RICE UNIVERSITY

Software-based Baseband Processing for Massive MIMO

by

Jian Ding

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Lin Zhong, Chair
Professor of Electrical and Computer
Engineering and Computer Science

Ashutosh Sabharwal
Professor of Electrical and Computer
Engineering

John Mellor-Crummey
Professor of Computer Science and
Electrical and Computer Engineering

Houston, Texas

August, 2019
Software-based Baseband Processing for Massive MIMO

by

Jian Ding

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Lin Zhong, Chair
Professor of Electrical and Computer Engineering and Computer Science

Ashutosh Sabharwal
Professor of Electrical and Computer Engineering

John Mellor-Crummey
Professor of Computer Science and Electrical and Computer Engineering

Houston, Texas

August, 2019
ABSTRACT

Software-based Baseband Processing for Massive MIMO

by

Jian Ding

Large-Scale multiple-input multiple-output (MIMO) is a key technology for improving spectral efficiency. However, it requires massive, real-time computation. Specialized hardware, e.g., FPGA, can efficiently process baseband data but has poor flexibility and programability. All existing solutions are based on dedicated, specialized hardware, e.g., FPGA, that is expensive, inflexible and difficult to program. In this thesis, we investigate a software-only solution that exploits recent CPU development supporting many cores and architectural extensions for fine-grained parallelism. We present MILLIPEDE, a high-performance framework for real-time, large-scale baseband processing on a many-core server. To achieve the high data rate and low latency promised by 5G, MILLIPEDE utilizes data parallelism and exploits architecture features, including memory hierarchy and SIMD extensions, to accelerate computations and data movements. We report a prototype on a 36-core server and evaluate its performance.
Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Lin Zhong, for his patience, guidance, encouragement and constructive critiques throughout my past years at Rice. His generous sharing of insights on all aspects of research, especially on how to select research problems, has always been motivating me to pursue hard problems in cutting-edge research areas.

I would like to thank Dr. Ashutosh Sabharwal and Dr. John Mellor-Crummey for serving on my thesis committee and offering insightful feedback. I also thank Dr. Joseph R. Cavallaro for his valuable suggestions on this thesis.

I would like to extend my sincere gratitude to Peiyao Zhao for his important contributions of code and ideas to MILLIPEDE. Peiyao built the predecessor of MILLIPEDE from which MILLIPEDE inherits the queue-based master-worker threading model. Before that, Songtao He and Caihua Li built two frameworks for massive MIMO baseband processing that implement BigStation’s idea of using both data and pipeline parallelism, which helped me gain insights into its inefficiency. I also thank Dr. Rahman Doost-Mohammady for his help on developing the ArgosV3 base station and his hardware expertises that give me valuable insights.

I am grateful to every member (past and present) of the RECG group. A special thanks to Dr. Clayton W. Shepard for his guidance and help during my early years at Rice, and his work on building and evolving the Argos platform, which opens up enormous new research opportunities, including the ones I have been working on.

My gratitude also goes to Dr. Ranveer Chandra, who advised me during my internships at Microsoft Research and helped me build my confidence and research expertises.

Finally, I thank my family, my parents and brother, for their unconditional support and
love that help me go though the good times and the hard times during my graduate study.
## Contents

Abstract ii  
Acknowledgments iii  
List of Illustrations vii  
List of Tables ix  

1 Introduction 1  

2 Background 5  
  2.1 RAN and Software RAN 5  
  2.2 Baseband 6  
    2.2.1 MU-MIMO and Massive MIMO 8  
    2.2.2 Pipeline and Data Parallelism 10  
    2.2.3 Performance Metrics 11  

3 Design 14  
  3.1 Design Principles 14  
  3.2 Not All Principles Are Equal 15  
  3.3 MILLIPEDE : Design Overview 17  
  3.4 Model-driven Configuration 19  
    3.4.1 Single-core Performance Models 20  
    3.4.2 Multi-core Pipeline Parallelization 21  
  3.5 Reducing Useless Work 24  
    3.5.1 Coarsening Job Granularity 24  
    3.5.2 Lock-free Data Access 26
4 Implementation

5 Evaluation

5.1 Methodology .......................................................... 31

5.2 Overall Performance .................................................. 34

5.2.1 Uplink Processing Latency ........................................ 34

5.2.2 Downlink Transmission Latency and Data Rate ............... 35

5.3 Performance Breakdown .............................................. 36

5.3.1 Pilot Symbol Processing ($T_{Pilot}$) ............................. 37

5.3.2 Zeroforcing ($T_{ZF}$) .............................................. 38

5.3.3 Uplink Data Symbol Processing ($T_{ULData}$) .................. 39

5.3.4 Explanation of MILLIPEDE’s Advantages ....................... 39

5.4 Performance Bottlenecks of MILLIPEDE ......................... 40

5.4.1 Message Synchronization Cost ................................... 40

5.4.2 Data Communication Overhead .................................. 41

5.4.3 Bottlenecks of External Library Calls. .......................... 45

6 Related Work ............................................................ 46

7 Conclusions ............................................................. 49

8 Future Work ............................................................. 50

Bibliography ............................................................... 52
Illustrations

2.1 Typical 5G PHY pipeline and frame structure in TDD mode. Within a typical frame, the system first estimates the channel, computes the precoder, and then carries out downlink/uplink data communication. 7

3.1 System overview. (a) MILLIPEDE processes the PHY pipeline on a single many-core server. (b) BigStation distributes the data processing blocks in the PHY pipeline to multiple groups of machines, where each machine only has a small number of cores. 17

3.2 Uplink frame processing schedule under different parallel schemes. MILLIPEDE’s data parallel only design processes frames without overlap. BigStation’s data and pipeline parallel design allows simultaneous execution of different procedures in different frames. 23

3.3 Uplink pipeline parallelization with different job granularities. Merging blocks operating on the same data dimension can reduce the number of synchronization points and data movement across cores. 25

5.1 Uplink processing latency of MILLIPEDE and BigStation. MILLIPEDE needs much fewer cores than BigStation to hit the performance bound, i.e., 5-ms processing time, of a 5-ms frame. 35
5.2 Downlink transmission latency and achievable data rate of MILLIPEDE and BigStation. Given the same number of cores, MILLIPEDE achieves significantly shorter downlink transmission latency and higher data rate than those in BigStation. .................................................. 36

5.3 Pilot symbol processing time over number of antennas using 30 cores. In all cases, pilot processing time is lower bounded by pilot receive duration. . 38

5.4 ZF processing time in uplink and downlink with 8 base station antennas. The ideal speedup in both (a) and (b) is equal to the number of cores assigned to execute doZF. In all cases, the practical speedup is close to the ideal speedup. ................................................................. 39

5.5 Uplink data symbol processing time. The speedup begins to drop after the uplink processing latency hits the performance lower bound. ............ 40

5.6 Processing time spent on synchronization. (a) Unit test under zero computation overhead. (b) Total time spent on synchronization within a frame using 25 cores. ........................................................... 42

5.7 Breakdown of CPU time spent in functions for 8×8 MIMO. CPU time does not vary much as the number of cores increases, which means MILLIPEDE has a good scalability over number of cores. Each color represents a sub-stage and the sub-stages stacked vertically belong to the same function. (Stacked bars: left: doFFT, middle: doZF, right: doDemod.) 43

5.8 Breakdown of CPU time spent in functions using 8 users and 30 cores. As the number of antennas increases, computation time increases due to a higher computation overhead; data pre/post processing time increases due to a larger data size. Each color represents a sub-stage and the sub-stages stacked vertically belong to the same function. (Stacked bars: left: doFFT, middle: doZF, right: doDemod.) ......................... 44

5.9 Performance bounds of library calls measured by VTune. ............... 45
3.1 PHY pipeline breakdown. ...................................... 26
Chapter 1

Introduction

The use of *dedicated* and *specialized* computing equipment is a long-standing, hard problem of mobile networks. Such use has shaped the industry in two important, undesirable ways. First, it significantly increases the cost of mobile networks, for both operators and subscribers. Operators have to invest billions of dollars upfront for an infrastructure with decent coverage. Subscribers have to pay for the time when the equipment lies idle, which is nontrivial [1–3]. Second, dedicated and specialized equipment leads to slow innovation. The billion dollar capital investment upfront not only raises the barrier of entrance into the industry but also discourages operators from updating the infrastructure when new technologies are available. Moreover, dedicated and specialized equipment is difficult to update because it is distributed and consists of less programmable hardware. These problems are only getting worse with each new generations of mobile networks. A key technology of 5G, massive MIMO, uses a large number of antennas so that many subscribers can use the same frequency band concurrently, improving the spectrum efficiency proportionally [4]. It requires massive computational power at the base station to process the high-speed data streams from all antennas in real-time, often in the form of expensive FPGAs and accelerators. It is not surprising at all that mobile network operators world-wide are slow in actually rolling out massive MIMO for 5G.

This work aims at a future in which *most, if not all, compute necessary for mobile networks is served by commodity data centers*. In doing so, we aim at dramatically changing the cost model of mobile networks from *capital-based* to *expensive-based*, lowering cost,
and accelerating innovation. As the core network functions have already been moving to the cloud, this work focuses on the radio-access networks (RANs), i.e., base stations, which shoulder the dominant majority of the computation necessary for mobile networks. In this vision, base stations should be disaggregated: only the absolutely necessary equipment, i.e., RF parts, will stay on the rooftop, called remote radio unit or RRU and operated by a tower vendor [5], while the compute equipment, called baseband unit or BBU, resides in a data center, operated by a cloud vendor. A mobile network operator can rent time from both vendors so that a startup operator can pay per use, instead of billion dollar capital investment upfront. We note that this vision is a natural yet radical departure from the decade old concept of C-RAN or Cloud-RAN [6, 7] that still leaves much of the physical layer function, and as a result, the majority of the computation, at the base station [8].

