Intelligent Networks for High Performance Computing

William Whitney Schonbein
University of New Mexico - Main Campus

Follow this and additional works at: https://digitalrepository.unm.edu/cs_etds

Part of the Digital Communications and Networking Commons

Recommended Citation
https://digitalrepository.unm.edu/cs_etds/108

This Thesis is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Computer Science ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.
Whit Schonbein

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Trilce Estrada University of New Mexico

Chairperson

Dorian Arnold Emory University

Ryan E. Grant University of New Mexico

Jinho D. Choi Emory University
INTELLIGENT NETWORKS FOR HIGH PERFORMANCE COMPUTING

by

WHIT SCHONBEIN

B.A., University of Wisconsin, 1994,
M.A., Washington University in St. Louis, 1999,
Ph.D., Washington University in St. Louis, 2002,
M.S., University of New Mexico, 2016

DISSERTATION

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Computer Science
The University of New Mexico
Albuquerque, New Mexico

December, 2020
ACKNOWLEDGEMENTS

I would like to thank the members of my committee, without whom this work would not have been possible: Dorian Arnold, Ryan Grant, Trilce Estrada, and Jinho Choi. My gratitude also extends to my collaborators, Matthew Dosanjh, Scott Levy, and Pepper Marts. Thanks to Matthew Dosanjh for identifying target kernels and calculating potential application speedups reported in Section 7.1, and to David DeBonis and Ronnie Garduño for providing the profiling data and source code for the study reported in Section 7.2.
INTELLIGENT NETWORKS FOR HIGH PERFORMANCE COMPUTING

by

Whit Schonbein

B.A., University of Wisconsin, 1994,
M.A., Washington University in St. Louis, 1999,
Ph.D., Washington University in St. Louis, 2002,
M.S., University of New Mexico, 2016

Doctor of Philosophy, Computer Science

There exists a resurgence of interest in ‘smart’ network interfaces that can operate on data as it flows through a network. However, while smart capabilities have been expanding, what they can do for high-performance computing (HPC) is not well-understood. In this work, we advance our understanding of the capabilities and contributions of smart network interfaces to HPC. First, we show current offloaded message demultiplexing can mitigate (but not eliminate) overheads incurred by multithreaded communication. Second, we demonstrate current offloaded capabilities can be leveraged to provide Turing complete program execution on the interface. We elaborate with a framework for offloading arbitrary compute kernels to the NIC: In-Network Compute Assistance (INCA). We show INCA can accelerate host applications by offloading components to the network. Moreover, INCA supports the offloading of autonomous machine learning kernels for predicting network properties, and by doing so, takes a significant first step towards realizing intelligent, adaptive networks.
LONG ABSTRACT

The past half-decade has witnessed a resurgence of interest in ‘smart’ network interfaces (‘SmartNICs’), i.e., NICs that not only move data, but also perform potentially complex work with or on that data. What sets this new wave of research apart from earlier attempts at diversifying network functionality is its scope. Where once making the network intelligent meant offloading characteristically network-oriented applications – packet forwarding, firewalls, segmentation, bits and pieces of specific protocols, collective communications, and so on – researchers are currently broadening the vision of what a smart network can be expected to do. Perhaps, for example, the network itself can handle web searches local to the cell phones that issued them, or perform edge detection for guiding autonomous vehicles, or instantiate machine learning algorithms to dynamically adapt itself to changing workloads.

Networks characteristic of high performance computing (HPC) are no exception to the trend towards increasing intelligence. Current state-of-the-art HPC network interfaces are smart in the traditional sense that they offload network applications such as collective communication or message demultiplexing. However, emerging products and proposals aim to expand this domain by including more flexible offloading capabilities, e.g., on-NIC CPUs executing user-defined kernels for manipulating data as it passes between the network and the host.

While on-NIC capabilities have been expanding, what exactly they can do for HPC is not well-understood. For example, to what degree do current smart capabilities address anticipated future paradigms such as multithreaded communication? Can existing on-NIC capabilities provide additional flexibility for supporting novel offloaded functionality, or must we appeal to general-purpose compute hardware such as CPUs? Finally, when such flexibility is secured, what can be done with it to service HPC applications?

In this work, we advance our understanding of the capabilities and contributions of smart network interfaces to HPC by addressing these questions. First, we show that current offloaded capabilities – specifically, message demultiplexing – can help mitigate (but not entirely alleviate) overheads incurred by allowing individual threads to engage in inter-process communication. Second, we demonstrate that message matching, when coupled with other standard HPC offloaded capabilities, can enable fully generalized (i.e., Turing complete) program execution on the NIC. That is, current offloaded smart capabilities typical of HPC NICs can be leveraged to provide the flexibility characteristic of today’s SmartNICs, without appeal to hardware such as FPGAs or CPUs. On this foundation, we build a framework for expressing and offloading arbitrary compute kernels to the NIC: In-Network Compute Assistance (INCA). INCA is unique in the SmartNIC landscape because it leverages existing offloaded, task-specific processing elements to provide Turing complete compute capabilities. Moreover, despite being located on the data processing pathway, these capabilities are not subject to deadlines imposed by network speeds. We show that INCA affords the possibility of accelerating host application performance by offloading parts of those applications to the network. Furthermore, we show that INCA supports the offloading of autonomous machine learning kernels capable of predicting network features, and by doing so, takes a significant first step towards realizing intelligent, adaptive networks. In short, once fully general-purpose compute
capabilities are secured, the future of SmartNICs in HPC looks promising.

To sum, the contributions of this work are:

1. A comprehensive survey of SmartNIC hardware, applications, and general architectures that situates contemporary SmartNICs in the offloading tradition.

2. The design and implementation of two benchmarks for assessing the performance impact of multithreaded MPI communication under standard halo exchange patterns. We show that the overheads incurred by a full multithreaded halo exchange involving 9 or 27 nodes can be approximated by a ‘low-cost’ benchmark that emulates the full exchange using only two nodes. In either case, these overheads are onerous for larger thread counts. Moreover, we use the low-cost benchmark to assess whether contemporary offloaded message-matching capabilities can mitigate these overheads, concluding that, while such a smart capability can reduce processing times, in some conditions it actually exacerbates the overhead.

3. In-Network Compute Assistance (INCA), a novel SmartNIC design that leverages ‘smart’ offloaded capabilities common to HPC – message matching, atomic operations, and triggered operations – to provide arbitrary program execution to assist host applications or execute in-network applications. On the basis of a formal model and proof of Turing completeness, we identify a modest set of changes to current Portals-compliant NICs sufficient to secure Turing completeness.

