BigStation: Enabling Scalable Real-time Signal Processing in Large MU-MIMO Systems

Qing Yang [*]
Microsoft Research Asia
and CUHK

Xiaoxiao Li [*]
Microsoft Research Asia
and Tsinghua University

Hongyi Yao [*]
Microsoft Research Asia
and USTC

Ji Fang [*]
Microsoft Research Asia
and BJTU

Kun Tan
Microsoft Research Asia

Wenjun Hu
Microsoft Research Asia

Jiansong Zhang
Microsoft Research Asia

Yongguang Zhang
Microsoft Research Asia

yq010@ie.cuhk.edu.hk {v-lxiaox, v-hoya, v-fangji, kuntan, wenjun, jiazhang, ygz}@microsoft.com

ABSTRACT

Multi-user multiple-input multiple-output (MU-MIMO) is the latest communication technology that promises to linearly increase the wireless capacity by deploying more antennas on access points (APs). However, the large number of MIMO antennas will generate a huge amount of digital signal samples in real time. This imposes a grand challenge on the AP design by multiplying the computation and the I/O requirements to process the digital samples. This paper presents BigStation, a scalable architecture that enables real-time signal processing in large-scale MIMO systems which may have tens or hundreds of antennas. Our strategy to scale is to extensively parallelize the MU-MIMO processing on many simple and low-cost commodity computing devices. Our design can incrementally support more antennas by proportionally adding more computing devices. To reduce the overall processing latency, which is a critical constraint for wireless communication, we parallelize the MU-MIMO processing with a distributed pipeline based on its computation and communication patterns. At each stage of the pipeline, we further use data partitioning and computation partitioning to increase the processing speed. As a proof of concept, we have built a BigStation prototype based on commodity PC servers and standard Ethernet switches. Our prototype employs 15 PC servers and can support real-time processing of 12 software radio antennas. Our results show that the BigStation architecture is able to scale to tens to hundreds of antennas. With 12 antennas, our BigStation prototype can increase wireless capacity by 6.8× with a low mean processing delay of 860μs. While this latency is not yet low enough for the 802.11 MAC, it already satisfies the real-time requirements of many existing wireless standards, *e.g.*, LTE and WCDMA.

[*]This work was performed while Qing Yang, Xiaoxiao Li, Hongyi Yao, and Ji Fang were research interns at Microsoft Research Asia.

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.

Categories and Subject Descriptors

C.2.1 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Architecture and Design—*Wireless communication*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

BigStation, MU-MIMO, software radio, parallel signal processing

1. INTRODUCTION

The proliferation of mobile devices like tablets and smartphones, along with many data-intensive applications, has created tremendous demands for high-speed wireless communication. It has been predicted that the amount of net traffic carried on wireless (*e.g.*, Wi-Fi and 3G/4G) will exceed the amount of wired traffic by 2015 [9]. To satisfy this demand, next-generation wireless networks need to be engineered with a capacity matching their wired counterpart, *e.g.*, to deliver giga-bits per second of throughput to each network user just like existing Ethernet.

One way to get more wireless capacity is to use more spectrum. However, it is well understood that wireless spectrum is a scarce resource and also shared among all wireless transmitters. Therefore, the capacity improvement from adding spectrum is still limited, and it is hard to keep up with traffic demands. A more promising approach is to increase spectral efficiency with *Multi-user MIMO (MU-MIMO)*. MU-MIMO allows multiple users to transmit signals concurrently. With multiple antennas, MU-MIMO access point (AP) will mesh the digital samples from all antennas together and jointly decode data for each user (Figure 1(a)). By adding more antennas to the APs, MU-MIMO has the potential to increase wireless capacity significantly – linearly with the number of deployed antennas. Indeed, the whole wireless industry is moving in this direction. For example, the 4G (LTE) standard [1] has defined MU-MIMO operations with eight antennas at the basestation, and the new Wi-Fi standard, IEEE 802.11ac, also specifies up to eight antennas to provide a 1 Gbps data rate to up to four users simultaneously. Recent literature further suggests the possibility of even larger-scale MU-MIMO systems with tens to hundreds of antennas to support tens of concurrent users [16–18]. However, how to build such a powerful AP and how well MU-MIMO may work in practice remain open research questions.

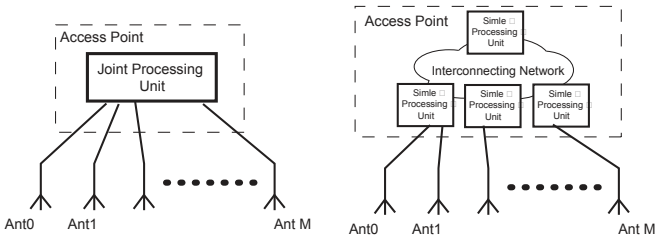


Figure 1: An AP with many (MU-)MIMO antennas. (a) Traditional AP design: A central unit jointly processes all sample streams from all antennas. (b) BigStation: Baseband sample streams and computation are parallelized among many simple processing units.

As the number of antennas on the AP increases, the demand for MU-MIMO processing grows accordingly, which imposes a huge challenge on the AP design. For example, an 802.11ac AP uses two MIMO antennas and a 160 MHz wide channel to support a 1 Gbps link to one user. To support 20 simultaneous 1 Gbps users, the same AP would need to have 40 antennas [1]. Collectively, these 40 antennas generate digital samples at 200 Gbps in real time, which would require the AP to have a processing capability of multiple trillions of operations per second for MU-MIMO decoding (Section 2.2). This, however, is far beyond the capability of any existing single computing device (*i.e.*, single processor or acceleration chip). Therefore, a scalable MU-MIMO system should explore parallelism in signal processing and employ an architecture to distribute the computation modules effectively among a number of simple processing units.

In this research, we propose such a scalable architecture, named BigStation, which extensively parallelizes the MU-MIMO processing across many simple and low-cost commodity computing devices (Figure 1(b)). Our design can incrementally scale out to support more MIMO antennas by proportionally adding more processing units and the interconnecting bandwidth. To reduce the overall processing latency, which is critical for wireless communication, we parallelize the MU-MIMO processing with a distributed pipeline based on its computation and communication patterns. At each stage of the pipeline, we further use *data partitioning* and *computation partitioning* to exploit the parallelism inside a processing unit as well as across multiple units.

As a proof of concept, we present the design and implementation of BigStation based on commodity PC servers and standard Ethernet switches. Besides serving as an instantiation of our scalable architecture, our exploration also provides a first study toward a large-scale software radio based centralized wireless infrastructure [2, 8], which holds the promise of reducing the cost and improving the efficiency of existing wireless networks. We have built a BigStation prototype with 15 PC servers connected to an 1/10Gb Ethernet switch, and software radio front-ends supporting 12 antennas. We have evaluated our prototype, and our main findings are:

- The BigStation architecture is scalable. Our benchmarks and analysis show that BigStation readily supports a few dozens of antennas with current mid-range PC servers. With more powerful high-end servers, we can scale BigStation to 100 antennas.
- With the distributed pipeline architecture, BigStation has a low mean end-to-end processing delay of 860 μs. While this latency may not be low enough to implement 802.11-type MAC layer acknowledgment, it already satisfies the real-time requirements for many existing wireless standards, *e.g.*, LTE and WCDMA.
- The capacity of a MU-MIMO system does not scale linearly if the number of the AP antennas (M) equals the sum of client antennas (N). The capacity may even decrease as N grows large due to wireless channel hardening [11]. However, the MU-MIMO capacity does scale linearly if the AP has more antennas ($M > N$). With 12 antennas, our BigStation prototype can support 9 ($M = 1.4N$) concurrent transmitters and increase the wireless capacity by 6.8× compared to a single-antenna radio.

The rest of the paper is organized as follows: Section 2 outlines MU-MIMO background and the system design challenges when the number of antennas grows large. We discuss our parallelization principles in Section 3. Section 4 presents the distributed pipeline architecture of BigStation. In Section 5, we apply our BigStation design principles in a system based on PC servers. Section 6 presents the implementation details and Section 7 evaluates our prototype. We discuss related work in Section 8 and Section 9 concludes the paper.

2. BACKGROUND

2.1 Multi-user MIMO

In a MU-MIMO system, a multi-antenna access point (AP) can provide simultaneous links to many independent clients over the shared wireless medium. Let M denote the number of antennas at the AP and N the total aggregate number of antennas from all active clients. As long as $N \leq M$, the MU-MIMO system can support up to N concurrent data streams, potentially achieving N times the capacity gain over single-antenna systems for the same channel width. In contrast, a single-user MIMO system can serve only one client at a time, where the capacity gain is bounded by the number of antennas at the client, which can be far smaller than N. We assume the MU-MIMO system is based on OFDM, the most popular wireless communication technology. OFDM subdivides the channel into many narrow orthogonal subcarriers. Since each subcarrier is narrow, its channel can be considered flat fading.

