is due to the fact that the data set increases substantially in the multi-segment scheme. The overall coded blocks being decoded become so large (4 MB per segment and 32 MB for the 8 active segments) that data accesses become memory-bound (the total Level 2 cache of all 8 cores is 24 MB).

Finally, our GPU-based multi-segment decoding can benefit from issuing more than one segment to each SM, *i.e.*, operating on 60 segments instead of 30 in parallel. This is due to the increased GPU utilization in the first stage of decoding, *i.e.*, calculation of $\mathbf{C}^{-1}$. By assigning two matrix inversions from separate segments to each SM, now two matrix inversions can proceed in parallel, improving the GPU utilization in the first stage (utilization in the second stage does not change much as forward multiplication is already a highly parallel task). The result is shown by the `GTX280-60Seg-n128` graph in Fig. 4.9 for $n = 128$, which clearly defeats the decoding performance of 30 segments, by up to a factor of 1.4. The graph is annotated by the first stage's workload share in the overall decoding task. It is obviously reduced compared to the 30-segment case, reflecting better utilization (for example from 64% to 48% at $k = 1024$). As the block size increases, the workload ratio of the first stage

decreases and the overall decoding rate gets closer to its encoding counterpart.

At this point, GPU-based decoding defeats the Mac Pro's decoding bandwidth across the board by a factor of 1.3 to 4.2. The advantage over single-segment GPU-based decoding in Fig. 4.6 is between a factor of 2.7 and 48.8. Higher gains are achieved at smaller block sizes, where single-segment GPU decoding did not perform very well. In more practical block sizes, 1024 bytes and more, the decoding gain is between 2.7 and 27.6.

4.2.3 Revisiting CPU-based encoding

For generating encoded blocks on a multi-core system, our CPU-based scheme in Chapter 2 and Chapter 3 partitioned the encoding task of "each coded block" among different threads, one thread per core, running in parallel. This ensured that a new encoded block is generated as fast as possible to achieve a maximum degree of parallelism at the coded block level. However, in a streaming server that generates a large number of encoded blocks for its downstream nodes (as in the GPU-based encoding scheme described in Sec. 4.2.1), the goal is to generate many coded blocks fast, buffer them and serve the downstream from the buffer over a longer period. In this case, each CPU's thread can encode fully coded blocks rather than partial blocks.

Fig. 4.10 compares the CPU-based encoding performance of the new full-block encode scheme against the original scheme, showing much better performance for small block sizes, apparently due to better performance of the memory prefetcher loading a longer sequence of data, *i.e.,* full blocks. However, both schemes essentially achieve the same encoding rate as the block size grows. Of course, the new scheme is the preferred one for a CPU-based streaming server, as it achieves a more consistent encoding bandwidth across a range of block sizes.

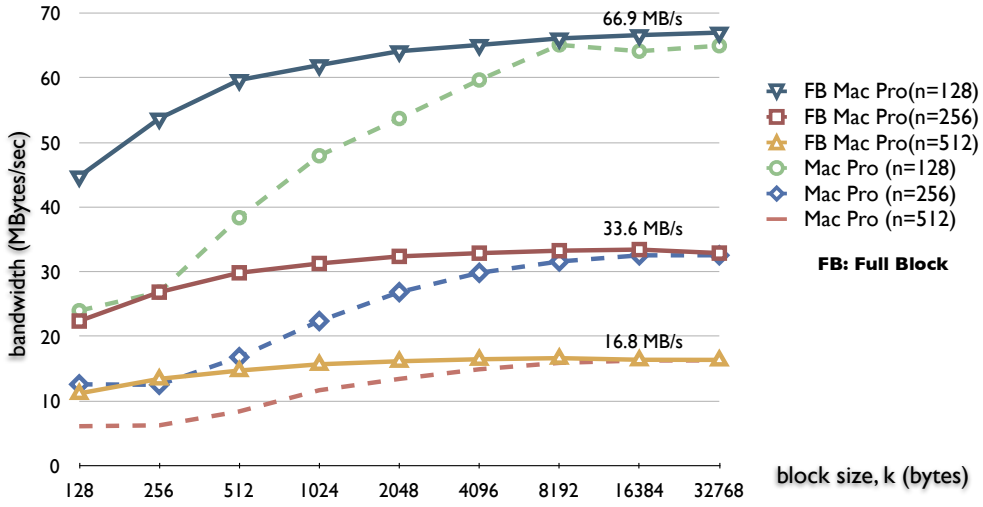


Figure 4.10: Parallel CPU-based encoding: full-block encoding vs. partitioned-block encoding.

Note that the new partitioning scheme essentially has the same overall computational complexity as the original one. The difference is in the usage scenario and system deployment. The original rationale behind partitioning the encoding process of each block in Chapter 2 was: (1) a new coded block should be *generated* as soon possible *on demand*; and (2) better caching performance is to be achieved by partitioning the data set among CPU cores. However, a more powerful server like our 8-core Mac Pro enjoys a much larger aggregate cache. In a streaming server deployment, the coded blocks can be generated in advance and buffered. This replaces *on-demand generation* in the original scheme with *on-demand delivery*.

4.2.4 Miscellaneous improvements

We now briefly review a number of miscellaneous issues we explored to further improve the coding performance.

Network encoding with both GPU and CPU

Due to the high degree of parallelism in the network encoding process, encoding can be employed by GPU and CPU in parallel, achieving encoding rates in proximity to the sum of the individual bandwidths on the GPU and the CPU. For example, by employing the optimized table-based encoding of Sec. 4.2.1 on the GTX 280 and the improved CPU-based encoding of Sec. 4.2.3 on the Mac Pro, encoding rates in proximity to the sum of the individual bandwidths can be achieved. This means encoding bandwidths close to 238 MB/s for $n = 128$, 119 MB/s for $n = 256$, and 60 MB/s for $n = 512$ blocks can be achieved in a combined GPU and CPU setup. However, the GTX 280 encoding rate is around 4.3 times that of a CPU-based solution on our 8-core Mac Pro server. This suggests an extra GTX 280 GPU, priced around US$$300$ at the time of this writing, leads to not only a much cleaner solution relieving CPU from heavy computation, but also a much better price/performance ratio.

Speeding up the decoding process with specialized instructions

As discussed in Sec. 4.1.2, the decoding process requires a search for the first non-zero coefficient of the current row (see **Step C** in Fig. 2.4). However, the search has to wait till all GPU threads working on the current coefficient row synchronize after **Step A** in Fig. 2.4 [64]. The search process can be accelerated if each thread reports its leading non-zero coefficient through an atomic minimum operation, `atomicMin`. GTX 280 is the first CUDA-enabled GPU that allows `atomicMin` operate on its shared memory, where we buffer the intermediate results.

Employing this optimization improves the decoding performance by around 0.6%. This is not surprising as **Step C** is not a major portion of the decoding workload after all.

Aggressive caching for decoding

Our decoding scheme aggressively caches various data structures on the shared memory to reduce accesses to the GPU memory. A more aggressive caching scheme would try to cache the entire coefficient matrix, because elements of $\mathbf{C}$ are the most frequent data referenced during the decoding process. However, having only 16 KB of on-chip shared memory per SM limits such scheme to coding configurations with $n = 128$ and less. With a number of creative techniques, we deployed full caching of the coefficients matrix beside caching of other essential data structures. The results show between 0.5% to 3.4% improvement in the decoding process. The smaller block sizes, *e.g.,* less than 1024, enjoy the most substantial gain, since the processing of the coefficient matrix consumes a larger share of their execution time.

Random number generation

GeForce 8800 GT used in Chapter 3 had support for only single-precision floating point operations. As described in Sec. 3.2.5, generation of random coefficients was also performed in the GPU. We ported our legacy CPU-based random generator to the GPU which was based on the pseudo random generator shipped in C Standard Library. In the GPU, however, integer division and modulo operations are costly operations. To perform the required scaling operation faster, we used a floating point division operation which obviously had to be done in single-precision. This forced us to also change our CPU-based random generator to use single-precision division so GPU-based encoding and CPU-based decoding can match each other.

GeForce GTX 280, however, added support for double-precision floating point. This means that our CPU-based random generator can revert back to its double-precision division. In particular, this restores backward compatibility with the previ-

ous distributions of our CPU-based network coding, *e.g.,* like the distribution in use by UUSee [12] which has thousands of active clients.

4.3 Summary

This chapter presented the best implementation of random network coding using GPUs reported in the literature. We presented a highly optimized table-based encoding scheme for the GPU that outperforms the loop-based algorithm on the same GPU by a factor of 2.2 across the board. The encoding advantage over a 8-core Mac Pro server is at least a factor of 4.3. Our multi-segment decoding scheme outperforms its 8-core Mac Pro counterpart by a factor between 1.3 and 4.2, significantly closing the gap with the encoding performance especially at large block sizes.

With a encoding rate of 294 MB/s at 128 blocks, more than 3000 downstream peers can be served at a streaming rate of 768 Kbps. Such a computation bandwidth can easily saturate two Gigabit Ethernet interfaces, making the GPUs the prime choice for streaming servers. With a much better memory performance than CPUs, a rapidly increasing number of cores, and much better performance/price ratio, GPUs are able to bring high-performance network coding to real-world applications.

Chapter 5

Random Network Coding on the iPhone

Pedersen *et al.* [60] 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 [43]. However, random network coding is able to offer substantially more flexibility, allowing coding over symbols and the use of multiple paths. On the other hand, 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 [34]. 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 chapter 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: the iPhone SDK published by Apple Inc. (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 chapter 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 chapter, 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 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, in both CPU and GPU fronts, are expected to offer a better point of tradeoffs and a smaller footprint incurred by network coding.

The remainder of this chapter is organized as follows. Sec. 5.1 presents our experiences implementing random network coding on the ARM architecture core, which the iPhone platform uses. Sec. 5.2 evaluates our implementation on both the iPhone 3G and the second-generation iPod Touch. Sec. 5.3 summarizes the chapter.

5.1 Random Network Coding on the iPhone Platform

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 chapter 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. 2.1, 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 work in Chapter 2, 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 mul-

ticore 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.

5.1.1 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.

5.1.2 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 [15]. 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 [16]. 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 [17].

We now propose a scheme, inspired by our work in Chapter 3, 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. 5.1.

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

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.

5.1.3 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.

5.1.4 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. 5.1. By examining the ARM assembly generated by the

```
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 5.2: Hand-tuned loop-based multiplication for ARMv6.

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. 5.2. 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 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.

5.2 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.

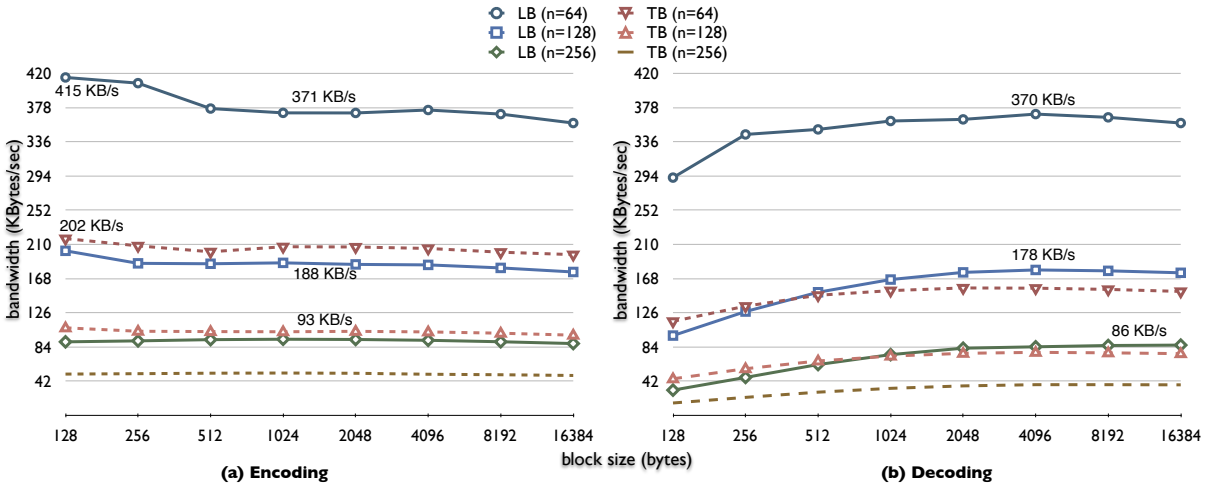


Figure 5.3: Coding bandwidth of loop-based (LB) and table-based (TB) for network **(a)** encoding; and **(b)** decoding processes on the 2nd generation iPod Touch.

5.2.1 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. 5.3. 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. 5.3(a) shows that encoding achieves its peak performance across almost all coding settings. It is not a surprise that Fig. 5.3(b) shows a lower decoding performance, since decoding needs to perform an $\mathrm{O}(n^3)$ 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.

These results have confirmed that the encoding performance of our loop-based implementation is mainly constrained by the computation limits of the ARMv6 core. The number of executed instructions to achieve an encoding bandwidth of 415 KB/s at $(n = 64, k = 128)$ can be estimated through the following first-order calculation:

$$\begin{aligned}
\text{inst. rate} &= \text{Computed words} \cdot \text{inst}_{\text{coded}-\text{word}} \\
&= (\text{BW}/4) \cdot (\text{n} \cdot (\text{inst}_{\text{pre}-\text{post}-\text{loop}} + \text{inst}_{\text{GF}-\text{multiply}})) \\
&= (\text{BW}/4) \cdot (\text{n} \cdot (7 + 6 + (\text{loop}_{\text{avg}} \cdot \text{inst}_{\text{iter}}))) \\
&= (415 \cdot 1024/4) \cdot (64 \cdot (13 + (7.02 \cdot 8))) \\
&= 470.243 \text{ million instructions}
\end{aligned}$$

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) as shown above. 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.

5.2.2 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. 5.4 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.

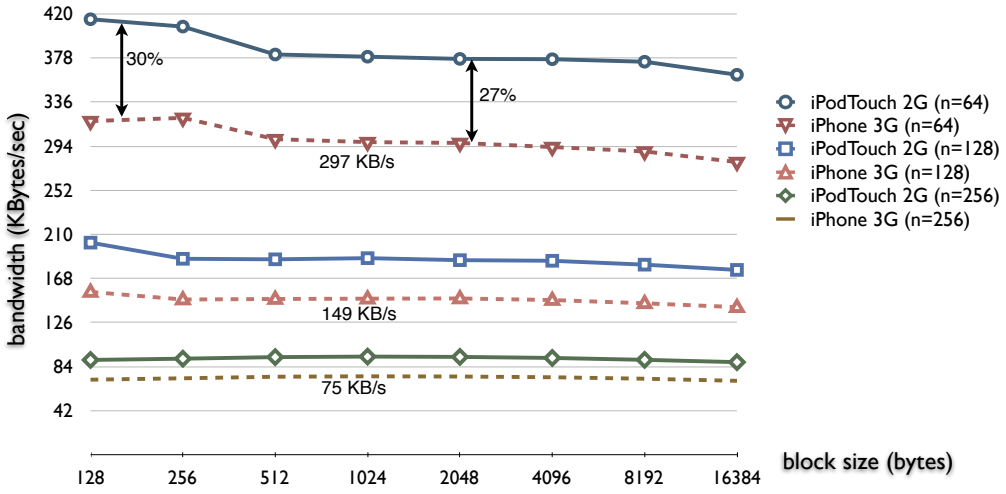


Figure 5.4: Coding bandwidth of loop-based network encoding: iPhone 3G vs. 2nd generation iPod Touch.

5.2.3 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.

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 5.1 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

Table 5.1: YouTube contents & steaming experiments

Clip name	Size	Bitrate	Length (min:sec)	CPU usage	Ingress rate
`History of the Internet`	320x180	300 Kbps (37.5 KB/s)	8:10	12.9%	61 KB/s
`Validation`	320x240	400 Kbps (50 KB/s)	16:23	14.1%	101 KB/s
`Obama's Inauguration`	640x360	620 Kbps (77.5 KB/s)	21:22	14.9%	112 KB/s

`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 5.1, 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 streaming 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. The ingress streaming rates are also different across our test video clips.

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 [54] 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 5.2 shows the average CPU usage in our experiments, with network coding

Table 5.2: CPU usage of network coded streaming.

	$(n = 128, k = 2048)$			
Clip name	Estimate	$d = 1$	$d = 2$	$d = 4$	$d = 0$
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	Estimate	$d = 1$	$d = 2$	$d = 4$	$d = 0$
Hist. of the Internet	21%	22%	18%	17%	12%
Validation	27%	28%	25%	22%	16%
Obama's Inauguration	43%	43%	38%	35%	22%

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 $\text{CPU usage}_{\text{Est}} = R/\text{BW}_{\text{enc}} + R/\text{BW}_{\text{dec}}$, where BW reflects the coding bandwidth from Fig. 5.3 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 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

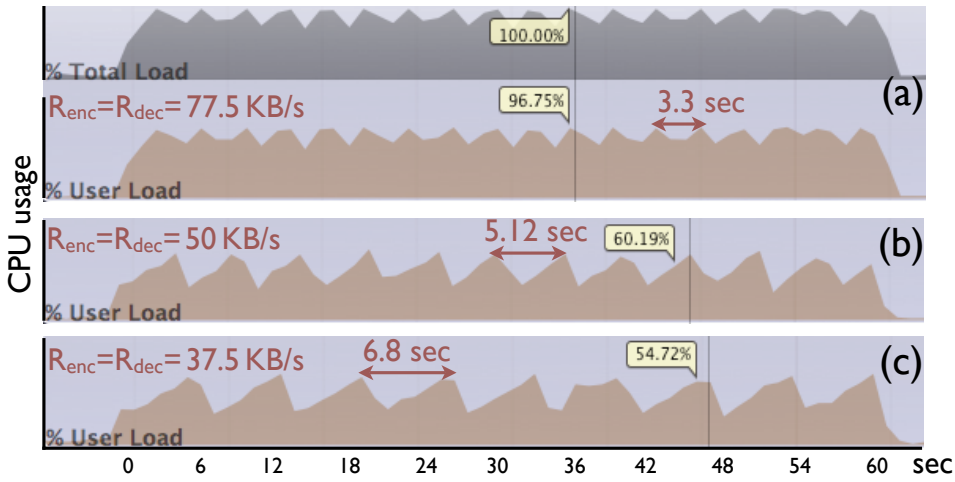


Figure 5.5: Instantaneous CPU usage for $d = 1$.

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. 5.5 shows the CPU usage over the course of our $(n = 128, k = 2048)$ experiments, at three different video bit rates. Fig. 5.5(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) [64], causing such a sawtooth phenomenon.

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

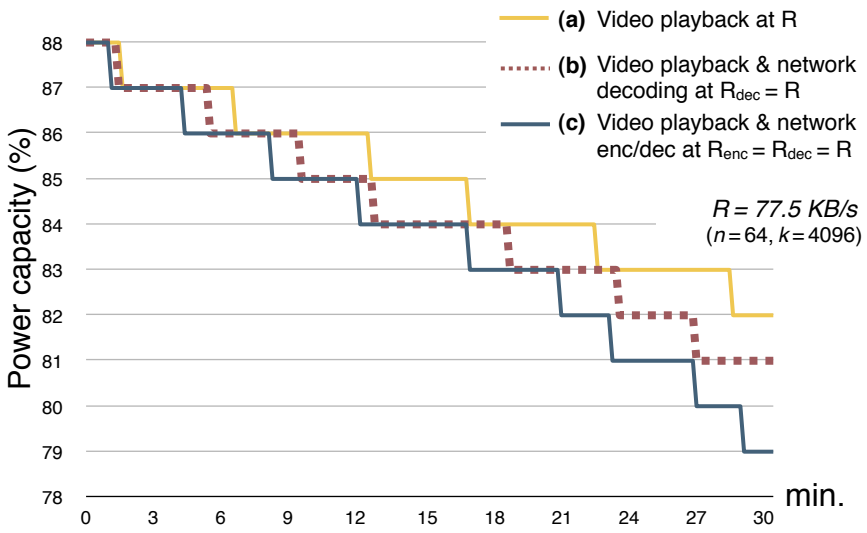


Figure 5.6: Energy consumption on the iPhone 3G.

power metering utility is running as a console application launched through a remote session into a "jailbroken" iPhone device 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. 5.6. The results reflect that about 33% of the reduction in energy reserves 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.

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.

5.3 Summary

This chapter presented 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. For example, the upcoming generation of iPhone may incorporate the more recent POWERVR SGX GPU, which carries programmable shaders, and may potentially assist to reduce the computational footprint of random network coding. More importantly, faster and more advanced processors, such as the ARMv7 core and the ARM Cortex-A9 multicore architecture, would further assist to improve the performance of network coding[8]. As we see it, as mobile hardware platforms advance in the future, the tradeoffs

[8]The performance of network coding on ARMv7 core, employed in the new iPhone 3GS, will be studied in Sec. 7.3.4.

will be more aligned in favor of using network coding on mobile devices for real-world multimedia streaming applications.

Chapter 6

Blizzard: A Scalable Emulation Framework

Peer-to-peer (P2P) protocols have received extensive attention in recent years. This reflects a shift of the algorithmic intelligence and bandwidth burden from dedicated servers to the end systems (*i.e.,* peers) at the edge of the Internet [70]. Higher performance, reliability and scalability of P2P protocols for many applications has resulted in development of new P2P protocols in both academia, *e.g.,* [22, 74, 34], and industry, *e.g.,* BitTorrent [10].

Since P2P protocols involve hundreds and even thousands of nodes, there exist many difficulties in their testing, troubleshooting, and performance evaluations which forces many researchers to limit their evaluation to simulation.

As an alternative to simulation, an *emulation framework* targets a controlled environment of a cluster of high-performance servers as a deployment alternative for evaluation of P2P protocols. The *iQua* lab in University of Toronto [9] has worked on a number of such emulation frameworks over the years. The first in this series was *iOverlay* which included "application layer multi-threaded message switching engines

and virtualized distributed nodes" at its core [48].

However, due to its internal design, iOverlay does not scale well when number of nodes emulated on a physical system increases or computational expensive P2P streaming protocols are deployed. *Crystal* has tried to address these issues by improving the scalability, and adding extra services to better facilitate rapid deployment of P2P streaming protocols, in particular [70].

As the Crystal design has become more mature, we have realized its potential not only as an emulation framework but also as a framework for more realistic and high-performance *network applications*. Though it can already achieve good level of scalability, emulating up to 300 nodes on a single server for a simple relay protocol [70], higher level of scalability and more efficient design are not out of reach. *Blizzard*, the third generation emulation framework in the *iQua* lab, builds on the strength of Crystal, specifically targets scalability and achieves an improvement "twice" and more over Crystal in most cases. It employs the relatively new *epoll event notification* scheme [1] in Linux as our primary tool towards better scalability. At the same time, we seek further speed improvements especially in the area of threading performance by aggressively avoiding the synchronization locks.

Although our main goal is to build a more scalable and efficient emulation framework, we have an eye on making Blizzard also usable for more general network applications such as high-performance streaming server. Our later experiments in Chapter 8 will employ both of these features.

The remainder of this chapter is organized as follows. We first overview the current design of Crystal in Sec. 6.1 as Blizzard still employs some of its components and designs. The basics of `epoll` interface and its potential advantages are overviewed in Sec. 6.2. Our new design for the architecture of Blizzard core is presented in Sec. 6.3. Section 6.4 compares the performance of experimental results of Blizzard and Crys-

tal. A cross-platform implementation of Blizzard is discussed in Sec. 6.5. Finally, we conclude the chapter in Sec. 6.6.

6.1 Crystal Overview

As Blizzard inherits many design decisions from Crystal, a rather detailed overview of Crystal is given here.

6.1.1 Crystal architecture

As a highly condensed peer-to-peer emulation framework, Crystal features the following highlights in 10279 LOC (lines of code including comments) [70]:

- Crystal implements the set of common elements that any peer-to-peer protocol researcher would have to implement, including multi-threading, message switching, timed and periodic event scheduling, network socket programming, and exception handling.
- Crystal is custom-tailored for server clusters, and includes facilities to automate the deployment, troubleshooting, and data collection of peer-to-peer protocols.
- Crystal focuses on the lack of reality in simulation and the lack of DSL peers and controllability in real-world testing, and *emulates* DSL-like peer upload and download capacities, end-to-end delays, as well as peer arrivals and departures.

The core of Crystal consists of four components [70]: (1) The *network*. The network provides low-level network I/O services and handles basic sockets-level tasks related to new incoming connections, broken connections, as well as the actual communication (send and receive operations). (2) The *engine*. The implementation of the engine

supports scalable emulation of bandwidth, delay, timed or periodic events, message switching, TCP and UDP traffic, as well as exception handling. (3) The *algorithm*. In Crystal, the new P2P protocol to be developed and evaluated is referred to as the *algorithm*. To minimize development time, Crystal provides a carefully designed and well-defined API between the engine and the algorithm, for the purpose of processing incoming messages, producing outgoing ones, as well as handling peer arrival and departure events. (4) The *libraries*. The libraries in Crystal include common programming elements and components to avoid "reinventing the wheel," and further reduce the time required to develop a new P2P protocol.

Each instance of Crystal, running on a physical node, emulates one or more peers each uniquely identified by an IP address and port number. A P2P application executes on a collection of such peers, running by one or several instance of Crystal on a single or across a cluster of physical node(s). The core of the P2P application is in the algorithm. To feed the algorithm, Crystal allows multiple TCP connections or UDP flows from multiple upstream peers. To transmit the messages sent by the algorithm, multiple TCP connections or UDP flows to their corresponding downstream peers may be established. It is important to note that the implementation of the Crystal engine on each peer consists of only two threads. First, the *network* thread is in charge of maintaining all the incoming and outgoing TCP connections or UDP flows, their corresponding FIFO queues, data sources, and has the capability of managing multiple data communication sessions as well. All incoming and outgoing network traffic are monitored by a single `select()` call with a specific timeout value. The timeout value of the `select()` call is tuned dynamically on-the-fly, and is critical to the scalable implementation of bandwidth emulation. Second, the *engine* thread processes head-of-line messages from incoming connections, and sends freshly produced messages to outgoing connections. The engine thread also interacts with the algorithm to

process incoming messages, and implements timed events. The engine thread forms natural producer-consumer relationships with the network thread. Figure 6.1 illustrates network and engine components and their interactions.

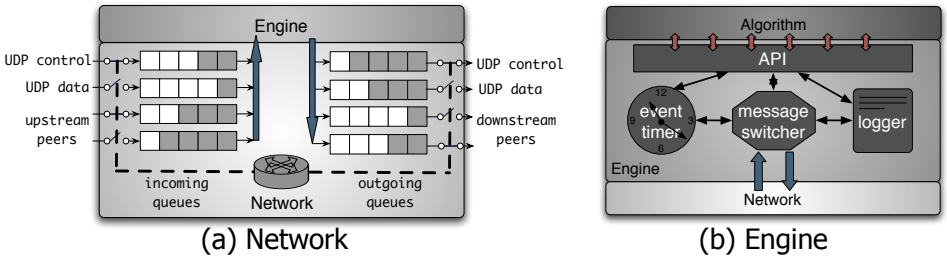


Figure 6.1: Crystal's Network and Engine components (from [70]).

There are no limitations in the Crystal implementation that preclude running more than one peer on each physical cluster node; in fact, such a way of running emulated peers is encouraged. Emulated peers do not need to periodically contact a central server for logistics or authentication. All logs are written to local file systems, then collected and analyzed by scripts after the experiment finishes. By minimizing the footprint of each emulated peer, Crystal can run hundreds of peers in one cluster node. Crystal is strived to be platform neutral and is currently portable across major UNIX variants (Linux, FreeBSD and Mac OS X), as well as Microsoft Windows.

6.1.2 Messaging in Crystal

In Crystal, peers communicate with each other through exchanging messages. Each message consists of a header (information like message type, sequence number, and payload size) and an optional payload. Each message goes through different components of Crystal which will be discussed more in the following sections.

Crystal has taken extra measures to improve performance of message handling. *First*, it guarantees that there exists no excessive data copying when messages are

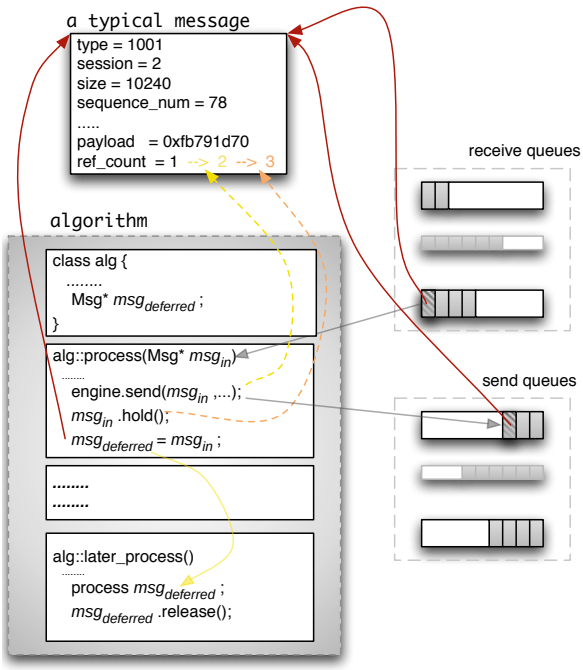


Figure 6.2: Reference counting and creating multiple copies of a message reference.

being received in Crystal from sockets. *Second*, Crystal implements extensive reference counting mechanisms to guarantee that a message is only *referenced*, and never copied, throughout its lifetime. Figure 6.2 shows an example scenario where the algorithm relays an incoming message to another peer node by putting it in a send queue while still keeping a message reference for a lengthy process at a later time.

Since a message can be referenced by multiple threads concurrently, additional complexities may arise with respect to synchronized access to messages. One obvious example is the need for thread-safe updates of the reference count in each message. Another example is the possibility of race conditions when a message is scheduled for sending by one queue, while it is actually being sent in another queue at the same time. *Finally*, certain P2P applications need to take advantage of hardware acceleration instructions such as SSE (for x86) and Altivec (for PowerPC). These instructions either require or prefer (for performance reasons) 16-byte aligned memory operands,

depending on the hardware platform and individual instructions. To support such hardware acceleration, the implementation of messages in Crystal supports memory-aligned buffer allocation, which can be optionally activated.

6.1.3 The Crystal core

The design of the Crystal engine, or its core, is based on a previous work on *iOverlay* [48], a lightweight middleware framework for developing overlay applications over the Internet. The design of iOverlay had employed blocking socket operations with a "new thread" for each connection, which corresponds to a "thread-per-client" concurrency model. It was observed that such scheme leads to an increasing number of threads that was proportional to the number of active connections. As the number of active peers increased, the overhead of thread context switching increased as well.

In contrast, Crystal requires only two threads for each emulated peer. Similar to event-driven designs of [59] and [71], this approach buys in the "extra complexity" of event-driven implementation in order to achieve higher performance. With less number of threads, the threading overhead decreases, such that Crystal will be more scalable than alternative VMM-based solutions to emulate a large number of emulated peers on a single physical cluster node.

Crystal employs two threads for each peer, the *network thread* and the *engine thread*. While the engine thread routes incoming messages to the algorithm for further processing, the network thread is free to respond to network-level events. This leads to very fast responses to socket events, and subsequently makes emulating high-throughput peers possible. The network thread is responsible for handling queues of new incoming and outgoing messages, sockets becoming ready, as well as the arrivals and departures of connections. The engine thread is mainly responsible for forward-

ing incoming messages to the algorithm (which implements protocol-specific logic), as well as handling all periodic or timed events that are registered with the engine.