4. The design of assembly and high-level languages for expressing INCA kernels, a compiler for transforming code written in the latter to the former, and an assessment – via a simulator interpreting INCA assembly code – of the performance of INCA executing a selection of HPC-centric kernels under various operating assumptions. We conclude that with the right selection of software and hardware optimizations, INCA kernel runtimes can, for some kernels, be made comparable to those of contemporary CPUs. Moreover, all else being equal, INCA runtimes will only decrease as network speeds increase.

5. An assessment of the potential speedups afforded by INCA for a selection of scientific applications or miniapps executing on the host. We show that, by offloading parts of host applications, INCA can in principle accelerate some applications by 35% or more. We also consider a second class of applications, those that utilize the network only infrequently during their runtime, meaning the network is usually idle. In a preliminary study, we profile one such application to locate functions that consume a significant portion of the overall runtime, and consider speedups afforded by offloading a subset of those calls to an INCA-enabled SmartNIC. Results from a simulation study show speedups of up to $1.13 \times$ and $1.23 \times$, without and with additional hardware acceleration, respectively.

6. The first known design and implementation of fully-offloaded machine learning kernels for enabling ‘self-learning’ networks by efficiently and accurately predicting local network traffic rates. Specifically, we profile the amount of data received by
the NIC during the execution of a selection of scientific applications, proxy applications, and miniapps, and show that machine learning kernels based on static or rolling linear regression can predict the amount of incoming data with an accuracy ranging from less than 2.5% normalized RMSE in the worst case, to less than 0.1% in the best-performing cases. Moreover, INCA kernels for generating predictions regarding future network traffic execute with runtimes less than 40 µs in the most computationally demanding scenarios, and in many cases, less than 1 µs, with memory requirements never exceeding 12 KiB.
# TABLE OF CONTENTS

List of Figures ........................................................................................................ xi

List of Tables .......................................................................................................... xii

1 Introduction ........................................................................................................ 1

2 Directions in networking .................................................................................... 7
   2.1 Hardware ...................................................................................................... 8
      2.1.1 Application-Specific Integrated Circuits ......................................... 9
      2.1.2 CPUs .................................................................................................. 10
      2.1.3 FPGAs .............................................................................................. 12
   2.2 Applications ................................................................................................. 13
      2.2.1 Core network applications ................................................................ 13
      2.2.2 Host applications .............................................................................. 18
      2.2.3 Independent applications ................................................................ 20
   2.3 Architectures ................................................................................................ 21
      2.3.1 On-path ............................................................................................. 21
      2.3.2 Off-path ............................................................................................ 23
      2.3.3 In-path .............................................................................................. 24
   2.4 Discussion: The path forward ................................................................. 25

3 Message matching ............................................................................................ 28
   3.1 Background ................................................................................................. 30
   3.2 Related work .............................................................................................. 33
   3.3 Analysis ...................................................................................................... 34
   3.4 Benchmarks and methodology .................................................................. 37
   3.5 Results ........................................................................................................ 41
      3.5.1 Resource usage .................................................................................. 41
      3.5.2 Items searched ................................................................................... 42
      3.5.3 Matching overheads ........................................................................... 45
      3.5.4 Discussion ......................................................................................... 48
   3.6 Case study: assessing hardware offloaded message matching ................. 49
      3.6.1 Discussion ......................................................................................... 52
   3.7 Conclusion ................................................................................................. 53

4 INCA: In-Network Compute Assistance ...................................................... 55
4.1 The big picture ................................................................. 57
4.2 The triggered operation machine ........................................ 59
4.3 The Turing completeness of TOM ........................................ 61
4.4 Implementing INCA .......................................................... 64
4.5 Conclusion........................................................................ 66

5 The INCA Ecosystem ............................................................... 68
5.1 The INCA ecosystem ........................................................ 68
5.2 INCA-Q ........................................................................ 69
5.3 INCA-A ........................................................................ 70
5.4 Compiler and interpreter .................................................... 72
5.5 Conclusion........................................................................ 73

6 INCA: Kernel Performance .................................................... 75
6.1 INCASim ......................................................................... 76
6.2 Kernels and INCASim configurations .................................... 78
   6.2.1 Kernels ................................................................... 78
   6.2.2 INCASim configurations ............................................. 79
6.3 Performance Evaluation ...................................................... 82
   6.3.1 Base and scratchpad performance .................................. 84
   6.3.2 Network speeds ....................................................... 85
   6.3.3 Reducing data movement ............................................ 85
   6.3.4 Hardware optimizations ............................................. 86
6.4 Conclusion........................................................................ 90

7 INCA: Host Applications ....................................................... 92
7.1 Offloading near-network functions ...................................... 93
7.2 NIC as co-processor .......................................................... 96
7.3 Conclusion........................................................................ 101

8 Independent Applications for Adaptive Networks .................. 102
8.1 Constraints on ML kernels ................................................ 104
8.2 Applications .................................................................... 105
8.3 Data collection .................................................................. 105
8.4 ML kernels ..................................................................... 106
   8.4.1 Ordinary LR ......................................................... 109
   8.4.2 Rolling LR ........................................................... 110
8.5 Results: Ordinary linear regression .................................... 112
   8.5.1 Method performance ............................................... 112
   8.5.2 INCA kernel resources ............................................ 114
8.6 Results: Rolling linear regression ....................................... 115
   8.6.1 Method performance ............................................... 119
   8.6.2 INCA kernel resources ............................................ 122
8.7 Related work ................................................................... 123
8.8 Conclusion........................................................................ 124
9 Conclusion and future work ..........................................................125

Appendices ..................................................................................130
  Appendix A. The INCA-A language .............................................. 130
  Appendix B. The INCA-Q Language ............................................. 136

References ..................................................................................141
LIST OF FIGURES

2.1 Diagram of an on-path NIC architecture. ........................................... 22
2.2 Diagram of an off-path NIC architecture. ........................................... 23
2.3 Diagram of an in-path NIC architecture. ........................................... 24

3.1 MPI message matching queues............................................................. 31
3.2 Singlethreaded and multithreaded nine-point halo exchanges. ................. 35
3.3 Average core hours for real-world and low-cost benchmarks. .................. 41
3.4 Median total number of items searched. ............................................ 42
3.5 Average number of searches by number of items searched. .................... 44
3.6 Median queue drain times................................................................. 45
3.7 Average time per search................................................................. 47
3.8 Comparison of message processing time with and without hardware accel-
eration. ......................................................................................... 50
3.9 Median time spent processing messages by message size....................... 51