In the uplink direction, all N antennas at the clients will simultaneously transmit symbols to the AP. These concurrent symbols add up at each of the M receiving antennas. Let $y_i^k(t)$ denote the received signal on antenna i on subcarrier k. Let $Y^k(t)$ be the vector $[y_1^k, y_2^k, \cdots, y_M^k]^T$. We have

$$Y^k(t) = H^k X^k(t),$$

where $X^k(t) = [x_1^k, x_2^k, \cdots, x_N^k]^T$ is the vector of transmitted symbols, and H^k is the $M \times N$ channel matrix on subcarrier k. Hereafter, for simplicity, we may omit the superscript k when there is no ambiguity. To decode each x_j, the AP needs to first compute the pseudo-inverse of H, $H^+ = (H^*H)^{-1}H^*$. Then, the AP should multiply H^+ with the received signal vector $Y(t)$ to obtain the transmitted symbols $X(t)$, as $H^+H = I$. This operation is called *spatial demultiplexing*, where receiver antennas collectively recover each symbol stream transmitted by each sender antenna. Finally, the AP will feed these spatial streams through a channel decoder, *e.g.*, Viterbi or Turbo decoder, to decode the information bits.

[1] As we will show later, the AP may need even more than 40 antennas to avoid channel hardening [11].

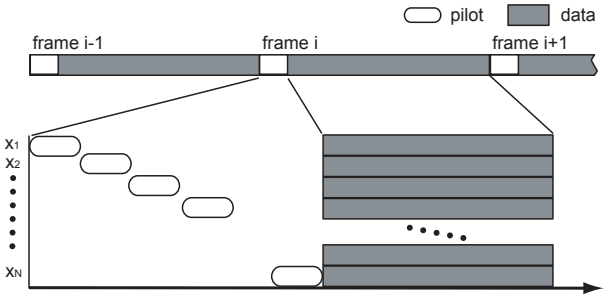


Figure 2: A typical frame format for MU-MIMO transmissions. Each transmitter will send an orthogonal pilot symbol for wireless channel measurement. After that, all data symbols are transmitted together.

In the downlink direction, the M AP antennas will simultaneously transmit to the N antennas at the clients. Similar to the uplink operations but reversed, the AP first encodes the information bits with a forward error correction (FEC) code (channel encoding). It also computes a pseudo-inverse of the channel matrix H, $H^+ = H^*(HH^*)^{-1}$. Then, for every outgoing symbol (after channel encoding), the AP performs *precoding* by multiplying the symbol with the channel inverse,

$$X'(t) = H^+X(t).$$

and transmits the precoded symbols instead. This way, the N receiving antennas will just receive the data symbols targeted at themselves, while the interference is canceled out as

$$Y(t) = HX'(t) = HH^+X(t) = X(t).$$

This decoding and precoding method is called *zero-forcing*, as it effectively removes the mutual interference among concurrent transmissions.

In a practical MU-MIMO system, transmissions are grouped into frames, as shown in Figure 2. Each frame has a preamble before the data symbols. During this preamble portion, each sender can transmit a known pilot symbol orthogonally for the receivers to learn the wireless channel H.

Although MU-MIMO is well-understood in information theory, only small-scale MU-MIMO (*i.e.*, $M < 10$) systems have been implemented recently [6, 19] and adopted in wireless standards, *e.g.*, 802.11ac [4], LTE [1], and WiMax. However, large-scale MU-MIMO setups, with tens or even hundreds of antennas, remain largely unexplored. One important reason is that, as the number of antennas increases, the computation requirement grows multiplicatively, and may far exceed the capability of a single computing device. Indeed, this challenge has motivated previous works [12, 18] to settle for a simpler algorithm (conjugate processing) to scale, but at the expense of significant performance loss. For example, as reported in [18], the capacity may be reduced by a factor of 4 compared to when using the aforementioned zero-forcing method. In the next subsection, we first outline the challenges for real-time MU-MIMO processing, and then we present our approach to address this challenge by parallelizing the zero-forcing MU-MIMO computation across many processing units.

2.2 Challenges for real-time MU-MIMO processing

From the above discussion, we can see that a MU-MIMO AP needs to handle a significant amount of signal processing. With more concurrent users supported, the computation load and the internal datapath bandwidth requirement grow multiplicatively. In the following, we study the magnitude of such computation and communication requirements in a MU-MIMO AP.

Inside a high-speed digital wireless communication system, each antenna will generate (or consume) a fairly large amount of high-fidelity digital samples. Depending on the channel width and the physical layer (PHY) design, this number may range from 416 Mbps (802.11g, 20 MHz channel) to 5 Gbps (802.11ac, 160 MHz channel), per antenna. If the AP has M antennas, the aggregate data rate of digital samples will be simply multiplied by M. For example, 802.11ac uses 2×2 MIMO over 160 MHz of wireless spectrum to deliver one giga-bit-per-second (Gbps) wireless link. To support 20 concurrent 1 Gbps wireless users, a MU-MIMO AP needs to have at least 40 antennas. The aggregate volume for the sample streams would exceed 200 Gbps.

Such a large amount of digital samples requires substantial computation to process. Let R be the digital sample rate per antenna, and W be the number of subcarriers in the wireless channel. Based on the MU-MIMO operations described previously, we can estimate the computational complexity as follows. Clearly, since we need to support N data streams, the computation complexity for the channel decoder is $O(NR)$. For spatial demultiplexing, which needs to compute a matrix vector multiplication, the complexity is $O(NMR)$. The complexity for channel inversion, which must be calculated for every frame, is $O(MN^2W/T_f)$, where T_f is the transmission time of a frame. To get a sense of how many cycles are actually needed, we can do some back-of-the-envelope calculations for the above 40-antenna MU-MIMO case. According to the 802.11ac specification, when the channel width is 160 MHz, we have $W = 468$ and $R \approx 5$ Gbps. We also have $N = M = 40$, to support 20 concurrent 1 Gbps users. On the uplink, decoding a single stream using a Viterbi decoder takes approximately 137 GOPS (operations per second) by one estimate [14]. Multiplying by the number of antennas, the channel decoding part requires approximately 5.5 TOPS in total. Spatial demultiplexing adds another 1.5 TOPS. If we assume a 2 ms frame transmission time, channel inversion needs another 269 GOPS. Adding them up, the AP will need a processing capability to support as many as 7.27 TOPS! The downlink is less demanding, but the estimate is still 1.7 TOPS.

These numbers are simply astronomical, far beyond the capability of a single piece of processing hardware today (or even in the near future given existing technology trends). As one data point for reference, state-of-the-art multi-core CPUs or DSPs on the market can only process on the order of 50 GOPS per chip. To build a MU-MIMO system to handle this kind of computation load and possibly scale up further, it will require serious thinking in the signal processing architecture and the system design, as well as non-trivial engineering efforts.

3. DESIGN PRINCIPLES

Our goal is to build a scalable AP architecture that can support a large number of MIMO antennas, say tens or hundreds. Our strategy to scale is to parallelize the MU-MIMO processing into many small pieces, each of which can fit into an available computing device (or a processing unit). As far as our architecture is concerned, such a computing device can be a general purpose processor (*i.e.*, CPU), DSP, FPGA, or even a custom-designed ASIC. However, as we will show later, the specific properties of MU-MIMO and wireless communications have placed fundamental constraints on how we can parallelize the processing. In this section, we start with a set of principles that guide our practical system design.

Distributed pipeline. One simple idea for parallelization is to partition the sample streams into blocks and send each block to a dif-

ferent processing unit for processing. Let us assume the sample block lasts for t_b seconds, and the module needs t_p time to process. Then, deploying $\lceil \frac{t_p}{t_b} \rceil$ processing units will provide enough capacity to process all sample streams. This simple solution sounds reasonable, as wireless transmissions are naturally separated into frames (Figure 2). In practice, however, sending a whole frame to a single processing unit can introduce a prohibitively long delay for real-time wireless communication. Taking the example we have used in Section 2.2, it can easily take 1s to process a single frame with a top-of-the-line processor (*e.g.*, an Intel 8-core CPU clocked at 3 GHz), while actual wireless protocols require a processing latency two or three orders of magnitude smaller, *e.g.*, 10 ms delay for WCDMA and 3 ms for LTE.

Given this delay concern, our first design principle should be: *The processing functions of a frame should be parallelized in a distributed pipeline*. At each stage of the pipeline, the computation is further distributed across multiple processing units. Since each unit may execute only a small portion of the computing task, the overall processing time is significantly reduced.