```
while peer is alive
    sleep till at least one queue becomes eligible for send or receive
    for every socket sock_i of a ready incoming queue recv_q_i
        add sock_i to active_socket_list_recv
    for every socket sock_i of a ready outgoing queue send_q_i
        add sock_i to active_socket_list_send
    select (active_socket_list_recv , active_socket_list_send , timeout)
    for every ready socket sock_i from active_socket_list_recv
        receive the incoming message msg_in from sock_i
        if msg_in is fully received
            add msg_in to the incoming queue recv_q_i
    for every ready socket sock_i from active_socket_list_send
        send out the head-of-line message msg_out of send_q_i over sock_i
        if msg_out is fully sent out
            remove msg_out from the outgoing queue send_q_i
    update network statistics
```

Figure 6.3: The network thread: in a nutshell.

The network thread

The network thread provides low-level network I/O services for Crystal. It handles basic sockets-level tasks related to new incoming TCP connections, exception handling related to broken TCP connections, as well as the actual communication (send and receive operations) for all active connections. The network thread supports both connection-oriented stream sockets and connectionless datagram sockets. As shown in Fig. 6.1, each TCP connection to an upstream or downstream peer corresponds to a queue in the network thread, implemented as a circular buffer of messages. UDP traffic has its own dedicated incoming and outgoing queues.

As shown in Fig. 6.3, the main body of the network thread consists of a loop running for its lifetime. Each iteration first prepares a list of all *active sockets* for the current

iteration. This list includes the server sockets, sockets from all *ready outgoing queues* and sockets from all *ready incoming queues*. The "readiness" of a queue is computed based on the per-link bandwidth limit of the corresponding TCP connection, and on the per-peer bandwidth limits as well. After the `select()` call is activated with a brief timeout, it will release as a result of one or more sockets, called *ready sockets*, from the active list becoming ready for I/O operations, or when the prescribed timeout expires. The network thread then proceeds to process all sockets that have become ready from the `select()` call. All TCP I/O operations are non-blocking, and the send or receive operation of a message in a queue does not necessarily finish in one iteration of the network loop.

The engine thread

The engine thread employs the network thread as a "producer" in a typical consumer-producer relationship with respect to messages. Its main responsibility is the dispatch of incoming messages, which are retrieved from the incoming queues, and routed to the appropriate message handlers. Some messages are handled by the engine itself, the remainder is destined for the algorithm. The engine also provides support for timed and periodic events for the algorithm, and it calls the registered "callback" functions provided by the algorithm when registered events expire. Fig. 6.4 describes the skeleton of tasks performed by the engine thread.

The delivery of all incoming messages destined for the algorithm and timer-based event notifications are serialized by the engine thread. This relieves the algorithm developer from being concerned about synchronizing its implementation with other threads. This advantage further simplifies the development of the P2P algorithm, since its implementation does not need to be concerned with *thread safety*. In each iteration of the engine thread, at most one incoming message from each incoming queue is processed (completely or partially), and subsequently removed after being

completely processed.

```
while peer is alive
    sleep till at least one incoming queue becomes non-empty or
          the next event has expired
    for every non-empty incoming queue recv_q_i
        select the head-of-line message msg_in from recv_q_i
        if msg_in is destined for the engine
            process msg_in in the engine
        else
            pass msg_in to the algorithm
        if msg_in is fully processed
            remove msg_in from recv_q_i
    if a pending timed event event_j has expired
       call event_j's registered callback function
```

Figure 6.4: The engine thread: in a nutshell.

The algorithm thread

The algorithm is usually implemented as an instance of an application-specific C++ class, within the engine thread. As the algorithm does not usually have its own thread, when it is provided with an incoming message to process or a timed event notification, it should not perform lengthy operations. Otherwise, the engine thread would be blocked, and will not be able to route other incoming messages or issue pending timer events during this period. If the algorithm needs to perform lengthy message processing or its logic requires its own thread, it can launch its own private thread(s). Similar to any other multi-threaded application with potential access to a set of shared data by multiple threads, each algorithm thread has to use proper synchronization construct to prevent potential race conditions.

6.1.4 Runtime interactions among Crystal components

The main application thread creates engine instances — one for each emulated peer — and launches their associated threads. Each engine thread has a corresponding network thread, which is launched during its initialization. From this point on, the network and engine threads communicate through a set of incoming and outgoing message queues which are protected against concurrent access by both threads.

Beside the usual `enqueue` and `dequeue` interfaces for queue access, the `peek` function allows the retrieval of head-of-line messages without actually removing them from the corresponding queues. This helps in the cases that a message cannot be fully processed in one step. For example, it may occur that only a portion of an outgoing TCP message is sent out by a socket send operation and the rest has to be sent by further iterations of the network loop. Another application of the `peek` function is by the engine thread when it routes a message to the algorithm. Since the algorithm can opt to not finish processing the message (*e.g.*, due to a lengthy operation) by returning *hold*, the engine thread only removes the head-of-line message after the algorithm has indicated that the message is no longer needed. When a message is not completely processed, it will be sent to the algorithm again in the next iteration of the engine loop.

Figure 6.5 shows an example of inter-thread communication, and showcases how the engine and the algorithm may interact with each other. In this example, an incoming message destined for the algorithm is inspected from the incoming UDP control message queue by the engine thread and routed to the algorithm instance. The algorithm processes the message, and could launch its own private thread, in addition to creating and sending a response message to the upstream peer through TCP. To schedule the message for sending, the algorithm calls the `send()` function of the engine

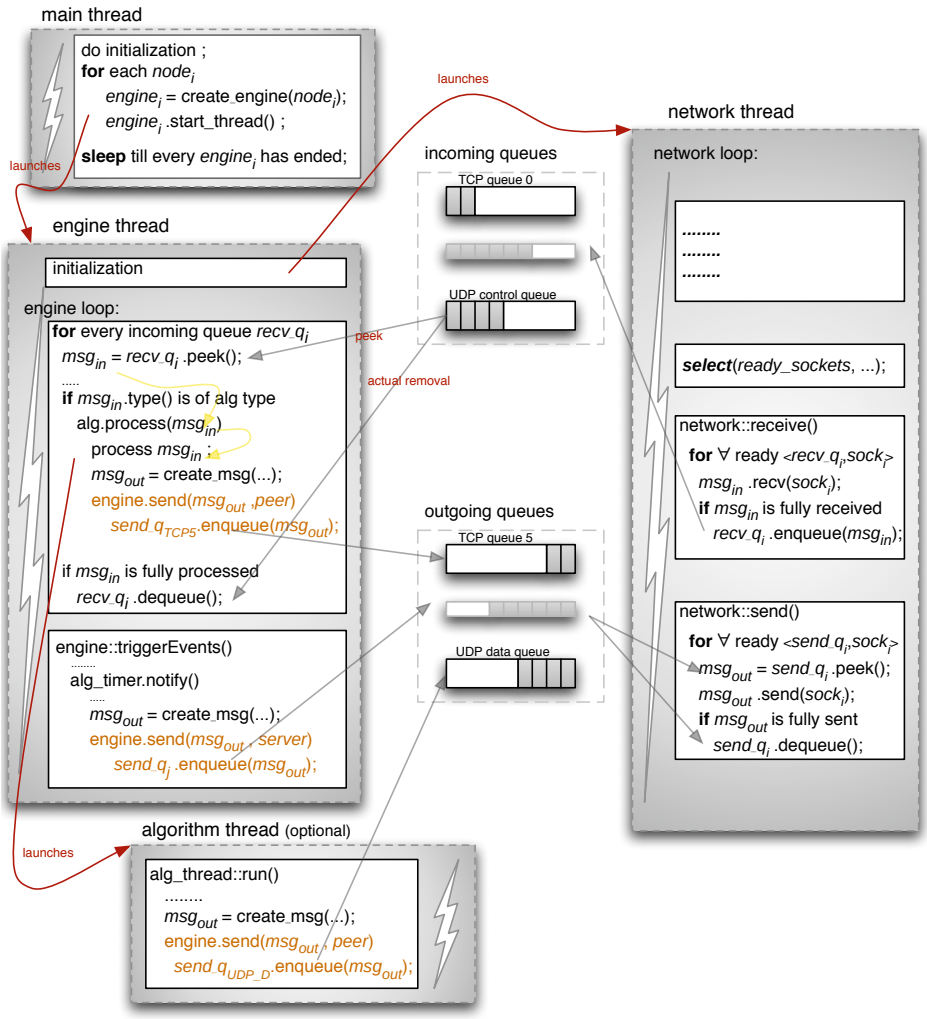


Figure 6.5: Runtime interactions within Crystal: an example.

instance, which subsequently selects the appropriate outgoing queue and appends the message to the end of the queue. Since the algorithm has indicated that msg_{in} is completely processed, engine would then remove the message from the queue.

6.2 `epoll` Overview

Crystal has adopted the traditional I/O operations scheme by using the `select` system call. This scheme's pros and cons are discussed first. Then the `epoll` interface of Linux is introduced to familiarize the reader with the actual `epoll`-based design of Blizzard to be presented in Sec. 6.3.

6.2.1 `select` API for networking I/O

A socket is one end-point of a two-way communication link between two programs running on the network [68]. In Unix and Linux, and unlike Microsoft Windows, sockets are identified by file descriptors and treated the same as regular files. Many Unix/Linux system APIs work transparently for any file descriptor regardless of the type of descriptor, whether it represents a socket or an actual physical file. Traditionally, network applications have heavily used the `select` system call to learn about the status of network sockets. This is especially true as most applications use synchronous I/O for networking. Asynchronous networking I/O has been traditionally considered to be more difficult to employ as it generally requires more states to track and events to respond to. As the need for high-performance networking has grown over the years, *e.g.,* in web-servers, Linux has gradually added more support for highly efficient asynchronous I/O.

Format of the `select` system call is shown in Figure 6.6 [4]. The API accepts a separate array of file descriptors, `fds`, for each set of read, write, and exception monitoring operations. It also accepts an optional timeout. The `select` system call will return whenever one or more sockets from the list of watched sockets receives an exception, or when read or write operations become possible for one or more sockets, or when the call times out. The application scans each array of file descriptors to learn

about the readiness of individual sockets. Then it can issue read or write calls on each ready socket. Since it already knows which sockets are ready for I/O, synchronous I/O calls can efficiently perform network operations after the `select` call.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Figure 6.6: Format of the *select* system call.

This simple scheme works well with small numbers of sockets. One `select` call tells us about the status of several sockets. It also allows efficient change of list of *under watch sockets* at any time. As described in Sec. 6.1.3, Crystal is not interested in the status of all existing sockets at all time. Sockets which their associated receive queues are full or their associated send queues are empty are not passed to the `select` call. This is also the case for sockets that the governing bandwidth rules for their associated links do not allow input or output operations at the current time and has to wait for certain period to be reactivated.

However, the scheme suffers from several deficiencies. *First*, it is not efficiently scalable to very large number of sockets. Supporting large number of real or emulated nodes commonly used in P2P applications is an important goal of Crystal. Preparation of array of file descriptors before making the `select` call and also scanning through them after return of the `select` call is of $O(s)$ order for s sockets. Since the calls to `select` and then the actual I/O operations are performed repeatedly within a loop, similar to what shown in Fig. 6.3, often the same set of sockets of interest are added to the `select` call across several iterations. This reflects the repeated waste happening at preparation of file descriptors list while "only a few" usually end up as ready sockets.

Second, a Crystal-based high-performance application that requires sending out many messages every second would suffer from this scheme. Unlike the *receive sockets*,

the sockets used for receiving data, the *send sockets* are added to the `select` call only when there is a pending data to be sent out on these sockets. Otherwise, the `select` call returns right away as the send sockets quickly become ready for further send. This happens as soon as their associated system buffers kept by the TCP/IP network stack becomes free after the data leaves the system.

By adding the send sockets to the `select` call only when there is an actual data for output, we prevent Crystal's *network thread* from becoming a busy loop and wasting CPU cycles on executing code unnecessarily. But this approach has its own drawbacks. When there is no network activity for a while, the `select` call times out. Crystal uses a timeout value of $100\ usec$ which is supposed to result in a reasonable rate for reexamining and updating the set of active sockets. But Linux is not a real-time operating system and does not provide high granularity timing operations. A timed-out `select` call will not return for a few milliseconds instead of the planned $100\ usec$. While a *select* call takes a few millisecond to time out, a new piece of data which has become available by the engine or algorithm thread has to wait for the next iteration of the network thread so it can finally adds its socket to the active sockets list. This issue is because there is no way to break the current `select` call.

As a result, a high-performance application which generates thousands of output messages per second, especially at regular intervals like a streaming server, will suffer from this blackout period and end up with a smaller sending rate. So far, we have used a work around for this type of applications by always adding the send sockets regardless of availability of data for send. This has remedied the problem to some degree but has caused the network thread to become a busy loop and waste CPU cycles.

6.2.2 `epoll` interface

The `poll` API [3] has a behavior quite similar to the `select` call. The main advantage of `poll` is that unlike the $FD\_SETSIZE$ limit of the `select` API, the number of sockets watched by `poll` does not have a hardcoded limit [44]. However, the `poll` API still suffers from the same $O(s)$ complexity deficiency as the `select` API. Later, a major improvement came in by introduction of `/dev/epoll` interface for Linux through a kernel patch but applications had to use the generic `ioctl` (I/O control) interface to use this feature. From Linux kernel 2.5.44, a dedicated system call was added for `epoll`.

The real merit for the `epoll` interface is that in many application scenarios similar to the cases in Crystal described earlier, the list of watched sockets does not change significantly from one `select` call to another. Most of the sockets of interest can be "added once" to the "list of watched sockets". Then, `epoll` provides a mechanism for "efficient notification" of the application about network status of the watched sockets. If an application scenario continuously changes its list of socket of interest, especially if the number of sockets are not huge, then it better suits to stick to the `select` scheme. For many other cases, the `epoll` interface is more beneficial with only few disadvantages. The most important drawback is its lack of portability to the mainstream Unix variants like Unix System V and Mac OS X which is based on BSD Unix. Also, depending on the actual application scenario, taking full advantage of the advanced features of the `epoll` interface might be challenging as explained in Sec. 6.3.

The `epoll` interface consists of several APIs [1] as shown in Figure 6.7. First, an application creates an instance of *epoll object* by calling `epoll_create` API. This object tracks a set of sockets for change of their network status. Each socket is individually added to the watch list along with the network event of interest (e.g., readi-

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events,
               int maxevents, int timeout);
```

Figure 6.7: Group of APIs for the `epoll` interface.

ness for read, write, connection failure,...) by calling `epoll_ctl` API. An instance of `epoll_event` structure, shown in Figure 6.8, specifies the socket file descriptor and type of the event. `epoll_ctl` is also used for removing a socket from the watch list. The real magic comes through `epoll_wait` API. It returns on availability of an event of interest or when the timeout expires. What differentiates `epoll_wait` from `select` is that the output parameter `events` reports back "only" the sockets that an event of interest has been detected on them. As a result, scan of the whole socket list is not required anymore which makes the cost of using `epoll` much lower than `select` when sockets of interest do not change often.

Another strength of `epoll` event distribution interface is its ability to behave both in *Edge Triggered (ET)* and *Level Triggered (LT)* modes [1]. For each socket added to the watch list by `epoll_ctl`, a notification mode has to be specified. When kernel detects a network event on a watched socket marked as level-triggered, all later calls to `epoll_wait` will return that socket again and again until an appropriate I/O operation changes back the status of that event. For example, a $EPOLLIN$ event, which indicates the socket has received incoming data currently sitting in a system buffer for retrieval, will be passed to all later `epoll_wait` calls unless a read operation is performed on the socket and all pending data is retrieved.

On the other hand, an edge-triggered event notification mode will report an

```
typedef union epoll_data {
   void *ptr;
   int fd;
   __uint32_t u32;
   __uint64_t u64;
} epoll_data_t;
struct epoll_event {
   __uint32_t events; /* Epoll events */
   epoll_data_t data; /* User data variable */
};
```

Figure 6.8: Format of `epoll_event` data structure.

$EPOLLIN$ event for the associated socket only for the `epoll_wait` call that is directly issued after the system detects an arrival event due to incoming data. (Note that no read operation has been performed on the socket yet.) The next $EPOLLIN$ notification will not come on the socket unless the system's internal buffer goes through another cycle of empty to non-empty. This means that a read operation has to remove first all the incoming data sitting in the system buffer. Then, a new $EPOLLIN$ event will be generated only after a new set of data is received. While this is a very useful operation mode, it brings on its own challenges too. Here we skip all the details and refer the reader to [1] for further details.

Each of the level-triggered and edge-triggered notification modes suits a different set of application scenarios. In Blizzard, we pick the edge-triggered mode as presented in the next section.

6.3 Blizzard Design with `epoll`

As briefly discussed in Sec. 6.2, the `epoll` interface can potentially help Crystal to improve its scalability and offer better support for high-performance applications. In this section, we expand the discussion and present Blizzard, our new design that uses `epoll` at its core.

6.3.1 How `epoll` fits in Blizzard

One could simply replace the `select`-based scheme in the network thread, shown in Fig. 6.3, with an `epoll`-based scheme that adds the sockets to a watch list, by frequent `epoll_ctl` calls, and then calls `epoll_wait` to inspect their status. Not only this approach does not take advantage of the full potentials of the `epoll` interface, but it also backfires because rebuilding the socket list at each iteration of the network loop requires more system calls than a single `select` call. Our real goal here is to take advantage of every feature of the `epoll` interface which can help us build a more efficient networking I/O in our emulation framework.

The rough idea is to add every socket only once to the watch list to minimize the number of `epoll_ctl` calls. Then we take advantage of the edge-triggered event notifications mode to be fully aware of the network status of every socket no matter if the other conditions of an actual I/O operation on the socket is met or not at the current time. Even if I/O operation on a socket is suspended till all the conditions are met, the maximum number of wasted notifications received during the socket suspension will not be more than one because we use the edge-triggered mode. After all, such notification will not be wasted anyway and will be recorded within Blizzard in the data structure associated with each socket. Now that the full network status of each socket is known, we attempt to schedule each I/O operation at the right time when all

the conditions of I/O operation are met.

The necessary conditions for a network I/O operation can be formalized as the following. For a send operation to be able to proceed on a socket, the following send conditions (SC) has to be valid:

- ***SC1*** The network state of the socket is ready for output, *i.e.*, the system output buffers for the corresponding socket is not full.
- ***SC2*** There is some pending messages in the Crystal's outgoing queue for the socket, *i.e.,* there is something to send out.
- ***SC3*** The emulated bandwidth constraint on the associated link and also the constraint on the overall node allows an outgoing message be sent out at the current time.

The conditions for a receive operation, receive conditions (RC), are the followings:

- ***RC1*** The network state of the socket is ready for input, *i.e.*, there exist some incoming data currently residing in the system buffer which is destined for the socket.
- ***RC2*** There is some empty slots in the Crystal's incoming message queue associated with the socket.
- ***RC3*** The emulated bandwidth constraint on the associated link and also the constraint on the overall peer bandwidth allows an incoming message be received at the current time.