**Thesis statement.** Recent CPU advances that support 100s of cores on a single server and advanced architecture extensions provide software sufficient computation power to perform massive MIMO baseband processing. This thesis takes an exploratory step toward moving massive MIMO baseband processing into software to address the challenges of dedicated and specialized hardware. We report the design and implementation of massive MIMO baseband software for many-core Intel x86-based servers, called MILLIPEDE. Evaluation shows that MILLIPEDE significantly outperforms the best publicly known software-based massive MIMO baseband, i.e., BigStation [9], in terms of both throughout and latency. It does so by making careful tradeoffs between data and pipeline parallelisms, optimizing data locality, and exploiting new architectural extensions.

MILLIPEDE is the fastest, most efficient software-based realization of massive MIMO baseband that is publicly known. First, it demonstrates for the first time that massive MIMO (up to 32 basestation antennas) is feasible with a single modern high-end commodity many-core server. Second, it contributes a key insight toward addressing the latency and data rate
challenges in 5G: we should dedicate all CPU resources to first process the earliest frame and maximize the use of data parallelism. This insight distinguishes MILLIPEDE from prior work such as Sora [10] and BigStation [9], where pipeline parallelism is heavily exploited because exploiting data parallelism within massive MIMO baseband processing incurs high inter-core or inter-server communication overhead. When implementing massive MIMO baseband with a single many-core server, such communication overhead is significantly lower and can often be concealed by the intensive computation. Third, to prioritize exploiting data parallelism, MILLIPEDE employs a queue-based master-worker threading model to flexibly control the CPU resource allocation to baseband processing. While this model has been widely employed for web server design [11, 12], MILLIPEDE is its first publicly known application to baseband processing. We carefully design and optimize each the building block, i.e., queues, master thread and worker threads, in the model for baseband processing. To further reduce latency, MILLIPEDE identifies code optimization opportunities and makes non-trivial optimizations that are tailored for massive MIMO baseband. The optimizations includes cache-aware optimizations that improve memory performance and code vectorization with SIMD (Single Instruction, Multiple Data) that accelerates code execution. Based on our experience of designing and implementing MILLIPEDE, we articulate three design principles that are generally applicable to latency-sensitive and throughput-sensitive stream workloads. We also employ performance models of MILLIPEDE to analyze its strength and weakness.

We evaluate MILLIPEDE with two complementary methods. First, we empirically compare MILLIPEDE against BigStation [9] on a same state-of-the-art server. Our experimental results show that MILLIPEDE can achieve much lower latency and higher data rate than BigStation given the same number of CPU cores. For example, given 20 cores, 16×8 MIMO, a 16-QAM modulation order, a 20 MHz bandwidth, and a 5-ms frame length,
in the uplink, both MILLIPEDE and BigStation can achieve a data rate of 454.1 Mbps, the maximum possible given the modulation order and bandwidth. MILLIPEDE finishes processing a frame within 4.999 ms, while BigStation takes 8.199 ms. In the downlink, MILLIPEDE has a latency of 2.667 ms and achieves a data rate of 234.4 Mbps, about 0.6× of the latency and 3× of the data rate by BigStation, respectively. With the insights given by the performance models, we find that the superior performance of MILLIPEDE is because MILLIPEDE maximizes multi-core speedup. We test the scalability of MILLIPEDE over the number of cores up to 30 and the number of base station antennas up to 32. In both cases, the practical performance of MILLIPEDE matches the performance predicted by the performance models. The maximum number of antennas supported is currently limited by the NIC we use, which supports up to 40 Gbps.

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background regarding baseband processing of mobile networks, especially massive MIMO. Chapter 3 and Chapter 4 present the design and implementation of MILLIPEDE, respectively. Chapter 5 reports an empirical evaluation of MILLIPEDE's performance against BigStation and an analytical evaluation of its performance model. Chapter 6 discusses related work. We discuss the limitations of MILLIPEDE and our future plan toward addressing them in Chapter 8.
Chapter 2

Background

We next provide necessary background about radio-access networks (RAN) and the edge data center where we intend to implement RAN functions in software.

2.1 RAN and Software RAN

A modern cellular network consists of two major components: the radio-access network (RAN) and the packet core network. As their names suggest, the RAN provides wireless access to clients via base stations that implement the PHY and MAC layer functions; the core network implements higher layer network functions such as mobility management, billing, and firewall.

In recent years, the core network implementation has already evolved from using specialized equipment to largely software-based realization, i.e., network function virtualization, that runs on shared commodity servers. The RAN, especially base stations, are still ruled by specialized, dedicated equipment, due to the demanding computation requirements of PHY and MAC functions. As a result, a base station often locally integrates all three types of resources necessary for its functions: radio-frequency (such as antennas, amplifiers, and DAC/ADC), hardware (such as FPGA and ASIC for PHY functions), and software (such as servers for MAC functions).

Many have envisioned that future base stations will be disaggregated: only the absolutely necessary equipment, i.e., RF parts, will stay on the rooftop while both hardware
and software can reside in a shared commodity data center in the same metro area. There are strong incentives to disaggregate base stations. First, technologies for RF, hardware and software evolve with different paces. Disaggregation makes it much easier to evolve each independently, accelerating technology adoption. Second, disaggregation makes it possible for multiple cells (base stations) to share computational resources and therefore amortize their cost. Because cells may have uneven workloads, sharing also improves resource utilization and therefore operational efficiency. The cost of RAN dominates the cost of cellular networks and is rising as new generations adopt more computational heavy solutions to improve spectral efficiency. Disaggregation allows this cost to be amortized and reduced. Third, a centralized RAN facilitates advanced schemes for base stations to collaborate (CoMP) to manage inter-cell interference and improve overall network capacity. Finally, when PHY and MAC functions are in a powerful data center outside the base station, it becomes attractive and feasible to implement many of them as software, sharing the data center with other services, which has the potential to further amortize the hardware cost and accelerate technology evolution.

In this project, we investigate a software realization of 5G PHY.

2.2 Baseband

Baseband processing in 5G is responsible for bridging time-domain IQ samples sent/received by the base station and data to/from users.

Similar to wireless PHY in earlier Wi-Fi and 4G/LTE standards, 5G PHY typically consists of a few data processing blocks. Figure 2.1 shows a block diagram of 5G PHY pipeline and its corresponding uplink and downlink frame structures in TDD mode.

As shown in Figure 2.1(b), a frame typically consists fixed-length symbols, The symbol length depends on the number of Orthogonal Frequency-Division Multiplex (OFDM)
subcarriers, where OFDM is the transmission scheme used in 5G. In each symbol duration, a base station simultaneously collects/sends time-domain IQ samples in the uplink/downlink for each of its antennas, where the number of IQ samples depends on the number of OFDM subcarriers and the length of guard interval that is used to eliminate intersymbol interference. The frame length, i.e., the number of symbols in a frame, is often determined by the channel coherence time during which the channel can be considered as unchanged.

There are three types of symbols. A frame usually starts with uplink pilot symbols, from which the base station obtains the channel state information and computes the precoder. The precoder is then used to generate the downlink symbols for each antenna from user data, i.e., precoding, and to extract the data for each user out of the uplink symbols from all antennas, i.e., equalization. Figure 2.1(a) shows the major signal processing blocks for each type of symbols. Precoder calculation is generally the most time consuming block in 5G since it performs expensive operations, e.g., matrix inversion, on the full CSI matrix estimated from multiple pilot symbols. The other blocks have less expensive operations, e.g, the equalization block perform a matrix-vector multiplication. More specifically, for \( M \) antennas serving \( K \) users, the precoder calculation block has a computational complexity of \( O(MK^2) \) due to matrix inverse, while the equalization block only has a complexity of
$O(MK)$. 

### 2.2.1 MU-MIMO and Massive MIMO

To improve spectral efficiency via higher spatial reuse, modern wireless standards employ a technology called multi-user MIMO (MU-MIMO). Using $M$ base-station antennas, MU-MIMO can serve $K$ users concurrently, given $M > K$. To explain how MU-MIMO works, let us use the downlink for example. Let $s$, a vector of $K$ elements, denote the data streams intended for the users. MU-MIMO *precodes* them into $s' = f(s)$, a vector of $M$ elements, each denoting the signal to send from a base-station antenna. While capacity-achieving nonlinear precoding methods have been studied, only linear methods are considered practical in the near future and have been adopted by modern standards such as 802.11, 4G/LTE and 5G. With linear precoding, we have:

$$ s' = Ws $$

where $W$ is a $M$ by $K$ matrix.

The precoding matrix $W$, or precoder, is computed using knowledge of the channel state information between the base-station antennas and users, which is also represented by a $M$ by $K$ matrix, $H$. The most widely studied precoder is *zeroforcing*, which ensures every user only receives the intended data stream, i.e., zero inter-user interference. For zeroforcing, the precoder is computed as

$$ W_{zf} = c \cdot H^* \left( H^T H^* \right)^{-1} $$

where $c$ is a constant factor to ensure no antenna exceeds the maximum allowable transmission power.
There are two challenges facing zero-forcing. First, the time to obtain the channel knowledge $H$ in FDD systems is $O(M \cdot K)$ because the users have to measure the downlink channel and send the measurement to the base station. This challenge is generally true for all MU-MIMO precoders, not just zero-forcing. Second, the precoder computation requires matrix inversion and multiplication that are not only expensive, $O(M \cdot K^2)$ but also difficult to parallelize. Importantly both channel measurement and precoder computation must be done within the channel coherence time during which the channel is supposed to be constant. For mobile users, the channel coherence time can be as short as several milliseconds [13]. As a result, both 802.11 and 4G/LTE have limited $M$: they support up to 8 by 8 MU-MIMO at most.