Data partitioning and computation partitioning. The ideal way to parallelize computation among multiple processing units at a given stage of the pipeline is *data partitioning*, which divides the digital samples into multiple independent data sets. Each data set is then sent to a distinct unit to be processed individually, until the results from all sets reach the barrier at the next pipeline stage and are synchronized there. In a MU-MIMO system, there are many opportunities to exploit parallelism through data partitioning: At the *channel inversion* and *spatial demultiplexing* stages, we can partition samples by subcarrier, each of which can be processed individually; while at the *channel decoding* stage, symbols belonging to one spatial stream are grouped together and processed by a separate channel decoder.

Data partitioning itself can provide enough parallelism as long as the individual data set can be processed in one processing unit in real time. However, if the computation for one data unit is still too much (*e.g.*, to invert a very large channel matrix with a dimension of 500×500), further *computation partitioning* should be performed to take advantage of any parallelization opportunity within the processing unit or across multiple units. In Section 5, we will discuss in detail how to partition the computation of MU-MIMO signal processing algorithms.

4. DISTRIBUTED PIPELINE

Figure 3 illustrates the distributed pipeline architecture of BigStation. The first stage in the pipeline consists of *front-side modules* (FS) which are directly attached to the antennas. The second stage is composed of *channel inversion modules* (CI) where the pseudo-inverse of the channel matrix is computed. The third and fourth stages include *spatial demultiplexing modules* (SD) and *channel decoding modules* (CD) for the uplink, or a set of *precoding modules* (PR) and *channel encoding modules* (CE) for the downlink. Each stage may further comprise multiple modules for parallel processing. The actual number of modules at one stage is flexible and configurable, depending on the computational load – a function of the MU-MIMO configuration and the signal processing algorithms used – as well as the processing capability of the hardware running these modules. Therefore, the architecture is highly scalable and agnostic to implementation choices. Each stage can scale horizontally, as the number of antennas or clients increases, or as the MU-MIMO algorithms evolve to be more computationally demanding.

In the uplink pipeline (Figure 3(a)), the FS module performs time synchronization to find the starting point of a frame. Then,

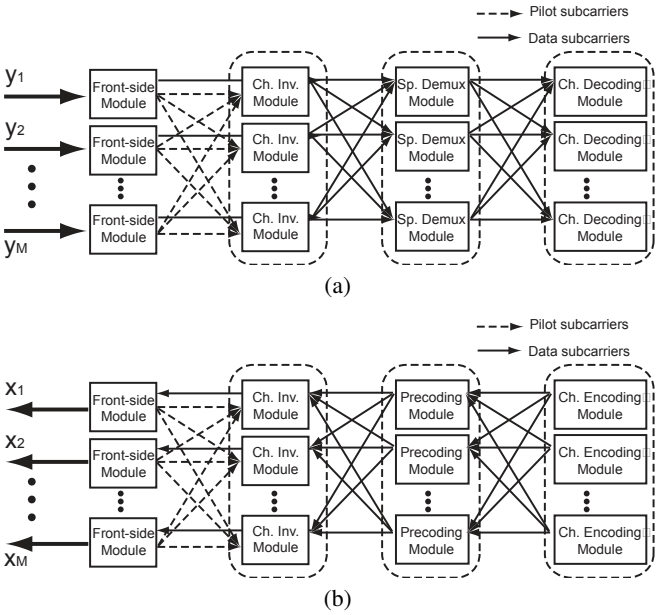


Figure 3: BigStation Architecture. (a) Up-link; (b) down-link.

it identifies each OFDM symbol and removes the time-domain redundancy, *i.e.*, the cyclic-prefix (CP). Next, it translates the time-domain symbols into frequency-domain symbols via FFT, and removes the frequency-domain redundancy, *i.e.*, the guard subcarriers. The pilot symbols are then sent to the CI modules for computing the pseudo-inverse of the channel matrix. Depending on the channel matrix dimension and the hardware processing power, each CI module may handle one or multiple subcarriers. Data symbols are directly sent to the SD modules. Similarly, the symbols are partitioned by subcarrier, and each SD module may handle only a subset of subcarriers. After obtaining the channel inverse from the CI modules, the SD modules extract each spatial stream from the input symbols. Symbols belonging to the same spatial stream are sent to the same CD module. The CD modules are responsible for mapping the symbols to soft bits and finally decode the original information bits.

On the downlink (Figure 3(b)), the concurrent information bitstreams are first encoded with FEC and then mapped to different subcarriers at the CE modules. Symbols on each subcarrier are then partitioned across different PR modules, where the symbols are *precoded* with the channel inverse – computed at the CI modules above. The precoded symbols on all subcarriers are collected at the FS modules, which then perform an IFFT and add CP to generate the time-domain OFDM symbols. Finally, all FS modules control the radio interfaces to transmit the waveforms simultaneously on all BigStation antennas.

5. BigStation ON PC SERVERS

As a proof of concept, we develop a BigStation system on a cluster of commodity servers. Besides producing an instantiation of our scalable architecture, our exploration also provides a first feasibility study of software radio based centralized wireless infrastructure [2, 8]. We use commodity multi-core x86 servers as physical processing units and standard Ethernet switches to connect them. Each PC server can host one or more signal processing modules as described in the previous section. Radio front-ends (antennas) are attached to the servers that run the FS modules. All signal process-

ing algorithms run in software on the PC servers in real time. In a sense, our BigStation implementation can be considered a large-scale PC-based software radio for MU-MIMO.

Building upon standard and commodity hardware components makes our system easy to scale. Not only can we incrementally add more servers and switches to meet growing capacity requirements, but also we align with current technology trends and take advantage of the latest offerings. Naturally, the BigStation architecture may be implemented using other technologies as well, *e.g.*, DSP or FPGA chips as the physical processing units. Given the limited processing power on each DSP or FPGA chip, however, we believe similar parallelized structures and methods will be needed. Therefore, the insights of our design and implementation will be general and valuable for making large-scale MU-MIMO a reality.

In this section, we present the detailed design. Specifically, we focus on exploiting parallelism to speed up processing at each stage of the distributed pipeline. Our strategy to parallelize involves data partitioning across servers (§ 5.1) and computation partitioning inside a server (§ 5.2) as well as across servers (§ 5.3).

5.1 Data partitioning

Data partitioning provides a simple method to leverage parallelism among multiple servers. As discussed earlier, in a MU-MIMO system, the channel inversion and spatial demultiplexing steps can be performed separately for each subcarrier, while the channel decoder should operate on symbols on *all* subcarriers that belong to the same spatial stream. That said, each spatial stream may be fed to a different channel decoder and decoded individually. Therefore, we partition the symbols by subcarrier at the channel inversion and demultiplexing stages, but by spatial stream at the channel decoding stage.

Data partitioning significantly reduces the computation and internal I/O requirements, since each server only needs to handle a small portion of the data now. With subcarrier partitioning, the I/O bandwidth required for one SD module (or a CI module) is $\frac{RM}{W}$, when serving M antennas. Take 802.11ac as an example, which has $W = 52$ subcarriers on a 20 MHz channel ($R = 416$ Mbps) or $W = 468$ subcarriers on a 160 MHz channel ($R = 5$ Gbps). Each subcarrier only consumes 10 Mbps of the bandwidth per antenna. With today's Ethernet technology (10 Gbps, and advancing to 40 Gbps), the data volume is hardly a bottleneck until $M > 1000$. With spatial stream partitioning, each decoding module may only handle one spatial stream. Even with a 160 MHz wide channel, the required bandwidth (~ 5 Gbps) may also be accommodated easily. Accordingly, the computational requirement is reduced by up to a factor of W for each CI or SD module and a factor of N for each CD module.

5.2 Computation partitioning inside a server

Today's high-end PC servers are predominantly built on shared-memory multi-core architecture. Therefore, we can further explore parallelism across multiple cores to speed up MU-MIMO signal processing. In this subsection, we study the three core signal processing algorithms used in a MU-MIMO system: 1) matrix multiplication, 2) matrix inversion, and 3) channel decoding, *e.g.*, Viterbi.

Matrix multiplication. The basic idea to parallelize matrix multiplication is to divide the matrices into a group of small blocks, and assign the multiplication of each block to a distinct CPU core. For example, to compute H^*H (an intermediate step in the channel inversion, see §2.1) with two cores, we can divide $H = (H_1 H_2)^T$.

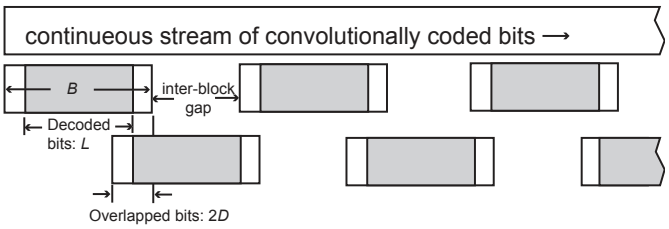


Figure 4: The parallelized Viterbi decoder.