Each instance of a Crystal engine consists of an engine thread and a network thread as discussed in Sec. 6.1.3. An instance of a Crystal-based application can encompass

multiple Crystal engines with each emulating a separate peer concurrently and independently from the other engines. Having two threads per peer results in a total of $1 + 2p$ threads for p peers, including the main application thread. However, all engine instances can ideally use a shared network thread instead of each having its own network thread. A network interface (including hardware, device driver and the kernel network stack) serializes the network operations at some point. Though some higher level processing of networking APIs can be executed in parallel threads, the lower level kernel-side processing will serialize at some point.

More importantly, having a single shared network thread will lower the total number of threads and should improve its scalability especially when hundreds of peers are emulated on a physical node. Even when multiple cores exist on a physical node, having a single dedicated network thread will allow other cores to service other threads, *e.g.,* engine threads and potential algorithm threads, more efficiently by decreasing the overhead of thread context switches. With a single shared network thread, the total number of threads decreases to $2 + p$ threads, almost half of the $1 + 2p$ threads in Crystal.

The sketch of such new scheme for network thread is shown in Figure 6.9. This network thread handles all sockets no matter which instance of the engine they belong to.

However, the above scheme would not work due to few issues. The main problem comes from the fact that executing `epoll_wait` call can not be asynchronously terminated. Let us assume the following scenario. There exists pending incoming data for $socket_j$, *i.e.,* condition *RC1* is met, and the incoming link does not have any emulated bandwidth constraint, *i.e.,* condition *RC3* is met too. But the Crystal's incoming queue does not have any empty slot left to accept the new data in form of one or more messages. As a result, the receive operation has to be delayed till the engine thread

```
while ( network thread is active ) {
      // check the network status of all sockets
   n = epoll_wait(pfd, events, maxevents, timeout);
      // for each network event i
   for ( i = 0; i < n ; i ++ ) {
      if ( all I/O conditions for events[i].data.fd socket is met ) {
         perform the socket I/O operation
      }
      else
         delay the I/O operation to a later time when all conditions are met
   }
}
```

Figure 6.9: Sketch of the initial idea for use of `epoll` interface in the network loop.

consumes a message from the incoming queue of $socket_j$, *i.e.,* till condition *RC2* becomes true. Now assume `epoll_wait` has blocked till a change in the network state is detected but no actual change of network state happens for some time. Since there is no way to signal the blocking `epoll_wait` call to return early, the engine thread has no way to signal the network thread to perform the receive operation.

Of course, the blocking `epoll_wait` will return after the expiry of its timeout. But relying on the timeout mechanism to schedule the I/O is not appealing for few reasons. *First*, the network thread still can not respond to the *RC2* and *SC2* conditions right away. *Second*, `epoll_wait` timeout granularity is even worse than the `select` call because the unit of timeout parameter is in seconds!

One could have used level-triggered network events instead of edge-triggered ones. Then `epoll_wait` would not block in this scenario since there exists a ready socket. But this approach suffers from excessive iterations of the network loop which

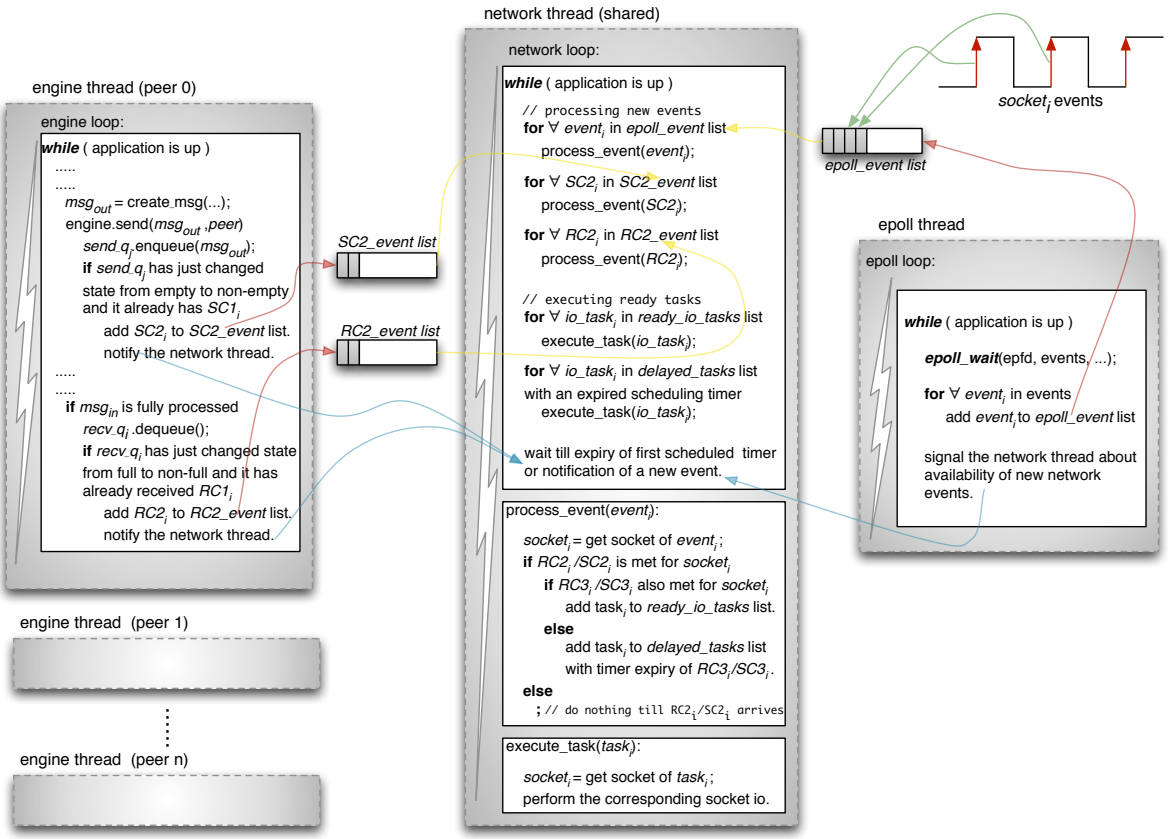


Figure 6.10: High-level view of the new `epoll` scheme.

will end up like a busy loop. One other possibility is to have the engine thread issue the network I/O operation itself after realizing that all conditions are met. This approach has two main flaws itself. *First*, it is against the earlier goal to centralize all network I/O operations in a single thread. *Second*, it cannot fully solve the problem because similar asynchronous event is needed for enforcing bandwidth constraint conditions, *i.e.*, *RC3* and *SC3*.

The revised solution separates `epoll`-based event retrieval from the actual network I/O operations. This is achieved by calling `epoll_wait` in a dedicated thread, referred to as *epoll thread*, and then passing the network events to the network thread through a dedicated queue for `epoll` events. This scheme is shown in Figure 6.10.

The network thread learns about *RC1/SC1* through the `epoll` events queue. Then the event is processed by checking other conditions of the socket. If *RC2/SC2* is not met yet, the socket will not be scheduled for network I/O at the current time. Otherwise, *RC3/SC3* condition is checked to see whether the governing bandwidth constraint allows the I/O operation to be scheduled right away. If so, the I/O task will be added to $ready\_io\_tasks$ queue. If the bandwidth constraint on the related link requires us to delay the I/O operation, we schedule a timer for the operation and queue the task to the $delayed\_io\_tasks$ queue instead. The network thread executes a single I/O task from each ready socket, *i.e.*, a socket that has all its three conditions for send or receive met. The $delayed\_io\_tasks$ queue is a list sorted based on the scheduling time of the I/O task. After performing the I/O tasks of the ready sockets, we check the target time of the tasks in $delayed\_io\_tasks$ queue to see if the right time has reached to perform any I/O task. Since the list is sorted according to the target time of each I/O task, we start checking the tasks from the head of the list and stop when a task's scheduled time is later than the current time.

With the *epoll thread*, the total number of threads in a Blizzard-base application emulating p peers will be $p + 3$ threads (assuming no algorithm thread is launched by the engines). There are more details into the actual operation of the network thread which we have to skip here. In the next section, we go through other design goals that were followed beside the `epoll`-based implementation.

6.3.2 Other design goals

Beside the `select`-based implementation, Crystal suffers from other deficiencies which are mostly addressed in Blizzard. Some of these improvements are made possible because of use of the new `epoll`-based design, such as having a shared network thread,

while other updates are independent of `epoll`-based implementation.

- **Producer/Consumer queues:** As described in Sec. 6.1, Crystal employs specialized message queues to queue the incoming messages till they are consumed, and also queues the outgoing messages till they leave the node. Currently, these queues are accessed by the engine and network threads without a strict producer-consumer rule. A direct advantage of enforcing a strict rule is the removal of the need for locking the queues at the critical sections. With the queues implemented through circular FIFOs, the queue has to be locked to keep its state consistent when a message is added or removed from a queue. However, if only a single thread always behaves as the producer for the queue and another thread as the only consumer from the queue, then no locking will be necessary. This task can be achieved without too much change in the base queue implementation. The engine thread can behave as a strict producer for the send queues while the network thread can be strictly the only consumer thread from the send queues. The case is the other way around for the receive queues. The network thread will be a strict producer for the receive queues while the engine thread is their consumer.

  This update did not turn out to be much problematic as there were not many instances which could not be turned into the producer-consumer relationship. Atomic operations are used to update the queue states when there exists the danger of race conditions leading to inconsistent states. Wrapper functions for atomic operations are created so these changes can be seamlessly portable to other OS platforms.

- **Removing excessive use of locks:** Crystal have used synchronization locks in many parts of its code to synchronize access to its common data used by multiple

threads. In many occasions, the locks are too coarse leading to performance degradation in high-performance applications. It turns out that many of such locks can be removed or changed to finer-grained scopes such that the possibility of threads waiting for each other can be reduced.

- **Avoiding locks for tracking *RC2/SC2* condition:** When a socket receives a *RC1/SC1* event notification through the `epoll` event queue, the network thread checks wether *RC2/SC2* condition is met or not. If not, the queue has to be marked such that the engine thread knows when it should notify the network thread by queuing an *RC2/SC2* event. This "marking operation" has to happen atomically with test of the queue state. Since use of the queue locks are about to be minimized as discussed above, a special value for m_size member of the queue is used to mark the queue in waiting mode at the same time the queue size is being checked. This operation is performed using an atomic *compare and set* construct which takes advantage of hardware locking of memory bus.

6.3.3 Implementation issues

The overall improvements turned out to be a bigger undertaking than what originally thought especially due to the inter-relationships between various improvements. Use of `epoll` in edge-triggered mode requires a very careful design. For example, if a notification edge is lost, no other I/O task will be performed on the associated socket which subsequently results in not receiving any further notification on that socket. Further, after receiving a ready notification edge, it is required to assume the socket will remain ready for transfer unless the socket I/O operation returns E_AGAIN which implies the underlying system buffer can not process the request at the moment, *e.g.*, due to no incoming data for receive or already full output buffers for send

operations.

The progress logs have been our main tool to test and validate the implementation and particularly the threads interleaving in Blizzard. Several race conditions were detected which required revise of the queuing construct. Also, some unexpected `epoll_wait` behaviors were encountered which initially broke the implementation logic. An example issue is shown in Figure 6.11 where a single UDP send operation of a few kilobytes (a UDP I/O operation is performed in a single socket I/O call) causes multiple ready notifications to the `epoll` thread during the duration of a single send call. Since the system send buffer is a few hundred kilobytes, we had the impression that the sending a single UDP message of let's say 10 KB should not cause so many notifications but it actually does.

This behavior of `epoll_wait` breaks some of our initial assumptions in a few ways. First, it breaks our handshake mechanism between the network thread and the engine thread. The engine thread sends the *RC2/SC2* events to the network thread but we assumed that when a socket is waiting for *RC2/SC2*, it would not receive further *RC1/SC1* notifications from the `epoll` thread. Second, processing these successive and redundant events is wasteful. To solve this issue, further release of `epoll_wait` calls in the `epoll` thread are avoided while actual network I/O is in progress in the network thread. Then the following `epoll_wait` call will tell us all about an up-to-date network state since the previous `epoll_wait` was issued. This solution needs careful implementation to avoid stalling network and `epoll` threads waiting for each other.

Overall, finalizing the design, implementations, test and debugging turned out to be a bigger undertaking and lengthier process than what we originally planned. In particular, reproducing some race issues with experiments involving hundreds of peers, implying hundreds of threads, turned out to be challenging.

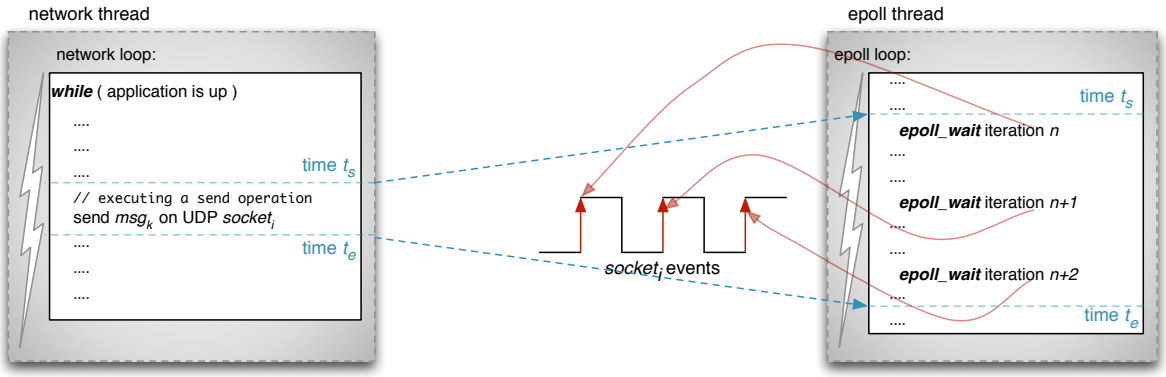


Figure 6.11: Receiving successive event notifications in response to a single socket send.

6.4 Experimental results

Now, a number of experiments are performed to evaluate Blizzard's scalability. These experiments compare the performance of Blizzard with Crystal. The same set of experiments reported in Crystal [70], measuring the CPU usage and maximum achievable throughput, are used as our benchmark here on the same dual-core Pentium 4 Xeon 3.6 GHz system. The system runs Ubuntu 8.10 Server Edition.

6.4.1 Blizzard vs. Crystal: Scalability

Similar to the setup in [70], a simple "relay protocol" is used with the source node feeding a number of peer chains, each consisting of 10 peers, all on a single physical node. The number of chains varies with the total number of emulated peers p in the experiment, and equals to $p/10$. In this relay protocol, each node forwards the incoming messages from its upstream node in the chain to its downstream node in the chain. Figure 6.12 shows such topology with 5 chains, *i.e.*, 50 peers in total.

The first experiment measures the maximum achievable streaming rate as the num-

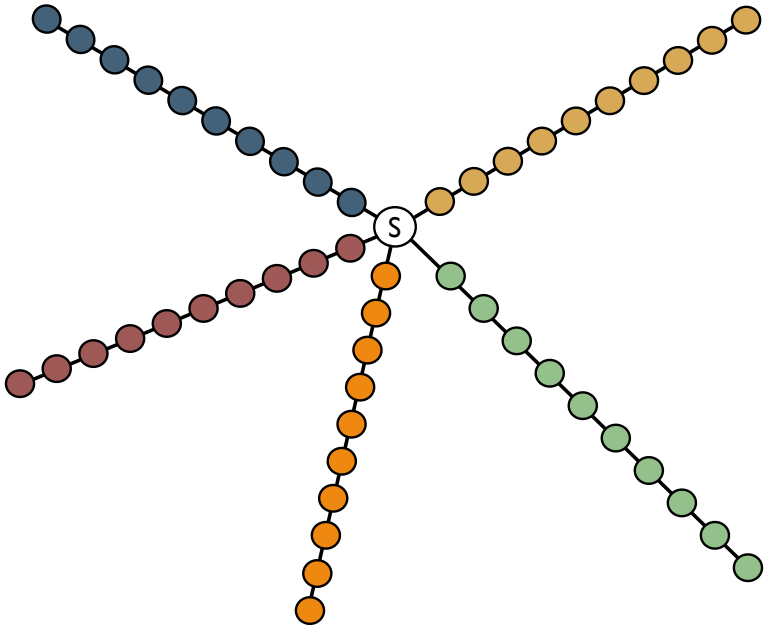


Figure 6.12: Experiment setup: Single source serving multiple chains of 10 peers.

ber of emulated peers increases. Because the links in the chains have no bandwidth limits or delay, the source node's streaming rate has no limit and the overall performance of the relay protocol is only limited by the available processor power of the CPUs in the physical node.

Similar to Crystal's experiments, the messages initiated from the source, which are subsequently relayed to the nodes along the chains, have a 1 KB payload [70]. Obviously, the cumulative streaming rate of all peers, shown in Fig. 6.13, decreases as the number of peers increases. This is because the processors need to serve more peers, *i.e.,* through p engine threads.

The advantage of Blizzard is obvious as it achieves between 1.9 and 3.4 times of the Crystal's cumulative streaming rate, up to experiments with 400 peers. Blizzard can easily maintain its performance with more peers and does not encounter significant decline even with 1000 peers while Crystal only reports experiments up to 400 peers

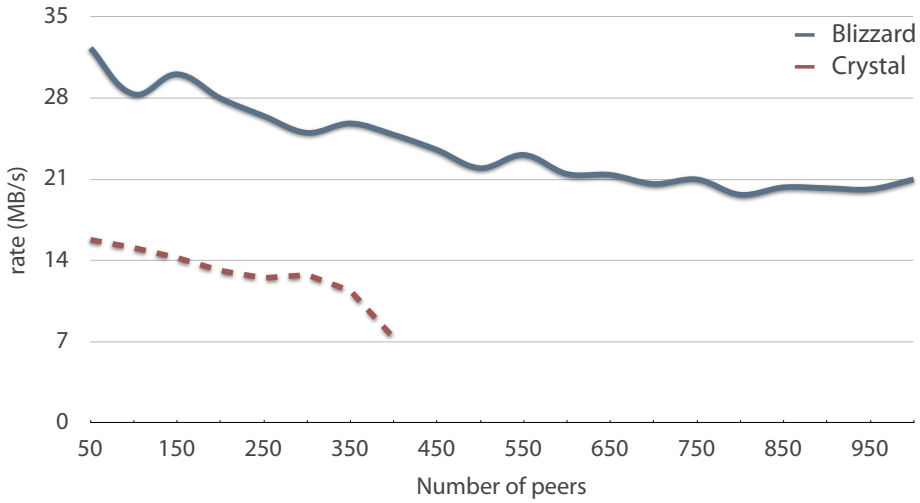


Figure 6.13: Blizzard vs. Crystal: Cumulative streaming rate with unlimited source.

due to some limitations.

Ideally, the cumulative rate should stay fixed but more peers imply more overhead, due to more threads to execute and more sockets to track, so the rate still decreases. An overall streaming rate of a few tens of MB/s might not seem very high initially. As we discuss later, a short message size of 1 KB causes more messages to be processed by the engines so the performance will be constrained by message handling within each engine and between the peers. Obviously, Blizzard show less overhead and better flexibility on that front.

Fig. 6.14 shows the per-peer streaming rate as the peers increase. Crystal concludes that for achieving at least 50 KB/s per peer, the test system should not run more than 250 peers. Blizzard hits the same 50 KB/s mark at 480 peers. With 1000 peers, each peer still can achieve a streaming rate around 21.5 KB/s.

In the next set of experiments, we use a constrained source instead of an unlimited source. The the source rate is increased from 50 KB/s to 3.0 MB/s on a single chain of peers, *i.e.,* 10 peers. With the same fixed message size of 1 KB as the previous set

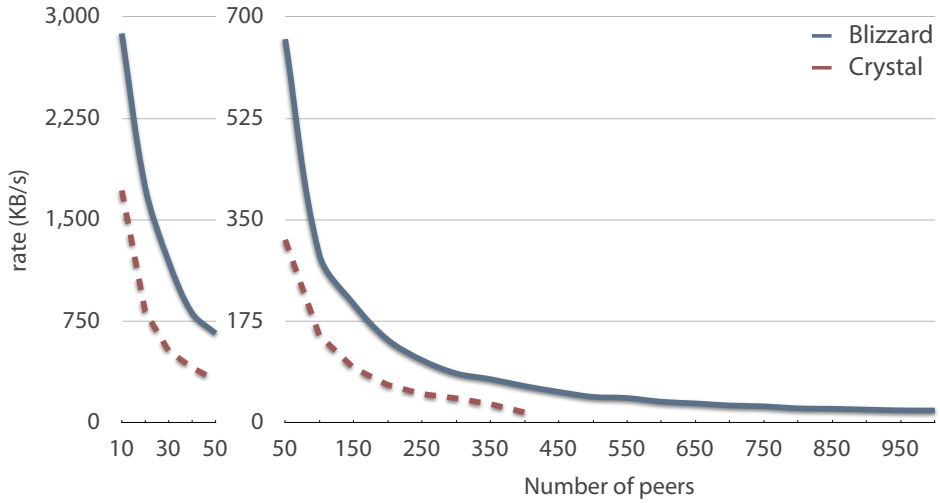


Figure 6.14: Blizzard vs. Crystal: Per-peer streaming rate with unlimited source.

of experiments, the gradual increase in the source rate results in more messages being forwarded to the downstream peers. More messages obviously leads to higher CPU load from each peer.

The performance results are shown in Fig. 6.15. At the initial streaming rates, Crystal experiences a much bigger CPU load compared to Blizzard, *e.g.*, up to 20% higher. This reflects the extra overhead of Crystal's network threads going over multiple queues and sockets regardless of their activity level. As the streaming rates increases, they perform better as more useful works are to be done in Crystal's network loops, *i.e.*, more message to process. Crystal's gap with Blizzard decreases gradually and stay at about 5%. With 20 peers, however, Blizzard's advantage becomes more evident as number of peers increase. For CPU load measurement, we use automatic sampling of the load based on a scheme to be described in Sec. 6.4.4. To make the comparison fair, we implement the same measurement scheme in Crystal.

6.4.2 CPU usage: A closer look

In Fig. 6.16, the CPU usage of the individual CPU cores, beside the average load, are shown for the same experiments in Fig. 6.15. Not surprisingly, different cores are

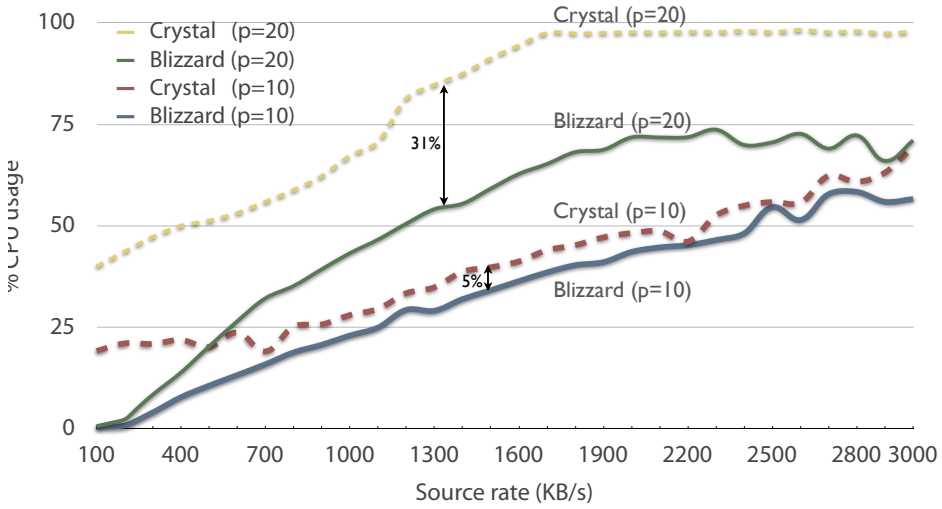


Figure 6.15: Blizzard vs. Crystal: CPU usage with varied the source rate.

assigned different portion of the load for each experiment as the system performs the thread scheduling.

To get a better idea about the CPU load in Blizzard, we experiment with explicit assignments of threads to cores. Explicit "thread pinning" can have its own pros and cons so it has to be done carefully. It can potentially improve the cache performance. On the other hand, it can lead to load imbalance as a fixed assignment can not adapt itself to system's dynamics.

Now we explicitly "pin the threads" to the CPU cores in a new set of experiments. Because the network thread serves all the peers, *i.e.*, all engine threads, we expect it to receive a bigger portion of the CPU load than other threads. As a result, we assign the network thread to the first core, separate from other threads. All the engine threads, one thread per peer, are assigned to the second core.