4.1 Portals processing pipeline (data structure view) .................................. 57
4.2 An INCA instruction.......................................................................... 58
4.3 Message processing flow for the triggered operation machine. ............... 59

5.1 The INCA ecosystem. ......................................................................... 69

6.1 The extended LogGP model. ............................................................... 77
6.2 The baseline INCA architecture. ......................................................... 80
6.3 The scratchpad INCA architecture. ..................................................... 81
6.4 INCA instruction rates. ...................................................................... 81
6.5 INCA SIMD unit............................................................................... 89

8.1 Examples of raw bytes received data.................................................... 107
8.2 Illustration of rolling linear regression.................................................. 110
8.3 Ordinary linear regression results. ....................................................... 113
8.4 The impact of startup plateaux............................................................. 114
8.5 Error as a function of window size for rolling LR. ................................. 116
8.6 Error for rolling LR methods.............................................................. 120
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of survey of SmartNIC applications.</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>Summary of survey of SmartNIC architectures.</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Message counts for multithreaded halo exchange.</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Triggered operation machine notation.</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Summary of the reduction of URM$s to TOM$</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>INCA kernels and instruction counts.</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>Table of INC$asim parameters.</td>
<td>82</td>
</tr>
<tr>
<td>6.3</td>
<td>INCA kernel runtimes.</td>
<td>83</td>
</tr>
<tr>
<td>6.4</td>
<td>Impact of network speed on kernel runtimes.</td>
<td>85</td>
</tr>
<tr>
<td>7.1</td>
<td>Potential impact of INCA on applications.</td>
<td>96</td>
</tr>
<tr>
<td>7.2</td>
<td>FPGA exponential offloaded latencies.</td>
<td>98</td>
</tr>
<tr>
<td>7.3</td>
<td>Potential speedups for applications with idle networks.</td>
<td>100</td>
</tr>
<tr>
<td>8.1</td>
<td>Mean probability of data arrival.</td>
<td>108</td>
</tr>
<tr>
<td>8.2</td>
<td>Best-performing window sizes for each prediction point.</td>
<td>118</td>
</tr>
<tr>
<td>8.3</td>
<td>Selected small and large windows sizes.</td>
<td>119</td>
</tr>
<tr>
<td>8.4</td>
<td>INCA runtimes for rolling LR methods.</td>
<td>122</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The past half-decade has witnessed a resurgence of interest in ‘smart’ network interfaces (‘SmartNICs’), i.e., NICs that not only move data, but also perform potentially complex work with or on that data. What sets this new wave of research apart from earlier attempts at diversifying network functionality is its scope. Where once making the network intelligent meant offloading characteristically network-oriented applications – packet forwarding, firewalls, segmentation, bits and pieces of specific protocols, collective communications, and so on – researchers are currently broadening the vision of what a ‘smart’ network can be expected to do. Perhaps, for example, the network itself can handle web searches local to the cell phones that issued them, or perform edge detection for guiding autonomous vehicles, or instantiate machine learning algorithms to dynamically adapt itself to changing workloads.

Networks characteristic of high performance computing (HPC) are no exception to the trend towards increasing intelligence. Current state-of-the-art HPC network interfaces are ‘smart’ in the traditional sense that they offload network applications such as collective communication or message demultiplexing. However, emerging products and proposals aim to expand this domain by including more flexible offloading capabilities, e.g., on-NIC CPUs executing user-defined kernels for manipulating data as it passes between the network and the host.
While on-NIC capabilities have been expanding, what exactly they can do for HPC is not well-understood. For example, to what degree do current ‘smart’ capabilities address anticipated future paradigms such as multithreaded communication? Can existing on-NIC capabilities provide additional flexibility for supporting novel offloaded functionality, or must we appeal to general-purpose compute hardware such as CPUs? Finally, when such flexibility is secured, what can be done with it to service HPC applications?

In this work, we advance our understanding of the capabilities and contributions of smart network interfaces to HPC by addressing these questions. First, we show that current offloaded capabilities – specifically, message demultiplexing – can help mitigate (but not entirely alleviate) overheads incurred by allowing individual threads to engage in inter-process communication. Second, we demonstrate that message matching, when coupled with other standard HPC offloaded capabilities, can enable fully generalized (i.e., Turing complete) program execution on the NIC. That is, current offloaded ‘smart’ capabilities typical of HPC NICs can be leveraged to provide the flexibility characteristic of today’s SmartNICs, without appeal to hardware such as FPGAs or CPUs. On this foundation, we build a framework for expressing and offloading arbitrary compute kernels to the NIC: In-Network Compute Assistance (INCA). INCA is unique in the SmartNIC landscape because it leverages existing offloaded, task-specific processing elements to provide Turing complete compute capabilities. Moreover, despite being located on the data processing pathway, these capabilities are not subject to deadlines imposed by network speeds. Finally, we show that INCA affords the possibility of accelerating host application performance by offloading parts of those applications to the network. Furthermore, we show that INCA supports the offloading of autonomous machine learning kernels capable of predicting network features, and by doing so, take a significant first step towards realizing intelligent, adaptive networks. In short, once fully general-purpose compute capabilities are secured, the future of SmartNICs in HPC looks promising.

We begin, in Chapter 2, by surveying the field of intelligent network interfaces, fo-
Focusing on three distinct aspects: the hardware used to underwrite NIC intelligence, the types of applications offloaded to that hardware, and the general architectures these solutions adopt. This survey underwrites the observation made above, that the concept of a SmartNIC has evolved from one involving NICs offloading task-specific, core network applications (e.g., packet forwarding, filtering, etc.) to one implicating flexible, general-purpose kernel execution. Moreover, this survey situates our proposed approach to enabling fully general on-NIC offloading – In-Network Compute Assistance – within the hardware, application, and architectural landscapes of current and past SmartNICs.

Trends towards larger core counts and faster but lower-capacity memory (e.g., high-bandwidth memory [1]) discourage traditional ‘MPI-everywhere’ approaches, where each Message Passing Interface (MPI) process instantiates its own set of communication resources. Instead, under a multithreaded strategy, a single set of resources is used, and multiple threads are permitted to engage in communication with other MPI processes. In Chapter 3, we design and implement benchmarks for assessing the message processing overheads incurred by multithreaded communication in a standard halo exchange. Moreover, we deploy one of these benchmarks to address whether contemporary message matching offloading hardware can help mitigate these overheads, thereby addressing the broader question of what SmartNICs can (and cannot) do.