Then, the result matrix

$$R = H^*H = (H_1^* H_2^*) \begin{pmatrix} H_1 \\ H_2 \end{pmatrix} = (H_1^* H_1 + H_2^* H_2)\,.$$

Each $H_i^* H_i$ operation can be assigned to a different CPU core and is much less complex compared to the original matrix multiplication.

Matrix inversion. The direct way to invert a square matrix is to use the Gauss-Jordan method. Its complexity grows with $O(N^3)$, where N is the size of each dimension of the matrix. Again, we can partition the computation by assigning different sets of rows to different CPU cores. Each core can perform Gaussian elimination on these rows independently [7]. With N units, the computational complexity can be reduced to $O(N^2)$.

Channel decoding. Parallelizing the decoding algorithm is straightforward for block-based channel codes, *e.g.*, Turbo and LDPC (Low-Density Parity Check) codes. In these coding schemes, bits are divided into blocks, *e.g.*, of a few hundreds to thousands of bits, and are encoded separately. We can simply assign each coded block to a separate CPU core (or a separate PC server) and achieve a higher aggregate throughput. However, parallelization becomes tricky for codes that work on a continuous bit stream, for example, the convolutional code widely used in wireless communication standards. Consequently, the corresponding decoding algorithm, *i.e.*, Viterbi decoding, cannot be parallelized naively as it needs to decode over a continuous bit stream as well. In this paper, we develop a trick that artificially partitions the bit stream into independent blocks of L bits, and assigns each block to a different core (or a separate PC server). The negative effect of this artificial partition is that it breaks the convolutional property of the code and may significantly reduce the decoding performance at the edges of blocks. Fortunately, this issue can be mitigated by overlapping the blocks as shown in Figure 4. Each block contains a prefix and a suffix of D bits each, which help the decoding path in between to converge to optimum. The Viterbi decoder processes the entire block, but only outputs the bits between the prefix and the suffix. According to the Viterbi theorem, when D is large enough, *i.e.*, $D \geq 5 * K$, where K is the constraint length of the convolutional code (typically $K = 7$ as in the 802.11 standard), with a probability close to 1, the decoded bits from blocks are identical to those from processing the entire bit stream [22].

We did further analysis to choose the right block size. Clearly, a larger block size would be more efficient as it amortizes the overhead of prefixes and suffixes, but a larger block also means a longer, undesirable decoding latency. We therefore aim at *fully utilizing the computational power while keeping the block size as small as possible*. This can be achieved if we keep the *inter-block gap* as small as possible. The ideal value of this gap is zero, meaning that a new block is scheduled on the same core right after the previous block is finished. Assuming the coded bit-stream comes at a rate of u, a decoder module can process at a rate of v, and there are m processing

units, the *inter-block gap* will be zero if

$$u\frac{B}{v} = mL.$$

This equation implies that during the processing of a block, exactly m blocks worth of bits will arrive, and each processing core can take a new block immediately after it finishes the current one. Taking $B = L + 2D$, we have

$$L^* = \frac{2Du}{mv - u}.$$

When the input rate is close to the processing capacity (*i.e.*, $u \to mv$), L^* will increase quickly. To prevent an unreasonable delay, we choose a bound $L_{max} = 2048$. With this upper bound, the prefix and suffix overhead combined is less than 3%.

5.3 Computation partitioning across servers

If data partitioning does not provide enough parallelism for each signal processing module to be executed on one PC server, we can further parallelize the operations across multiple servers. This can happen when M becomes very large – large enough that processing the smallest data set (*e.g.*, a single subcarrier) may still require more than the processing power of a PC server.

Channel inversion. The parallel matrix inversion algorithm discussed previously can be used across servers, where we send subsets of rows to different servers and each server can perform Gaussian elimination in parallel. However, one difficulty is *pivoting*. In Gaussian elimination, pivoting exchanges two rows, so that the diagonal entry used in elimination is nonzero and has, preferably, a large magnitude. To pivot across servers is cumbersome, as it may cause all servers to exchange information about their rows of data, incurring a heavy overhead on the network. Fortunately, the channel matrix in MU-MIMO is Hermitian (*i.e.*, H^*H) and pivoting is not necessarily needed [7]. The servers only need to broadcast the row that is used for elimination for each iteration. Since each row will be sent at most once for one elimination iteration, the communication overhead is bounded by the size of the matrix.

Another concern for parallelizing matrix inversion across servers arises from the need to synchronize among all servers at every elimination iteration. Given the non-deterministic delays in Ethernet, this could cause blocking in many servers and hold up the completion of channel inversion. However, these servers need to process a series of matrices, one for each new incoming frame. With a carefully designed multi-threaded algorithm, a server can immediately work on the next matrix if it is blocked on the current one. As a consequence, the aggregate throughput will not be affected.

Spatial demultiplexing. As M grows, a PC server may not be able to handle spatial demultiplexing for M antennas even on a single subcarrier. If a server only has enough power to compute the multiplication of a $K \times K$ matrix with a K-vector in real time, an SD module in this system will only be able to handle $K < M$ antennas. Therefore, the computation of any x_i should be separated to $\lceil \frac{M}{K} \rceil$ servers. We show how this can be done in Figure 7. The M FS servers send $y_1, \cdots, y_M$ to the first group of $\lceil \frac{M}{K} \rceil$ SD servers. Each SD server then computes a partial result for $x_1, \cdots, x_k$, and sends them to be further combined at another layer of servers. Repeatedly, we construct $\lceil \frac{N}{K} \rceil$ such groups to output all $x_1, \cdots, x_N$. The whole operation proceeds in two phases. In the first *spread* phase, each FS server multicasts y_i to the intermediate $(\frac{i-1}{K} + 1)^{th}$ servers in all $\lceil \frac{N}{K} \rceil$ groups. Each intermediate server then computes the partial results for K spatial streams. In the second *reduce* phase, the partial results are combined accordingly at another N servers to generate the final x_j.

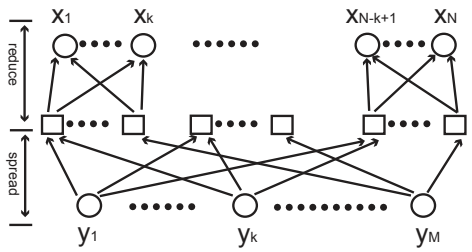


Figure 5: Deep distributed pipeline for spatial demultiplexing for an extremely large M. Some communication links are omitted for clarity.

5.4 Putting it all together

To summarize, here is how the distributed processing pipeline is constructed in BigStation:

Uplink:

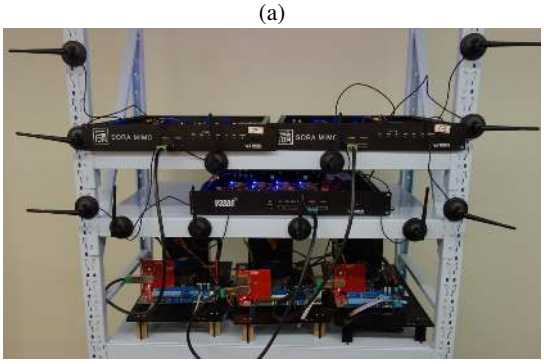
- For pilot symbols after FFT, the FS server divides subcarriers into $\frac{W}{c_i}$ groups and sends each group to a distinct CI server, assuming each server can handle c_i subcarriers.
- For data symbols after FFT, the FS server divides subcarriers into $\frac{W}{c_s}$ groups and sends each group of symbols to a distinct SD server, assuming each server can handle c_s subcarriers.
- Each CI server performs channel inversion on the received pilot bits and sends the result to the corresponding SD server.
- Each SD server separates the spatial streams from incoming symbol streams, and sends symbols belonging to one spatial stream to one CD server.
- Each CD server collects symbols from all subcarriers for one spatial stream and performs channel decoding.

Downlink:

- Each CE server generates channel coded bits and maps them onto symbols on each subcarrier. It divides subcarriers into $\frac{W}{c_s}$ groups and sends each group of symbols to a distinct PR server, assuming each server can handle c_s subcarriers.
- Each CE server also divides channel state information of all subcarriers into $\frac{W}{c_i}$ groups and sends each group to a distinct CI server, assuming each CI server can handle c_i subcarriers.
- Each CI server performs channel inversion on the received channel state information and sends the precoding vectors to the corresponding PR server.
- Each PR server performs precoding on incoming symbol streams and sends symbols belonging to one spatial stream to a distinct FS server.
- The FS servers cooperatively transmit the precoded symbols simultaneously.