Fig. 6.17 shows the CPU load results for the same set of experiments as of Fig. 6.16 but with pinning enabled. Because there is not much thread contention in this experiment, *i.e.*, only about a dozen of threads compete with each other in this setup with 10

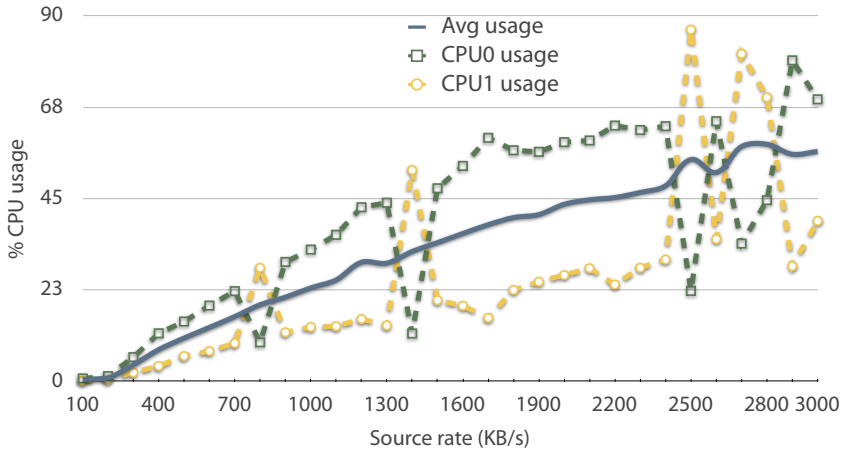


Figure 6.16: Individual CPU usage with varied source rate.

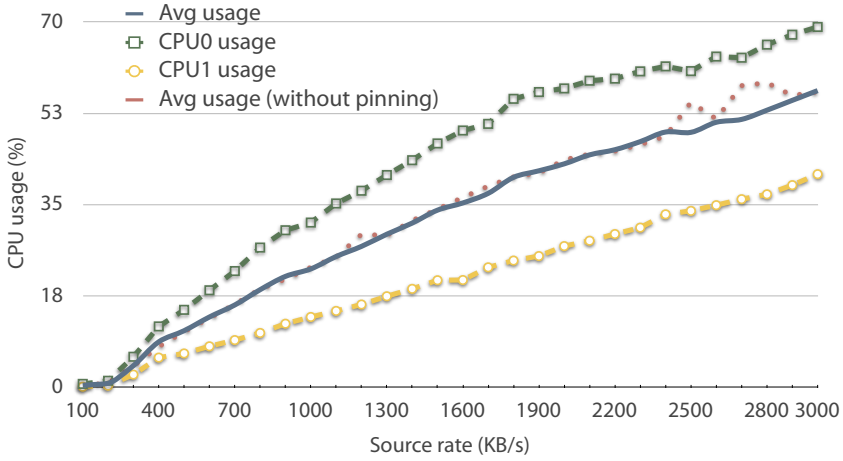


Figure 6.17: Individual CPU usage with varying source rate. The network thread is pinned to *CPU 0* and all engine threads to *CPU 1*.

peers, the average load is not changing. However, the result reveals that the network thread, running on *CPU 0*, consumes more than twice of the engine threads altogether, running on *CPU 1*. This suggests that the network thread can be more susceptible to thread contention in larger experiments, for example.

Now we repeat our first set of experiments, with unlimited source rate, but with thread pinning enabled. The result is shown in Fig. 6.18. Because the source has no limit on its streaming rate, it serves the chains as fast as it can and only limited by the

CPU power. As observed, the network thread reaches 100% load on *CPU 0* across the board. Because the network thread is not interrupted by the engine threads anymore, its sole use of a CPU core improves the overall system performance significantly. This is evident in the higher utilization of the cores by comparing the average CPU usage with and without thread pinning enabled.

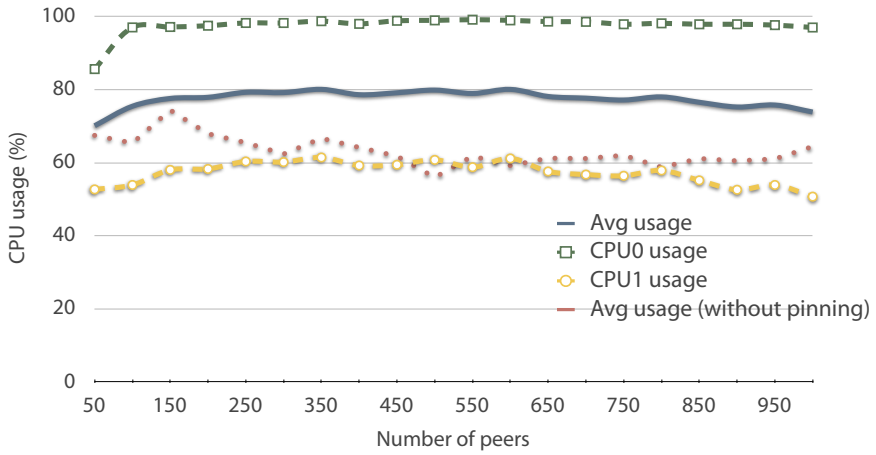


Figure 6.18: CPU usage with unlimited source rate and increasing number of peers. The network thread is pinned to *CPU 0* and all engine threads to *CPU 1*.

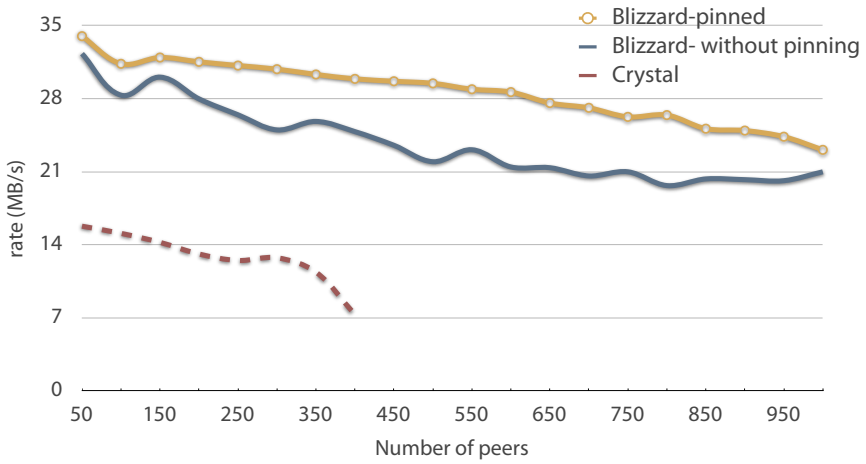


Figure 6.19: Blizzard (with an without pinning) vs. Crystal: Cumulative streaming rate with unlimited source.

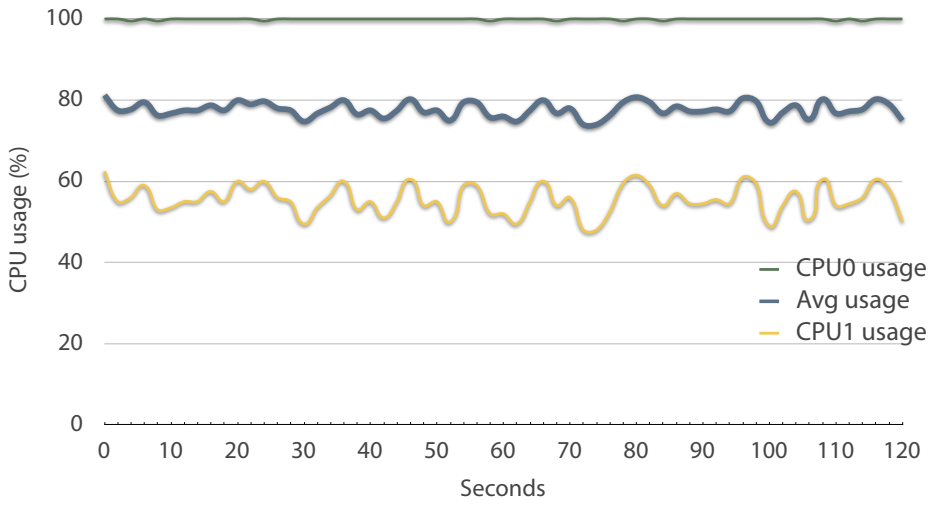


Figure 6.20: CPU usage with unlimited source rate and 100 peers. The network thread is pinned to *CPU 0* and all engine threads to *CPU 1*.

Fig. 6.19 shows the cumulative streaming rate of the experiment. Obviously, the higher utilization of the CPU cores with pinning enabled leads to another 10% to 30% improvement, in most cases, over the case without pinning. These results and Fig. 6.18 imply that the network thread is the bottleneck that limits the overall system utilization to 80% albeit its much better performance compared to Crystal. However, this is not a major issue as for having a finer granularity of control over the network thread, one can break the experiment to multiple processes and pin the network thread of each process such that thread contention in the cores are reduced.

Fig. 6.20 takes a closer look at the CPU usage for the experiment with 100 peers over time instead of the overall behavior of different sized experiments of Fig. 6.18. Without much surprise, similar behavior as the overall load is observed during the 120 seconds length of the experiment with the network thread placing a 100% load on the first CPU core.

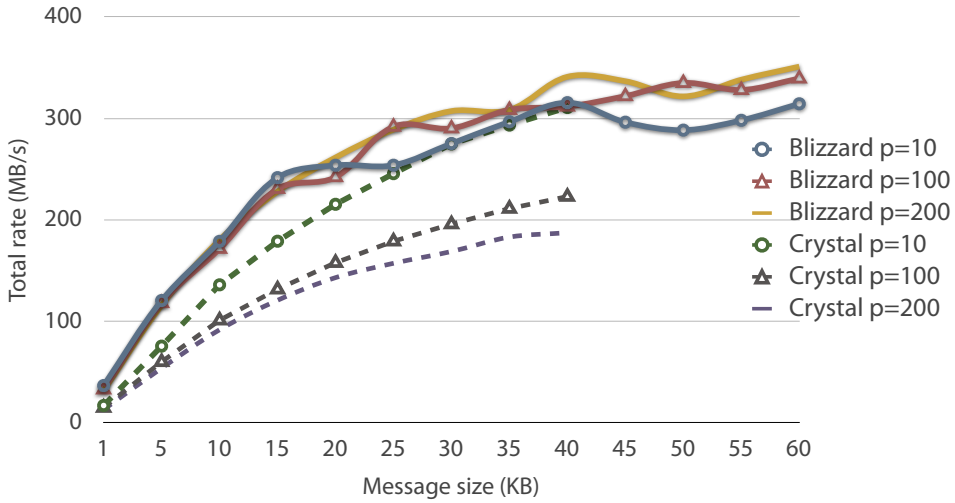


Figure 6.21: Blizzard vs. Crystal: Cumulative streaming rate with unlimited source and variable message size.

6.4.3 Message switching capacity

The number of messages that the core of Blizzard or Crystal can handle also depends on the message size. In the next set of experiments, the total switching bandwidth and also number of handled messages are investigated. The same unlimited source experiment is performed here but with varying message sizes, instead of the fixed 1 KB messages. Fig. 6.21 shows the overall streaming rate for the topologies with 10, 100 and 200 peers. The overall steaming rate, the cumulative data rate "exiting" all peers, is half of the total data exchanged among peers, *i.e.*, each sent message is received by another peer so the total messaging bandwidth is twice the overall streaming rate. We use streaming rate here to be in synch with the results reported in Crystal [70].

Initially, at small message sizes, the switching capacity of the emulation framework restricts the number of messages that can be issued at the send time and processed at the receive time. This message switching rate is shown in Fig. 6.22. For example, about 74000 messages are sent or received per second with message size of 1 KB in the

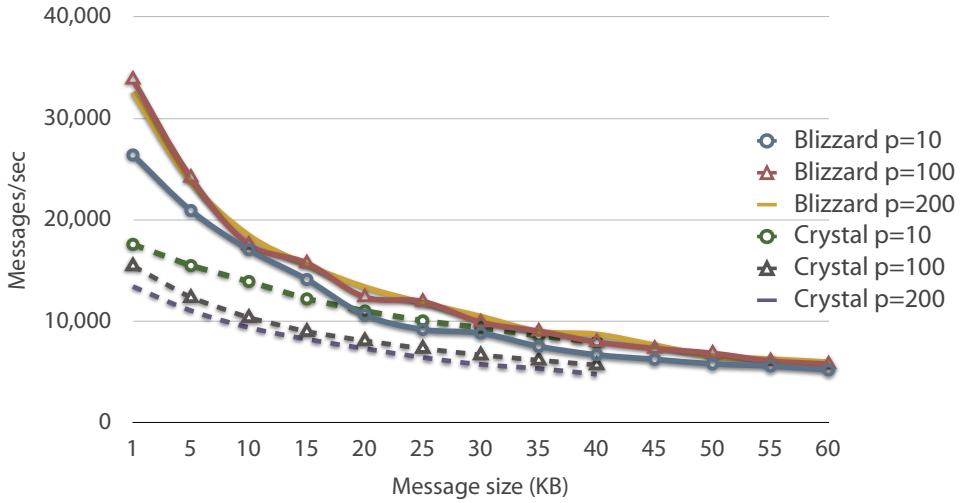


Figure 6.22: Blizzard vs. Crystal: Switching capacity (number of messages per second) with varying message size.

10-peer experiment. As the message size increases, the overall streaming/switching rate, in terms of bytes/sec, increases despite the fact that the actual switching capacity, in terms of messages/sec, goes down due to overheads associated with handling larger-sized messages.

The overall switching rate starts to saturate around 340 MB/s which can be attributed to the memory access bandwidth, to the most part. As the messages go through the network stack, at least two copy operations happen: one to the system's network stack, at the send time, and another from the system layer, at the receive time. Such rate is not far off noting that each copy takes a read and write operation and the maximum bandwidth of the system memory is 3200 MB/s, with PC-3200 memory modules.

To measure memory bandwidth performance, we develop a memory benchmark in a similar setup as our experiments, *i.e.*, for copying messages of 1 KB length. This benchmark achieves copy rates of 646.3 MB/s and 742.6 MB/s in single and dou-

ble threaded setups, respectively. These results confirm that memory has become the main bottleneck in our experiments in Fig. 6.21. This is because a switching rate of 340 MB/s implies a memory copy rate of 680 MB/s which is quite close to our memory benchmark results.

Because the memory/system becomes the bottleneck as the message size grows, the advantage of Blizzard over Crystal decreases. However, Blizzard's advantage is still clear with larger experiments, *i.e.,* 100 and 200 peers. They almost achieve the same performance as the 10-peer experiment while in Crystal, they experience a performance cut.

6.4.4 Discussion and miscellaneous issues

Not surprisingly, Blizzard has a clear advantage over Crystal especially on the scalability front. Though all development aspects of Blizzard is complete now, investigation of few stress test issues, *e.g.,* the rate oscillations in Fig. 6.21, and some tuning remain as future works. Nonetheless, this does not prevent the development of Blizzard-based real-life streaming applications to be presented in Chapter 8.

For running large experiments with hundreds of peers, some of the system settings has to be adjusted. Unlike Crystal experiments, the Blizzard experiments shown here all run in a single process. Having a single process has been helpful particularly for debugging purposes and tracing the race conditions, *e.g.,* through a single log file. Each peer requires at least three sockets, beside the per-connection sockets, for its listening port, and UDP data and control ports. Because sockets are treated as file handles in Linux, these sockets are counted towards the maximum allowed file handles used by the process. Also, each peer can employ its own log file to record its private logs rather than using the shared process-wide log. With only few hundred

peers, all these file handles quickly consume the maximum 1024 file handles allowed by default. As a result, the maximum per process limit on the file handles has to be increased by the `ulimit` command [6]. This command requires system administrator privilege for increasing the limit.

Further, the default per-thread stack size of 8 MB restricts the number of threads that can be launched in a single process. For example, creation of new threads starts to fail after having around 350 threads, *i.e.,* nearly 3 GB of virtual address space only for the thread stacks. As a result, the per-thread stack size was reduced to 2 MB with `ulimit` command. This reduction of stack size will easily accommodate launch of approximately 1000 threads required by the 1000-peer experiment reported in Fig. 6.13.

Problem investigation and debugging of a complex system as Blizzard is a very challenging task. This is the case particularly because hundreds of threads are present, in experiments involving hundreds of peers, and many race conditions can not be reproduced easily. Using file-based logs has helped to trace the progress but has also shown its own drawbacks as it slows down the execution, often masking the bugs.

To tackle this issue, we have developed a more sophisticated logging service in Blizzard that can switch between file-based and memory-based logs. The memory-based logging system has a much smaller scheduling footprint on execution of the experiments. These memory-based logs, typically of order of hundreds of megabytes, will be eventually written to files at the end of experiment or when the allocated memory has filled up. We have also realized that a central memory log is often helpful to collect traces and track the multiplexing of threads progressing in parallel. This central log requires access to the memory log be synchronized. Instead of traditional synchronization locks, we have used fast hardware-based spinlocks. Nevertheless, we still have run to some bugs that become masked even with low-overhead memory-based logs. Fixing such bugs sometimes took up to several weeks as we had to repeat

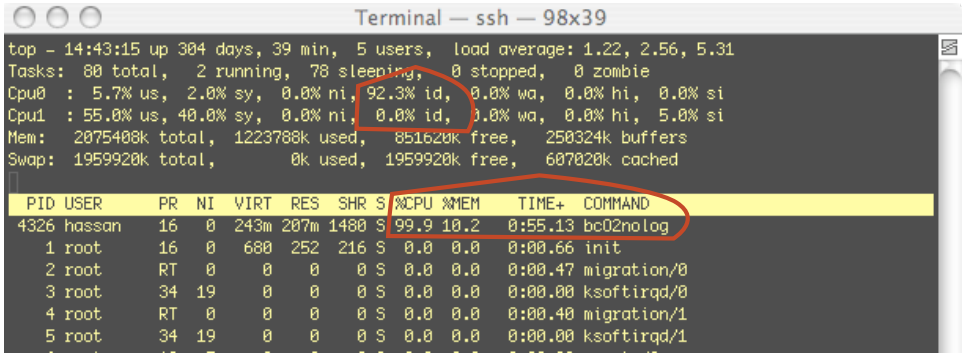


Figure 6.23: CPU usage reported by the Linux `top` tool.

the experiment many times and wait till only luck brings us the right condition, *e.g.*, non-corrupted stack frames, in GDB [2] for further investigation.

And finally, Blizzard has added support for automatic query of the CPU load. It employs portion of the source code from the Linux `top` command [5] which queries the load of individual CPU cores along with the average load from the `/proc/stat` path. To have a more accurate account of the CPU load, Blizzard profiles the process load through the system-wide "idle usage"' of the core(s) instead of the per-process usage. As shown in Fig. 6.23, the idle usage implies both user-mode and system-mode usages and allows per-core account of the system load on multi-core systems. Of course, such scheme is only valid when all background processes, *e.g.*, system services, do not consume significance portion of the CPU cycles.

6.5 Cross-platform Blizzard

Originally, Blizzard was intended as a highly scalable emulation framework specifically targeted for cluster of computers running Linux. However, as we started to build real-life coding-based applications (as the ones reported in Chapter 8) across multiple platforms, we realized that having the same code stream will make the maintenance

much easier than keeping separate code streams for each platform.

For a cross-platform Blizzard, we have first ported the system level mechanisms for threading, synchronization, networking, time tracking, etc. to different platforms. We have created an abstraction layer that hides all such differences and presents a higher level interface for the rest of Blizzard. Although time-consuming, this process has not been too challenging and successfully performed for Linux, Windows, Mac OS X, and iPhone OS. The main challenge is the portability of the `epoll` interface.

Fortunately, it turns out that the `kqueue` interface [14, 47] of Mac OS X allows us to detect the status of sockets in a similar fashion as `epoll`. The port to OS X does not turn out much difficult. We are pleasantly surprised to realize that iPhone OS fully supports the `kqueue` interface too. As a result, Blizzard works on Linux, Mac OS X, as well as iPhone OS. The platform-dependent parts of the code are conditionally compiled based on the platform. The final code stream of Blizzard contains roughly 17K lines of C++ code.

We have also looked into a Windows-based variant of Blizzard's asynchronous I/O core. However, fitting an asynchronous I/O scheme based on the *I/O completion ports* of Windows into Blizzard does not turn out to be easy. This is mainly due to lack of efficient APIs, such as `epoll` and `kqueue`, for detecting the status of sockets. As a result, a Windows port of Blizzard remains a future work.

6.6 Summary

This chapter presented the motivation and design for *Blizzard*, a new design for a scalable emulation framework. Blizzard has intended to improve the scalability of the framework to higher number of emulated nodes, *e.g.*, for emulation of P2P protocols. Further, it has kept eyes on gearing itself towards real and high-performance

applications by employing efficient techniques with low CPU overhead. The primary improvement was achieved through employing the relatively new *epoll* interface of Linux for tracking and distributing network events. This new interface has given us new opportunities to create a more efficient architecture with a totally redesigned core. At the same time, we have sought further improvements especially in the area of threading performance by aggressively reducing the use of synchronization locks.

We have compared the performance results of Blizzard with Crystal through several experiments. In almost all fronts, Blizzard has clear advantageous over Crystal. Development of Blizzard was a big undertaking with a lengthy verification and troubleshooting process, however. A new cross-platform addition to Blizzard now supports Mac OS X and iPhone OS beside the original implementation targeted for Linux.

Blizzard's basic services, packaged in roughly 17K lines of C++ code, significantly reduces the complexity of development and evaluation of sophisticated protocols. As an example, the coding-based on-demand and P2P video streaming experiments presented in Chapter 8 have taken only about 4000 lines of code, excluding *Tenor* coding library.

Chapter 7

Tenor: Making Coding Practical from Servers to Smartphones

From the perspective of computational capabilities of off-the-shelf hardware, we have recently witnessed a slew of state-of-the-art hardware advances, from servers to smartphone devices. On the extreme of dedicated servers, Graphics Processing Units (GPUs) have evolved to programmable general-purpose *throughput computers*, as shown in NVIDIA's Tesla GPU architecture. A single NVIDIA GTX 280, for example, attains a peak performance of 933 GFLOPS. On the extreme of mobile smartphone devices, the ARM Cortex-A8 core, being used in the iPhone 3GS and Palm Pre devices, includes a full Single-Instruction, Multiple-Data (SIMD) implementation called NEON. Moore's Law dictates that these isolated 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 with a coding toolkit with

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.

In this chapter, 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 on all platforms, where applicable. Tenor can be readily used as a black box without any knowledge of its implementation details. Or it can be fine-tuned and customized for specific needs of new coding-based applications.

Throughout this work, 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. Later in Chapter 8, we will build real-life streaming applications with Tenor and evaluate them.

The remainder of this chapter is organized as follows. Sec. 7.1 reviews the background of off-the-shelf hardware that Tenor supports. Sec. 7.2 presents the Tenor toolkit. Sec. 7.3 through 7.5 present the coding techniques (random network coding, LT codes and Reed-Solomon codes respectively), their implementations on various platforms along with their performance evaluations. Sec. 7.6 compares the pros and cons of the coding techniques supported by Tenor. Finally, Sec. 7.7 concludes the chapter.

7.1 Hardware Acceleration 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 (single-instruction, multiple data) instruction sets offer substantial assistance in Tenor, as they allow a single operation — such as floating point/integer arithmetic and logical operations — be performed on multiple data in a parallel fashion on specialized hardware units. SIMD instruction sets have become widely available on all modern commodity processors.

As commodity processors have migrated to multi-core architectures, our other parallelization venue is to utilize multiple processing cores. As our coding techniques are CPU-intensive without much I/O intervals in between, our multi-threaded implementations employ one thread per core and assign a portion of the workload at different granularities to each active thread.

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 [52]. NVIDIA's Tesla architecture is the most popular GPU architecture that enables high-performance parallel computing applications, written in the C language using the Compute Unified Device Architecture (CUDA) programming model and development tools [26], and is now considered to be the "most ubiquitous" supercomputing platform. Comparing to the CPU, 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 comparable and even 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. For example, the DDR3 memory found in the GTX 280 has a peak performance of 155 GB/s, compared to a meager 6.4 GB/s of DDR2-800 memory found on many servers.

The high-end desktop product GeForce GTX 295 boasts 480 processing cores and memory bandwidth of 233 GB/s. Even mobile GPU products offer many cores and high memory bandwidth these days, namely GTX 285M with 128 processing cores and memory bandwidth of 61 GB/s or the midrange GeForce GTX 9600M with 32 processing cores.

Beyond performance reasons, our interest is further stimulated by the relative low cost of mainstream GPUs as compared to multi-core CPUs. As an example, the mainstream NVIDIA GeForce GTX 280 used in this work retails for approximately $1/10$ of the cost of our Mac Pro server, with dual Quad-core 2.8 GHz Intel Xeon CPUs. Besides, due to the specific GPU design that schedules its threads in hardware, the performance of GPU-based coding is not affected by competing threads and background tasks. This makes GPUs very attractive for online multimedia systems, such as high-end servers for live video streaming where guaranteed performance is needed.

Mobile devices:

ARM processors are the most widely-deployed processors for embedded and mobile devices. Among them, ARMv6 architecture is used in a wide variety of mobile devices — prominent examples include the iPhone, iPhone 3G, 2nd generation iPod-Touch, Nokia N95, Nokia N800, Microsoft Zune, Motorola Razr2 V9, HTC TyTN II, and the Android-based HTC Magic.