In Chapter 4 we describe how In-Network Compute Assistance – INCA – leverages offloaded message demultiplexing and other existing task-specific network applications to provide on-NIC general-purpose compute capabilities. To this end, we develop a formal computational model – the triggered operation machine – and demonstrate it is Turing complete. On the basis of this model, we articulate a set of modest adjustments to existing NIC hardware sufficient to enable INCA-style offloading.

Chapter 5 builds on the prior theoretical results to develop an ecosystem for expressing INCA kernels to be executed on NIC hardware. Based on the formal model, we define a low-level ‘assembly’ language – INCA-A – for defining INCA kernels, and implement
an interpreter for this language. To assist in developing INCA kernels, we also define a high-level language – INCA-Q – and a compiler for translating kernels written in INCA-Q into INCA-A.

The remaining chapters are dedicated to putting INCA to work. In Chapter 6, we assess the performance of INCA-enabled SmartNICs by developing a simulation model, identifying a set of representative kernels (matrix multiplication, convolution, etc.), and interpreting the execution of these kernels within the simulation. In the process, we consider various software and hardware optimizations, and their impact on INCA kernel runtimes. In Chapter 7, we identify two possible scenarios where INCA could enable host application speedups by offloading parts of those host applications, and assess those speedups. Finally, in Chapter 8, we show that INCA enables adaptive, intelligent networks by offloading machine learning techniques for predicting network traffic entirely to the NIC.

To sum, the contributions of this work are:

1. A comprehensive survey of SmartNIC hardware, applications, and general architectures that situates contemporary SmartNICs in the offloading tradition, and underscores the novelty of In-Network Compute Assistance. Whereas contemporary SmartNICs invariably adopt one of two standard designs – on-path, where processing elements are placed on the packet processing pathway, and off-path, where they are located off that pathway – INCA represents a synthesis of the two by leveraging on-path processing elements to provided compute capabilities that are nonetheless not constrained by network speed.

2. The design and implementation of two benchmarks for assessing the performance impact of multithreaded MPI communication under standard halo exchange patterns. We show that the overheads incurred by a full multithreaded halo exchange involving 9 or 27 nodes can be approximated by a ‘low-cost’ benchmark that emulates the full exchange using only two nodes. In either case, these overheads are
onerous for larger thread counts; e.g., approaching or exceeding 1 ms. Moreover, we use the low-cost benchmark to assess whether contemporary offloaded message matching capabilities (as provided by NVIDIA Mellanox ConnectX-5 NICs [2]) can mitigate these overheads, concluding that, while such a smart capability can reduce processing times, in some conditions it actually exacerbates the overhead.

3. In-Network Compute Assistance (INCA), a novel SmartNIC design that leverages ‘smart’ offloaded capabilities common to HPC – message matching, atomic operations, and triggered operations – to provide arbitrary program execution to assist host applications or execute in-network applications. On the basis of a formal model and proof of Turing completeness, we identify a set of modest changes to current HPC NICs based on the Portals network programming API [3] sufficient to secure general-purpose compute capabilities.

4. The design of assembly and high-level languages for expressing INCA kernels, a compiler for transforming code written in the latter to the former, and an assessment – via a simulator interpreting INCA assembly code – of the performance of INCA executing a selection of HPC-centric kernels under various operating assumptions. We conclude that with the right selection of software and hardware optimizations, INCA kernel runtimes can, for some kernels, be made comparable to those of contemporary CPUs. Moreover, all else being equal, INCA runtimes will only decrease as network speeds increase [4].

5. An assessment of the potential speedups afforded by INCA for a selection of scientific applications or miniapps executing on the host. We show that, by offloading parts of host applications, INCA can in principle accelerate some applications by 35% or more. We also consider a second class of applications, those that utilize the network only infrequently during their runtime, meaning the network is usually idle. In a preliminary study, we profile one such application to locate functions
that consume a significant portion of the overall runtime, and consider speedups afforded by offloading a subset of those calls to an INCA-enabled SmartNIC. Results from a simulation study show speedups of up to 1.13× and 1.23×, without and with additional hardware acceleration, respectively.

6. The first known design and implementation of fully-offloaded machine learning kernels for enabling ‘self-learning’ networks by efficiently and accurately predicting local network traffic rates. Specifically, we profile the amount of data received by the NIC during the execution of a selection of scientific applications, proxy applications, and miniapps, and show that machine learning kernels based on static or rolling linear regression can predict the amount of incoming data with an accuracy ranging from less than 2.5% normalized RMSE in the worst case, to less than 0.1% in the best-performing cases. Moreover, INCA kernels for generating predictions regarding future network traffic execute with runtimes less than 40 µs in the most computationally demanding scenarios, and in many cases, less than 1 µs, with memory requirements never exceeding 12 KiB.
Chapter 2

Directions in networking

The primary purpose of a network interface card (NIC) is to push and receive data to and from the network. For an ‘ignorant’ NIC – i.e., one that services only these functions – anything more complicated (such as protocol enforcement, packet processing, filtering, demultiplexing, etc.) is the responsibility of the host CPU. The past quarter-decade has witnessed a series of increasingly complex network interfaces with enhanced processing capabilities designed to relieve host CPUs of some of these tasks. Originally, this offloading was modest, including fixed-field packet forwarding, packet filtering, or simple header manipulations. More recently, this trend in so-called ‘SmartNICs’ (also sometimes referred to as ‘intelligent NICs’ (iNICs) or ‘data processing units’ (DPUs)) has culminated in NICs with complex processing engines, including fully-programmable CPUs, FPGAs, or even specialized ‘engines’ dedicated to tasks such as neural network processing.

The term ‘SmartNIC’ can be traced at least as far back as the mid-1990s, where it was used to refer to NICs offering basic offloaded functionality such as TCP (de)segmentation, header matching for purposes of flexible forwarding, error checking, or simple packet modification [5], [6]. So, as a starting point, let us define a SmartNIC (/iNIC/DPU) as a network interface or host adapter that offloads functionality traditionally performed by a host CPU.
This is obviously a broad definition, and during the course of this survey of Smart-NICs, we will identify a narrower conception that better captures the contemporary (ca. 2020) understanding of the term. To accomplish this, we view the concept of a SmartNIC from three different perspectives: (1) hardware (Section 2.1), (2) the tasks, functions, or services offloaded (Section 2.2), and (3) general architecture (Section 2.3). The results of this survey motivate and situate the present project both historically and within the contemporary SmartNIC landscape.