Because the spectral efficiency (and the cell capacity) improves according to the smaller of $M$ and $K$, there is a strong incentive to scale up the base station to tens or even hundreds, a technology known for 5G as massive MIMO. To address the first challenge of channel measurement, massive MIMO systems often assume a TDD architecture in which the downlink and uplink use the same band. In such systems, downlink channels can be estimated from uplink channels leveraging calibration and channel reciprocity [4]. As a result, the time complexity becomes $O(K)$. To address the second challenge of precoder computation, one can resort to a much simpler but less performant precoding method, called conjugate, where $W = W_{\text{conj}} = c \cdot H^*$, when the channel coherence time is short [14]. Or with increasingly available computational power, especially in the shared data center, one would expect this challenge to become tractable with proper design. This project focuses on how to realize zero-forcing in an efficient and scalable way in software.
2.2.2 Pipeline and Data Parallelism

An obvious way to address the computation challenge in massive MIMO systems is to leverage the rich parallelism in the physical layer and the parallel resources available in modern processors.

Pipeline parallelism exists between any two directly connected data processing blocks when their execution can be overlapped. For example, the execution of the demodulation block for the current symbol can be overlapped with the execution of the equalization block for the next symbol. This parallelism is widely adopted in existing baseband processing implementations, such as Sora [10], BigStation [9], and Atomix [15], to distribute different stages of PHY pipeline to different cores or machines. It improves the overall throughput but does not reduce execution time of a block. It fits well for small-scale systems where processing time of a block is not the bottleneck or distributed systems where communication cost is high.

Data parallelism exists within each data processing block where the same operator can be applied to different subsets of data concurrently. Compared with pipeline parallelism, distributing computations of each block to multiple cores can potentially increase buffer size and result in worse data and code localities. There are abundant opportunities to distribute independent data subsets within a block to multiple cores. The FFT block is antenna-parallel data, performing FFT on the time-domain IQ samples from each antenna independently. After FFT, the processing becomes subcarrier-parallel because modern physical layers employ OFDM signals that divide a wide frequency band, e.g., 20 MHz in 4G/LTE, into narrowband subcarriers, e.g., 15 KHz in LTE, and encode a data stream with each independently. 4G/LTE physical layer has 1200 subcarriers. 5G allows a bandwidth of 100s MHz, leading to many thousands of subcarriers.

We note that data parallelism can be exploited in various granularities in modern pro-
cessors. At the instruction level, most modern processors provide SIMD instructions that can process multiple data points simultaneously. Modern processors also feature multiple cores so that data parallelism can be exploited with current threads. We exploit both levels in this work.

### 2.2.3 Performance Metrics

5G imposes much demanding requirements on the latency and data rate a base station must serve its users. *User plane latency* is the one-way transmit time between a user and the base station. *Data rate* is the number of bits per second communicated between the base station and its users. 5G requires a user plane latency of 1 ms for ultra-reliable and low-latency communications (URLLC) and 4 ms for enhanced mobile broadband (eMBB) [16, 17]. Its target of peak data rate is different for different frequency bands. For sub-6 GHz, which has a maximum bandwidth of 100 MHz, a downlink peak spectral efficiency of 30 bps/Hz, which is the minimum requirement in 5G [16, 17], corresponds to a peak data rate of 3 Gbps.

An implementation of baseband processing impacts on the above two metrics in two related ways. First, there may not be enough computational resources to process the frames at the rate required by the upper layer. This will lead to latency increase (due to queuing) and data rate loss (due to frame drops). Second, even there are enough computation resources, the critical path of the implementation contributes to the latency for uplink and causes potential data rate loss in downlink, due to the base station’s idle time. Because we assume there is abundant processing power to the baseband processing as is in our prototype, we focus on analyzing the impact of critical path latency below.

*Metrics for data rate.* Given abundant processing power, data rate loss only happens in the downlink. Uplink and downlink differ in data rate loss due to their different data depen-
dences between data receiving/transmission at the remote radio head (RRH) and baseband processing at the baseband unit (BBU). In the uplink, data receiving at RRH does not have data dependency on BBU. Therefore, once the BBU’s baseband processing is fast enough to handle all the data received by the base station, there is no data rate loss. In contrast, in the downlink, the RRH needs to wait for downlink data from the BBU to transmit out valid data. Due to the critical path in the baseband processing, it is unavoidable that the RRH has some idle time or wastes some time transmitting invalid data, which results in data rate loss in the downlink. To capture the impact of critical path on data rate, we first introduce downlink transmission latency, \( D_{DL} \), referring to the processing time between frame starts and downlink data is available for base station. Then we can use it to describe the downlink achievable data rate, which is a fraction of the ideal data rate without PHY processing overhead, given as

\[
R_{\text{achievable}} = \frac{T - D_{DL}}{T} R_{\text{ideal}}
\]  

(2.1)

where \( T \) is the frame length.

*Metrics for latency.* The baseband processing time affects uplink and downlink latency differently. In the downlink, processing time does not adds extra latency to a frame’s transmission because frames are sent out by the base station and always have a fixed schedule no matter what the processing time is. In contrast, in the uplink, a frame’s processing time \( T_{\text{processing}} \) and queuing delay \( D_{\text{start}} \) can both contribute to the frame’s user plane latency, where the queuing delay refers to the time a frame is waiting to be processed after arrival. To capture the impacts of both \( T_{\text{processing}} \) and \( D_{\text{start}} \), we use a metric, uplink processing latency, which is the time from the frame’s first packet arrives at the input of PHY pipeline, to when the whole processed frame is available at the output of PHY pipeline, given by
\[ D_{UL} = D_{\text{start}} + T_{\text{processing}} \]  \hspace{1cm} (2.2)

Since the frame itself has a duration \( T \), uplink processing latency would be at least \( T \) even when there is no PHY processing overhead. To describe the extra latency introduced by PHY processing overhead, we introduce a metric, \textit{uplink extra latency}, to remove the impact of frame duration, given by

\[ \Delta D_{UL} = D_{UL} - T \]  \hspace{1cm} (2.3)
Chapter 3

Design

Our goal is to build a high-performance software framework for massive MIMO baseband, called MILLPEDE, that achieves the latency and data rate requirements by 5G. As a portion of 5G’s user plane latency, the uplink processing latency of PHY must be kept small enough to meet 5G’s low-latency goal. We must minimize the data rate loss due to processing latency. In the uplink, our framework should process frames fast enough to avoid frame drop. In the downlink, we should reduce downlink transmission latency to reduce base station’s idle time waiting for data to transmit.

3.1 Design Principles

Prior work has explored software-based baseband processing for small-scale wireless systems [10, 18] or for large MIMO systems that satisfies latency requirement in earlier wireless standards, e.g., 4G [9]. The new latency requirement and adoption of massive MIMO in 5G pose new challenges for processing baseband data in software and render the prior work inadequate. We design MILLPEDE to address these challenges based on the following principles:

Earliest frame first. To meet 5G’s stringent latency requirement, we want to optimize each frame’s processing latency. As we know, a frame’s processing latency depends on both the processing time and the queuing delay. MILLPEDE processes frames in a first-in, first-out (FIFO) order to let each frame get the maximum amount of CPU resources
when being processed, thus minimizing its processing time. This approach can result in longer queuing delay compared to an alternative method that a frame’s processing starts when the frame arrives at the server. However, due to the natural of baseband traffic that frames arrives one after another without overlap, a trivial queuing delay is achievable when MILLIPEDE finishes processing a frame shortly after receiving all the packets in the frame, thus resulting in an optimized processing latency with this principle applied.

*Keep cores busy.* Given a single machine with limited resources in terms of CPU and memory bandwidth, we must efficiently utilize them to deal with the intensive workload in 5G baseband.

*Minimize resources spent on useless work.* that do not advance the useful computation but maintain states properly, e.g., context switch, busy spinning and cache miss.

### 3.2 Not All Principles Are Equal

We note that these principles often do not lead to the same design decisions. To achieve the best performance, design decisions should be tailored for the characteristics of a workload, e.g., computational intensity, processing time distribution and data dependences. It takes delicate balancing between communication overhead and data parallelism to find the best choice for a certain workload. When absolutely necessary, we favor the speedup of data parallelism over communication overhead.

Our design choices (Chapter 3.3) primarily follow Principle 1 to reduce each frame’s processing latency. To follow Principle 1, i.e., to let the earliest frame use all available cores when being processed, we must exploit data parallelism to distribute a frame’s workloads to all available cores, which results in our occasional violation of Principle 2 and 3. The violation of Principle 3 happens because of the unavoidable message synchronization and data communication overheads introduced by the use of data parallelism while
the violation of Principle 2 happens when a core responsible for task dispatch is stalled due to message synchronization and other cores waiting for new tasks become idle. These extra overheads and extra core idle time are small compared to the intensive computation overhead of massive MIMO baseband processing and thus only slightly hurt the overall performance.