In cases where a single server cannot handle the computation for a single subcarrier (*i.e.*, c_i or $c_s < 1$), the corresponding server may be replaced by a deeper pipeline of servers as discussed in Section 5.3.

(a)



(b)

Figure 6: BigStation radio front-end built from Sora MIMO kit. (a) Sora MIMO Kit. (b) BigStation radio front-end, containing three Sora MIMO Kits.

6. IMPLEMENTATION

6.1 Hardware platform

For our study, we have used two commodity PC models. One is a desktop PC with an Intel Core i7 3770 CPU and 8 GB memory on an ASUS P8Z77-M motherboard. The second is a Dell server with an Intel Xeon E5520 CPU (2.26 GHz, 4 cores) and 16 GB memory. The desktop PCs are primarily used as front-side (FS) servers and are connected to the software radio front-ends with multiple antennas. The radio front-end is based on the newly developed Sora MIMO Kit, as shown in Figure 6(a). Each Sora MIMO Kit integrates a Sora radio control board (RCB [20]) and 4 phase-coherent RF daughter boards, whose maximal operating bandwidth is 20 MHz each. The RCB is connected to a PC with an external PCIEx4 cable. We use 3 Sora MIMO kits to support 12 MU-MIMO antennas (Figure 6(b)). These 3 kits are synchronized by an external clock source.

The Dell servers are used as channel inversion (CI) or spatial demultiplexing (SD) servers. We have 15 such servers in total, all connected to a Pronto 3290 Ethernet switch, which has 48 1 Gbps ports and 4 10 Gbps ports.

The FS servers also connect to the same switch. Ideally, all FS servers should use 10 Gbps connections, since each FS server would generate sample data at 1.6 Gbps (416 Mbps $\times$ 4 antennas). Unfortunately, we do not have enough 10 Gbps ports on the switch – three out of the four 10G ports are connected to CI and SD servers (§ 7.2). Therefore, we use four 1 Gbps ports instead. All our PC servers are running Windows Server 2008 R3.

6.2 Underlying software support

SIMD library. Our signal processing software is implemented using the signal processing library from Sora SDK [20], which has been highly optimized for SIMD-capable Intel CPUs. We have extended the library to support parallel algorithms among multiple cores (§5.2).

Table 1: Communication over multiple cores (Gbps).

# of cores	Receive	Send	Receive & Send
1	5.9	9.2	2.4 (R) / 7.8 (S)
2	8.6	9.4	5.1 (R) / 7.0 (S)
4	9.2	9.4	5.9 (R) / 6.8 (S)

Parallelizing communication across cores. Besides the computation, the underlying software should also handle the communication among BigStation servers. This is especially critical for the CI servers and the SD servers as they are required to receive/send data from/to all FS servers. In the following, we focus on the CI and SD servers. Since both CI and SD servers are equipped with 10 Gbps NICs, ideally we would like the server to be able to handle full-speed traffic on both the uplink and the downlink, at a total throughput of 20 Gbps. However, such an amount of data traffic cannot be handled by a single CPU core in our PC server. As a consequence, we need to further exploit multi-core parallelism to handle network traffic as well.

We study the impact of using multiple cores on network communication experimentally. In our experiments, we let one SD server receive 12 digital sample streams generated from all FS servers as fast as possible. Since we are focusing on the communication performance, we instruct the SD server to directly send the received digital samples to a CD server without performing *spatial demultiplexing*.

Table 1 summarizes the results. Although we can send fast enough to saturate the link (9.2 Gbps) with a single core, the receiving throughput is only about 5.9 Gbps. Since now CPU is the bottleneck, multiplexing sending and receiving on the same core may reduce throughput in both directions and also cause huge unfairness between the uplink and the downlink (Table 1 row 1, column 3). Using two CPU cores, we can almost achieve the full link speed for either sending or receiving. However, with simultaneous sending/receiving operations, the total throughput we can get in both directions is 12 Gbps, despite the theoretical maximum of 20 Gbps. We have carefully checked our code to avoid any interlocking between our sending and receiving procedures. Therefore, we believe there are some interactions inside the Mellanox driver/NIC. Unfortunately, both the driver and the NIC are closed to us, which prevents us from finding the exact reasons. Adding more cores for communication does not improve the performance any further. This is reasonable as now the NIC becomes the bottleneck. Therefore, in our implementation, we use two threads, each of which is pinned to one physical core, to handle incoming and outgoing traffic separately.

Another potential issue for the SD server is incast TCP collapse. This is because the SD server may need to receive data from many TCP sessions from the FS servers. For example, in our case, there are 12 concurrent TCP connections synchronized at one switch port. The short-term burstiness from many TCP connections may overflow the switch buffer, causing intensive packets losses, TCP retransmissions, and even TCP timeouts. This potential incast problem can have significant adverse impacts on the performance of BigStation. While other researchers have suggested various ways to solve the TCP incast problem by modifying TCP or adding ECN tuning on the switch [23], we adopt a simple application-level flow control mechanism to avoid this problem. In standard TCP, the receiving side maintains a window that controls how many packets can be sent to this server without receiving an ACK. The window size is carefully chosen so that it will avoid buffer overflow for the underlying switch, but at the same time deliver good throughput. Since BigStation uses a dedicated Ethernet and all traffic patterns

Table 2: Comparison of different locking mechanisms in a SD server. There are four computing threads and the processing throughput is measured by MSps (sample-per-second).

naive	write-back	lock-free
62.2	78.15	81.45

are known, the receiving server can simply distribute its window equally among all of its upstream servers. Since the number of packets in aggregate will not exceed the switch buffer capacity, the incast problem is avoided. Further, our application-level flow control helps to keep the Ethernet switch buffer occupancy at a very low level, and therefore reduce the Ethernet communication latency.

Lock-free computing structure. After the receiving thread reads data from the network, it will put them into local buffers. Then, the computing threads need to read data from all input queues and compute a result for each of the output queues. How should the computing threads interact with these data queues? Figure 7 illustrates two possible ways to connect the servers and queues. In Figure 7(a), the computing thread can read symbols from all input queues; once it gets a data block from one queue, it computes partial results for all output queues, *e.g.*, an SD server calculates the products of an entire column of H^+ and the input symbol block of Y. However, this may create heavy contention for the output queues as multiple computing threads may try to write to the same output queue at the same time. While the output queues can be protected by locks, these locks will be heavily contended for.

A carelessly implemented locking mechanism may significantly reduce the system performance. For example, in our initial implementation, the computing thread would lock a buffer in the output queue, perform the computation, write back the results in the buffer, and then release the lock. We call this scheme *naive locking*. *Naive locking* significantly reduces the system processing throughput as it locks the buffer for an unnecessarily long period.

A better scheme is using *write-back locks*. In this scheme, the computing thread calculates the result in a temporary buffer first. Only after the computation of an entire data block does the thread acquire the lock on the output queue, write back the results, and release the lock. This approach requires a computing thread to maintain an additional buffer for each output queue, but will greatly reduce the locking time.

The best approach, however, is to avoid locks completely. As shown in Figure 7(b), while each computing thread is still able to read from all input queues, it is only responsible for updating a small group of output queues. Since each output queue is assigned to only one computing thread, contention is avoided.

Table 2 compares the processing throughput of an SD server with different locking schemes. We can see the lock-free scheme has the best performance, while naive locking can reduce the processing throughput by 23%.

Thread and core allocation. A final question to ask is how we assign the computing threads to CPU cores. We have tried a few different configurations. We find that mixing the computing and communication threads on one CPU core, or on two hyper-threading cores that share the same physical core, would significantly reduce the communication throughput. This is because both the computing and the communication threads will compete fiercely for the CPU resource. This result leads to our first rule: *Isolating the communication and computing threads on different physical cores*. Second, we find that assigning only one computing thread to a core has the best performance. This is reasonable as our computing thread has been highly optimized to maximize CPU utilization. Therefore, assigning more threads to the same core will only incur additional overhead (*e.g.*, context-switching). Also, we find utilizing hyper-threading for computing threads does not increase the overall processing throughput. Nor does it decrease the performance. Therefore, our second rule is: *The number of computing threads should be between the numbers of physical and hyper-threading cores that are dedicated to computation*.

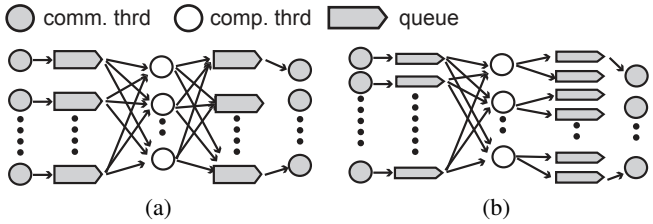


Figure 7: Software pipeline in the processing server.