This survey comes with several caveats. First, we do not attempt to catalog every instance of a network appliance that (self-identified or otherwise) qualifies as a SmartNIC (cf. [7]). Instead, the present goal is to highlight salient general features of the SmartNIC landscape. Second, SmartNICs are one aspect of a broader collection of research into ‘smart’ network appliances in general, such as programmable routers, switches, or middleboxes [8]–[13] and active networks [14]–[21]. Building on these technologies, researchers have proposed offloading tasks such as consensus algorithms [22], [23], collective communication/data aggregation [24], [25], caches or distributed key-value stores [26]–[28], policy [29], intrusion detection [30], and network monitoring [31]. With occasional exceptions, this survey is limited to network interfaces or host adapters, i.e., to discrete or integrated network appliances residing on network endpoints. Finally, because the focus of this work is on high-performance computing (HPC), we emphasize research in that domain.

### 2.1 Hardware

The first dimension along which SmartNICs vary is the hardware they use to enable offloading. For purposes of this survey, we distinguish between three broad categories: application-specific integrated circuits (ASICs), central processing units (CPUs), and field-programmable gate arrays (FPGAs).
2.1.1 Application-Specific Integrated Circuits

ASICs are specialized circuits designed to carry out specific, ‘hardwired’ tasks, e.g., executing CRC checks on packet headers of a predefined size, incrementing counters, or performing packet reassembly. Despite being task-specific, ASICs may nonetheless be ‘configurable’ in the sense that within that limited domain of functionality, various parameters may be adjusted. An example of a configurable ASIC that is extremely common in high-speed networks is a content addressable memory (CAM) or ternary content addressable memory (TCAM). CAMs and TCAMs allow for efficient table lookups, e.g., for Layer 2 and Layer 3 forwarding, applying access control lists (ACLs), packet classification, or message demultiplexing in HPC systems [32]–[34]. These units qualify as ASICs because they service, in hardware, one and only one task: table lookup. They are ‘configurable’ because entries can be added, removed, or modified.

Examples of contemporary NICs that may be ASIC-based are those designed to implement the Portals network programming API [3], [35]. The Portals specification identifies several primitive capabilities – message matching, basic arithmetic and logical operations, etc. – that support offloading network applications. Because the context is HPC and the high-speed networks supporting HPC applications, it is reasonable to expect these capabilities will be implemented in hardware (cf. Chapter 4). Netronome Agilo SmartNICs [36], [37] are sometimes identified as ASIC-based, but for reasons discussed below, we categorize them as involving CPUs.

In contrast to other types of hardware, ASICs have the advantage of being fast (e.g., a CAM lookup takes 5 or fewer nanoseconds [38], [39]) and hence are good candidates for latency-sensitive tasks requiring processing at line rate. However, because ASICs must be designed from the ground up to fulfill novel tasks, they can incur significant development costs and their time-to-market may be slow. Moreover, despite being configurable, ASICs are also relatively inflexible, e.g., in comparison to standard CPUs. This
means they cannot easily be adapted to changes in protocols (perhaps impacting their

time-in-market), and are poor candidates for prototyping.

Motivated by the desire to retain the benefits of ASICs (speed, energy-efficiency),
while enabling additional flexibility, application-specific instruction-set processors (ASIPs)
are defined as ASICs extended to be configurable through software, i.e., ‘programmable’ [40].

By being programmable, the promise is that the same hardware can be re-configured to
serve different roles within the same narrow task domain. While ASIPs may represent a
further degree of configurability beyond that illustrated by content-addressable memory,
because they are still designed to service a specific application, for purposes of this survey
we place ASIPs in the same category as (configurable) ASICs, reserving the notion of
‘programmable’ for hardware that can execute arbitrary programs.

2.1.2 CPUs

Any given SmartNIC is likely to include a general-purpose CPU, if only to service con-
trol functions. However, some smartNICs utilize CPUs as the primary hardware for
supporting offloaded applications. Often, the instruction sets and capabilities of these
CPUs may be extended or tailored to provide network-specific instructions such as CRC
checks, matching bit fields, traversing data structures, or queue management [41]. Insofar
as these ‘network processors’ (NPs, or ‘network processing units’ (NPUs)) have special-
ized, application-specific aspects, they are comparable to ASICs. However, because they
are based on general-purpose compute hardware configurable through software, these
NICs can provide flexibility lacking in ASIC-based approaches.

In HPC, influential earlier-generation examples of NICs with programmable pro-
cessors include those used by Meiko CS-2 [42], Myrinet [43], Quadrics [44], [45], and
SeaStar [46], [47] networks. Meikos and Quadrics NICs included versions of the Elan
network interface, which (in the Quadrics incarnation) provided a 32bit RISC processor
designed to foster the offloading of arbitrary higher-level messaging libraries. Similarly,
Myrinet NICs provided a 133MHz processor that could be used to offload various network functions [48]. Finally, according to [47], SeaStar featured a 500MHz processor programmable through firmware; while the primary purpose of this processor was to support the offloading of functions identified in the Portals API, its inherent flexibility offered the possibility of offloading other sorts of network services [49]. In each of these cases, the flexibility of CPU-based offloading was a motivating design consideration.

More recent examples of SmartNIC designs featuring general-purpose CPUs include NVIDIA’s Mellanox Bluefield and Bluefield-2 SmartNICs [50], [51], Broadcom’s Stingray PS225 SmartNIC [52], and Marvell’s LiquidIO II series (using the cnMIPS chip which is heavily specialized for network processing) [53]. A more recent example of an HPC-oriented SmartNIC design featuring general-purpose CPUs is sPIN (Streaming Processing in-Network) [54]. On a sPIN NIC, incoming packets are initially processed by a header handler, which then dispatches each packet to a different CPU core executing user-defined packet handlers. When the work is completed, the (potentially modified) packets are pushed to host memory using DMA. Again, the emphasis is on flexibility: packet handlers are written in high-level languages (e.g., C), and compiled to execute on the NIC.

A final example are Netronome’s Agilio SmartNICs [36], [37]. These NICs embody the ‘match-plus-action’ processing model popularized by software-defined networking [8], [9]: incoming packets are classified by configurable packet processors and dispatched to programmable cores (e.g., 60 cores with 8 threads/core) for further processing. These cores are domain specific, targeted towards network processing and security (consequently, they are sometimes referred to as ‘ASIC-based’ [55]) However, they are also programmable using standard software tools (e.g., C, P4), and the vendor notes these specialized cores could in principle be replaced with general-purpose RISC cores [56], [57]. For these reasons, we include them as examples of CPU-based SmartNICs.