The ARMv6 family of processors only features a plain 32-bit ALU, and does not in-

clude any SIMD support. Very recently, however, the ARMv7 architecture, also known as the ARM Cortex-A8, have been used in smartphone devices such as the iPhone 3GS, and features full SIMD support, called NEON. NEON has opened up opportunities in the Tenor toolkit towards high-performance implementations of coding. Beyond mobile devices, our implementations will be equally applicable to ARM-based digital set-top boxes.

7.2 *Tenor*: An Overview

Tenor includes high-performance implementations of a number of coding techniques: *random linear network coding* (RLC), *fountain codes* (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 cores. 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 allow seamless use of coding techniques in applications. An algorithm just needs to be set up with coding parameters (*e.g.*, block size and the number of blocks per segment), before it proceeds with the encoding or decoding processes. The rest of the coding process is handled transparently by Tenor. This allows the application, such as a media streaming system, to experiment with different coding techniques with minimal algorithm modifications. As a result, accurate performance comparison across different coding techniques can be performed quickly.

Beside heavy use of hardware acceleration and optimization of individual coding schemes, extra system-level measures have been taken to improve the performance.

For example, Tenor avoids data copying during the coding process by *in-place* coding *from* the source data segment directly *to* the outgoing message.

The model:

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 a fixed number of k bytes, the block size. To encode a new coded block x_j of k bytes, *code* $\mathbf{C_j} = [\mathbf{c_{j1}}, \mathbf{c_{j2}}, \cdots, \mathbf{c_{jn}}]$, consisting of n coefficients, is chosen and employed to combine the source blocks "somehow" into x_j. The decoding process processes the coded blocks one-by-one, *e.g.*, as they are received from the network. The original source segment can be fully rebuilt when n or more coded blocks, depending on the actual coding technique, are successfully decoded. The process is repeated for the following 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, 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 coding technique, a baseline reference implementation without acceleration, and various high-performance versions are provided in *Tenor*.

Fig. 7.1 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), number of threads to

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

Figure 7.1: Basic interface in the Tenor toolkit.

launch for multi-threaded coding, and buffer ownership modes. 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 coded blocks, *i.e.,* linearly independent blocks. The *reset* method resets the internal states of the codec to prepare it for encoding/decoding a new segment. Other variations of encode/decode methods exist as well, in order to facilitate various deployment scenarios.

There exist other variations of the same methods in the library. For example, the *encode* method of Fig. 7.1 receives a contiguous buffer for the original segment which is the typical usage scenario for a server node. However, the original content might sit in a 2-dimensional array as non-contiguous blocks. Other methods allow encode/decode of a number of coded blocks at once.

We now present each coding technique in Tenor and discuss their performance results.

7.3 Random Linear Network Coding

With random linear codes, code $\mathbf{C_j}$ is a set of randomly chosen coding coefficients in GF(2^8). For generating coded block x_j, $\mathbf{C_j}$ is employed through the linear combination shown in Eq. 2.1. 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 then recovers the original blocks $\mathbf{b}$ through Eq. 2.2. A random seed is often used to identify each coefficient row. Through Gauss-Jordan elimination, the decoding process can occur progressively as the coded blocks arrive.

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 bottlenecks in random network coding. To accelerate this costly operation, in our work in Chapter 2, we were the first to explore the use of a loop-based approach in Rijndael's finite field, rather than using the htraditional `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 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 [64]. In [65, 67], we explored high-performance implementations of RLC on the GPUs with various loop-based and table-based schemes. These works have shown that, by taking advantage of hundreds of GPU cores, encoding rates up to 279 MB/s and decoding rates up to 242 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. Having almost no SIMD support on ARMv6 cores used in smartphones

devices such as the iPhone 3G, our previous work in Chapter 5 have still managed to improve the network coding performance using a heavily hand-tuned loop-based implementation. We have concluded that RLC was feasible on the ARMv6 platform, albeit at less challenging configurations.

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

7.3.1 Recoding capability

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 if 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. 8.4, we will show how recoding helps a P2P live streaming application to serve peer nodes with coded content faster, such that they can meet their playback deadlines.

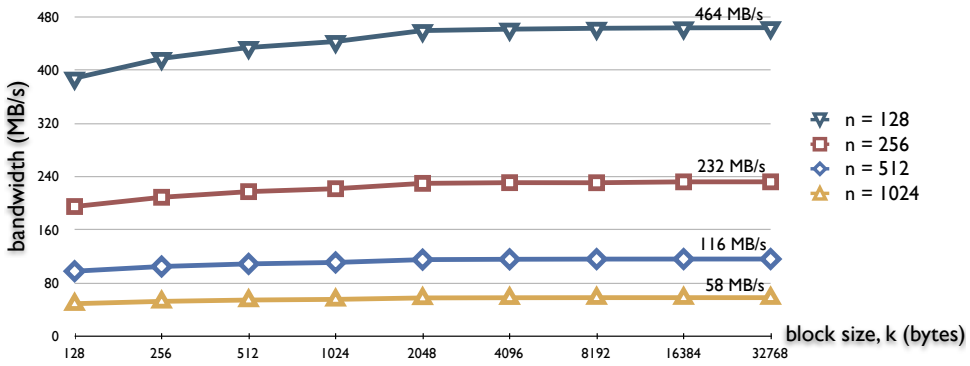


Figure 7.2: Multi-GPU encoding with GTX 280 and GTX 260.

7.3.2 Multi-GPU network coding

Because the encoding process in random network coding is a highly parallel problem, extra GPU power can be exploited to achieve even higher performance. Fig. 7.2 shows the raw encoding performance for a system running a GTX 280 and a GTX 260 in parallel. Since CUDA requires each GPU device 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 notoriously difficult settings such as $n = 1024$, an encoding rate of 58 MB/s can be achieved which is capable of serving over 600 nodes at a streaming rate as 96 KB/s.

7.3.3 Network coding on streaming servers

The encoding performance results in Chapter 3 and 4 and multi-GPU results in Sec. 7.3.2 reflect 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 GPU[9]. In Tenor, we are able to address challenges related to such system overhead in practical live and Video-on-Demand (VoD) streaming systems.

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. However, 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 an *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 result in linearly dependent content, leading to redundancy.

In order to best implement on-the-fly GPU based RLC encoding in VoD systems, we consider a $(n = 128, k = 4096)$ RLC configuration in a VoD streaming server, 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 VoD client requests $c = 128$ blocks of a video segment, the raw encoding rate is 265 MB/s.

To maximize streaming performance, it helps to have an in-depth understanding of GPU-based coding processes in streaming servers. When a client node requests c

[9]The coding bandwidth results in Chapter 3 and 4, also reported in [65, 67], reported each 1,000,000 bytes as a MB. From here on, we properly assume each MB as 1,048,576 bytes.

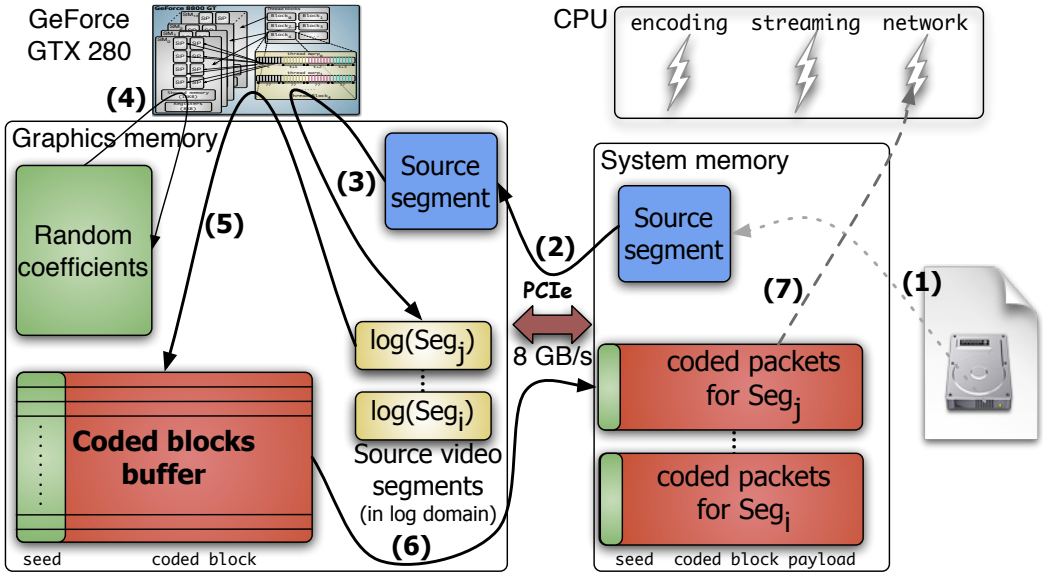


Figure 7.3: Steps of a VoD GPU-based encoding.

coded blocks of a video segment from the VoD server, the following steps, shown in Fig. 7.3, 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 `log` domain [65]; (4) generating c rows of random coefficients; (5) encoding the blocks to generate c coded blocks; (6) retrieving the coded blocks, of a total of $c \times k$ bytes, from the GPU memory to the system memory; (7) packetizing the coded blocks and sending them to the client based on the streaming algorithm.

Steps (1) and (7), disk access and the streaming mechanics, are not the subject of our discussion here. If steps (2) through (6) are performed serially, the encoding process of step (5), *i.e.*, the useful work, will take 1893 microseconds for $c = 128$, 79% of the overall execution time. This slows down the effective coding rate by 27% to 209 MB/s. Although this rate still has a clear advantage over the meager 62 MB/s performance on an 8-core server without GPUs, we still wish to improve the performance by minimize

the overhead.

The number of c coded blocks requested by a client could vary based on different factors, mainly on number of servers seeding the client, and the individual link bandwidth the servers to the client, *i.e.*, the client requests more content from the server that it has better connection to. To make the discussion here more tractable and look at the individual step above, we assume a single-server setup such as a YouTube server that now serves network coded blocks to the client. As such, at least $c = n$ coded blocks need to be generated and sent to the client. To simplify the discussion here, we ignore the possibility of loss of coded packets over the communication channel, e.g., assuming TCP protocol for transport, and also we ignore the possibility of linear dependence between blocks so $c = n$ coded blocks guarantees decodabilty.

Steps (1) and (7), disk access and the streaming mechanics, are not the subject of our discussion here. Steps (2) and (3) need to happen only once for a video segment as long as the pre-processed form remains in the GPU memory. Rather than sending the encode requests to the GPU over a period of time (e.g., just before sending a coded block to the client), generating n coded blocks at once results in better performance obviously to lower overhead due to less number of PCIe bus transactions and GPU kernel executions. With the $(n = 128, k = 4096)$ configuration, we make the following observations. Transfer of the source segment to the GPU memory and transforming it to the `log` domain, steps (2) and (3) together, takes 328 usec, around 14% of total time for encode of 128 blocks. Generation of random coefficient, step (3), has a very low complexity. To reduce kernel launch overhead further, we generate many rows of random coefficients at once and buffer the results in the GPU memory for the next encoding requests. For example, generating 15360 rows of coefficients takes only 1064 usec. These many coefficient rows are enough to encode $c = 128$ coded blocks for each of a 120 video segments set. This means an amortized per segment overhead of less

than 9 usec for generation of random coefficients.

Generating $c = 128$ coded blocks, step (5), takes 1893 usec, 79% of the total time. This number effectively means that by generating only 128 coded blocks, a raw encoding rate of 265 MB/s can be achieved which is down by around 5% from the 279 MB/s result in Fig. 4.8.[10] This is due to the small size of the encoding task, *i.e.,* just 128 blocks rather than 1000's of blocks. Also, the encoding achieves its best result when number of coded blocks is a multiple of Stream Multiprocessors in the GPU (GTX 280 has 30 SMs and $n = 128$ is not multiple of 30; generating 150 coded blocks reach 276 MB/s). Retrieval of $n = 128$ (step (6)) coded blocks takes 167 usec, 7% of the total time.

If all tasks, steps (2) to (6), are performed serially, the overall encoding rate will go down by another 27% to 209 MB/s. Even at this lower rate, the GPU still has a clear advantage over CPU-based network encoding. As shown in Fig. 4.10, the best asymptotic rate (generating a few thousands coded blocks) for CPU-based encoding on an 8-core server achieves a meager 62 MB/s at $(n = 128, k = 4096)$ which is far lower than even a serial execution of encoding[11]. Although such overall encoding rate of 209 MB/s can suffice the encoding needs of many applications by easily saturating a Gigabit Ethernet interface, we try to find ways to close the gap with the raw encoding performance.

Cache-based scheme for VoD: Nowadays, GPUs come with significant amount of RAM. Our GTX 280 graphics card has 1 GB of DDR III memory. Even more recent GPUs, such as the GeForce GTX 295, come with up to 1.8 GB of memory. Such large amount of GPU memory triggers this idea that we can cache the source segments in the GPU memory. Setting aside GPU memory for coefficient buffers, coded blocks

[10] In Fig. 4.8, the coding bandwidth is reported as 293.7 MB/s assuming a MB is one million bytes. Here we use the true MB as 1,048,576 which results in 279 MB/s.

[11] Similar to footnote 10, Fig. 4.10 reported the encoding rate as 65.1 million bytes which is 62 MB.

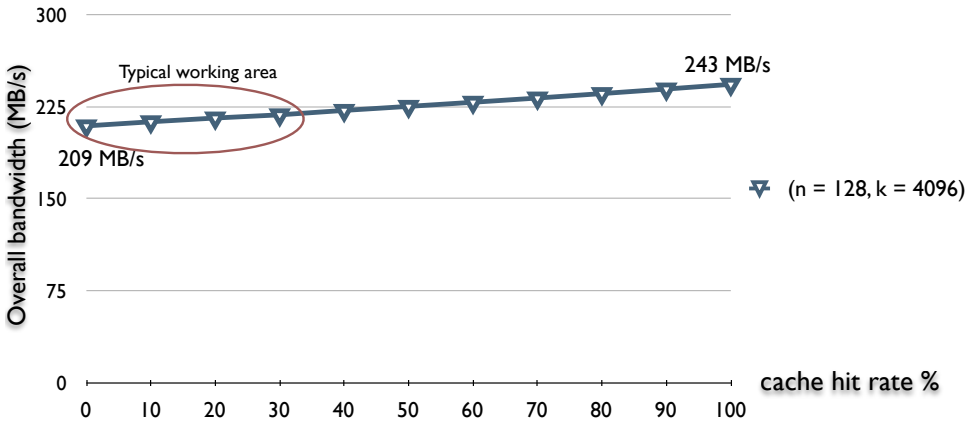


Figure 7.4: Overall encoding performance with varying cache performance.

results, etc. leaves us with enough space to store up to 1923 source segments, transformed to the `log` domain, in our GPU memory. Since each source segment needs to be transformed to the `log` domain once without the need to reference the original source anymore, the transformation can be done in place leaving us with enough room for 1923 different segments. This suggests that if another client requests a video segment that is already resident in the GPU memory (due to recent access by a previous client), we no longer need to repeat steps (2) and (3). Effectively, the GPU memory can be used as a cache of source segments. A simple data structure at the application side can track the segments currently cached at the GPU side. Each cached segment can be identified by its *content id* and its *segment index* within the content. Then only at a cache miss, we need to load the segment from the disk, step (1), transfer it to the GPU memory, step (2), and then transform it to the `log` domain, step (3). Simple replacement schemes such as least-recently-used can be employed. At replacement time, we simply overwrite an old segment with a new one.

Fig. 7.4 shows an estimate of the overall coding performance with the cache hit rate changing from 0% to 100%. Not surprisingly, the cache performance has a linear

effect on the overall coding rate achieving 243 MB/s at 100% hit rate, still 9% below the raw encoding rate of 265 MB/s.

However, the VoD systems does not show such locality. In reality, few clients are watching the same content and even fewer ones around the same temporal point within that content such that they can benefit from the cached segments before they are replaced by new segments. More advanced caching schemes can track the contents that are watched by multiple clients with a replacement policy that favors them. Another scheme can favor the initial segments of each content because of the intuitive observation that the earlier segments of video contents are normally accessed more often than the later segments as many users start watching a content from the beginning but do not watch it to the end. Even at a cache hit rate of 50%, the overall coding rate would drop to 225 MB/s which is only 8% better than the serial execution of 209 MB/s we discussed in the previous section.

Even ideal scenarios such as flash crowds, when a new content released to the system, are unlikely to improve the overall locality within the whole system, for example towards a 70% hit rate. The following section, however, proposes an optimization technique that make such caching scheme effectively redundant.

Overlapped scheme: 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 means that the kernel executions of step (3) through (5) can proceed in parallel with steps (2) and (6). Obviously these parallel subtasks can not belong to the same encoding tasks. Even steps (2) and (6) of different encoding tasks can proceed in parallel as PCIe is a full-duplex bus. 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 encoding tasks at the same time. We use CUDA's *streams* to manage such concurrency [26]. A *stream* is a

sequence of operations that execute in order. Different streams, however, can execute their operations concurrently with respect to one another [26]. We assign our encoding steps to streams, *i.e.,* the pipeline stages, according to the following: *Stream 0* performs step (2) and (3) for the new task that needs its source segment transferred to the GPU memory and then transformed to `log` domain. *Stream 1* performs steps (4) and (5) to generating the 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 time slot.

Of course, not all stages of the pipeline will be full at anytime. For example, if there is no pending task to transfer a source segment to the GPU memory, *Stream 0* will be empty in the next time slot. However, *Stream 1* can still proceed with generating coded blocks for the previous task. Double buffering is used for the output buffer that holds the coded blocks. This is because when *Stream 2* retrieves the coded blocks of $task_{i-2}$ (assuming the new task entering the pipeline is $task_i$), *stream 1* will be writing to the output buffer generating coded blocks for $task_{i-1}$. So we need two separate output buffers with buffers swapping their positions after each time slot.

This design did not go that smoothly in the beginning. We used CUDA's `cudaMemcpy2DAsync` API, which allows copy of rows of data when the source or destination blocks have different pitches [27]. This scheme was ideal for retrieving the coded blocks right into the payload of the packets that we later send to the client nodes. However, it turned out that `cudaMemcpy2DAsync` does not perform efficiently (it most likely behaves like a synchronous call). We ended up using the flat version of memory copy, `cudaMemcpyAsync`, instead. Now, at the application side, we need an extra system-to-system memory during the packetization phase.

With this overlapped scheme alone (without using any caching scheme) the overall encoding rate goes up to 262 MB/s. This rate is only 1% below the raw encoding rate

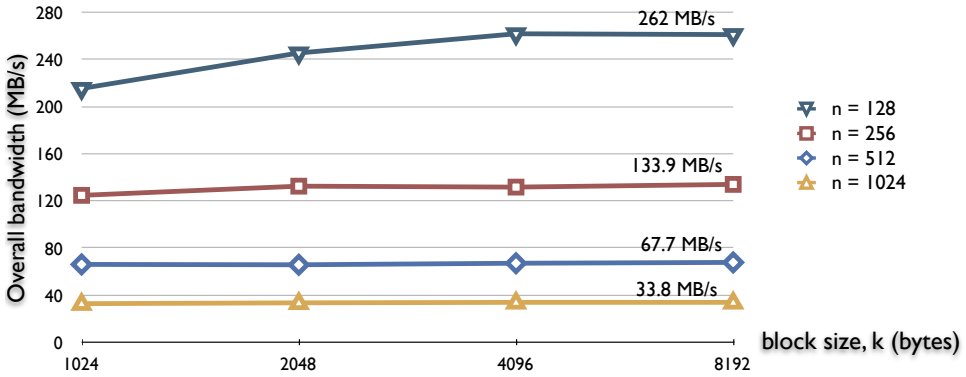


Figure 7.5: Pipelining in VoD streaming servers.

of 265 MB/s. The main reason for this small difference is the fact that transforming the source segment to the `log` domain, Step (4), is a GPU kernel call that is eventually executed by the GPU serially with regards to Step (5), the encoding operation.

With such pipelined processing, the overall encoding rate is substantially improved to 262 MB/s, only 1% below the raw encoding rate of 265 MB/s with $c = 128$. Fig. 7.5 shows the encoding performance for the VoD system of Fig. 7.3 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 reported in Chapter 4. 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 $(n = 128, k = 4096)$

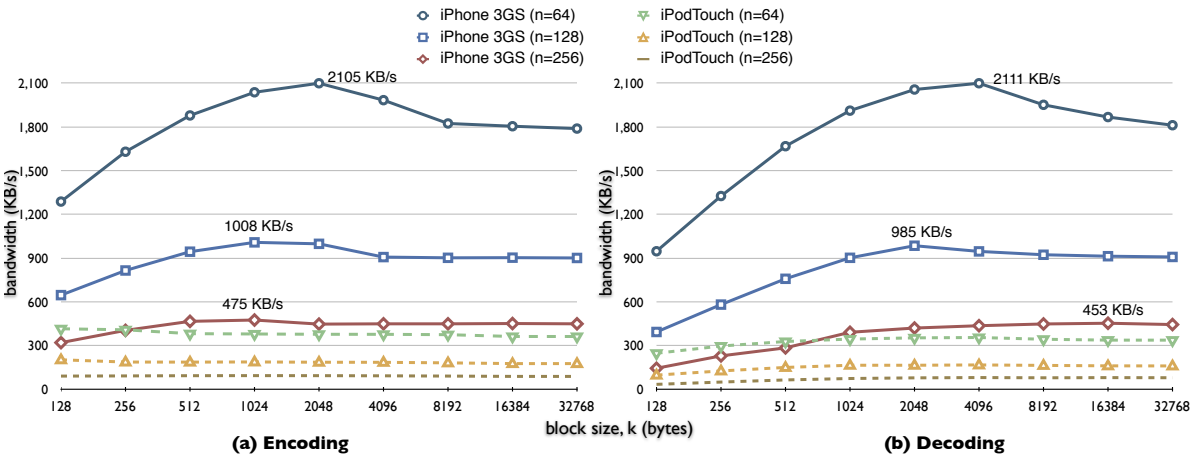


Figure 7.6: Network coding performance of the new iPhone 3GS in comparison with the 2nd generation iPod Touch.

setting.

7.3.4 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 in Chapter 5. In comparison, the *iPhone 3GS* and *Palm Pre* smartphones, released in summer of 2009, have both used a 600 MHz ARM Cortex-A8 as the application processor, coupled with a PowerVR SGX graphics 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 [16]. We wish to include a NEON-optimized RLC implementation in the Tenor toolkit, and evaluate its performance. Now we evaluate our NEON-optimized RLC implementation.

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 architecture without NEON support. While it is encouraging, an inspection of the machine code reveals that the compiler does not generate the most efficient code. In Tenor, we have included a NEON-optimized implementation by hand-tuning portions of GF-multiplication through inline assembly, which improves the performance by another 46%, up to 5.7 times advantage over the iPod Touch, as shown in Fig. 7.6. The coding rate drops when the working set increases beyond the 256 KB Level-2 cache. As the iPhone 3GS is running at 600 MHz, only 13% faster than the iPod Touch running at 533 MHz, such a performance improvement is directly due to the addition of NEON SIMD instructions. Comparing the accelerated loop-based coding to the legacy table-based implementation reveals an advantage between 3.5 to 6, which is quite consistent with our results on desktop CPUs [64].

We are pleasantly surprised by the performance of the ARM Cortex-A8: with a *single* SIMD unit at 600 MHz, it achieves nearly 1 MB/s at $(n = 128, k = 4096)$, while an Intel Xeon with *three* SIMD units at 2.8 GHz, achieves 9.4 MB/s. Such high coding rates, 2 MB/s for $n = 64$ and 1 MB/s for $n = 128$, open up new opportunities for real-life streaming applications with network coding 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.

7.4 Accelerated 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 [53]. Per our earlier model in Sec. 7.2, the encoding of x_j is linear combination of source blocks $[b_1, b_2, \ldots, b_n]$ such that the coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jn}]$ are chosen such that d_j number of them are one and the rest are zero.

With the linear combination done in GF(2^8), we effectively combine d_j number of the source blocks by xor-ing them. With probability $1 - \delta$, $n + \sqrt{n}(\ln(n/\delta))^2$ such coded blocks are sufficient to decode the original segment successfully. 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 [55], in sharp contrast with RS codes. The main advantage of LT codes is their low complexity. In average, degree d_j is equal to $\ln(n/\delta)$. Even with n as high as thousands of blocks, the average degree will only be a few tens of blocks to be xor-ed for each coded block, *e.g.*, 17 at $n = 10240$, and 12 at $n = 1024$. In contrast, RLC requires n linear combinations in $GF(2^8)$. The *robust soliton distribution* is commonly used for the degree distribution, it guarantees a sufficient number of *degree-one* coded blocks, the coded blocks that are equal to one of the original blocks b_i, coded blocks with $x_j = b_i$, to "kickstart" the decoding process and eventually reconstruct the original source blocks.

7.4.1 LT codes on the CPU

Implementing the LT encoder is quite straightforward as we select a number of source blocks based on the degree distribution and xor them. Like RLC, a random seed can be used to convey the selected code, which identifies the code degree and the subsequent indices of the 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 associ-

ated 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 retrieved first and then 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 a $(n = 10240, k = 1024)$ setting, for example, 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 encoding 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 implementation, 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, optimized with specialized instructions, instead of simply following the linked lists. 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 multithreaded implementation of the encoding process by launching one thread per core, similar to our multithreaded RLC in Tenor. Partitioning the coded blocks, however, brings very little benefit, especially at smaller block sizes. For example, at $(n = 10240, k = 1024)$, 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 assigning each block fully to one of the 8 threads. At the same setup, the encoding rate increases to 447 MB/s, a speedup of 2.7.