6.3 Link layer operations

Our current BigStation prototype employs a very simple TDMA MAC. Each TDMA time slot is 2 ms long and fits one frame. The slot can be dynamically allocated to uplink or downlink transmissions by a simple packet scheduler. For each uplink frame, all transmitters need to send out an orthogonal training symbol (pilot) for BigStation to learn the channel state information (CSI) (Figure 2). Each training symbol is 8 μs long and contains a repeated pattern like the 802.11 long training symbol (LTS), which can be used to estimate the carrier frequency offset between each transmitter and BigStation.

BigStation relies on channel reciprocity to obtain downlink CSI from the uplink channel measurements. We use an approach similar to that in [18] to calibrate the coefficients between the uplink and downlink channels. Basically, an internal calibration is performed first among all antennas on BigStation, after which an equivalent downlink channel matrix can be derived from the uplink channel measurements. This internal calibration is only needed once when BigStation boots up. We omit the algorithm here and refer the interested readers to [18] for the details.

BigStation maintains a database to store all CSI. Once a new channel measurement is taken (*e.g.*, through an uplink transmission), the database is updated. The CSI is removed after the channel coherence time. In this work, we manually set this time to 20 ms. We defer the dynamic estimation of the channel coherence time to future work. When scheduling downlink MU-MIMO transmissions, all selected clients should have a fresh CSI record taken within the coherence time.

7. EVALUATION

7.1 Micro-benchmarks

We first evaluate the capability of our existing servers for signal processing in BigStation. By benchmarking the server performance, we try to answer the following question: *how many servers do we need to build a BigStation with a given capacity?* Specifically, we consider three example configurations: *Medium scale*, 100 Mbps to 6 users; 2) *Large scale*, Gbps to 10 users; and 3) *Ultra-large scale*, Gbps to 50 users. The parameters of these three configurations are listed in Table 3. We perform all our experiments on the Dell servers with Intel Xeon E5520 CPUs (§6). Additionally, for large scale and ultra-large scale settings, we also consider another high-end server configuration with more CPU cores. For example, the latest Dell server is equipped with 32 cores [3].

Table 3: Example configurations of BigStation

	Channel width	W	M	Rate per spatial stream
Medium scale	20 MHz	52	12	54 Mbps
Large scale	80 MHz	234	40	293 Mbps
Ultra-large scale	160 MHz	468	100	585 Mbps

Parallel algorithms. As discussed earlier, the complexity of channel inversion increases with $O(N^2M)$. Therefore, it is more likely to become a bottleneck as M increases (and N increases accordingly, *i.e.*, $N = M$ in the worst case). Figure 8(a) shows the processing time of the channel inversion on a single CPU core with different N values. Clearly, we can see that the processing time increases quickly with N, although the absolute processing time is actually affordable when N is modest (< 50). For example, when N is 12, inverting a single channel matrix takes merely $10\mu s$. Recall that only one channel inversion is computed for every frame. So a single core is able to handle about 200 subcarriers, if the frame length is 2 ms. When N is 40, the channel inversion time for a single subcarrier increases to $236\mu s$. Still using $2ms$ frames as an example, a single core can handle 8 subcarriers. When N grows to 100, the invertion time rises to 3.3 ms, and a single core is not able to handle even one subcarrier in real time. Parallel processing among multiple cores is then essential. Figure 8(b) shows the processing time of inverting a single channel using multiple cores. We can see that with more cores, the processing time is reduced proportionally. For example, when there are four cores to invert a channel matrix in parallel, the processing time for $N = 100$ can decrease to $607\mu s$. One 4-core PC server can handle 3 subcarriers.

Figure 9 shows the spatial demultiplexing throughput. With a single core, the demultiplexing throughput for 10 spatial streams from $M = 10$ antennas is around 4 Gbps, sufficient to support 50 subcarriers. The throughput, however, reduces to 888 Mbps and 400 Mbps, when M is 40 and 100 respectively. We can similarly improve the processing speed with multiple cores. With 4 cores, our server can speed up processing by 4 times to 3.2 Gbps or 8 subcarriers ($M = 40$) worth, and 1.6 Gbps or 1 subcarrier ($M = 100$).

We further evaluate our parallelized Viterbi algorithm in Figure 10. Similarly, the decoding throughput increases linearly with the number of cores. With 4 cores, our server can deliver a throughput of 283 Mbps.

Summary. Based on the above micro-benchmarks, we can extrapolate the number of servers needed to construct BigStation at different scales. We note that in all three example configurations, the computation is the bottleneck. However, as discussed in §6.2, we still need to allocate one or two cores on each server to handle the network traffic. Table 4 summarizes the results.

Although our design can scale even with low-end, 4-core servers, we have not considered the network cost. Indeed, we expect the cost of network devices to be significant, but this issue can be mitigated by upgrading servers. Given the existing trend of server technologies, we expect more cores to become available even for low-cost commodity servers. With more cores per server, the number of total required servers decreases proportionally, thereby reducing the cost of network devices. All in all, we conclude that our architecture can scale to tens to hundreds of antennas with very wide channel widths.

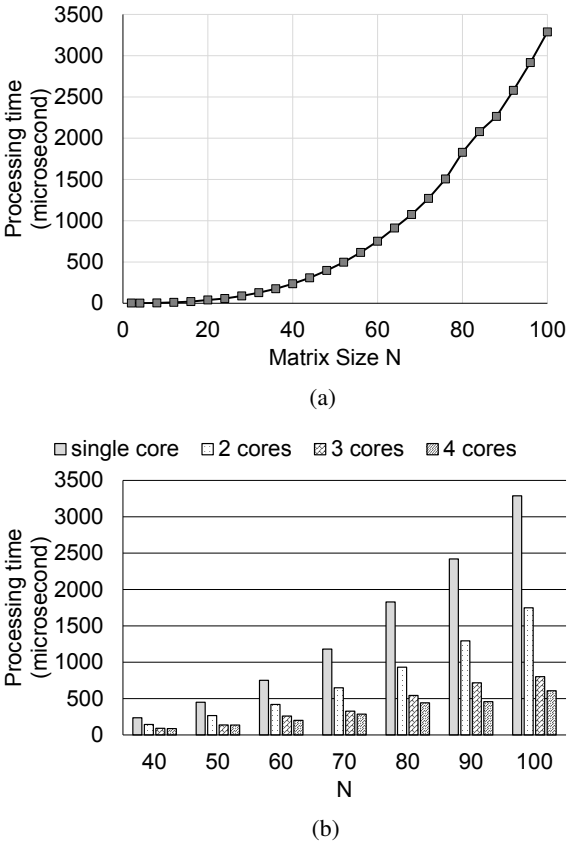


Figure 8: Processing time of matrix inversion. (a) Using a single CPU core. (b) Using multiple cores.

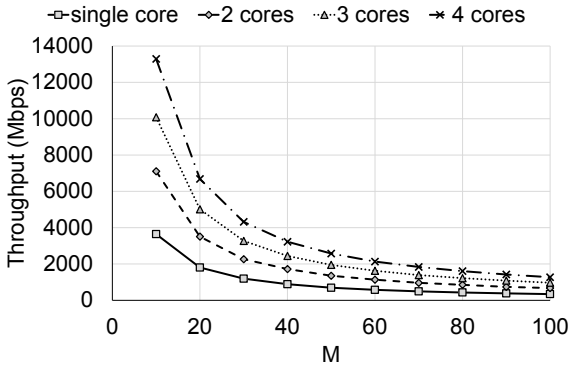


Figure 9: Spatial demultiplexing throughput using multiple cores, $M = N$.

Table 4: # of servers to construct BigStation

	4-core servers			32-core servers		
	CI	SD	CD	CI	SD	CD
Medium scale	1	2	4	1	1	1
Large scale	15	30	80	2	4	10
Ultra-large scale	156	468	300	20	59	25

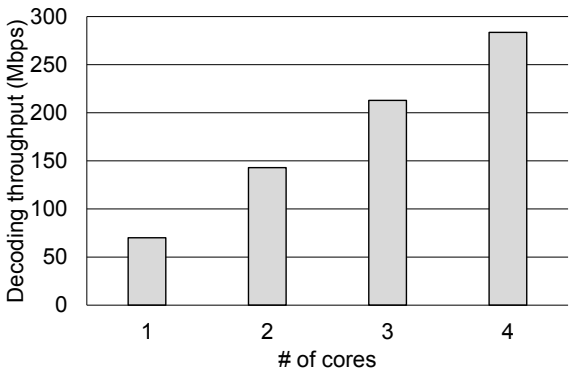


Figure 10: Decoding throughput of the parallel Viterbi algorithm using multiple cores.