As already emphasized, the primary advantage of deploying CPUs for NIC offloading
is flexibility. In contrast to ASICs (configurable or otherwise), this flexibility means CPU-based SmartNICs can be readily adapted to changing network offloading demands, and used for prototyping. However, CPUs also tend to be slower than ASICs on similar tasks (e.g., table lookups [58]).

2.1.3 FPGAs

Field programmable gate arrays (FPGAs), when deployed on NICs, aim towards providing the flexibility of CPUs with the speed of ASICs. An FPGA comprises a set of millions of logic elements (LEs), each of which has a lookup table and several registers. Each LE can be configured to compute arbitrary combinatorial logic. The FPGA also has onboard block RAM (megabytes), as well as a variety of digital signal processors for performing more complex mathematical operations (numbering in the thousands). FPGAs are typically a magnitude of order slower than CPUs, but they make up for this by trading the temporal organization of the CPU for a ‘spatial’ or massively parallel organization made possible by the LEs and DSPs. FPGAs are thus a popular choice for the design and implementation of hardware for network offloading.

Underwood et al. [59], [60] propose an intelligent NIC equipped with an FPGA for performing operations on data as it moves through the NIC. Other examples of FPGA-enabled SmartNICs include the NetFPGA platform for education and research [61], the various accelerator cards offered by Xilinx [62], [63], NVIDIA’s Mellanox Innova-2 Flex card [64], and the NICs used to implement Microsoft’s Virtual Filtering Platform [65], [66]. Also within Microsoft, the Catapult network appliances are FPGAs placed on the data path between local switch and NIC [67].
2.2 Applications

In addition to hardware, SmartNICs can be distinguished according to the types of tasks or applications that have been offloaded (or proposed for offloading). These can be grouped into three categories: core network applications, host applications, and independent applications. As with our discussion of SmartNIC hardware, we do not claim these categories are mutually exclusive, i.e., there may exist edge cases situated on the boundaries.

2.2.1 Core network applications

For purposes of this survey, a core network application is broadly defined as one that is involved in the ingress or egress of data to and from a host application, i.e., in performing a task that addresses the traditional notion of what a NIC is supposed to do, namely, deliver data. This class of applications includes tasks at all layers of the OSI model, and extends to communication middleware such as the Message Passing Interface (MPI) and OpenSHMEM [68], [69]. However, the majority of offloaded core network applications target the lower layers, e.g., L2 and L3.

Core network functions include CRC checks, atomic operations (e.g., decrementing TTL, bit shifts, addition), segmentation/desegmentation, traffic shaping and packet scheduling (QoS), forms of congestion control (e.g., rate adjustment), receive-side scaling or packet steering, aspects of firewall implementation (e.g., dropping packets), offloading collective communications such as multicast or reduction, encryption/decryption, message demultiplexing (e.g., MPI message matching), TCP offloading, and other forms of protocol processing. Since the majority of these functions involve forms of packet classification – i.e., matching on header or payload bit fields for purposes of differentially processing those packets – we include in the category of core network functions tasks that naturally cohere with this processing strategy, e.g., applying access control lists or
performing forms of intrusion detection. However, we also acknowledge that some of these applications may span multiple categories.

Core network functions comprise the bulk of applications offloaded to SmartNICs. In the remainder of this subsection we highlight several classes of offloaded network applications.

2.2.1.1 Basic packet processing

Basic packet processing applications involve tasks such as forwarding, filtering, receive-side scaling (RSS) [70], simple header manipulation (e.g., decrementing TTL), or segmentation. The capacity to handle applications such as these is the foundation of the notion of a SmartNIC. For example, in an early use of the term ‘smart network interface’, Connery et al. [5] patent a method for performing on-NIC TCP segmentation. The bar for qualifying as ‘smart’ is not high.

Most if not all contemporary NICs include hardware support for some form of basic packet processing. Recent work in this vein includes intelligent forwarding of packets to available cores [71], and support for offloading network virtualization [65], [72].

2.2.1.2 TCP offloading

As the name implies, TCP offload engines (TOEs) are designed to offload TCP functionality. The primary motivation behind TOEs is to free up CPU resources and hence accelerate host applications by avoiding interrupts or the need to calculate checksums, perform out-of-order packet assembly, or copy data. Despite being an occasionally contentious subject ([73]–[75]), work on TOEs has continued over multiple decades. For example, Connery et al. [5] described a method for offloading TCP (de)segmentation, Ang [76] explored the potential benefits of offloading TCP congestion control (i.e., windows) as well as (de)segmentation. Hoskote et al. [77] proposed a chip design for offloading the ingress side of a TOE, and Freimuth et al. [78] presented a system that offloads the entire
TCP/IP stack. Feng et al. [79] performed a performance analysis of a then-current TOE that utilizes a combination of TCAM and CPU configurable through firmware, showing that the TOE provides higher bandwidth and lower latency than the kernel implementation of the stack. The iWARP protocol builds on top of TOEs to bring zero-copy, kernel-bypass, RDMA to TCP/IP, and Rashti et al. and Grant et al. [80], [81] show how this can be extended to operate using UDP, and hence avoid the overhead of maintaining state. Jang et al. [82] propose a hybrid TOE architecture that provides ASIC-based Tx and Rx engines coupled with programmable processors to provide flexibility to adapt to changes in the protocol. More recently, Moon et al. [83] explore the benefits of offloading not the entire TCP/IP stack, but rather ‘peripheral’ TCP functions such as connection setup and teardown (in addition to the usual checksums and (de)segmentation), showing that doing so makes performance on short-lived connections comparable to that of longer-lived connections.

2.2.1.3 Security

Another class of core network applications that has received significant attention for SmartNIC offloading are distributed security applications such as cryptography, IPsec, firewalls, and intrusion detection systems (IDS). While some of these applications may involve nothing more than header matching (e.g., basic firewalls), others have more intensive memory or processing requirements, e.g., matching strings in payloads may require TCP stream reconstruction and more complex searches. For example, Clark et al. [84] use a network processor ([85]) for initial header-based filtering, TCP dsegmentation, and implementing a decision procedure for determining whether the incoming data should be subject to further analysis, and an FPGA for implementing the more computationally-demanding process of searching payloads for suspect patterns. Bos et al. [86] investigate a signature detection system that executes without assistance of an FPGA, and Bruijn et al. [87] demonstrate an offloaded solution that provides deep packet inspection (e.g., for
detecting polymorphic buffer overflows) at gigabit rates. Friedman and Nagle [88] present a firewall implementation that runs on the local SmartNIC. Burnside and Keromytis [89], [90] advocate for offloaded cryptography services, focusing on discrete cards but noting the functionality could be provided by the NIC itself. Chaignon et al. [91] address the problem of supporting distinct security domains on a single server (e.g., for different virtual machine instances), proposing a means for offloading nearly arbitrary packet filters to on-NIC CPUs while maintaining fairness under the standard run-to-completion model. Dimolianis et al. [92] implement a DDoS attack detection method using match-plus-action programs on a contemporary Netronome Agilio SmartNIC; the system is capable of handling millions of messages per second on 10 Gb/s links.