There are alternative design choices if we give the priority to Principle 2 or 3. (i) To make cores as busy as possible, we need to make sure the cores always have available tasks to execute, which leads to a design choice that new tasks are created as early as possible once their data dependences are satisfied. This could run against Principle 1 because these new tasks may belong to a later frame. Furthermore, load balance also matters to make all cores equally busy, which may lead to a design choice that allows core-to-core task stealing. This could violate Principle 3 because task stealing introduces extra communication overhead. ZygOS [19] is an example that favors load balance over communication overhead. (ii) Targeting at minimize useless work leads us to assign a frame to a single core so that communication overhead is minimized. With such a design choice, following Principle 1 is not possible on a multicore machine. It is still possible to apply Principle 2 by assigning different frames to different cores, which violates Principle 1 because now the earliest frame and the later frames are processed simultaneously. Principle 3 is favored in prior work on software-based network packet processing such as RouteBricks [20] and IX [21], whose workloads have a short and deterministic processing time.

The above design choices made by primarily following Principle 2 or 3 are not desirable in this work because they can result in longer frame processing time while low latency is a major performance goal we want to achieve.
3.3 MILLIPEDE: Design Overview

We present a parallel computing framework called MILLIPEDE for massive MIMO baseband processing. MILLIPEDE follows the principles described above to achieve both low latency and high data rate. Unlike BigStation, which distributes baseband processing to multiple machines with a combination of data and pipeline parallelism as shown in Figure 3.1(b), MILLIPEDE assumes a single many-core machine and relies primarily on data parallelism as shown in Figure 3.1(a).

MILLIPEDE employs a queue-based master-worker model, which consists of a master thread, a pool of worker threads, and multiple FIFO queues. The master thread generates jobs, which are then consumed by the worker threads. There are two groups of worker threads, one for networking and the other for computation. We dedicate a minimum number of threads for networking, just enough to support baseband traffic. For example, one core is enough to support 8x8 LTE-advanced traffic that goes through 40 GbE NICs.

The master thread and the worker threads communicate through the FIFO queues consisting of messages. A message is a small piece of data that includes a job type and a location offset. A worker thread decides which computation job to execute and which data
location to access based on the information within the message it dequeues. The master thread maintains a set of execution statuses to track the progress of PHY pipeline processing and schedule new jobs. It relies on the replied messages from the worker threads to update the statuses. This model has been shown to be effective in handling heavy traffic with low latency in web server applications [11, 12]. MILLIPEDE is the first to apply it to wireless baseband processing considering the new challenges in 5G and the trend of modern hardware that a single machine tends to have more and more cores.

We note the prior work on software-based baseband processing does not adopt the model behind MILLIPEDE, for two reasons. First, this model could introduce significant overhead due to message synchronization and data communication when implementing earlier wireless protocols (e.g., 802.11a [10]) in software, which is less computationally intense than massive MIMO baseband by several orders of magnitude. The only software implementation of large-scale MIMO with 12 antennas, i.e., BigStation, focuses on its architecture’s scalability in distributed systems and each machine only has a small number of cores, where the synchronization and communication overhead is significantly higher. Therefore, BigStation chooses to spread the PHY pipeline across multiple machines and lets each machine process a subset of blocks, as shown in Figure 3.1(b).

In order to keep cores busy, we pin every thread to a physical core and keep all the threads busy executing jobs. The threads are alive until program exits and each thread has a one-to-one mapping to a physical core to avoid CPU resource competition. An alternative of keeping cores busy is to allow OS to migrate threads to idle cores, which could introduce large context switch overhead and poor cache locality [22].

To keep the worker threads busy, MILLIPEDE breaks down data processing blocks in the PHY pipeline into smaller jobs by exploiting data parallelism and distributes them to all worker threads. Each worker thread runs an infinite loop repeatedly looking for
jobs to execute while the job dispatcher generates jobs in time to make sure the worker
treads always get new jobs. We use dynamic scheduling to compensate for performance
asymmetry among multiple cores so that all the worker threads are kept equally busy. A
set of single-producer multiple-consumer queues are employed to hold different types of
jobs generated by the master thread. They can be accessed by all the worker threads so that
each job is dynamically assigned to whichever idle thread.

MillipeDE reduces CPU time wasted on useless work through a series of optimiza-
tions. In order to remove unnecessary synchronization between the job dispatcher and
worker threads, the data processing blocks are grouped and small jobs are merged for ex-
ecution. We reduce the cost due to message synchronization and data communication be-
tween the threads through lock-free queues and lock-free data access in shared memory. We
exploit cache locality to reduce cache misses, which largely facilitates data access. None
optimizations are novel per se but are crucial in achieving MillipeDE ’s high performance.

3.4 Model-driven Configuration

Here we introduce performance models based on a single core for uplink and downlink,
respectively. We will use them to analyze MillipeDE ’s scalability on a multi-core system
and show the benefit of applying the earliest frame principle by comparing MillipeDE
against an alternative design choice made by BigStation. In the models, we consider ideal
speedups and assume zero overhead of message synchronization between the master thread
and worker threads and data communication between worker threads. In practice, however,
these overheads are unavoidable and will reduce the scalability of MillipeDE to deviate
from model predictions. We will show how we reduce these overheads in Chapter 3.5.
### 3.4.1 Single-core Performance Models

We first mathematically model single-core performance, i.e., uplink processing latency and downlink transmission latency, which are two key performance metrics as discussed in Chapter 2.2.3. The models capture the data dependences of frame structure and the speedup limitations introduced by them.

To simply the discussion, we group each frame’s processing blocks into three pipelined procedures: (i) **Pilot**: pilot symbol processing, (ii) **ZF**: precoder calculation using zero-forcing as the precoding method, and (iii) **DataProc**: uplink/down data symbol processing. We use $T_{\text{Pilot}}$, $T_{\text{ZF}}$ and $T_{\text{ULData}}/T_{\text{DLData}}$ to refer to processing time spent in these procedures, respectively.

Since **Pilot** is a common procedure for uplink and downlink, we first model its completion time of $T_{\text{Pilot}}$. Due to its data dependence on pilot symbol data, it is lower bounded by pilot receive duration. The model of **Pilot** is then given as

$$T_{\text{Pilot}} = \max(D_{\text{start}} + T_{\text{ChEst}}, \frac{s_p - 1}{s}T + t_{RX})$$  \hspace{1cm} (3.1)

where $D_{\text{start}}$ is the queuing delay from a frame’s arrival to when its processing starts, $T_{\text{ChEst}}$ is the total time spent on pilot symbols’ FFT and channel estimation within a frame, $s_p$ is the number of pilot symbols, $s$ is the total number of symbols, and $t_{RX}$ is the duration to receive all packets within a single symbol, which is a ratio of the total size of a symbol to NIC throughput. Note that the lower bound in Eq. 3.1 is an approximation, where we have ignored the extra time spent on processing the last arrived packet since it is negligible compared to other items in the equation.

Next, we look at the overall uplink and downlink processing time.

**Uplink.** We use the three procedures in the uplink to model the uplink processing
latency. Due to data dependence, it is lower bounded by frame receive duration. The model of uplink processing latency is then given as

$$D_{UL} = \max(T_{Pilot} + T_{ZF} + T_{ULData}, \frac{s - 1}{s}T + t_{RX})$$  (3.2)

Similar to Eq. 3.1, in Eq. 3.2 we have ignored some extra time spent on processing the last data symbol.

Downlink. We use the three procedures in the downlink to model the downlink transmission latency. Due to the frame-based schedule at the base station, it is upper bounded by the frame duration, $T$, corresponding to a downlink data rate of $0$. The model of downlink transmission latency is then given as

$$D_{DL} = \min(T_{Pilot} + T_{ZF} + t_{DLData}, T)$$  (3.3)

where $t_{DLData}$ is one downlink data symbol’s precode and IFFT time. We ignore modulation, encoding and scrambling time in this equation since they do not have data dependence on the precoder and therefore can be performed anytime when CPU is not busy.

3.4.2 Multi-core Pipeline Parallelization

Different design choices of parallelizing the PHY pipeline to multiple cores can result in different frame processing latencies. Here we compare the design choices of MILLIPEDE and BigStation to demonstrate the advantage of MILLIPEDE in achieving 5G requirements.
MILLIPEDE maximizes data parallelism by using all cores to process the PHY pipeline of the earliest frame. As discussed in Chapter 2.2.2, rich data parallelism within each data processing block allows us to distribute each processing block to all cores attached with worker threads. The worker threads can then be scheduled to sequentially process blocks of the earliest frame and move to the next frame only after the current frame is finished. Figure 3.2(a) shows an example of MILLIPEDE ’s frame processing order in the uplink.

Due to high inter-machine communication overhead, BigStation favors the use of pipeline parallelism and limits the use of data parallelism. It distributes blocks to different groups of machines and parallelize a block’s processing only within the group of machines it is assigned to. Figure 3.2(b) depicts the uplink frame processing schedule using this data and pipeline parallel combined design. Compared to our data-parallel only design, two major differences are queuing delay and frame processing time. In BigStation’s design, frames has no queueing delay, i.e., each frame processing starts immediately once its data arrives, which is because the core subset responsible for ChEst has finished processing the previous frame when the current frame arrives. However, it takes longer to process a frame in this design because each block is parallelized to less cores.