In Fig. 7.7-(a), graphs marked as *CPU-accel* and *CPU-th8* respectively present the accelerated and 8-threaded accelerated encoding rates for $n = 10240$ and $n = 1024$ across a range of block sizes. As the block size k increases, the encoding rate increases initially but then drops across the board. This drop roughly happens when the working set becomes too large to fit the 6 MB L2 cache available for each pair of cores in our 8-core system. The working set is dominated by the segment size, *e.g.*, 10 MB at $(n = 10240, k = 1024)$ or 8 MB at $(n = 1024, k = 8096)$.

In Fig. 7.7-(b), *CPU-accel* and *CPU-opt* respectively present the accelerated and graph-optimized decoding rates for $n = 10240$ and $n = 1024$. The advantage of better graph maintenance is obvious, in particular, at smaller blocks. However, the gap between *CPU-accel* and *CPU-opt* decreases as the block size becomes larger. This is due to the increase of block operations workload as k increases.

7.4.2 LT codes on the GPU

An efficient port of LT codes to the GPU turns out to be challenging even for the encoding process, which is simpler than the decoding process. GPU threads perform well when all threads of each *thread block* [26] follow exactly the same execution path. The randomness of the degree distribution, however, causes the encoding of different coded 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, which are then passed to the GPU along with the source blocks.

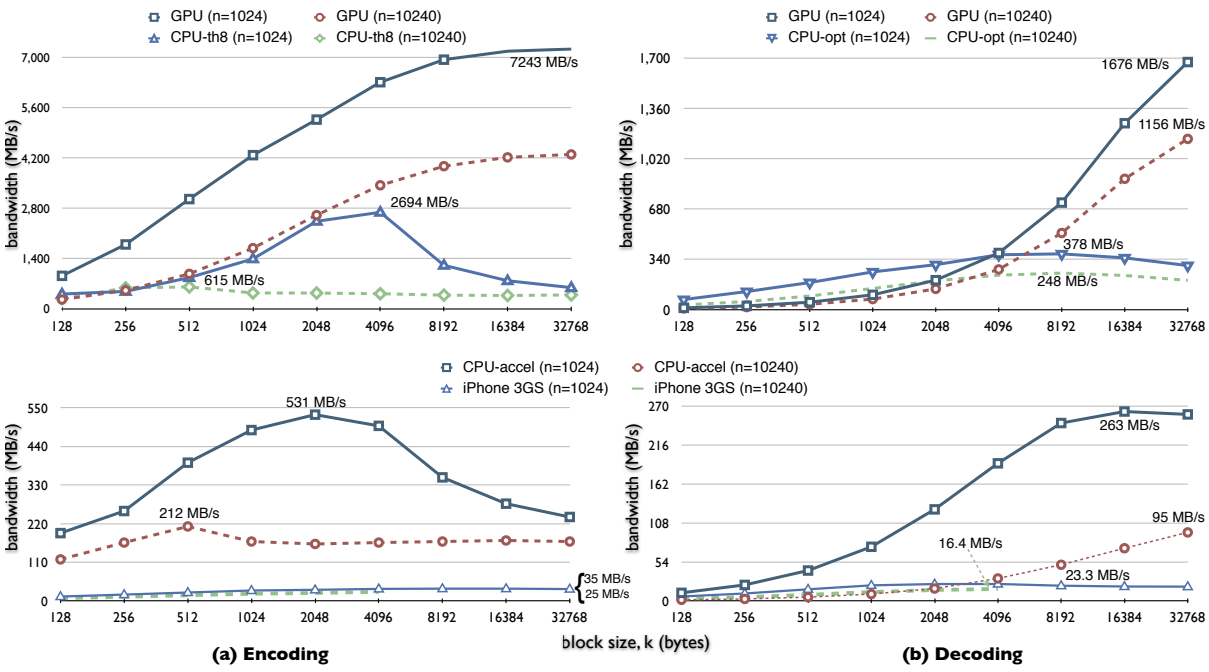


Figure 7.7: Coding bandwidth of accelerated fountain (LT) codes on different platforms.

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, which is a minor improvement over the single-threaded CPU-based rate of 165.5 MB/s at the $(n = 10240, k = 1024)$ benchmark. In the $n = 10240$ experiment, 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" the encoding tasks 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 generations to the GPU. Some minor,

but non-intrusive, modifications are made to the encoding to make it suitable for GPU-based coding (*e.g.,* to fit the robust soliton's CDF in GPU's constant memory). Multiple GPU kernels are called sequentially to perform different stages of encoding: 1) generate code degree for all coded blocks according to the robust soliton distribution; 2) sort the task queue according to the degree of coded blocks with higher degree blocks placed at the front of the task queue; 3) generate the codes, *i.e.,* indices of the source blocks to be selected for each coded block; 4) perform the actual coding. The new fully GPU-based encoding now achieves up to 1697 MB/s at the $(n = 10240, k = 1024)$ benchmark. Steps 1 through 4 respectively take 1%, 2%, 58% and 39% of the GPU execution time reflecting the fact that code generation is the most time-consuming part. The upper chart of Fig. 7.7-(a) presents the results of our best GPU-based encoding scheme for $n = 10240$ and $n = 1024$ with encoding rates up to several Gigabytes per second.

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 our GPU-based 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. 7.7-(b). The reason for this behavior is lack of enough parallelism in our GPU-based block operations as we process one block at a time and not many threads can be launched at smaller block sizes. Similar observation was made with network decoding in Chapter 3.

One particular benefit of GPU-based decoding can be illustrated in the following scenario. Assume an LT-based client-server bulk content delivery over high-capacity links, *e.g.,* software distribution from the main server to mid-servers at $(n = 10240, k = 4098)$ setting, *i.e.,* segment size of 40 MB. If the delivery rate is 32 Mbps (4 MB/s),

the segment can be delivered in little bit more than 10 seconds (due to the network overhead). However an artifact of LT decoding is that its progressive decoding is effectively not possible and the bulk of the decoding load happens at the very end. This means that not much decoding is done over the 10 seconds delivery time. Even at a decoding rate of 234 MB/s at this setting, the CPU will be effectively busy for nearly 171 msec in the last stage of decoding. This long busy time could lead to loss of incoming messages, *i.e.,* if the delivery is over UDP, unless decoding is interleaved with intentional sleeps in the middle. GPU decoding can become handy as the CPU can delegate decoding to the GPU and stay responsive to other tasks. Even with our current CPU-GPU decode scheme, where CPU bears about 20% of the decoding load at $(n = 10240, k = 4098)$ setting with a 273 MB/s decoding rate, the CPU will be busy for only about 29 msec, an 83% reduction in CPU time.

It is important to remind that the presented coding rates here are raw encoding rates and do not include transit times over the PCIe bus. Similar pipelined processing, as in Sec. 7.3.3, can be used to pipeline GPU encoding with bus transactions. However, we have observed a maximum PCIe throughput of 3.2 GB/s in our system, despite the 8 GB/s theoretical bandwidth of 16-lane PCIe. As a result, the effective throughput of our results in Fig. 7.7-(a) will be capped by 3.2 GB/s.

7.4.3 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 architecture (the iPhone 3GS in particular), taking advantage of its NEON SIMD instructions for block xor operations. The achieved results are shown in Fig. 7.7 for $n = 10240$ and $n = 1024$. At $n = 10240$, we are not able to evaluate coding rates at block sizes of 8 KB and higher,

due to the limited 256 MB memory on the iPhone 3GS is not enough to host the source, destination and coded segments. At $k = 8192$, the source segment reaches to 80 MB let alone other buffers required for the coded blocks and the decoded content.

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 $(n = 64, k = 32\text{KB})$ or LT codes at $(n = 1024, k = 2\text{KB})$. At these settings, iPhone 3GS achieves a RLC decoding rate of 1800 KB/s, while LT decoding achieves 31.2 MB/s, a 17 times advantage. Of course, a nearly 10% network overhead of LT codes at $n = 1024$, as well as its lack of the recoding capability, can be a prohibitive factor in some systems.

7.5 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 linear combinations of the source blocks in Galois Field. In RLC, the coefficient codes 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. The decoder can reconstruct the original segment by receiving any n coded blocks. Choice of m normally depends on the expected erasure rate with m/n ratio called *coding rate* [55]. 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 the row associated with the coded block so the code matrix $\mathbf{C}$ is rebuilt after receiving n coded blocks. After calculating $\mathbf{C^{-1}}$, the source segment is recovered.

In general, the erasure probability has to be estimated first to choose the *coding rate*

m/n [55] and build the generator matrix. 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, *i.e.*, the inverse matrix can be calculated. There are well-known generators, such as Vandermonde and Cauchy matrices, 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 [49] takes $4.5n^2$ operations compared to n^3 with Gaussian elimination. On the other hand, fast computation of the inverse of the generator does not improve performance much, because it comprises a very 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 (also limited to $n < 256$). 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 $2^{16} - 1$.

7.5.1 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 level-1 cache. In comparison, the `log/exp` table sizes are only 256 and 512 bytes in $GF(2^8)$. 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 in Chapter 2. This is obviously due to the fact that loops now have up to 16 iterations instead of 8. A threaded implementation is also provided in Tenor by partitioning individual blocks.

The accelerated and threaded coding results are shown as respectively *CPU-accel* and *CPU-th8* graphs in Fig. 7.8. To avoid complicating the figure further, we skip the baseline implementation results[12]. The results follow the RLC trend, except that the coding rates are halved due to longer iterations. Similarly, the performance also halves with doubling n since the linear combinations will operate on twice as many source blocks. We have used the Vandermonde matrix as the generator matrix in our benchmarks. Sparser generators will result in higher performance.

[12]For an external baseline implementation, see *RSCODE project* [8]. *RSCODE* uses the traditional table-based GF-multiplication. It supports error-correction setups in $GF(2^8)$. Its performance is essentially the same as our baseline RLC results in $GF(2^8)$ reported in Chapter 2 and [64].

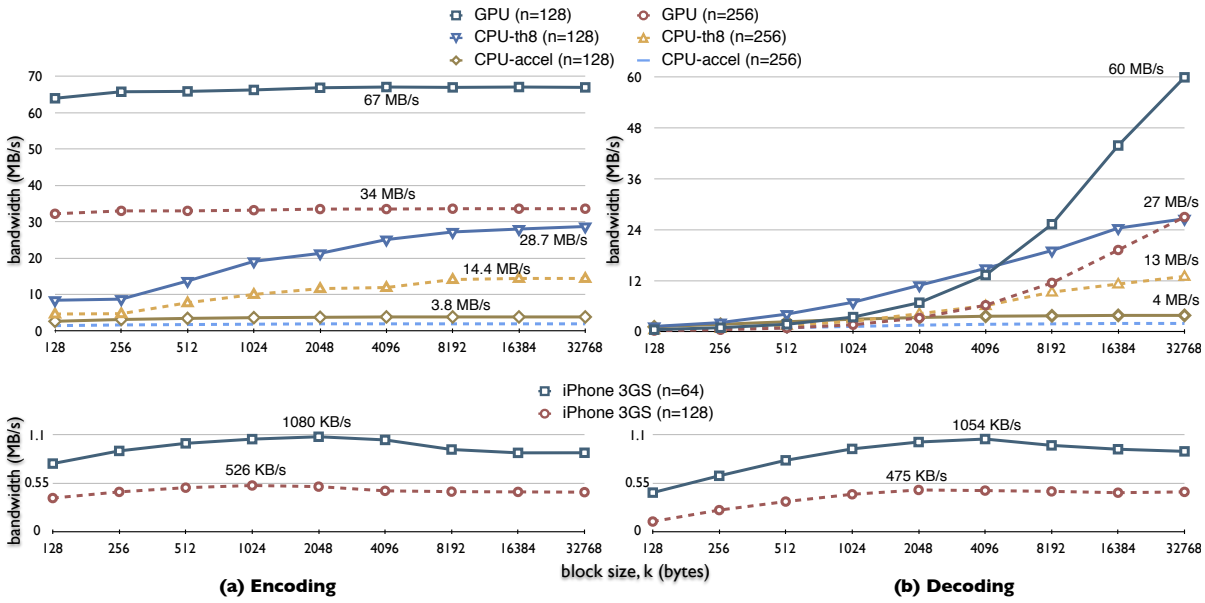


Figure 7.8: Coding bandwidth of accelerated RS codes in $GF(2^{16})$ on different platforms.

7.5.2 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 (SM) of the GPU. As a result, an optimized table-based scheme as in Chapter 4 is no longer possible. Instead, we follow the loop-based approach of Chapter 3 but operate in $GF(2^{16})$ instead. Since each GPU core has a 32-bit ALU, we encode a word-length portion of a coded block by each GPU thread. Each thread loops n times and performs a coefficient-by-word GF-multiplication at each iteration. Each word is treated as two short integers in $GF(2^{16})$. To create the irreducible polynomial mask, for the upper and lower short integers when they are about to overflow, we use the fast 24-bit multiplication.

The encoding performance is shown in Fig. 7.8-(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, how-

ever, the GPU-based decoding performance, shown in Fig. 7.8-(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.

7.5.3 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 employs NEON SIMD instructions in the ARM Cortex-A8 instead of SSE2/AltiVec SIMD in desktop CPUs. The encoding and decoding graphs are shown in the bottom charts of Fig. 7.8. Without much surprise, the coding results reflect approximately half of what have been achieved in random network coding with the same number of blocks.

7.6 Tradeoffs: Selecting a Coding Technique

In this section, we briefly overview the main properties, from system development point of view, of the three coding techniques supported by the Tenor toolkit. Fig. 7.9 summarizes the pros and cons of these techniques. The cross and check marks respectively indicate no/weak support, and good support of a feature.

Each criteria is discussed in more details. Our discussion here is a general discussion of the pros and cons of the techniques as the exact complete setup of the application, *e.g.*, lossy vs. non-lossy protocol or segment size, will determine the superiority of a technique.

- **Computation complexity:** The main advantage of LT codes over the other two

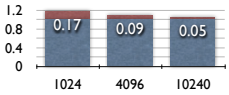
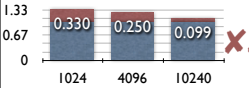
	LT codes	RLC	Reed-Solomon
Computation complexity	✔✔	✔	✔
Instantaneous CPU usage (decoding)	✘ 	✔ 	✔ 
Avg. % of blocks overhead	 ✘	✔	✔✔
Max. % of blocks overhead	 ✘✘	✔	✔✔
Large segments	✔	✘	✘
Med/small segments	✘	✔	✔
Recoding	✘	✔✔	✘✘
Packet overhead	✔✔ (random seed)	✔ (random seed, full coef row)	✔✔ (code id)

Figure 7.9: Features of LT codes, RLC, and Reed-Solomon codes.

techniques is its far lower computational complexity. For generating (or decoding) a coded block, a few source blocks, of logarithmic order, are xor-ed together. With RS codes and RLC, all n source blocks are linearly combined in GF domain which is a far more complex operation compared to simple xor in LT codes. At the same segment size, for example, LT codes achieve at least an order of magnitude better coding bandwidth than RLC. For a generic generator matrix, RS codes in GF(2^8) achieve the same coding performance as RLC. However, the maximum 256 blocks possible with RS codes in GF(2^8) could limit its application in some scenarios, *e.g.*, over a lossy transport, and require GF(2^{16}) instead. The coding performance of RS codes in GF(2^{16}) is half of RLC which can still operate in GF(2^8) for most cases. Further, it has been shown in [54] that the coding coefficients of RLC can be as sparse as a $7 - 10\%$ density setting. Noting

all results presented here and the previous chapters use fully dense coefficients, except the explicit discussion of Table 5.2, the use of sparse codes significantly close the performance gap of RLC and LT codes.

- **Instantaneous CPU usage for decoding:** In the decoding process, unlike the encoding process, the computation load of decoding a coded block varies. In RLC and RS codes, decode of the i-th coded block requires $2i + 1$ row operations, per Gauss-Jordan elimination. As a result, the decoding complexity linearly increases as the coded blocks arrive. In LT codes, however, receive of the earlier coded blocks only leads to minor updates in the sparse graph state. The mass block operations happen only at decoding of the very last few blocks. This leads to a sudden sharp increase in the CPU load, *i.e*, 100% CPU usage for periods up to few hundreds of milliseconds in typical setups. Such behavior might not be desirable for application and would require proper planning.

- **Average network overhead:** RS codes have no network overhead, *i.e*, any n coded blocks can decode a segment. The blocks coded with RLC might occasionally encounter a linearly dependent block, *i.e*, useless block, which effectively wastes the bandwidth. The probability of receiving a linearly dependent block is theoretically equal to $1/d$ with d equal to the size of Galois Field, *e.g.*, 256 for GF(2^8). As discussed in Sec. 7.4, LT codes require a significant number of extra blocks, beyond n, to be able to decode a segment. This leads to extra use of network bandwidth, *i.e*, network overhead, and varies with the number of blocks n used in the setup. It could go as low as 5% for $n = 10240$ or as high as 17% for $n = 1024$.

- **Maximum network overhead:** For LT codes, the network overhead could significantly vary from one segment to another. For $n = 10240$ as an example,

Although the average overhead is around 5%, we have observed segments experiencing overheads as much as 10%. Such big variance of the overhead could cause serious practical issues in system development.

- **Segment size:** LT codes are particularly favorable for large segments because higher n leads to lower network overhead. In contrast, higher n increases the computation load of RLC and RS codes linearly. Although they can use larger block size k to accommodate larger segment size, other limitations, such as the maximum packet size, could restrict use of large block size. While LT codes perform poorly with low n, which implies lower segment size in general, RLC and RS codes encounter lower computation overhead with low n. That is why LT codes are generally favored for bulk data distribution rather than time critical tasks such as streaming.

- **Recoding capability:** Recoding is a unique feature of RLC which distinguishes it from the other two techniques. In particular, it can lead to faster propagation of time critical data, such as live video streaming.

- **Packet overhead:** Assuming each coded block is transmitted in a packet, the packet overhead for identifying the code is quite small for LT and RS codes. In RS codes, an index of few bytes length can identify a code from the generator matrix. A random seed, *e.g.,* of four bytes, is sufficient to identify a code in LT codes. In RLC, however, a random seed can identify a code when the intermediate node does not participate in coding, *i.e.,* recoding is disabled. With recoding, the full coefficient row of n bytes, has to be embedded in the packet to identify the code. As a result, this $\frac{n}{n+k}$ overhead is a factor to consider when the coding setting n, k is selected.

As a realistic example for discussing various tradeoffs, let us consider a video-on-demand media streaming application with streaming rate of 800 kbps. As the segment size directly implies the initial buffering delay, a segment has to be fully received and decoded before it is passed to the codec for video decoding and the eventual presentation. A typical 5 seconds buffering delay implies a segment size of around 512 KB. A UDP-based setup can use the reliability provided by coding with a TFRC-based [29] TCP-friendly flow control algorithm. With RLC, the block number of $n = 128$ is known to have a very low computation requirement. This implies a block size of $k = 4096$ bytes. With LT codes, however, an $n = 128$ setup will lead to huge network overhead, e.g., of around 40%. A $(n = 1024, k = 512)$ setup will incur around 17% of network overhead due to coding. RS codes in GF(2^8), implying a coding space of only 256 coded blocks, does not seem a favorable technique in such setup with lossy transport unless the error rate is predetermined and known to be low. Further, a 4-byte random seed for both LT codes and RLC (assuming RLC is used without recoding), 16 bytes of protocol header and 20 bytes of RTP/UDP header will lead to an overall packet overhead of $\frac{40}{40+4096} = 0.97\%$ for RLC which is much smaller than $\frac{40}{40+512} = 7.2\%$ for the corresponding LT setup. On the other hand, packet sizes of smaller than Ethernet MTU (Maximum Transmission Unit), around 1500 bytes, have been found to have better penetration than larger UDP packets that the layer-2 network has to send in multiple MTU-sized chunks. To have a $k = 1024$ bytes in the RLC setup, however, would imply $n = 512$ which is known to be too computationally expensive particularly for less capable devices such as iPhone.

The above discussion tried to show various tradeoffs that has to be considered in picking a coding technique and its practical setting. In the above case, for example, RLC seems to have an overall advantage over LT unless the platform's processing power is very limited to support RLC or the communication links have extra band-

width to accommodate the overheads associated with LT codes.

7.7 Summary

This chapter presented *Tenor,* highly optimized and accelerated designs and implementations of RLC, RS and LT codes on a wide range of hardware and software platforms. By providing coding rates ranging from few MB/s for smartphones, to hundreds of MB/s for general-purpose CPUs, and up to thousands of MB/s for GPU-based servers in typical coding settings required for streaming and content distribution applications, Tenor can be deployed as *the coding facility* across all nodes of client-server and P2P systems.

To increase the portability of Tenor on the GPUs, an OpenCL port [45] can be added to Tenor. In particular, this can lower the CPU load for high capacity peers (that can serve a large number of peers) by facilitating GPU-based coding even if they have non-CUDA GPUs, such as ATI GPUs.

Chapter 8

Real-life Video Streaming with Tenor

Chapter 7 presented *Tenor*, our 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. It is cross-platform with support on Linux, Windows, Mac OS X, and iPhone OS, and supports both 32-bit and 64-bit implementations on all platforms. By providing high coding rates across different devices and platforms, it is ideal for deployment as *the coding facility* across all nodes of client-server and P2P systems. This chapter changes gears and presents the attained level of performance in real-life systems based on the *Tenor* toolkit.

In order to validate the effectiveness of the Tenor toolkit, we have built a coding-based on-demand and live media streaming system with a GPU-based 8-core Intel server, thousands of emulated clients, and a small number of actual iPhone and iPod Touch devices. Our experiences with this system, presented in this chapter, 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 work, 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 ARM Cortex-A8 architecture is sufficient to support random linear coding with a realistic range of parameter settings.

The remainder of this chapter is organized as follows. First, Sec. 8.1 introduces our overall goals and performance evaluation setup. Sec. 8.2 presents our system design of large-scale coding-based on-demand media streaming. Sec. 8.3 experiments with on-demand streaming on Smartphones. Sec. 8.4 extends the experiments to coding-based P2P Live Streaming. Finally, Sec. 8.5 concludes the chapter.

8.1 Performance Evaluation

Having *Tenor* and its high-performance implementations of coding techniques at our disposal, we now develop coding-based VoD and P2P applications in real-life setups that eventually deliver and playback real video. We deploy our prototypes in a LAN with realistic setups and push them to their performance limits. Here, our goal is not to build the most sophisticated protocols. For example, we use TCP-based delivery in our LAN deployments rather than UDP, which has better penetration properties in the Internet. The computation performance aspects of the system is of our main interest. However, our relatively simple protocols in a LAN-based setup can be the basis for more sophisticated protocols for an Internet deployment.

To push the performance of our GPU-based VoD server to the limits, we deploy it in a large experiment with thousands of clients emulated on a cluster of Linux-

based nodes. To facilitate deployment of such large experiments, we use *Blizzard*, our high-performance framework presented in Chapter 6. We use Blizzard's scalable emulation capabilities for: 1) emulating thousands of client/peer applications on a cluster of computers; 2) facilitating scalable servers that can efficiently communicate with thousands of clients. Blizzard framework provides basic services, *e.g.*, managing multiple connections with independent message queues such that slow connections do not slow other connections, a messaging subsystem that can carry payloads of various sizes, supporting several emulated nodes at various ports of a physical node with bootstrapping mechanism at arbitrary start and stop times, separate threads for emulated nodes so they can run in parallel and behave as independent nodes, as well as maintaining network topologies. With such services from the framework, the streaming protocol becomes much thinner and also makes maintenance of the protocol easier.

Scalability is the main challenge for both deployment schemes in our streaming setup, *i.e.*, server or emulated clients. Blizzard, of roughly 17K lines of C++ code, provides a "scalable network framework" for our system. Tenor, consisting of roughly 23K lines of C++ code, provides a "scalable coding facility" that complements Blizzard. Blizzard and Tenor together can push our system to limits on both ends, network and computation limits. Because both Blizzard and Tenor are cross-platform utilities, the implementations of our streaming protocols are mostly indifferent to platform. With the help of Tenor and Blizzard, our VoD and P2P streaming protocols, to be presented in the following sections, result in only about 4000 lines of code.

8.2 Large-Scale VoD Streaming

Our setup for a large-scale VoD client-server experiment, which incorporates coding as a part of its streaming protocol, is shown in Fig. 8.1. The server node is a Mac Pro

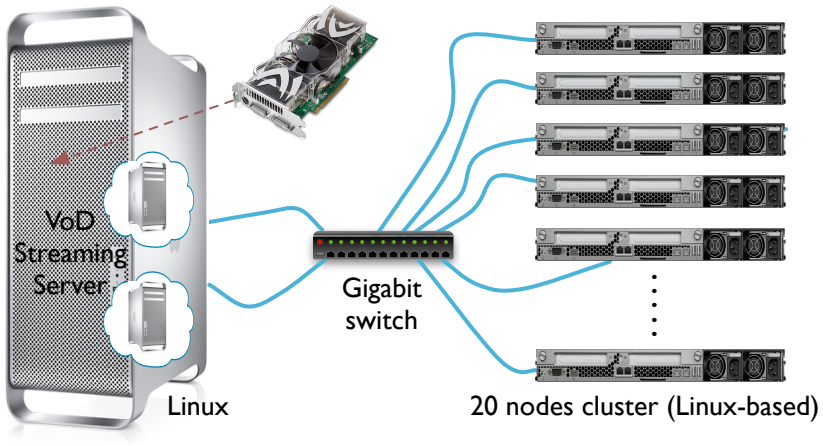


Figure 8.1: Large scale VoD experiment on a cluster.