2.2.1.4 Collective and rendezvous communication

Collective communications (i.e., ‘collectives’) include broadcast, multicast, reduction, barrier, and other paradigms involving the dissemination or retrieval of data to or from multiple distributed processes. Rendezvous communication occurs when a sender cannot assume a receiver has available buffer space to handle the payload, so must initiate a ‘request-to-send’ to the receiver, and wait for a ‘clear-to-send’ response, at which point the payload proper can be sent. Both forms of communication present opportunities for overlapping host computation with communication by offloading the latter onto the NIC. For instance, in a tree-structured broadcast, a process in the middle of the tree is responsible for passing incoming data to its children; if this process can be autonomously handled by the NIC, CPU resources are made available to continue servicing the host application. Likewise, computation can be overlapped with rendezvous communication by offloading the process of confirming available buffer space.

With its embedded CPU, the Myrinet NIC [43] prompted significant research on the offloading of collective and rendezvous communication. For instance, Verstoep et al. [93], Bhoedjang et al. [94], Buntinas et al. [95] each explore strategies for offloading
multicast. Keppitiyagama and Wagner [96] and Tourancheau and Westrelin [97] consider how on-NIC CPUs can facilitate asynchronous MPI progress, and hence enable computation/communication overlap in multicast or rendezvous communication. Similarly, Buntinas et al. [98], [99] research performance benefits for offloading barriers onto Myrinet processors.

Outside of the Myrinet sphere of influence, Graham et al. [100] describe how Mellanox ConnectX-2 hardware facilitates the offloading of barriers (cf. [24]). NICs based on the Portals network programming interface [3] include ‘triggered operations’, e.g., the capacity to issue sends or perform other actions when a local buffer has been filled, independently of the host processor (cf. Chapter 4). Using these building blocks, Hemmert et al. [101] implement offloaded barrier and broadcast collectives, Underwood et al. [102] do the same for allreduce, and Barrett et al. [103] provide an initial attempt at handling rendezvous communication.

2.2.1.5 Message matching

The final type of offloaded core network applications we identify is that of message matching. Broadly construed, this category could include any offloaded packet forwarding mechanism based on matching bit fields, but here we target matching as defined by high-level communication middleware, primarily the Message Passing Interface (MPI) [69]. Under MPI, incoming messages are demultiplexed, and payloads placed in the correct buffers, according to a process identifier (the ‘rank’) and a user-specified tag. Since MPI is by far the most common user-level network interface used by developers of scientific applications, MPI message matching has been the subject of considerable research into on-NIC offloading.

Tourancheau and Westrelin [97] show that offloading matching to an on-NIC CPU can allow MPI to progress independently of the host application calling into the library, enabling offloaded rendezvous communication. Underwood et al. [32] propose a TCAM-
based architecture for offloading message matching, and Tanabe et al. [104] explore an offloading strategy based on separating headers and payloads. Klenk et al. [105] consider how GPUs may accelerate matching if current MPI semantics can be relaxed, and current Atos Bull and NVIDIA Mellanox ConnectX-5 NICs provide offloaded MPI message matching functionality [2], [35].

2.2.2 Host applications

In the previous subsection we considered the offloading of core network applications, which include functions ranging from low-level packet processing to communication middleware. A second class of offloaded applications has received less attention: host applications. The characteristic feature of this type of offloaded application is that rather than targeting network functionality, it encapsulates some part of the host application.

For purposes of illustration, a simple example of an offloaded host application is one that performs specialized data packing and unpacking. For instance, an application process may need to exchange data with neighboring processes, where this data comprises elements of different sizes, irregularly distributed across local memory. To perform a send, the application may first allocate a temporary buffer, pack this buffer by copying the various elements into the contiguous memory, and then send the contents of the buffer to a receiving process. Likewise, the receiver must unpack the payload, placing the contents into the appropriate memory locations. Because these packing and unpacking routines are not part of the communication software stack, offloading them to the NIC constitutes offloading part of the host application.

While there has been significant research into offloading host applications to specialized appliances (e.g., discrete cards for accelerating mathematical operations or neural network computation [106]–[110]), host application offloading to network appliances has received less attention. Indeed, because of the relative rarity, the novelty is sometimes explicitly noted, e.g., by Dang et al. [60], in their work on offloading consensus algorithms
to intelligent switches.

Because of their importance in reducing datacenter server workload, the most popular host application targeted for offloading is key-value stores or caches, such as Memcached [111]. For example, Kim et al. [112] and Choi et al. [113] exploit increases in on-NIC memory to cache recently accessed web data, potentially avoiding expensive transactions over PCIe. Fukada et al. [114], Chalamalasetti et al. [115], and Istvan et al. [116] explore FPGA-based NICs for offloading aspects of Memcached, e.g., using an on-NIC FPGA (and associated DRAM) as an additional, near-network memory cache; if no match is found on the NIC, the request is passed to the host for processing as usual. Noting that processing key lookups is a bottleneck, Lim et al. [117] propose an ASIC solution, to be included on the NIC, for accelerating request handling. Similarly, the central motivating case for the FlexNIC architecture proposed by Kaufmann et al. [118] is accelerating Memcached by providing the NIC with the capability to inspect headers and intelligently direct lookup requests to cores dedicated to specific key ranges. Lavasani et al. [119] propose implementing a ‘fast path’ for processing Memcached requests on-NIC, passing the request to the host CPU only if necessary. Li et al. [120] explore using SmartNIC offloading capabilities to extend RDMA semantics to include primitives for doing key lookup and allowing the NIC to retrieve values directly from main memory (rather than caching those values on-NIC). Liu et al. [121] implement an offloaded key-value store and distributed transactions as part of their work on the iPipe SmartNIC programming model. Likewise, Choi et al. [122] use Netronome Agilio SmartNICs to offload database transactions and decrease tail latencies. Roughly, the strategy is a copy of a transaction sent to the master is kept on the local NIC, and the master then negotiates with the NIC to determine any ordering constraints, thus allowing the client to proceed as if the transaction had already been finalized. Finally, in HPC environments, key-value stores form the basis for global address spaces. With that target in mind, Larkins et al. [123], [124] propose leveraging the offloaded message demultiplexing provided by NICs implementing
the Portals network API [3] to serve as a key-value store for implementing a version of such a space.