**Multi-core speedup.** Next we mathematically explain why MILLIPEDE and BigStation differ in processing time. The fundamental reason is that they differ in speeding up the parallelizable items, i.e., $T_{ChEst}, T_{ZF}, T_{ULData}$ and $t_{DLData}$, in Equation 3.2 and Equation 3.3. To simplify the discussion, we consider an ideal speedup where these items can be accelerated by a factor of the number of cores they are distributed to. In MILLIPEDE ’s data parallel only design, each parallelizable item gets a speedup of $n$, where $n$ is total number of cores. The parallelized computation time of each item is given as

$$T_i(n) = \frac{T_i(1)}{n}, T_i(1) \in \{T_{ChEst}, T_{ZF}, T_{ULData}, t_{DLData}\}$$ (3.4)
Figure 3.2: Uplink frame processing schedule under different parallel schemes. MILLIPEDE’s data parallel only design processes frames without overlap. BigStation’s data and pipeline parallel design allows simultaneous execution of different procedures in different frames.

Where $T_i(1)$ refers to the computation time of $i$th item on a single core.

In BigStation’s data and pipeline parallel design, each item is parallelized to a core subset with $n_i (n_i < n)$ cores. Therefore, each item has a longer parallelized computation time, $T_i(n_i) (T_i(n_i) > T_i(n))$.

**Required number of cores.** The speed up difference in MILLIPEDE and BigStation results in their difference in the minimum number of cores required to achieve a certain performance, i.e., uplink processing latency or downlink data rate. MILLIPEDE can achieve more efficient CPU resource usage than BigStation.

In the uplink, the lower bound of processing latency results in a cost-performance trade-off, i.e., after increasing the number of cores to a certain boundary, $T_{\text{processing}}$ is bounded by the frame structure (as shown in Equation 3.2), thus leading to little latency reduction by further increasing the number of cores. With a larger degree of multi-core speedup,
MILLIPEDE can reach this boundary with much less cores than BigStation.

In the downlink, to achieve a certain data rate $R_{\text{achievable}}$ (defined in Equation 2.1), there is a minimum requirement of number of cores to reduce $T_{ZF}$ and $t_{DLData}$ in Equation 3.3. Due to the larger degrees of speeding up $T_{ZF}$ and $t_{DLData}$, MILLIPEDE has the advantage to achieve the required data rate with less cores than BigStation.

**Impact of communication overhead.** In the performance models, we do not include the communication overheads. Since MILLIPEDE distributes each block to more cores, it is possible that MILLIPEDE has higher data communication and message synchronization overheads than BigStation. How much the actual difference of communication overhead largely depends on the practical implementation and the platforms MILLIPEDE and BigStation are deployed on. If considering a many-core server for MILLIPEDE and a distributed system where each machine has a small number of cores for BigStation, MILLIPEDE can always have lower communication overhead than BigStation. If MILLIPEDE and BigStation are both deployed on a many-core server, however, the communication overhead in MILLIPEDE can be higher than BigStation. Even in this case, parallelizing to more cores is still beneficial if the communication overhead is relatively small compared with the intensive computation overhead.

### 3.5 Reducing Useless Work

Here we show the key methods we use to reduce message synchronization and data communication overheads.

#### 3.5.1 Coarsening Job Granularity

Data processing blocks in the pipeline have very different computation time. When distributing a small block to multiple cores, synchronization overhead may kill speedup. To
keep the execution of small blocks parallelized while minimizing communication costs, we merge small blocks based on data dependences and data dimensions they operate on. The basic rule is: a thread should execute blocks along the pipeline sequentially until data dimension for the next block is different from the current block or new data dependence arises. An example of merging blocks in the uplink based on this rule is shown in Figure 3.3. There are two benefits of doing this. First, we can reduce data communication overhead between adjacent blocks since data migration across cores is avoided. Second, we can eliminate unnecessary synchronization cost since the worker threads do not need to send synchronization message back to the master thread before the current execution ends.

Figure 3.3: Uplink pipeline parallelization with different job granularities. Merging blocks operating on the same data dimension can reduce the number of synchronization points and data movement across cores.

As shown in Table 3.1, data processing blocks in both uplink and downlink can then be divided into five stages, where ChEst and ZF are shared by uplink and downlink. We define job as the processing unit at the worker threads. The choice of job granularity of each stage is based on the tradeoff between synchronization cost and computation overhead. For stages with less computation overhead, e.g., Demod, we combine the computation of multiple subcarriers into one job while keeping the number of jobs large enough to be
distributed to all worker threads.

Table 3.1 : PHY pipeline breakdown.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Blocks</th>
<th>Data parallelism</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChEst</td>
<td>FFT, ChannelEst</td>
<td>Antenna</td>
<td>per ant</td>
</tr>
<tr>
<td>ZF</td>
<td>PrecoderCal</td>
<td>Subcarrier</td>
<td>per SC</td>
</tr>
<tr>
<td>FFT (UL)</td>
<td>FFT</td>
<td>Antenna</td>
<td>per ant</td>
</tr>
<tr>
<td>Demod (UL)</td>
<td>Equalization Demodulation</td>
<td>Subcarrier</td>
<td>per (m) SCs</td>
</tr>
<tr>
<td>Decode (UL)</td>
<td>Decoding</td>
<td>Code block</td>
<td>per block</td>
</tr>
<tr>
<td>Mod (DL)</td>
<td>Encoding Modulation</td>
<td>Code block</td>
<td>per block</td>
</tr>
<tr>
<td>Precode (DL)</td>
<td>Precoding</td>
<td>Subcarrier</td>
<td>per (m) SCs</td>
</tr>
<tr>
<td>IFFT (DL)</td>
<td>IFFT</td>
<td>Antenna</td>
<td>per ant</td>
</tr>
</tbody>
</table>

3.5.2 Lock-free Data Access

Since the worker threads all need to load or store data from the main memory when processing the PHY pipeline, data race may happen if the worker threads are accessing the same piece of data simultaneously. We avoid data race without using locks by ensuring only one thread is accessing a location in the main memory at any time. This is achieved through data parallelism and the centralized job dispatcher. With data parallelism, the worker threads operate on independent datasets, which ensures that no two threads access the same piece of data at the same time. In the centralized task dispatcher, we use a set of checkpoints for data dependence check and only generate new tasks when data dependence is satisfied, which guarantees that data will only be accessed after the previous access is done.
3.5.3 Cache-aware Optimizations

Temporal locality for intermediate results. After we merge blocks into stages, the outputs from the merged blocks become temporary results that only belong to the thread which executes the corresponding job. This gives us an opportunity to utilize cache temporal locality. We allocate a minimum piece of buffers to each thread to hold the temporary results. With a small size, these buffers have more chance to stay in caches, thus largely reducing the costs of data load and store.

Temporal and spatial locality for matrix transpose. We exploit both temporal and spatial localities to speed up matrix transpose, which is required between two data processing blocks operating on different data dimensions. We use the CSI matrix transpose between the FFT and ZF stages to explain the benefit of doing this.

A naive way is to directly transpose the whole matrix. To take care of the data dependence of the CSI matrix on all subcarriers, all pilots and antennas, we need to add an extra matrix transpose stage between FFT and ZF. Thus, before executing transpose, the worker threads must finish the FFT stage for all antennas and all pilots, and the master thread must synchronize all results and generate the new tasks execute the matrix transpose. This approach adds extra data movements, i.e., from multiple cores that execute the FFT stage to cores that perform the transpose and then back to cores that execute the ZF stage. It also adds CPU idle time to the worker threads due to the extra synchronization overhead.

We notice that these data movements and the extra synchronization overhead can be easily avoid if we directly store results in the FFT stage to corresponding locations in the transposed matrix. The ZF stage then only needs to access data in the transposed matrix for its computation. However, this still involves some redundant work. When two worker threads are simultaneously storing data of the same subcarrier but different antennas, they may access the same cache line, which leads to cache false sharing.
We avoid cache false sharing by exploiting the spatial locality of cache lines. More specifically, we let the FFT stage perform part of the transpose by storing cache lines consisting of adjacent subcarriers to a partially transposed matrix and let the ZF stage finish the transpose by gathering data from the partially transposed matrix.

*Non-temporal output data.* The outputs of all the stages do not have temporal locality since they are possible to be consumed by whichever threads in their next stages. Data movement from one core’s cache to memory and to another core’s cache is expensive. A better approach is to bypass the caches and directly write output data to memory, which shortens the data movement path. We use SIMD instructions with non-temporal hint to explicitly control data store.
Chapter 4

Implementation

We implement MILLIPEDE from scratch in C and C++. The baseband processing framework contains 4K lines of code, running as a user-space application in Linux. MILLIPEDE relies on pthreads APIs and a fast lock-free concurrent queue library [23] to create the threading model as described in Chapter 3.3. pthreads, in contrast to automatic parallelization APIs such as OpenMP, can give us the maximum flexibility of thread and data management. An efficient implementation of FIFO queues is the key to achieve low synchronization overhead in MILLIPEDE. We implement two types of queue using the lock-free queue library. The job queues are single-producer, multi-consumer, where the master thread is the only producer and the worker threads are the consumers. The result queue is single-consumer, multi-producer, where the master thread is the only consumer and the worker threads are the producers.

We implement each type of job in Table 3.1 as a function that can be executed by any of the worker threads. We name the functions as doFFT, doZF, doDemod, doDecode, doMod, doPrecode and doIFFT respectively. We combined ChEst and FFT into one function, i.e., doFFT, since they share a large piece of code. Each function reads input data from a buffer location in shared memory, performs computation, and writes output data back to another buffer location in shared memory. We use a set of ring buffers to hold data from a few frames so that the framework is able to tolerate performance variations and jitters. MILLIPEDE iterates through the ring buffers in runtime and always processes data of the oldest frame. The computations are implemented with state-of-the-art high-performance
libraries. We use Intel Math Kernel Library (MKL) [24], which has been optimized for state-of-the-art SIMD architectures, for the computation of FFT, and a linear algebra library, Armadillo [25] linked with Intel MKL, for matrix operations.