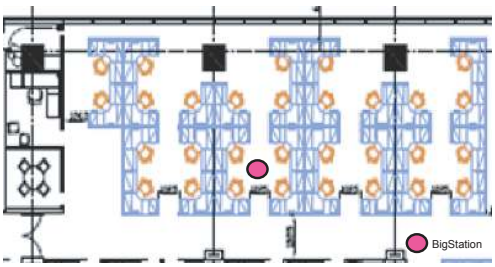


Figure 11: Layout of our testing environment. The BigStation prototype is mounted on a mobile rack in the middle of a cubicle. Clients are not marked as they move around in nearby cubicles.

7.2 System performance

Testbed. We have built a medium-scale BigStation with 12 antennas on our 15-server platform (§6.1). We deploy one CI server and two SD servers – each SD server handles 26 subcarriers. All SD and CI servers are connected to the 10G ports on the Pronto Ethernet switch. We deploy a decoding server on each of the remaining 12 PC servers.

We test our prototype in a typical office environment with cubicles. Figure 11 shows the layout of our testbed. We have also deployed 9 single-antenna clients in nearby cubicles around BigStation. Since the clients are close to BigStation, the signal-to-noise ratios (SNRs) between the client antennas and BigStation are high, usually between 20 $\sim$ 30 dB.

Sum peak rate. The first question we ask is *does large-scale MU-MIMO even make sense?* Can we indeed linearly scale wireless capacity with more antennas on BigStation? We let increasing numbers of clients send data packets to BigStation. Then, BigStation tries to decode each spatial stream and finds out its peak rate, *i.e.*, the maximal modulation rate it can support on each spatial stream. Since we use 802.11a modulation rates, the peak rate is capped at 54 Mbps for each stream. For each experiment, we collect 500 frames.

In the first experiment, we always let $M = N$. This case is interesting as it can fully utilize the antennas on BigStation. To do so, we randomly pick N antennas from the 12 antennas on BigStation and use only these N sample streams to decode packets. To our surprise, the sum peak rate of N spatial streams does not scale as we expected ("dot" line in Figure 12). When N is small, *i.e.*, 2 and 3, the capacity seems to increase linearly – with a small slope. When N becomes larger, the sum peak rate remains unchanged or even decreases! The reason behind this observation lies in the random antenna selection for MU-MIMO operations in BigStation, which induces *wireless channel hardening* [11]. In an $M \times N$ MU-MIMO system, when N is large, the sum rate can be modeled as follows [11]:

$$C = \sum_{n=1}^{N} \log\left(1 + P/[N(\mathbf{H}^*\mathbf{H})^{-1}{}_{n,n}]\right) \quad (1)$$

$$\approx N \log\left(1 + P/(M/N - 1)\right) \quad (2)$$

where P is the total transmission power and $\mathbf{H}$ is the channel matrix. Equation 2 converges to a constant value when $M = N$ [15].

To avoid channel hardening, in our second experiment, instead of limiting $M = N$, we use all 12 antennas regardless of how many senders there are. Interestingly, we see the sum peak rate indeed grows linearly as more senders transmit (solid line in Figure 12). When there are 9 concurrent senders, the sum peak rate increases by 6.8× compared to a single-antenna setup! Therefore, *BigStation indeed linearly scales wireless capacity if $M > N$.*

The next natural question to ask is how many antennas should BigStation have in order to support any given N clients? To answer this question, we vary antenna configurations and measure the condition number of the wireless channel matrix. The condition number shows how well the channel matrix inverse can be used to demultiplex spatial streams [21]. Well-conditioned channels, whose condition numbers are small (close to 1), can decorrelate the spatial streams without much distortion. Conversely, a large condition number will significantly reduce the SNRs of spatial streams after

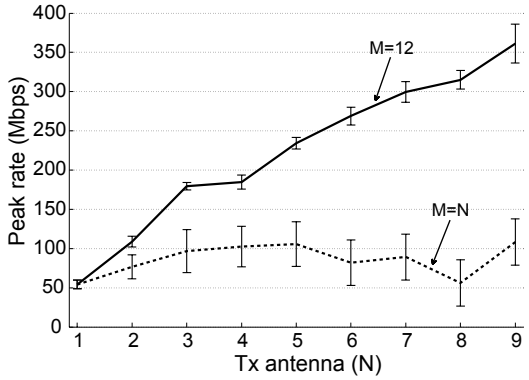


Figure 12: Sum of peak rate of BigStation. Error bar shows the standard deviation.

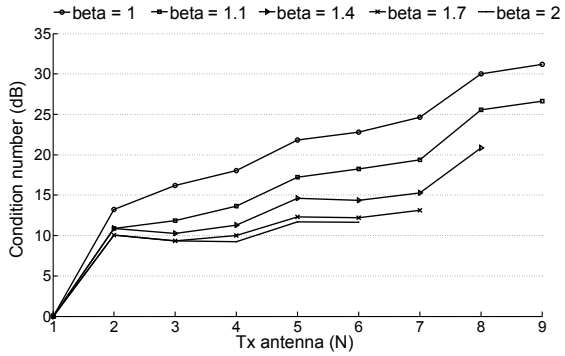
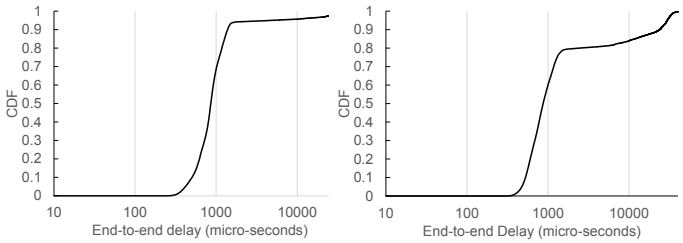


Figure 13: Channel matrix condition number with different N, $M = \beta N$.



(a) (b)

Figure 14: CDF of the end-to-end delay of BigStation. (a) Under a light load. (b) Under a heavy load. The x-axis is in logarithmic scale.

spatial demultiplexing. Let $M = \beta N$. Figure 13 shows the channel condition number with increasing number of transmitters under different β values. We find that with slight over-provisioning, *i.e.*, $M = 1.1N$, the channel condition number decreases prominently, by as much as 5 dB. Further increasing M will result in better-conditioned channels. However, when $\beta > 1.4$, the reduction becomes less pronounced.

System delay. Besides the processing throughput, the processing latency is another critical metric for wireless communications. In the following, we characterize the delay performance of BigStation. To measure the overall processing delay, instead of sending to a CD sever, we let the SD servers send symbols back to the FS server after spatial demultiplexing. The FS server can timestamp both the original symbol generation and the return of the corresponding demultiplexed symbol, and compute the delay. We note that this measured delay excludes the channel decoding delay, which is fixed to be 9 μs according to our benchmark.

We first measure the processing latency of BigStation in a light load situation, where the FS servers generate a frame every 10 ms. Figure 14(a) shows the cumulative distribution function (CDF) of processing delay. We can see that the mean processing latency is as low as 860 μs, and the 90th percentile is below 1.4 ms. However, when the traffic load becomes heavier, a heavy-tailed delay CDF appears. Figure 14(b) shows the latency measurements when the FS servers continuously generate back-to-back frames. We can observe that while the mean latency is still around 860 μs, a small portion of the frames may experience excessive delay (the 90th percentile is 20 ms). After a closer examination, we find this behavior is due to TCP retransmissions. Under a heavy load, the underlying network may occasionally see packet losses and TCP retransmissions. The SD server requires symbols from all antennas before it can finalize the output. Therefore, even if one TCP connection slows down, the entire MU-MIMO frame, as well as a few subsequent frames, is delayed.

Table 5 summarizes the delay breakdown of various components in BigStation. The data presented here are measured in the light load situation. In this case, we find the network delay is actually small ($\sim 300\mu s$). This is because our application-level rate control can keep the network queues small. Instead, most of the delay is incurred while the symbol packets are waiting in queues on the CI/SD servers. This behavior is also because the SD (CI) server requires symbols from all antennas before deriving final results. Therefore, the variance in the packet transmission times from different FS servers will translate into a queuing delay on the SD (CI) server.

In conclusion, BigStation has a low mean processing delay ($< 1ms$). While this delay may not be sufficiently low for 802.11 MAC layer ACK, which requires micro-second level latency, it already satisfies the real-time requirements for many other wireless protocols, *e.g.*, LTE and WCDMA. Finally, we note that when the system is heavily loaded, the processing delay exhibits a heavy-tailed distribution, where a small portion of the frames may experience excessively long delays. This long tail latency can be mitigated with resource over-provisioning, but at the expense of low system efficiency. Alternatively, we can apply many techniques developed for predictable service times in the distributed systems community to control the latency in our distributed MU-MIMO processing [5,10]. We defer a deep investigation in this direction to our future work.