(running Linux on two Quad-core 2.8 GHZ Xeon processor, 8GB of memory, two Gigabit Ethernet, and GTX 280 graphics card with 1GB of graphics memory) connected to a 20-node cluster (each running Linux on a Quad-core 3.0 GHz Xeon processor, and 2 GB memory) through a Gigabit Ethernet switch.

In the server, we employ the highly efficient GPU-based encoding scheme of Sec. 7.3.3, which was specifically designed such that coding rates very close to the raw rates can be achieved despite thousands of unique contents served in typical VoD systems. On the clients, we employ accelerated CPU-based decoding along with *Blizzard* to emulate up to hundreds of clients on each cluster node.

This VoD setup, if used with a single server application, very much resembles a YouTube streaming server with the clients requesting different video contents and the server providing them over TCP after an initial buffering delay. 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 in this

experiment as we emulate the clients on cluster nodes.

An important benefit of coding is that multiple servers can simultaneously serve a receiver with very 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. The servers are emulated as two server applications listening on separate ports of our physical server. 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 of our server. Initially, we assigned each separate IPs, on the same subnet, and intended to bind each server application to one of the interfaces. However, such setup turned out to be problematic in Linux arising issues such as ARP flux at the clients, routing at the server, etc. Eventually, we opted for *interface bonding* which allows aggregating multiple network interfaces into a single *logical* bonded interface with a single IP but a 2 Gb/s capacity.

We use *single* hard-drive with a plain setup. Because optimizing the disk read access is not our goal here, we do not wish the disk-to-memory load of the video contents affect our streaming performance of our experiments. As a result, we cap the number of unique video contents in our experiments to 200 contents. Each client, up to 3000 in our experiments, picks one of the 200 video contents at random. At 96 KB/s streaming rate, this results in a disk-to-memory transfer rate of about 20 MB/s with the rest of the reads served from the disk cache[13].

8.2.1 The streaming protocol

When a client starts, it requests a content from both servers. We use two server applications both running on our physical server, each listening to a different port of

[13]To establish a context, the effective I/O bandwidth of SATA disk drives is usually between 30 and 70 MB/s.

the *logically bonded interface*. This effectively emulates a YouTube-like scenario, however, with multiple servers. In an Internet deployment, each server can be placed at different geographic/topology location. As a server's link bandwidth varies over time, *e.g.*, due to increased load or a network bottleneck, the other server(s) can make up for it. With coding, however, block reconciliation, the traditional tracking of individual missing blocks, becomes unnecessary simplifying the streaming algorithms. This is because coding removes the identity of individual blocks and any coded block, regardless of its server origin, effectively contributes the same amount to decoding. Receiving *enough number* of coded blocks suffices to rebuild the source 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 task queues. After the initial request from a client, a request for loading the first segment of the related video file is queued to a dedicated thread, *file server*. After the segment is loaded to the main memory, the task will be delegated to Tenor's *encode server*, which performs GPU-based encoding of the loaded segment. The encoding process is executed in several stages in the pipeline fashion described in Sec. 7.3.3 with another queue managing the pending encoding requests. Each request is for a set of coded blocks to be generated from the loaded video segment. After the coded blocks are retrieved from the GPU, they will be buffered in the system memory and gradually pushed to the client according to the *streaming algorithm*. As streaming for a client reaches to a certain threshold in the active segment, the server automatically generates a new request for disk load and subsequent GPU encoding of the following video segment. To ensure smooth transition to the next segment, the coded blocks are double-buffered in the system memory: one buffer for the *current segment* which is currently streamed out, and another buffer for receiving the coded blocks of the *future segment* from the GPU. The streaming algorithm, both

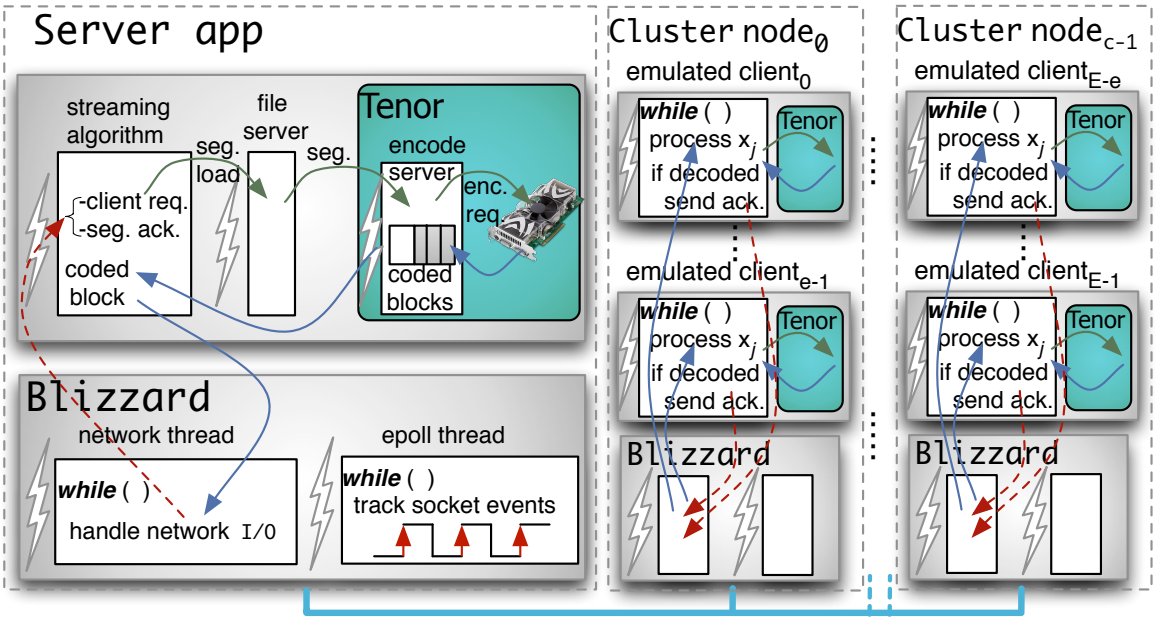


Figure 8.2: Server and emulated clients based on the Blizzard framework.

on the server and client sides, relies on Blizzard for its messaging subsystem and the low-level network operations. Fig. 8.2 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 the code, *i.e.*, a random seed, associated with the coded block. At the client side, the coded packets are decoded as they arrive. After each segment is successfully decoded by the client, an acknowledgement message is sent to both servers so the servers move to the following segment. We use segments of 512 KB length, which are worth of 5.33 seconds of media at a streaming rate of 96 KB/s. The segment size implies an *initial buffering delay* of 5.33 seconds, an acceptable delay for our experiments.

We use two different coding techniques to compare their performances. Our first

coding technique is RLC, presented in Sec. 7.3. We use a network setting of $(n = 128, k = 4096)$. At such setting, GTX 280 can serve at 261 MB/s in the real VoD setup of Sec. 7.3.3, *i.e.,* over 2600 clients each at 96 KB/s streaming rate. In the GPU encoder, we generate slightly higher number of coded blocks for each segment, effectively $c = 130$, to accommodate the rare case that a client needs further blocks after encountering a linear dependent block. Receiving a total of 128 linearly independent coded blocks, from all servers, will result in successful decoding of each segment. Each client uses the single-threaded SSE2-accelerated RLC codec of Tenor. The computation complexity of decoding is manageable for few hundreds of clients on each cluster node.

Our second coding technique is the LT codes. Albeit its lower computational complexity, LT codes incur a network overhead, *i.e.,* more than n coded blocks are necessary to decode a segment. When codes are selected randomly, such overhead is around 17% at $n = 1024$, for example. One need to use higher n to decrease the overhead, *e.g.,* $n = 4096$ lowers the overhead to 9%, and $n = 10240$ to 5%. However, large n is not suitable for a 512 KB segment, *e.g.,* at $(n = 1024, k = 512)$, a block size of 512 bytes will incur 56 bytes of header, message and TCP/IP headers included. This is why in general, the LT codes are more suitable for distribution of large files, where the segments can be large, rather than streaming applications. However, a large LT codes space is not needed for our experiment because we do not have loss of coded blocks, delivery is over TCP, and we have limited number of servers. Instead of random codes, we find and employ *fixed* LT codes that can operate in the same $(n = 128, k = 4096)$ setting as RLC but with a low network overhead such as 11.7%, *e.g.,* decodability with 143 coded blocks. This allows us to use LT codes as a benchmark against RLC though difficult to extend to more general solutions.

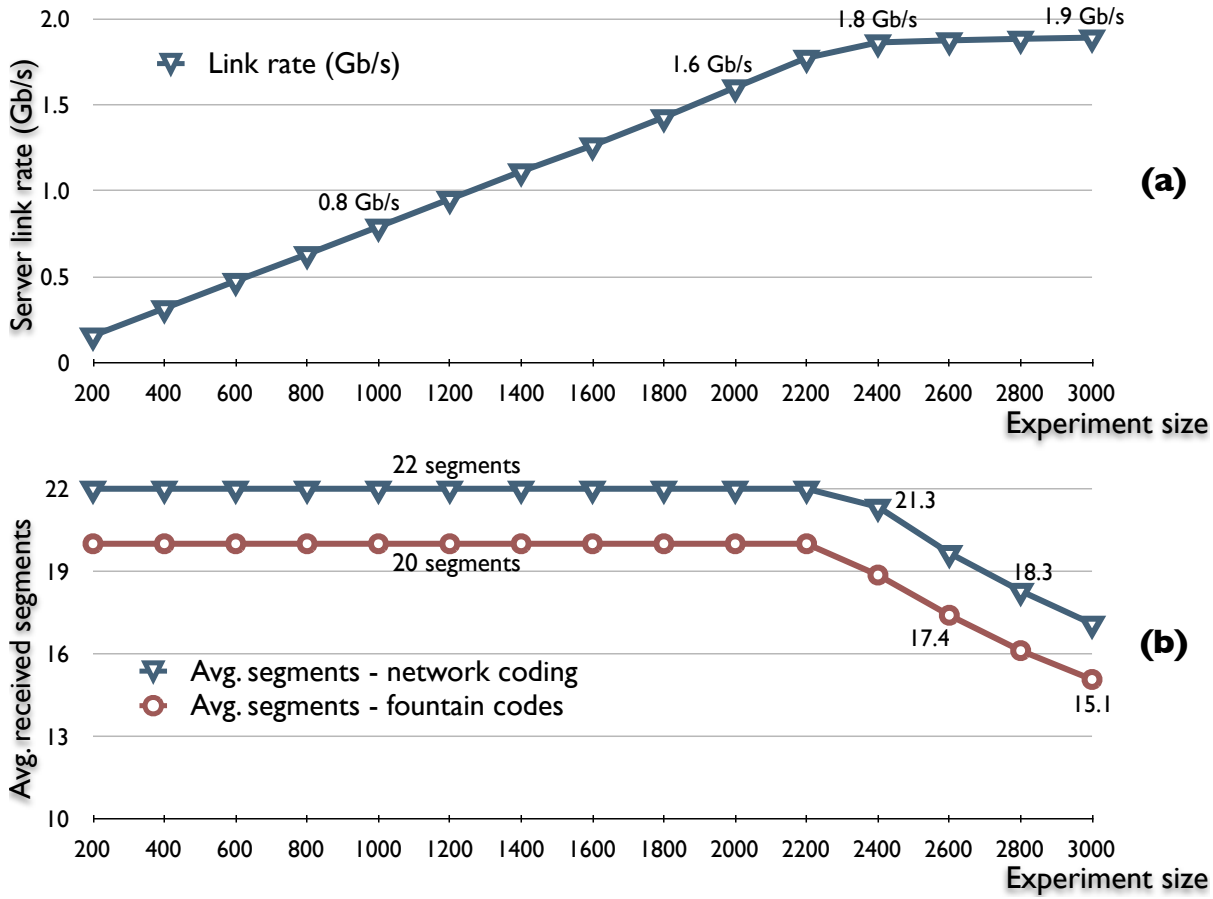


Figure 8.3: Link rate and segment delivery across different experiments.

8.2.2 Experimental results

We perform a number of VoD streaming experiments with each of the RLC and LT codes techniques. We changes the number of the clients in the system from 200 to 3000*i.e.*, emulating 10 to 150 clients on each of the 20 nodes of the cluster, and evaluate the streaming performance. In each experiment, all clients start their streaming sessions within a 5 second interval, *i.e.*, in the experiment with 2000 clients, clients join the system at a rate of 400 clients per second. Each session lasts for 120 seconds.

Fig. 8.3-(a) reflects the servers aggregate link rate, over the bounded network interface, across different experiment sizes. Since both coding schemes operate at the

same setting, the link bandwidth rate is very close for both set of experiments so one group is shown in the figure. However, the average number of fully decoded segments varies between the coding techniques as shown in Fig. 8.3-(b). With RLC, each client's 96 KB/s streaming rate is fully utilized and all clients receive the $120/5.33 = 22.5$ segments streamed for them and manage to fully decode 22 segments. Of course, this is true as long as the link capacity from the server applications is not saturated. When the experiment size grows beyond 2200 clients, the total number of coded blocks leaving the server can not increase beyond the 2 Gb/s link rate. Although the clients increase, the total number of delivered segments stays almost fixed, *e.g.,* around 51200 for RLC, so the average decreases. This eventually leads to decrease of the average number of fully decoded segments. At nearly 1.9 Gbps, we are quite close to the 2 Gbps link capacity. Because both servers share the same link, each provides roughly half of every client's streaming rate. The GPU never reaches its encoding limits as the link saturates from 2200 clients.

Beside the initial request from the clients, all the incoming data to the servers are only due to the acknowledgement messages sent by the clients. This incoming data reaches 16 KB/s at most, *i.e.,* the 3000-client experiment, which is only 0.0067% of the outgress rate.

For LT codes however, the average number of fully decoded segments is only 20 at its best. 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. Taking into account the partially received segments, the effective average of 20.1 implies that the streaming rate of 96 KB/s can effectively deliver no more than 85.7 KB/s of video content with our LT codes. With random linear network coding however, the whole 96 KB/s streaming rate is almost perfectly utilized. To be exact, linear dependent blocks happen but scarcely. We observe no more than 0.0033% of the received coded

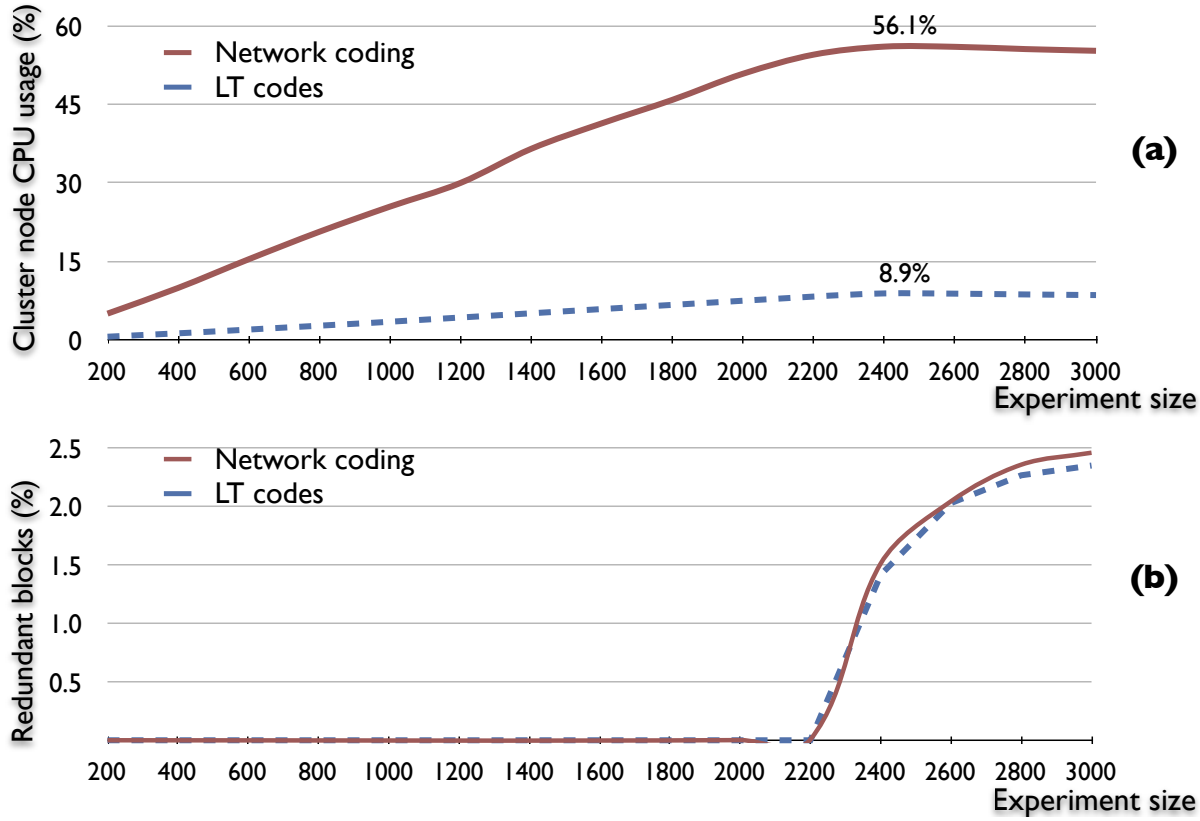


Figure 8.4: CPU usage and redundant blocks across different experiments.

blocks to be linearly dependent, *i.e.*, at most 0.0033% of the 96 KB/s streaming rate is wasted due to the linear dependence. Alternatively, the average number of linearly dependent blocks received by a client is only 0.097 in the whole 120-second period.

Fig. 8.4-(a) shows the main advantage of LT codes compared to network coding by reflecting the average CPU usage of the cluster nodes that emulate the client applications. As number of emulated clients increases with bigger experiments, the CPU usage (mainly dominated by the decoding load) increases almost linearly. This continues till it saturates beyond the 2200-client experiment as the servers link become the bottleneck and the rate of the received blocks no longer increase. LT codes show over 6 times advantage compared to random network codes. Such advantage could

be a decision factor in some setups, in particular embedded applications which use processors with low computation capability.

Fig. 8.4-(b) reflect the percentage of *redundant blocks* received by the clients. Redundant blocks are the on-the-fly coded blocks which belong to the previous segment but received after the segment is successful decoded. They essentially waste the network bandwidth so we want to keep them low. In all non-congested experiments, the redundant blocks are virtually zero. This is due to two factors: *first*, fast decoding mode of our accelerated decoding such that the acknowledgement message can be sent out quickly even before the payload of the last block is decoded; *second*, the link delay between our server and the cluster nodes is very low, only a few milliseconds due to the LAN setup, while the period of arrival of the coded messages is $\frac{4KB}{96KB} = 41.7$ ms. Obviously, the issue will be more pronounced in an Internet deployment where the link delay could go up to few 100's of millisecond. The number of redundant blocks increases from the 2200-client experiment as the link congestion leads to late delivery of the acknowledgement messages to the servers although the ACKs are issued much earlier by the client much earlier. This delay eventually leads to a backlog of coded blocks for each TCP stream at the servers.

For efficient deployment of two server applications on our physical server, we carefully pin the critical threads of each server application to different CPU cores. For example, the network thread, which does the actual network send/receive operations, experiences almost linear increase in its CPU usage as the number of active clients increases. Its CPU usage reaches over 90% from experiments of 2400 peers and more.

8.3 VoD on Smartphones

In our next set of experiments, we use an iPhone 3GS in a similar VoD setup as Sec. 8.2.1. We essentially use the same code stream as Blizzard can support iPhone OS, as discussed in Chapter. 6,

In the same VoD setup as Fig. 8.1, we add a wireless router to the LAN. Now our iPhone 3GS, like other clients, initiates a streaming session with both server applications, however, through a WiFi connection. The streaming session runs for 120 seconds at 96 KB/s. We proceed with both RLC and LT-based experiments. Our focus is on the performance of the iPhone 3GS.

8.3.1 RLC-based VoD on iPhone 3GS

First, the same (n=128, k=4096) RLC setup as Sec. 8.2.1 is deployed. We observe that only 21 segments are successfully decoded by the mobile device rather than the 22 segments received by other clients, running on the cluster nodes in the same LAN. With the iPhone 3GS, the average inter-segment arrival is 5.53 sec compared to the ideal 5.33 sec. Further investigation shows that the behavior is due to the latency of the *segment acknowledgement* messages which travel over the wireless medium. Basically, the latency of the wireless hop leads to delayed turnover from the current segment to the next. At the servers, a full RTT elapses from the send of the last coded block, the block that leads to successful decode of a segment, till receive of the segment acknowledgement message (ignoring the iPhone 3GS's *decoding latency* which is around 8 ms, much smaller than RTT). In comparison to the wireline clients, server's RTT to the mobile device is much higher, *e.g.*, in order of 150 ms against 2 ms. Because the RTT is much larger than the server's inter-block send interval of 41.7 ms, few extra coded blocks will be wasted while the the acknowledgment message is in transit from the client.

This is also seen in the higher number of redundant blocks, *i.e.,* 99 vs. almost zero for the wireline clients.

One way to overcome the wireless hop latency in our setup is to assume *implicit acknowledgment*, instead of explicit one, when a server has sent its share of coded block according to the streaming rate. The client now sends negative-acknowledgments (NACKs) to the server when it decodes a linear dependent block, or when it observes a server is falling behind its expected share of streaming rate, e.g., due to congestion. Not surprisingly, the NACK-based scheme compensates for the wireless hop latency and all 22 segments are successfully received.

The overall CPU usage of our application is around 16%. About 10% of this is due to network coding, per the results in Fig. 7.6, and the rest to our streaming protocol.

8.3.2 LT-based VoD on iPhone 3GS

As we observed with LT-based experiments of Sec. 8.2.2, our LT codes incur a network overhead of around 11.7%. This results in 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 content with LT codes.

However, the advantage of LT codes becomes obvious when we monitor the CPU usage which is only 7%. Less than 1% of this usage is due to LT decoding, and the rest to our protocol. This is significantly lower than the 16% usage of our application with RLC coding.

8.3.3 Real VoD streaming with playback

So far, all client applications in our experiments verified the validity of their decoded segments by comparing them against locally stored content. Now we also add video playback capability to our clients such that the content can be displayed in interactive sessions. Designing a full-fledged video playback application is beyond the main goals of this work. Ideally, we do not want to get involved with the mechanics and details of video playback of streaming video, which is a platform dependent operation. Our quickest solution, albeit its extra overhead, is to use a standard media player, such as QuickTime, and provide our video content to it through its "HTTP progressive download and playback" feature. By setting up our client application to function also as a lightweight HTTP server, the media player will read the video content from our client's HTTP server while the client receives the content from the VoD servers. As QuickTime supports progressive download, it playbacks the content as it becomes available. This leads to two applications at the client side, the streaming client and the player.

Now, the user requests a video content from our client application through the URL entered in QuickTime, for example. The client translates the HTTP GET request to a message sent to the server. When the client decodes a segment successfully, it will send the segment to QuickTime in response to the HTTP GET request it has received. On the iPhone OS, however, we use a slightly different scheme by embedding a "video controller" object in our application instead of a separate media player. These implementations were not straightforward. First, enabling a web-server in our Blizzard framework had to be done as a hack initially as Blizzard expected all incoming messages to carry Blizzard header. This requires us to exclude incoming HTTP GET requests from other messages deep down in the core of Blizzard. Second, while

QuickTime's HTTP GET requests come in order progressively seeking more data, after an initial query about the size, iPhone's video controller behaves differently. It sends subsequent HTTP GET requests when it doesn't receive a prompt response, *e.g.,* during segment buffering/decoding, on multiple TCP connections and confuses our light web-server. We had to use a "packet sniffer" tool, such as *Packet Peeper* [7], to analyze the requests and keep more states in our client such that it can properly satisfy the video controller's requests. Obviously, our playback scheme, based on HTTP progressive download, leads to extra CPU overhead but it has been the easiest way to integrate video playback into our system without the need to use more difficult lower-level mechanisms.

Now our client applications have two operating modes: **(1)** The old fashioned way, *e.g.,* as in Sec. 8.2, where the client receives the name of the video content as a parameter at the launch time, decodes and verifies the video segments as they arrive but and skips the playback. The segment verification is done through byte-by-byte comparison with a local copy of the content. **(**2) The new mode in which the user enters the content name as an URL, *e.g.,* in QuickTime from the client's listening port. The content names arrives to the client application as part of the HTTP GET request. When a segment is fully decoded by the client, it first verifies the content and then sends it to the player, *e.g.,* QuickTime, in response to the pending HTTP GET requests. As discussed, this mode requires careful implementation as unlike a regular file download, the content is coming on the fly and served to the player at the same time.