To support real-time requirement of baseband processing, we make the following efforts to improve the performance of our framework.

*Application level.* We align memory allocation of buffers to avoid cache false sharing among threads. We rely on both automatic and manual code vectorization to speed up processing. With an -O3 option, the compiler can automatically take care of basic optimizations and vectorize the SIMD-friendly libraries to achieve the maximum speedup. For operations that cannot utilize the high-performance libraries, including data type conversions in `doFFT` and `doIFFT`, and computations in the channel estimation block, we manually implement and vectorize them with SIMD intrinsics, which is about 40 lines of code. Because these operations have shorter latency than that of memory access, we interleave them with memory access to hide their latency. We use 256-bit AVX2 instructions, which are the maximum-width instructions supported by our testbed, to maximize data parallelization.

*OS level.* We run our program as an RT process to remove negative impacts from context switches. We turn off turbo boost and CPU idle states to reduce CPU variations. To improve load balance, we turn off hyper threading and only use one thread on each core.

*Networking stack.* We apply a series of optimizations for the Linux networking stack. We attach NIC TX/RX queues to the cores assigned for TX/RX threads and explicitly steer packets to these cores. To steer the packets, we use NIC’s ntuple filtering feature for RX flows and Transmit Packet Steering (XPS) feature for TX flows. We set MTU size to 9000 to avoid packet fragmentation. To reduce the packet transmission latency between NIC and MILLEPEDE, we disable interrupt coalescing.
Chapter 5

Evaluation

We aim to answer the following questions via prototype-based evaluation: (i) How good is MILLIPEDE in practice, i.e., what latency and data rate can be achieved? (ii) Why MILLIPEDE is good? (iii) What are the performance bottlenecks of MILLIPEDE?

5.1 Methodology

High performance packet generator. We implement a high performance packet generator in software to emulate a massive MIMO base station, especially the ArgosV3 base station. As introduced in Chapter 2.2, a base station follows a symbol-based frame structure to produce/consume time-domain IQ samples. In the ArgosV3 base station, in each symbol duration, the base station uses $M$ UDP packets to send/receive the IQ samples of all $M$ antennas to/from the server, one packet per antenna. Each packet contains data of a single symbol. To emulate the base station, the packet generator produces a UDP packet per antenna and generate a frames over multiple consecutive UDP packets. Each packet consists of a 64-byte header to store frame, symbol and antenna indexes, and $p$ 32-bit IQ samples, where the indexes are the necessary information to record where a packet comes from and $p$ is equal to the number of OFDM subcarriers. In the packet generator, $M$, $p$, symbol duration, number of symbols in a frame and frame duration are all configurable parameters. The packet generator uses the RDTSC instruction, which gives 64-bit CPU timestamps, to precisely control the packet duration that determines the symbol duration. We can then
control the frame duration by changing the packet duration and the number of symbols in a frame. According to our measurements, the deviation between measured frame duration and desired frame duration is less than 1 µs, e.g., for a 5-ms frame with 70 symbols, the average error is 0.2 µs with a standard deviation of 0.26 µs.

**Baseline.** We use BigStation [9] as the baseline and compare MILLIPEDE against it on a same many-core machine. BigStation’s original implementation targets at a distributed system where each machine only has a small number of cores. In such a system, the synchronization cost among multiple machines is a major bottleneck to prevent BigStation from making intensive use of data parallelism. The recent advances of CPU architectures allow a single server to equip with 100s of cores, which makes it possible to move all baseband processing into a single machine. Therefore, we implement our own version of BigStation on a single many-core machine, where the synchronization cost has been significantly reduced, while remaining its use of both pipeline and data parallelisms. We compare MILLIPEDE, which maximizes the use of data parallelism, against BigStation, which limits the use of data parallelism in favor of pipeline parallelism, to see whether and how the lower synchronization cost changes the balance between exploiting data vs. pipeline parallelism. To allow a fair comparison, we implement BigStation on the same server as MILLIPEDE and let it share the same low-level implementation, e.g., functions and libraries. The only difference between our implementations of BigStation and MILLIPEDE is the core assignment policy. In our implementation of BigStation, we assign each stage of the PHY pipeline, as shown in Table 3.1 to a fixed, dedicated group of cores, which then exploit data parallelism within the stage. As a result, non-overlapping groups of cores exploit the pipeline parallelism by processing different stages of the PHY pipeline simultaneously. In contrast, in MILLIPEDE we maximize data parallelism by allowing each stage to be processed on any available core.
Experimental setup. In our experiments, we use two Supermicro servers directly connected through 40 Gbps Ethernet NICs. We run the baseband processing framework on one server and the packet generator on the other. The symbol and frame configurations we use are similar to those in LTE, i.e., 20 MHz bandwidth, 16-QAM modulation order, 2048 subcarriers, within which 1200 subcarriers carry valid data and the others are used as guard bands to prevent inference. We use the packet generator to generate packets following an LTE-like symbol-based frame structure. We use a frame length of 5 ms to evaluate MILLIPEDE’s performance. This frame length allows the channel to be measured every 5 ms, necessary even for pedestrian mobility, which has a coherence time of 7 ms [13]. As shown in Figure 2.1(b), each frame consists of pilot symbols and data symbols. A 5-ms frame consists of 70 symbols. We implement time-orthogonal pilots where a pilot symbol only contains the pilot from a single user. Therefore, the total number of pilot symbols is equal to the number of users. For data symbols, we consider two extreme cases where a frame only has uplink data symbols and where a frame only has downlink data symbols symbols, to separately evaluate uplink and downlink performance. In the uplink, we run end-to-end experiments with the packet generator generating pilots and uplink data while the framework consuming them. In the downlink, the framework receives pilots from the packet generator, consumes them and then generates downlink data. We evaluate downlink data rate based on the downlink transmission latency measured within the framework, which is not end-to-end yet but allows us to compare the data rate loss introduced by different framework designs, i.e., BigStation and MILLIPEDE.

Performance measurement. We use the RDTSC instruction to measure timestamps for performance profiling. Since the master thread is responsible for tracking pipeline processing progress, we insert timestamps in it to measure a frame’s overall performance, i.e., uplink processing latency or downlink transmission latency, as well as its breakdown
into smaller procedures, e.g., pilot processing time. We insert timestamps in the worker
threads to measure processing time spent in computation functions, e.g., doFFT. For each
experiment, we collect data from 8000 frames and average the timestamps.

5.2 Overall Performance

We first evaluate the black-box performance of MILLIPEDE. Our experimental results
show MILLIPEDE significantly outperforms BigStation, achieving lower uplink processing
latency and higher downlink data rate.

5.2.1 Uplink Processing Latency

We first show MILLIPEDE achieves shorter uplink processing latency than BigStation, even
with fewer cores. We measure the uplink processing latency of MILLIPEDE and BigStation
with different number of cores that are attached with worker threads. We vary the number
of cores from a minimum number, which avoids frame drop, to a number that hits the per-
formance bound indicated in Equation 3.2. For BigStation, given a total number of cores,
we adjust the numbers of cores assigned to different groups to find a best core allocation
policy that allows BigStation to achieve the lowest processing latency. We also vary the
number of base station antennas from 8 to 32 for both MILLIPEDE and BigStation.

Figure 5.1 plots the uplink frame processing latency over number of cores. MILLIPEDE
significantly outperforms BigStation in two aspects: (i) Given the same number of cores,
MILLIPEDE can achieve much lower latency than BigStation, especially when the number
of cores is small, e.g., with 13 cores the latency difference can be larger than 4ms. (ii) MIL-
LIPEDE can achieve a small latency, e.g., 100 µs, with much fewer cores than BigStation.
To achieve less than 100 µs latency, MILLIPEDE only needs 13 cores for 8×8 MIMO, while
BigStation needs 25 cores. As the number of base station antennas increases, BigStation
Figure 5.1: Uplink processing latency of MILLIPEDE and BigStation. MILLIPEDE needs much fewer cores than BigStation to hit the performance bound, i.e., 5-ms processing time, of a 5-ms frame.

requires more than 30 cores to achieve less than 100 µs latency, which can easily exceeds the number of cores on a single server.

5.2.2 Downlink Transmission Latency and Data Rate

We next show MILLIPEDE can achieve shorter downlink transmission latency and higher downlink data rate than BigStation. Figure 5.2 depicts the maximum achievable data rate and downlink transmission latency. The data rate is computed from the downlink transmission latency. We first use the downlink transmission latency to compute the number of downlink data symbols left after downlink transmission starts and before a frame end, and then convert this number to the data rate. Note that this data rate is an ideal estimate under the given downlink transmission latency. In practice, we expect it to be lower considering the data transmission latencies from the server to the base station and real-world channel conditions. Since BigStation and MILLIPEDE suffer from these sources of data rate loss in the same way, we ignore them in Figure 5.2 and only focus on comparing their data rate loss caused by the downlink transmission latency. Similar to the uplink performance, MILLIPEDE significantly outperforms BigStation to achieve shorter transmission latency and
higher data rate, especially when there is a small number of cores. In the downlink, what is different from the uplink performance is that adding more cores is always beneficial. With 30 cores, MILLIPEDE can achieve 74.2% of the ideal 16QAM data rate while BigStation can only achieve 64.5%. As the number of base station antennas increases, the performance gap between MILLIPEDE and BigStation also grows, meaning that MILLIPEDE outperforms BigStation more when processing heavier workloads.