Table 5: Delay of components in BigStation. Unit is in (μs). The number in the brackets is the 90th percentile.

CI server		SD server		CD server
Net	CI	Net	SD	
280 (410)	680 (1,190)	330 (450)	550 (990)	9 (9)

8. RELATED WORK

MU-MIMO has been extensively discussed in the information theory literature. Small-scale MU-MIMO (*i.e.*, $M < 10$) has been implemented and experimented with in many real systems [6, 19]. Recent wireless standards, *e.g.*, 802.11ac [4], LTE [1], WiMAX *etc.*, have proposed to include small-scale MU-MIMO in their future evolution. While small-scale MU-MIMO can improve the wireless capacity to some extent, this improvement has been fundamentally limited by the number of antennas on the AP. With the growing demand of mobile traffic, it is desirable to build large-scale MU-MIMO systems that employ tens or hundreds of antennas and improve spectral efficiency by at least an order of magnitude.

The issue of scaling MU-MIMO systems has recently received much interest from both the theory community [12,13,17] and system builders [16, 18]. JMB [16] proposed to scale a MU-MIMO system by federating a number of small APs. The authors proposed a distributed algorithm to synchronize the phases of several APs. Therefore, these APs can perform joint beamforming and send concurrent packets to different users. However, JMB does not consider the scalability of the AP. As discussed earlier (§2.2), channel inversion and precoding processing will soon become bottlenecks with increasing M and N.

Realizing this difficulty, Hoydis *et al.* have proposed *massive MIMO* [12]. Assuming an infinite number of antennas on the AP, they show simpler conjugate beamforming can deliver the same performance as zero-forcing beamforming. Argos [18] is a large-scale MU-MIMO system with 64 antennas that employs conjugate beamforming. The authors have developed a local precoding method that distributes the conjugate beamforming operations to each radio module (similar to the FS module in BigStation). However, with a finite number of antennas on the AP, conjugate beamforming incurs a significant performance loss compared to zero-forcing beamforming. In their experiments, the performance loss can be as large as a factor of 4. Further, this local method only applies to the downlink, and the uplink processing does not yet scale. Although Argos removes the need for channel inversion (at the expense of performance), it still relies on a central module for spatial demultiplexing and decoding. As we have shown earlier, these two operations are more computationally demanding and more likely to be bottlenecks. In contrast, BigStation does not compromise the performance, but addresses the MU-MIMO scaling challenge by parallelizing the computation and communication among many simple

and low-cost devices. Consequently, BigStation achieves incremental scalability by adding more computing devices. Our experiment on a BigStation prototype with 12 antennas shows a peak rate gain of 6.8× compared to the single-antenna radio. In comparison, Argos only reports a 5.7× capacity improvement with 64 antennas, due to suboptimal conjugate processing.

The comparison in theory between zero-forcing and the conjugate processing (also called matched filter) is presented in [13]. Our results agree with [13], but we use real measured data from a practical large MIMO system.

BigStation is also related to much parallel computing work. Many schemes to parallelize the digital signal processing in BigStation have been previously studied in other contexts [7]. However, as far as we know, BigStation is the first work to parallelize MU-MIMO operations to scale the system to tens or hundreds of antennas.

9. CONCLUSION

This paper presents BigStation, a scalable architecture for large-scale MU-MIMO systems. Our strategy to scale is to extensively parallelize the MU-MIMO processing on many simple and low-cost commodity computing devices. Therefore, our design can incrementally scale to support more MIMO antennas by proportionally adding more processing units and interconnecting bandwidth. After carefully analyzing the computation and communication patterns of MU-MIMO, we parallelize MU-MIMO processing with a distributed pipeline to reduce the overall processing delay. At each stage of the pipeline, we further use data partitioning and computation partitioning to increase the processing speed.

We have built a BigStation prototype with 15 PC servers and standard Ethernet switches. Our prototype can support real-time MU-MIMO processing for 12 antennas. Our benchmarks show that the BigStation architecture is able to scale to tens to hundreds of antennas. With 12 antennas, our BigStation prototype can increase the wireless capacity by 6.8× with a low mean processing delay of 860 μs. This latency already satisfies the real-time requirements of many existing wireless standards, *e.g.*, LTE and WCDMA.

10. ACKNOWLEDGMENT

We sincerely thank our shepherd, Brad Karp, and the anonymous reviewers for their valuable comments and suggestions.

11. REFERENCES

[1] 3GPP TS 36.201-820: Evolved Universal Terrestrial Radio Access (E-UTRA); Long Term Evolution (LTE) physical layer; General description.

[2] C-RAN: The Road Towards Green RAN. *http://labs.chinamobile.com/cran/wp-content/uploads/CRAN_white_paper_v2_5_EN(1).pdf*.

[3] HP ProLiant DL560 Gen8 . *http://h10010.www1.hp.com/wwpc/us/en/sm/WF06b/15351-15351-3328412-241644-3328422-5268290-5288630-5288631.html?dnr=1*.

[4] IEEE Standard for Local and Metropolitan Area Networks Part 11; Amendment: Enhancements for Very High Throughput for operation in bands below 6GHz. *IEEE Std P802.11ac/Draft 4.0*, 2012.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of NSDI*, 2012.

[6] E. Aryafar, N. Anand, T. Salonidis, and E. W. Knightly. Design and experimental evaluation of multi-user beamforming in wireless LANs. In *Proceedings of MobiCom*, pages 197–208, New York, NY, USA, 2010. ACM.

[7] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 2003.

[8] S. Bhaumik, S. P. Chandrabose, M. K. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, and T. Woo. CloudIQ: A framework for processing base stations in a data center. In *Proceedings of MobiCom*, pages 125–136, 2012.

[9] Cisco Inc. Cisco Visual Networking Index (VNI): Forecast and Methodology 2011-2016. *Cisco, http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white _paper_c11-481360_ns827_Networking_Solutions_White_Paper.html*, 2012.

[10] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), 2013.

[11] B. Hochwald and S. Vishwanath. Space-Time Multiple Access: Linear Growth in the Sum Rate. In *Proceedings 40th Annual Allerton Conf. Communications, Control and Computing*, 2002.

[12] J. Hoydis, S. ten Brink, and M. Debbah. Massive MIMO: How many antennas do we need? In *Allerton Conference on Communication, Control, and Computing*, September 2011.

[13] H. Huh, G. Caire, H. Papadopoulos, and S. Ramprashad. Achieving "Massive MIMO" Spectral Efficiency with a Not-so-Large Number of Antennas. *IEEE Transactions on Wireless Communications*, 11(9):3226 –3239, September 2012.

[14] J. Neel, P. Robert, and J. Reed. A Formal Methodology for Estimating the Feasible Processor Solution Space for A Software Radio. In *Proceedings of the SDR Technical Conference and Product Exposition*, 2005.

[15] C. Peel, B. Hochwald, and A. Swindlehurst. A Vector-perturbation Technique for Near-capacity Multi-antenna Multi-user Communication — Part I: Channel Inversion and Regularization. *IEEE Transactions on Communications*, 53(1):195–202, 2005.

[16] H. S. Rahul, S. Kumar, and D. Katabi. JMB: Scaling wireless capacity with user demands. In *Proceedings of ACM SIGCOMM*, pages 235–246, 2012.

[17] F. Rusek, D. Persson, B. K. Lau, E. Larsson, T. Marzetta, O. Edfors, and F. Tufvesson. Scaling Up MIMO: Opportunities and Challenges with Very Large Arrays. *IEEE Signal Processing Magazine*, 30(1):40 –60, January 2013.

[18] C. Shepard, H. Yu, N. Anand, E. Li, T. Marzetta, R. Yang, and L. Zhong. Argos: Practical many-antenna base stations. In *Proceedings of MobiCom*, pages 53–64, 2012.

[19] K. Tan, H. Liu, J. Fang, W. Wang, J. Zhang, M. Chen, and G. Voelker. SAM: Enabling Practical Spatial Multiple Access in Wireless LAN. In *Proceedings of MobiCom*, 2009.

[20] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: High performance software radio using general purpose multi-core processors. In *NSDI 2009*.

[21] D. Tse and P. Vishwanath. *Fundamentals of Wireless Communications*. Cambridge University Press, 2005.

[22] A. J. Viterbi and J. K. Omura. *Principles of digital communication and coding*. McGraw-Hill, 1979.

[23] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in data center networks. In *Proceedings of CoNEXT*, pages 13:1–13:12, 2010.