In a new experiment with RLC at $(n$=128, k=4096), the VoD servers stream real video clips encoded with H.264 at 768 kbps (96 KB/s), audio and video included. The CPU usage with playback is around 34% on the iPhone 3GS, which 17% of it is due to `mediaserverd` process which handles video decode and playback in iPhone OS, and the rest to our client app. On an Intel 1.83 GHz Core Duo system, the CPU usage

of the client application is 2% and the QuickTime player takes 12%.

8.4 P2P Live Streaming with Coding

Peer-to-peer (P2P) applications can benefit from coding even more than client-server systems. In a P2P system, each peer is served by multiple peers but there exist lots of dynamism in the lifetime of the serving peers. Unlike the servers, the peers come and go as they wish, and have varied degree of connectivity. As a result, traditional block reconciliation schemes have a tough time in time-sensitive systems such as streaming and usually resort to lengthy initial buffering to increase their robustness.

Now we depart from our client-server system and extend our experiments to a P2P fashion with the clients now potentially feeding other nodes. As the coding process takes very little CPU usage on typical modern processors, e.g., the iMac CPU usage in Table 8.1, we mainly focus on practicality of P2P systems on our mobile devices from computational complexity point of view. A VoD system requires all serving nodes, server or peer, to store the full or partial video contents. As this is not a likely scenario with current mobile devices, we change our setup to a P2P live streaming scenario instead, *i.e.,* all peers watch the same content. In such setup, the seed servers stream out a video stream to a number of clients which further propagate it to their neighbors. P2P systems have normally thousands of nodes participating with highly dynamic topologies. Here, we use a few nodes in static topologies mainly to investigate the performance of our least capable nodes, *i.e.,* our mobile devices, in a P2P setup. Beside decoding the incoming stream, a peer now can serve other peers resulting in higher coding load.

In live streaming, contrary to VoD, all peers follow the same timeline and the coded blocks can not be generated from previously existing segments on the local file sys-

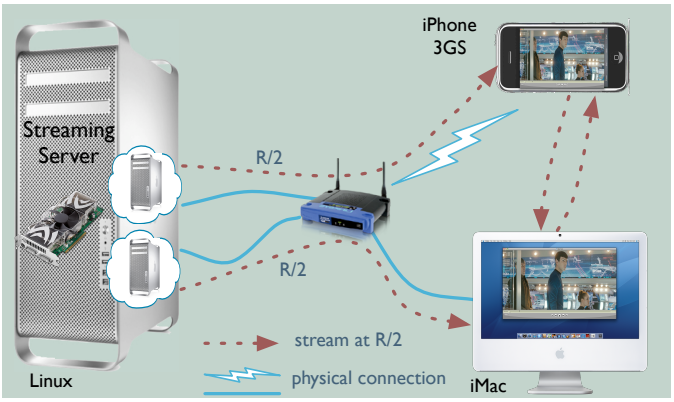


Figure 8.5: First prototype: Live P2P streaming with RLC.

tem, for example. A peer serves its neighbors although it has not fully decoded the segment yet. 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 same coded blocks from different paths. Instead, the peers "recode" their existing coded blocks although the segment is not fully decoded yet. Recoding, a unique feature of RLC which does not exist in LT or RS codes, generates new coded blocks from whatever blocks received so far.

In P2P systems, the server load drops significantly, *e.g.*, down to 10% to 20% of client-server systems. However, this allows the system size grow larger, in order of tens of thousands. In live streaming, especially, many peers are fully or partially fed by the server. Although the same content is intended for all the peers in the session, generating limited number of coded blocks and sending them to multiple peers would not work. This eventually leeds to heavily linear dependent blocks at downstream peers as the upstream peers have essentially recoded duplicate blocks. As a result, the servers should independently generate every coded block, *i.e.*, by using different codes in the code space. GPU-based encoding is still a prime choice for such demanding system.

P2P experiments:

In our first prototype, we use the simple topology shown in Fig. 8.5, with two servers in 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 content rate, $R/2$. Each peer receives the rest of the content through the other peer so it effectively decodes a full stream at rate R and recodes at a rate of $R/2$.

The iPhone 3GS and the iMac peer, with an Intel 1.83 GHz Core Duo processor, use our NEON/SSE2 accelerated recoding scheme of Sec. 7.3.1. 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 the 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 setup of Sec. 8.3.3, which is due to the extra recoding load of $R/2$.

Each peer receives the coded blocks from a server and a peer. The blocks originating from the servers identify their random coefficients simply through a random seed while the recoded blocks coming from the peers have to carry the whole 128 coefficients. The decoding process accepts both types of coded packets transparently, a new feature of RLC discussed in Sec. 7.3.1. Similar experiment with an iPod Touch, instead of the iPhone 3GS, results in delivery of 21 segments. This is directly due to the lower processing capability of the iPod Touch. Its CPU is 100% utilized as a result of the extra recoding load.

Now we also add an iPod Touch and iPhone 3G to our topology as shown in Fig. 8.6 with all three mobile devices on the same wireless LAN. As iPod Touch and iPhone 3G, running at 533 MHz and at 400 MHz respectively, both use the older ARM1176 processor without NEON support, they are not powerful to serve other peers beside

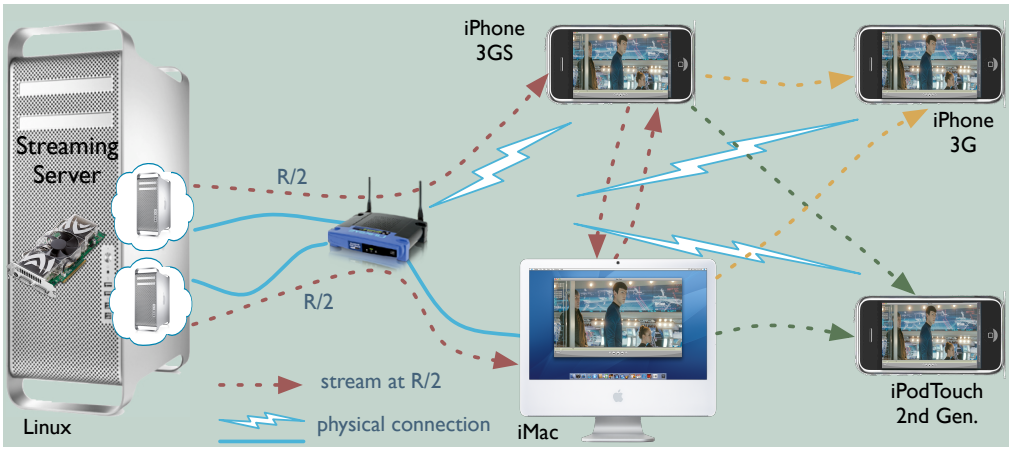


Figure 8.6: Second prototype: Live P2P streaming with RLC.

decoding and displaying their own content. That is why we deploy them as the sink nodes.

The iMac and iPhone 3GS nodes in the P2P setup of Fig. 8.6 now serve three downstream peers with recoded blocks effectively decoding at R and recoding at $3\dot{R}/2$. Table 8.1 summarizes number of successfully decoded segments for $(n = 128, k = 4096)$ and the less CPU intensive $(n = 64, k = 8192)$ setting, along with the CPU usage of our client app, excluding the usage due to playback which adds another 12% to 17%. $(n = 128, k = 4096)$ is too demanding for our least capable node, *i.e.*, 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 enough CPU power for a RLC setting of $(n = 64, k = 8192)$. However, having larger block sizes as $k = 8192$ has other trade-offs, *e.g.*, leads to bigger waste when block overshoot happens at segment transition time. Obviously a desktop PC as the iMac or a recent handheld device as iPhone 3GS are in good position to serve even more peers while the year-old iPod Touch and iPhone 3G better fit as consumers without serving others. Overall, we can conclude

Table 8.1: P2P experimental results.

	downstream	$(n = 64, k = 8KB)$		$(n = 128, k = 4KB)$	
Peer	nodes	CPU usage	Segments	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

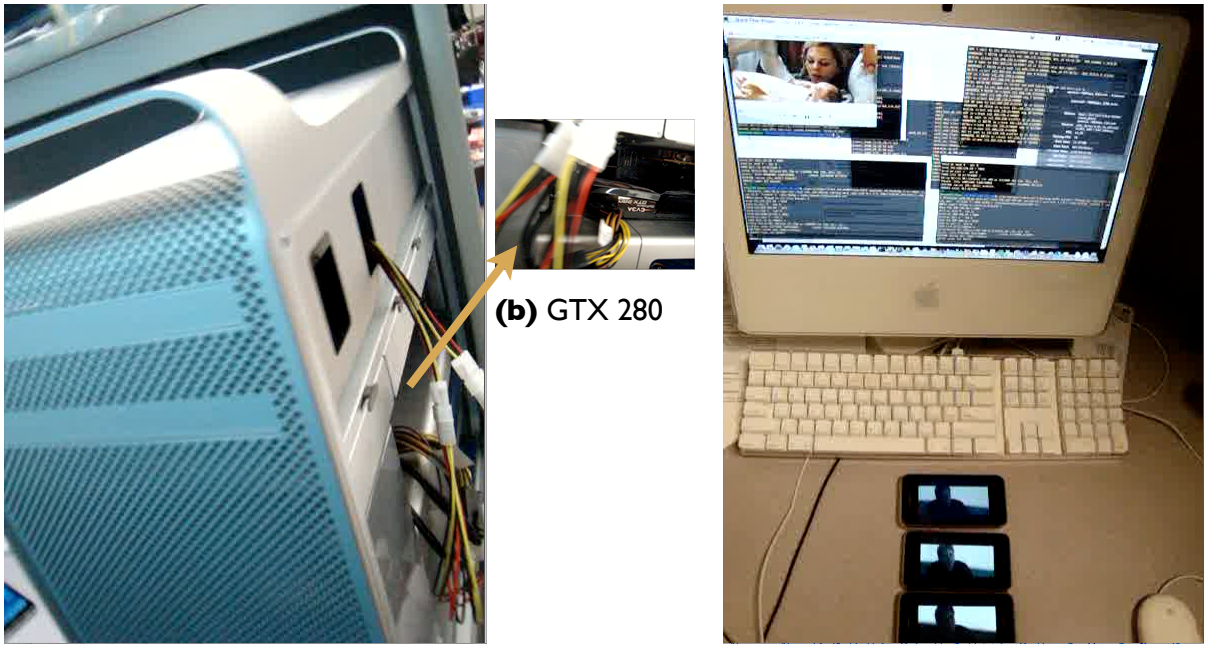
that with Tenor, P2P systems with coding support can be realistically deployed even on today's mobile devices although the computational capability of the device has to be considered when the mesh topology is formed.

So far, all our RLC experiments used fully dense codes, with non-zero coefficients. Similar to our discussion in Sec. 5.2.3, the computation load can be brought even lower with sparser codes, in expense of slight increase in the probability of linear dependency.

Finally, Fig. 8.7 show a snapshot of our second prototype, depicted in Fig. 8.6, playing a clip from *Star Trek*. The GPU-based streaming server is shown at the left and all peers at the right of the figure.

8.5 Summary

This chapter presented our coding-based on-demand and live media streaming systems from a GPU-based server to up to 3000 emulated nodes, and to iPhone devices with actual video playback. Our experiments, closely following the real-life examples, offered excellent illustrations of Tenor components in action, and their benefits



(a) Linux-based streaming server

(c) The peers

Figure 8.7: Snapshot of coding-based live P2P streaming with the setup shown in Fig. 8.6. The peers in **(c)** (from top to bottom) are an iMac desktop, iPhone 3GS, iPod Touch, and iPhone 3G.

in rapid system development. Tenor toolkit, our scalable coding facility, along with Blizzard, our scaleable network framework, particularly allowed us to push the performance of our cross-platform coding toolkit to the limits allowed by off-the-shelf hardware. With the help of GPUs, we were able to saturate two Gigabit Ethernet interfaces with coding performed in real time.

Throughout this work, our experiments have shown that off-the-shelf hardware is sufficiently sophisticated to bear the computational load of coding tasks in real-life scenarios.

Chapter 9

Concluding Remarks

9.1 Conclusions

It has been well recognized that innovative coding techniques has the "theoretical potential" to improve "network performance". With coding, the system's "resilience" to node departures and packet losses significantly improve. Also, multiple servers can simultaneously serve a single receiver with a substantially "simplified design of block reconciliation protocols". These properties are ideal for content distribution, both as media streaming or bulk data. Several coding techniques, such as fountain codes and network coding, have gained popularity for this purposes over the recent years. 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 serve thousands of clients, and in "smartphone" devices where hardware limitations prevail. To date, however, there has been no commercial real-world systems reported in

the literature that take advantage of the power of network coding.

This research has represented the first attempt towards a high-performance design and implementation of network coding. The objective of this work has been to explore the computational limits of network coding in off-the-shelf modern processors, and to provide a solid reference implementation to facilitate commercial deployment of network coding. The final work, packaged as a toolkit code-named Tenor, includes high-performance implementations of a number of coding techniques: random linear network coding (RLC), fountain codes (LT codes), and Reed-Solomon (RS) codes in CPUs (single and multi core(s) for both Intel x86 and IBM POWER families), GPUs (single and multiple), and mobile/embedded devices based on ARMv6 and ARMv7 cores. Tenor is cross-platform with support on Linux, Windows, Mac OS X, and iPhone OS, and supports both 32-bit and 64-bit platforms.

Tenor has strived to push the performance of cross-platform coding toolkit to the limits allowed by off-the-shelf hardware. With achieving coding rates ranging from few MB/s for smartphones, to hundreds of MB/s for general-purpose CPUs, and up to thousands of MB/s for GPU-based servers in typical coding settings required for streaming and content distribution applications, Tenor can be deployed as the *coding facility* across all nodes of client-server and peer-to-peer systems.

In order to validate the effectiveness of the Tenor toolkit, we have built coding-based on-demand and live media streaming systems with GPU-based servers, thousands of clients emulated on a cluster of computers, and a small number of actual iPhone devices. To facilitate deployment of such large experiments, we have developed *Blizzard*, a high-performance framework, with the main goals of: 1) emulating hundreds of client/peer applications on each physical node; 2) facilitating scalable servers that can efficiently communicate with thousands of clients. Our experiences 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 mobile devices.

Have we reached an era when coding techniques can be performed in networked systems and applications without serious concerns of their practical complexities? As this dissertation has thoroughly explored various avenues, the answer is pleasantly yes.

9.2 Future Directions

This work has mainly focused on the coding framework and the computation performance aspects of the system have been of our main interest. The deployed network protocols were less emphasized and simple protocols were developed as proof-of-concept deployment of the coding framework. For example, the prototypes were deployed in a LAN with TCP-based delivery. In the Internet, however, UDP-based delivery has better penetration properties but requires more advanced protocols. An Internet deployment, involving thousands of "real peers", will require significantly more robust protocols, *e.g.,* to be able to sustain dynamic peers, delay and bandwidth variations in the communication links.

Future directions of this research can change gears directly towards developing "sophisticated coding-based protocols" for different application scenarios. They can include coding-based P2P, peer-assisted, and client-server systems for applications such as *VoD media streaming*, *live media streaming*, and *bulk content-distribution*, *e.g.,* for Internet-based live and VoD systems, IPTV broadcast and VoD services, distribution of large software releases, and HD movies for offline viewing.

Efficient use of coding in P2P and the traditional client-server systems can sig-

nificantly improve the performance of such systems. As an example, by lowering the initial buffering delay, P2P live streaming can be deployed in highly scalable systems. Similarly, with coding, quality-of-service of client-server VoD systems can be improved when bandwidth varies.

Our solutions so far have targeted application-layer protocols by taking advantage of application processors or GPUs for coding purposes. However, not all potential target devices carry sophisticated processing cores, *e.g.,* with SIMD support. With more practical protocols that can justify the use of coding in wider application domains, new interesting problems will arise. For example, what is the most power-efficient design to perform coding on lower-end devices? Do we need to require better application processors or promote specialized hardware? A direct beneficiary of such designs will be wireless networks, where numerous theoretical works have shown the benefits of coding. To avoid the computational complexity of network coding, however, they have resorted to simpler but less efficient schemes, such as XOR coding. As physical-layer protocols running in the baseband processors rarely have access to sophisticated processing cores, power-efficient hardware designs of network coding with various levels of hardware/software co-design, from custom to accelerated designs, are interesting problems to explore.

On the IPTV front, the popularity of VoD contents has been increasing continuously, in contrast to live contents. Higher demand for VoD directly translates to higher demand on the servers and communication links. Because "full upgrade" to high bandwidth fiber is not in sight in many communities, P2P VoD systems emerge as a viable alternative to the traditional client-server VoD such that set-top boxes are used as peers to aid more efficient video dissemination. Such P2P setups are highly correlated to Internet-based P2P systems and can benefit from coding. Further, coding-based protocols can consider practical cross-layer issues more intelligently such that better

overall quality is achieved rather than the traditionally separate network delivery and video coding layers as [36, 37].

This research has tried to promote the development of new coding-based systems and protocols through a comprehensive toolkit with coding implementations that are not just reference implementations. Instead, they have attained high-performance and flexibility to find widespread adoption. The future research will complete Tenor's coding framework with "real-life protocols" and build more advanced and realistic prototype systems. By making efficient coding more accessible to both the academia and the industry, this research aims to make coding ubiquitously available in a variety of content delivery systems that can benefit from it.

Bibliography

[1] *Linux Manual Pages for* epoll *Interface.*

[2] *Linux Manual Pages for* GDB: The GNU Project Debugger.

[3] *Linux Manual Pages for* poll *API.*

[4] *Linux Manual Pages for* select *API.*

[5] *Linux Manual Pages for* top *Command.*

[6] *Linux Manual Pages for* ulimit *Command.*

[7] *Packet Peeper* Network Protocol Analyzer. SourceForge.net.

[8] *RSCODE project*: An Implementation of a Reed-Solomon Error Correction Algorithm. SourceForge.net.

[9] *iQua Research Group, University of Toronto*, http://iqua.ece.toronto.edu.

[10] *BitTorrent*, http://www.bittorrent.com/.

[11] *PPLive, Shanghai SynaCast Media Tech Co, China*, http://www.pplive.com/en/about.html.

[12] *UUSee Inc., Beijing, China*, http://www.uusee.com.

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

[14] Apple Inc. *Mac OS X Reference Library Manual Pages for* kqueue *Interface.*

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

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

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

[18] A. C. Begen, N. Glazebrook, and W. Ver Steeg. A Unified Approach for Repairing Packet Loss and Accelerating Channel Changes in Multicast IPTV. In *Proc. IEEE Consumer Communications and Networking Conf. (CCNC)*, Jan. 2009.

[19] A. C. Begen, N. Glazebrook, and W. Ver Steeg. Reducing Channel-Change Times with the Real-Time Transport Protocol. *IEEE Internet Computing*, 13(3), May/June 2009.

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

[21] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proc. of ACM SIGCOMM*, pages 56–67, 1998.

[22] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, October 2003.

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

[24] Y. R. Chen, R. Jana, D. Stern, B. Wei, M. Yang, and H. Sun. Zebroid: Using IPTV Data to Support Peer-Assisted VoD Content Delivery. In *Proc. of the 19th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2009)*, June 2009.

[25] P. Chou, Y. Wu, and K. Jain. Practical Network Coding. In *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.

[26] NVIDIA Corporation. *NVIDIA CUDA: Programming Guide, Version 2.0*, July 2008.

[27] NVIDIA Corporation. *NVIDIA CUDA: Reference Manual, Version 2.0*, June 2008.

[28] R. Doverspike, G. Li, K. Oikonomou, K.K. Ramakrishnan, R. K. Sinha, D. Wang, and C. Chase. Designing a Reliable IPTV Network. *IEEE Internet Computing*, 13(3), May/June 2009.

[29] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proc. of ACM SIGCOMM*, August 2000.

[30] C. Fragouli, J.-Y. Le Boudec, and J. Widmer. Network Coding: An Instant Primer. *ACM SIGCOMM Computer Communication Review*, 36, January 2006.

[31] Freescale Semiconductor. *AltiVec Technology Programming Interface Manual*, June 1999.

[32] C. Gkantsidis, J. Miller, and P. Rodriguez. Anatomy of a P2P Content Distribution System with Network Coding. In *Proc. of the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[33] C. Gkantsidis, J. Miller, and P. Rodriguez. Comprehensive View of a Live Network Coding P2P System. In *ACM Internet Measurement Conference (IMC 2006)*, 2006.

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

[35] T. Granlund and P. L. Montgomery. Division by Invariant Integers Using Multiplication. In *ACM SIGPLAN Notices*, volume 29, pages 61–72, June 1994.

[36] J. Greengrass, J. Evans, and A. C. Begen. Not All Packets Are Equal, Part I. *IEEE Internet Computing*, 13(1), Jan/Feb 2009.

[37] J. Greengrass, J. Evans, and A. C. Begen. Not All Packets Are Equal, Part II. *IEEE Internet Computing*, 13(2), Mar/Apr 2009.

[38] T. R. Halfhill. Parallel Processing With CUDA. *Microprocessor Report*, January 2008.

[39] 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.

[40] Intel Corporation. *IA-32 Intel® Architecture IA-32 Intel Architecture Optimization Reference Manual*, April 2006.

[41] Intel Corporation. *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture*, April 2008.

[42] Sachin Katti, Dina Katabi, Hari Balakrishnan, and Muriel Medard. Symbol-level Network Coding for Wireless Mesh Networks. In *ACM SIGCOMM 2008*.

[43] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Medard, and Jon Crowcroft. XORs in The Air: Practical Wireless Network Coding. In *Proc. of ACM SIGCOMM*, 2006.

[44] Dan Kegel. The C10K Problem. http://www.kegel.com/c10k.html.

[45] Khronos Group. *OpenCL 1.0: The Open Standard for Parallel Programming of Heterogeneous Systems*, Oct. 2009.

[46] R. Koetter and M. Medard. An Algebraic Approach to Network Coding. *IEEE/ACM Transactions on Networking*, 11(5):782–795, October 2003.

[47] Jonathan Lemon. *Kqueue: A Generic and Scalable Event Notification Facility*. FreeBSD Project, 2001.

[48] B. Li, J. Guo, and M. Wang. iOverlay: A Lightweight Middleware Infrastructure for Overlay Application Implementations. In *Proc. of the Fifth ACM/IFIP/USENIX International Middleware Conference (Middleware 2004)*, October 2004.

[49] Jin Li. The Efficient Implementation of Reed-Solomon High Rate Erasure Resilient Codes. In *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP '05)*, pages 1097–1100, March 2005.

[50] S. Y. R. Li, R. W. Yeung, and N. Cai. Linear Network Coding. *IEEE Transactions on Information Theory*, 49, 2003.

[51] Z. Li, X. Zhu, A. C. Begen, and B. Girod. Peer-assisted Packet Loss Repair For IPTV Video Multicast. In *Proc. of ACM Multimedia*, October 2009.

[52] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. In *IEEE MICRO*, volume 28, March-April 2008.

[53] 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.

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

[55] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[56] P. Maymounkov, N. Harvey, and D. Lun. Methods for Efficient Network Coding. In *Proc. of 44th Annual Allerton Conference on Comm., Control, and Computing*, Sep. 2006.

[57] Carl Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, 2001.

[58] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2), 2008.

[59] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. of Annual Usenix Technical Conference*, June 1999.

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

[61] R. Roth. *Introduction to Coding Theory, 1st Ed.* Cambridge University Press, 2006.

[62] P. Sanders, S. Egner, and L. Tolhuizen. Polynomial Time Algorithm for Network Information Flow. In *Proc. of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2003)*, June 2003.

[63] H. Shojania and B. Li. *Tenor:* Making Coding Practical from Servers to Smartphones. In *Submitted to USENIX NSDI 2010*.

[64] H. Shojania and B. Li. Parallelized Network Coding With Hardware Acceleration. In *Proc. of the 15th IEEE International Workshop on Quality of Service (IWQoS)*, 2007.

[65] H. Shojania and B. Li. Pushing the Envelope: Extreme Network Coding on the GPU. In *Proc. of the 29th International Conference on Distributed Computing Systems (ICDCS '09)*, June 2009.

[66] H. Shojania and B. Li. Random Network Coding on the iPhone: Fact or Fiction? In *Proc. of the 19th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2009)*, June 2009.

[67] H. Shojania, B. Li, and Xin Wang. Nuclei: Graphics-accelerated Many-core Network Coding. In *Proc. of the 29th IEEE Conference on Computer Communications (INFOCOM 2009)*, April 2009.

[68] Sun Microsystems. *Java Tutorial Lesson: All About Sockets*.

[69] Neal Wagner. *The Laws of Cryptography with Java Code*. http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf, 2003.

[70] M. Wang, H. Shojania, and B. Li. Crystal: An Emulation Framework for Practical Peer-to-Peer Multimedia Streaming Systems. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS'08)*, 2008.

[71] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[72] Wikipedia. Finite Field. http://en.wikipedia.org/wiki/Finite_field.

[73] Wikipedia. Network Coding. http://en.wikipedia.org/wiki/Network_coding.

[74] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. CoolStreaming/DONet: A Data-Driven Overlay Network for Peer-to-Peer Live Media Streaming. In *IEEE INFOCOM*, March 2005.