5.3 Performance Breakdown

In this section, we breakdown the overall performance with the help of the performance models discussed in Chapter 3.4. We seek to get insights about why MILLIPEDE is better than BigStation. As discussed in Chapter 3.4, the main advantage of MILLIPEDE is that it allows each parallelizable item to parallelize to all cores. Here we test whether this is true in practice. We look at the durations of major procedures that determine the overall performance: pilot symbol processing, zero forcing, and uplink data symbol processing. We ignore the downlink data symbol processing procedure in Equation 3.3 because it is
only for a single symbol and is small compared to other procedures. We evaluate how the multi-core speedup of the parallelizable items, $T_{ChEst}$, $T_{ZF}$ and $T_{ULData}$, in Equation 3.4 can help improve the overall performance.

5.3.1 Pilot Symbol Processing ($T_{Pilot}$)

We measure pilot symbol processing time as well as pilot receive duration of both BigStation and MILLIPEDE. For BigStation, we measure the performance under two different configurations of the number of cores assigned to execute doFFT: (i) using the minimum number of cores that allows the framework to work, which is the case under the best core assignment policy; (ii) using 5 cores that are enough to hit the lower bound. We vary the number of base station antennas to test how it affects the lower bound. In all experiments, we use 30 cores that are attached with worker threads.

Figure 5.3 plots the pilot processing time and pilot receive duration as the number of antennas increases. For the pilot receive duration, we see a good match between the theoretical values and the measured values for both BigStation and MILLIPEDE. It increases linearly over the number antennas because the total size of packets in a symbol increases linearly. We also observe that the pilot processing time results of BigStation with 5 cores for doFFT and MILLIPEDE are both very close to the pilot receive time. The small gap, which is around 10 µs, is the time spent on processing the last arrived packet and message synchronization. For BigStation only using the minimum numbers of doFFT cores, the gap becomes slightly larger. The takeaway message from this figure is that the pilot processing time $T_{pilot}$ in Equation 3.1 is mainly limited by the frame structure, not the multi-core speedup of the parallelizable item $T_{ChEst}$. In most cases, we can use the pilot receive duration to approximate $T_{pilot}$. 
Figure 5.3: Pilot symbol processing time over number of antennas using 30 cores. In all cases, pilot processing time is lower bounded by pilot receive duration.

5.3.2 Zeroforcing ($T_{ZF}$)

We evaluate how the practical speedup of $T_{ZF}$ matches the ideal speedup expected in the performance models. According to Equation 3.4, the ideal speedup is equal to the number of cores assigned to execute $doZF$. We evaluate the speedup for both BigStation and MILLIPEDE. For BigStation, we use best core assignment policies for uplink and downlink, which results in different number of cores assigned to execute $doZF$ in the uplink and downlink.

Figure 5.4 plots the speedup of $T_{ZF}$ over the total number of cores. We observe that the practical speedup closely matches the ideal speedup in all cases. As the number of antennas increases, the deviation between practical and ideal values becomes larger. In addition, we can see that MILLIPEDE has significantly larger speedups than BigStation when using the same total number of cores in both uplink and downlink. For example, when using a total of 15 cores in the uplink, MILLIPEDE can achieve a speedup of 14.42, corresponding to a compute time of 2047 $\mu$s, while BigStation can only achieve a speedup of 6.96, corresponding to a compute time of 4262 $\mu$s.
Figure 5.4: ZF processing time in uplink and downlink with 8 base station antennas. The ideal speedup in both (a) and (b) is equal to the number of cores assigned to execute doZF. In all cases, the practical speedup is close to the ideal speedup.

5.3.3 Uplink Data Symbol Processing ($T_{ULData}$)

The evaluation method of $T_{ULData}$ is similar to that of $T_{ZF}$. However, the speedup of $T_{ULData}$ over the total number of cores, as shown in Figure 5.5, has very different trends from those in Figure 5.4. In both BigStation and MILLIPEDE, the speedup stops to grow when the total number of cores hits the lower bound of uplink processing latency. The significant drop of MILLIPEDE’s speedup when using more than 13 cores is due to the increasing speedup of $T_{ZF}$ and a fixed 5-ms frame duration. Figure 5.5 together with Figure 5.4 indicate that the lower bound of uplink processing latency only impacts the uplink data symbol processing procedure while the zeroforcing procedure can still benefit from adding more cores. Another observation from Figure 5.5 is that the speedup of $T_{ULData}$ in MILLIPEDE significantly outperforms that in BigStation.

5.3.4 Explanation of MILLIPEDE’s Advantages

With the previous model-based evaluations on the procedures of pilot symbol processing, zeroforcing, and uplink data symbol processing, we can explain the reasons why MILL-
Figure 5.5: Uplink data symbol processing time. The speedup begins to drop after the uplink processing latency hits the performance lower bound.

MILLIPEDE significantly outperforms BigStation. In the uplink, the speedups of $T_{ZF}$ and $T_{ULData}$ in MILLIPEDE are much larger than those in BigStation, thus leading to much shower overall uplink processing latency. In the downlink, the advantage of MILLIPEDE is mainly in the ZF procedure, which lets it to achieve much higher data rate than BigStation.

5.4 Performance Bottlenecks of MILLIPEDE

In this section, we evaluate the sources that result in MILLIPEDE’s performance loss. Two major sources are message synchronization and data communication. Since the performance of MILLIPEDE heavily relies on the external library calls from Intel MKL, we also look at the performance bounds of the library calls.

5.4.1 Message Synchronization Cost

To evaluate the synchronization cost, we first do a unit test by removing all the data access and computation operations in the computation functions and remain enqueue, dequeue and the pipeline scheduling procedures in the master thread. The measured latencies come from the synchronization operations, which include enqueue, dequeue and status tracking in the master thread, and enqueue and dequeue in the worker threads. We assign each worker
thread with a single job and repeat the job dispatch and result synchronization processes for 8000 times and average the results. We vary the number of cores attached with worker threads to evaluate its impact on synchronization cost. As shown in Figure 5.6(a), the synchronization cost increases with more cores. This is because contentions among worker threads happen more frequently when more threads access the same queue concurrently.

We then look at the total time spent on synchronization when processing a frame. We use the uplink performance as an example. The synchronization cost is computed from the total frame processing time minus the time spent in computation functions doing data access and compute operations. Such a computation is only reasonable when the uplink processing latency is not constrained by the frame structure. Therefore, we reduce symbol duration in our measurements to avoid hitting the performance bound.

Figure 5.6(b) shows that the synchronization cost grows almost linearly as the number of antennas increase, indicating that adding more antennas reduces MILLIPEDE's scalability. When using 32 antennas, the synchronization cost occupies 5.5% of the total processing time. We find the cost mainly comes from the synchronization of FFT results. With more antennas, there are more FFT jobs. Hence, the worker threads are more likely to have contentions.

5.4.2 Data Communication Overhead

In this evaluation, we exclude the synchronization cost and only measure processing time within computation functions. To isolate the data communication overhead, we breakdown the functions into two or three sub-stages. The sub-stages can be categorized into three groups: computation, data pre-processing and data post-processing, where data pre-processing and data post-processing mainly performs memory operations. The computation sub-stage performs computation-intensive operations. It includes our own implemen-
Figure 5.6: Processing time spent on synchronization. (a) Unit test under zero computation overhead. (b) Total time spent on synchronization within a frame using 25 cores.

Impact of number of cores. We use 8 antennas in this experiment. Figure 5.7(a) plots the total CPU time of all cores spent in the three uplink functions when processing a 5-ms frame. Figure 5.7(b) plots the corresponding CPU time for a thread to execute a single task, i.e., processing data of one antenna in doFFT and one subcarrier in doZF and doDemod. In both figures, the CPU time does not vary much as the number of cores increases, showing that MILLIPEDE has a good scalability over number of cores. doZF is the most time-consuming function for both a single task and a frame. However, it has the lowest data communication overhead. doDemod is not expensive for executing a single task but has a
large total CPU time due to the large number of Demod jobs. More specifically, doDemod need to be executed for a large number of subcarriers and symbols within the frame. It also need to access a large amount of data and therefore has relatively high data communication overhead. The computation in doFFT is the cheapest among the three uplink functions. However, a large portion of its time is spent on data communication, which is because it needs to access data in all packets and symbols.

**Impact of number of antennas.** Next, we evaluate how the scale of MIMO system affects MILLIPEDE’s performance. We increase the number of base station antennas $M$ from 8 to 64, where 64 is a typical massive MIMO setting, and use 8 users and 30 cores for all the experiments. Note that MILLIPEDE currently does not support the traffic of 64 antennas with the sampling rate we use for fewer antennas, which is because the data throughput of 64 antennas is larger than the throughput of the 40 Gbps NIC. Hence, for the measurement with 64 antennas, we reduce the sampling rate by increasing symbol duration in the packet generator while keeping the other parameters the same.
Figure 5.8: Breakdown of CPU time spent in functions using 8 users and 30 cores. As the number of antennas increases, computation time increases due to a higher computation overhead; data pre/post processing time increases due to a larger data size. Each color represents a sub-stage and the sub-stages stacked vertically belong to the same function. (Stacked bars: left: doFFT, middle: doZF, right: doDemod.)

As shown in Figure 5.8(a), the total CPU time spent in all three functions grows with more antennas. The CPU time of all sub-stages within the doFFT function grows almost linearly over $M$, which is because the number of FFT jobs depends linearly on $M$ while a single task’s execution time remains almost the same, as shown in Figure 5.8(b). The reason for the growth of CPU time spent in doZF and doDemod functions is different from that of doFFT: the number of jobs remains the same over $M$ while each job’s data access time and computation time both grow as the matrix size increases.

We notice there is an abnormal growth of doZF function’s CPU time when $M = 16$. According to our profiling results in VTune, this is caused by a poor code vectorization when $M$ falls in the range of 15-20, which is likely to be a bad implementation of matrix inversion in Intel MKL.
5.4.3 Bottlenecks of External Library Calls.

We use VTune to further study the data communication overhead within external library calls, which consist of functions implemented in Armadillo and Intel MKL. More specifically, we look at three library calls: \texttt{pinv}, \texttt{fft} and \texttt{equalize}, corresponding to the computation of zero-forcing, FFT and equalization in \texttt{doZF}, \texttt{doFFT} and \texttt{doDemod}, respectively. \texttt{pinv} is a matrix pseudo-inverse function and \texttt{equalize} corresponds to a matrix multiplication function, both implemented in Armadillo and accelerated by Intel MKL. \texttt{fft} corresponds to an FFT function implemented in Intel MKL. Figure 5.9 shows the profiling results of \texttt{pinv}, \texttt{fft} and \texttt{equalize}. For all three library calls, data accesses (memory operations) cause CPU to stall for some time, which, however, is not the major bottleneck. The performance of \texttt{fft} and \texttt{equalize} are mainly core-bound, which is caused by insufficient compute resources or data/instruction dependencies. The performance of \texttt{pinv} is mainly front-end-bound, which means the CPU is mainly stalled in the front-end and might be caused by the large code size of \texttt{pinv}.  

![Figure 5.9: Performance bounds of library calls measured by VTune.](image)
Related Work

Software-based baseband processing. Existing software-based frameworks [10, 18] have demonstrated the possibility to achieve hardware-comparable baseband processing performance. They target at earlier Wi-Fi standards, e.g., 802.11a, which do not involve beamforming techniques and therefore only require a small number of CPU cores for data processing. BigStation [9] considers newer LTE and Wi-Fi standards that adopt beamforming and presents a distributed architecture for baseband processing of large MU-MIMO. However, its inefficient use of CPU cores and inefficient TCP communication among servers prevent it from further scaling up and meeting the stringent latency requirement of 5G. In contrast, MILLIPEDE relies on a single many-core server and employs an efficient threading model to dedicate CPU resources to the earliest frame, thus can meet a certain latency requirement with less cores than BigStation. As shown in Chapter 5, to achieve the same low latency for $8 \times 8$ MIMO, MILLIPEDE only needs 13 cores while BigStation needs 25 cores. Some open-source projects such as OpenAirInterface [26] and srsLTE [27] have been working on software realization of the physical layer. However, they do not support for massive MIMO yet. Intel’s FlexRAN [28] is a reference design for software-based base stations. Due to the lack of publicly available information, we are not able to compare it with MILLIPEDE. What is available publicly [29] provides highly optimized libraries for baseband processing, which, however, does not support massive MIMO. It does provide a high-performance LDPC implementation, which is a part has not been optimized in MILLIPEDE. Therefore, we expect MILLIPEDE to benefit from FlexRAN’s LDPC
implementation.

*Real-time Massive MIMO systems.* Existing massive MIMO base stations rely heavily on specialized hardwares to achieve real-time performance. LuMaMi [30] is an FPGA-based massive MIMO testbed that supports up to 100 antenna and 12 users. With intensive computation acceleration and fast Peripheral Component Interconnect Express (PCIe) based data communication, the delay between downlink transmission start and pilot receiving is compressed to be shorter than two symbols, i.e., 142.8 µs. Right now baseband processing in software can be 10× slower. However, there is a large space for further improvements with the programability and flexibility in software, which can reduce the performance gap between software and hardware. For example, historical CSI data can be exploited to predict the precoder for the current frame, which can largely relieve the burden of real-time computations. In this paper, we focus on evaluating the case where these further improvements have not been applied.

*Latency and throughput optimization.* Web servers and cloud application servers share similar performance goals of high throughput and low latency with MILLIPEDE, thus leading to a similar optimization space, e.g., choosing threading model, removing inefficiencies in a standard OS, and optimizing network stack. (i) The choice of threading model plays a critical role in optimizing latency and throughput for web and cloud application servers. Prior work [11, 12, 31] has extensively studied performance tradeoffs of threading models in different application scenarios. We make our choice of threading model based on these studies and adapt the model to the unique workload pattern and data dependences in 5G baseband. (ii) The authors in [32] summarize the inefficiencies from hardware, OS and application that add to tail latency. Since MILLIPEDE runs on a similar server environment, these inefficiencies also affect MILLIPEDE’s performance, e.g., interference from background processes can introduce context switch overheads. To eliminate them, MIL-
LIPEDE adopts common methods as those in prior work, detailed in Chapter 4. (iii) There have been extensive efforts on reducing latency contributed by the network stack, including kernel bypass (e.g., DPDK [33]) and OS optimization (such as IX [21], ZygOS [19], and Shinjuku [34]). MILLIPEDE is orthogonal to them since it mainly focuses on optimizing the part after packets are received in the user space.

*Computational kernel optimization.* We implement MILLIPEDE with computational kernels accelerated by Intel MKL [24], which is a widely adopted library for high-performance computing and provides significant speedup. Besides Intel MKL, there have been extensive studies on computational kernel optimization [24, 35–37]. Intel MKL may not be the one that has the best performance, e.g., SPIRAL [37] presents better performance than it. MILLIPEDE can benefit from these optimized kernels to achieve better computation performance.
Chapter 7

Conclusions

This work presents MILLIPEDE, a software-based framework for real-time baseband processing of massive MIMO on a single many-core server. Evaluation of MILLIPEDE on a 36-core machine shows that it achieves significantly lower latency and higher data rate than the state of the art and can scale up to use all available cores effectively. MILLIPEDE achieves this by maximizing the use of data parallelism for massive MIMO baseband while eschewing pipeline parallelism in the baseband processing. MILLIPEDE employs a carefully designed threading model to scale to many cores and a series of non-trivial cache-aware optimizations to cope with the memory bottleneck. Our experiments show that MILLIPEDE is able to support real-time baseband processing for $8 \times 32$ MIMO with a single 36-core server.
Chapter 8

Future Work

Implementing a complete end-to-end system. The current prototype of MILLIPEDE takes the first step to verify the feasibility of moving massive MU-MIMO baseband processing into software. Our next step is to make MILLIPEDE completely end-to-end, which requires MILLIPEDE to integrate with the ArgosV3 base station and have a complete physical layer pipeline. First, we plan to add a radio mode in MILLIPEDE to let it get data from the ArgosV3 base station instead of the packet generator. We will complete the physical layer pipeline by adding the LDPC encoder and decoder, which is a missing part in the current prototype. We plan to adopt the highly-optimized LDPC implementation from Intel FlexRAN [29].

Further accelerating computations. We plan to extend MILLIPEDE to support 512-bit SIMD extensions, i.e., AVX-512, which are the widest instructions supported by the state-of-the art Intel x86-based servers. We expect it to provide more instruction-level acceleration than the 256-bit AVX2 currently used in MILLIPEDE. Another direction we will explore is to replace the current 32-bit floating-point arithmetic with 16-bit fixed-point arithmetic. What makes 16-bit fixed-point arithmetic promising is that it allows us to use 2× of SIMD data parallelism as well as 0.5× of the memory bandwidth in 32-bit floating-point arithmetic, thus accelerating computations as well as reducing memory overhead. Furthermore, since the IQ samples from the ArgosV3 base station only have 12 bits, we do not expect MILLIPEDE to lose accuracy with 16-bit fixing-point arithmetic.

Improving threading model design. As shown in Figure 5.6(b), the synchronization
cost in the current design grows fast with more base station antennas, which will become a non-trivial overhead if we further increase the number of antennas to 64 or 96. To reduce this cost, we plan to improve MILLIPEDE 's threading model design in two ways. First, we will use static task scheduling to replace dynamic task scheduling so that there will be less contentions when the worker threads enqueue/dequeue jobs. Second, we will improve the job scheduling policy in the master thread to help hide part of the synchronization cost.

Separating design from implementation. Our vision is to enable a future where given an arbitrary configuration of massive MU-MIMO parameters (number of antennas, frame structure, etc.), MILLIPEDE can automatically explore the vast design space of architecture to find an optimized implementation for any combination of compiler, OS and implementation language. To achieve this, we will need a high-level language that allows the user to specify massive MU-MIMO parameters, and a compiler that can automatically find an optimized implementation taking care of pipeline scheduling, task parallelizing and all the code optimization we manually applied to MILLIPEDE right now. Existing studies, such as Ziria [18], Halide [36] and TVM [38], use domain specific languages (DSLs) to separate design from implementation. Some of them also provide compilers to automatically optimizing code, such as Ziria and TVM. Ziria is designed for earlier wireless standards and thus does not provide the extensive optimizations we applied in MILLIPEDE. TVM provides a compiler that can generate optimized code from a large search space for deep learning applications. MILLIPEDE can potentially draw inspirations from both Ziria and TVM to design a DSL and an optimizing compiler that fit the PHY pipeline of massive MU-MIMO.
Bibliography