Other possible host application offloads have been less well-explored. Some of the more complex intrusion-detection systems surveyed in Section 2.2.1.3 could be viewed as host application offloading. Underwood et al. [59], [60] offload FFT to FPGA-equipped intelligent NICs. By embedding the FPGA in the path from host memory to NIC, the NIC is able to perform the necessary matrix transformations on the fly, as data is moving between network and host. More recently, Di Girolamo et al. [125] utilize the sPIN SmartNIC architecture to accelerate the packing and unpacking of application-defined MPI datatypes. Glebke et al. [126] present research into offloading convolution for purposes of edge detection in mobile vehicle guidance to a match-plus-action NIC (Netronome Agilio). Sanvito et al. [127] analyze CNN network processing to determine when the host CPU is least efficient, and propose methods for offloading those stages to the NIC, essentially finishing the network computation as the results are delivered to their final destination.

2.2.3 Independent applications

The final category of applications offloaded to SmartNICs is the natural extension of offloading parts of host applications: let the NICs themselves be the hosts. In other words, treat on-NIC compute resources as independent from the host CPUs, allowing them to autonomously execute arbitrary applications. For example, if not needed to accelerate network or host applications, these resources could be allocated like any other compute resource by a job scheduler. Alternatively, they could be used to execute on-NIC kernels for dynamically tuning the network as host applications execute.

Recent work in network offloading points towards a future where network compute resources are treated as independent. For example, Bhattacharyya et al. [128] propose offloading functions as diverse as clickstream analysis, financial data analysis, and sensor
data processing. While in this work these applications are not implemented on Smart-NICs, they cite sPIN [54] as a potential platform.

In cloud computing, ‘serverless compute’ refers to services that allow developers to deploy code without having to consider the full infrastructure they run on. In principle, one can execute arbitrary code under such a service (although in practice, there are limits placed on time and memory). Choi et al. [55] propose a mechanism by which SmartNICs handle the execution of these so-called ‘lambda’ workloads. SmartNICs are again targeted for the execution of arbitrary workloads that may be independent of applications executing on the host.

As a final example, Microsoft Catapult inserts an FPGA in the pathway between switch and NIC [67], [129]. In addition to accelerating ‘local’ applications on the near host, these resources can be allocated to handle ‘global’ applications, i.e., for purposes independent of the local host. While not technically a SmartNIC according to the definition adopted here, this work nonetheless illustrates how SmartNIC compute resources may be treated as independent of their hosts.

2.3 Architectures

The third and final dimension along which we compare SmartNICs is their general architecture. In this section we distinguish between three broad categories of SmartNIC architecture, discussing the benefits and limitations of each.

2.3.1 On-path

The first category of architecture is on-path in the sense that processing elements (PEs) are situated on the packet processing pipeline [121]. Figure 2.1 provides an abstract representation of an on-path architecture. In this scenario, incoming data passes through some initial handler $A$, is directed to one or more processing elements (PE), and then
handed to $B$ for delivery to the host. On-path architectures are perhaps the most common architecture in the SmartNIC ecosystem, because they cohere with the original goal of processing packets in-flight: packets arrive and are forwarded, classified, filtered, checked for errors, etc., and then delivered.

![Figure 2.1: Abstract representation of an on-path SmartNIC architecture.](image)

The decision to place PEs on the processing pipeline has consequences regarding the nature of the work that can be done. First, because these PEs are engaged only when there is traffic in the pipeline, these resources are wasted when the network is idle. Second, processing is fundamentally \textit{deadline-based}. That is, when data arrives at the PE, it must be ready to handle that data, for otherwise the data stream may be invalidated, resulting in an unrecoverable, undefined data state. Since the PE is expected to keep up with line rate, there is a limited amount of time the PE has to perform work. For example, for a 200 Gb/s network with 256B packets, a single issue 2.5 GHz core can execute approximately 26 instructions before it must be ready to service the next packet.

There are several strategies for addressing this deadline. One is to arrange PEs in a pipeline: each PE is subject to the deadline, and the chain of PEs is responsible for executing the entire program. For example, extending the previous example, given a 200 Gb/s network and 256B packets, a chain of 10 2.5GHz cores could execute more than 260 instructions on each packet. Netronome Agilo SmartNICs are an example of a contemporary SmartNIC that adopts a pipelined architecture [36]. A second strategy is to arrange PEs in parallel; each additional PE buys another gap’s worth of processing time. For example, 32 cores with an IPC of 1, operating in parallel on a 200 Gb/s network with 256B packets, each core can execute at most 819 instructions before it must be available to process the next packet. sPIN follows this strategy [54].
Of course, none of these strategies remove the deadline; they only delay it. This leads to a third consequence, regarding code portability. For instance, when network speeds increase to 400 Gb/s [4], all else being equal, the number of available instructions in the parallel example just given is nearly halved, to approximately 420. Unfortunately, as network speeds increase, we cannot reasonably expect core clock speeds or the number of cores to increase proportionally. Consequently, packet processing kernels will not necessarily be forward-compatible.

### 2.3.2 Off-path

The obvious way to avoid network-imposed deadlines is to move the processing element off the packet processing pipeline. Figure 2.2 is an abstract depiction of such a NIC: incoming data passes from $A$ to $B$, at which point it can either proceed to the host, or be diverted to the processing element. Contemporary SmartNICs that utilize this architecture include NVIDIA’s Mellanox Bluefield data processing units [50], [51] and Broadcom’s Stingray NICs [52]. In each of these cases, the processing element is a CPU executing a version of Linux, and communication with other components in the NIC (e.g., $B$ and the host) uses the same mechanism as the host OS (e.g., InfiniBand Verbs).

![Figure 2.2: Abstract representation of an off-path SmartNIC architecture.](image)

Off-path architectures avoid deadlines by moving processing out of the pipeline that imposes them in the first place. This has two consequences. First, since one can no longer assume the PEs will operate within those deadlines, the system must allow for more state than an on-path architecture (where the required memory can be carefully...
tailored to the task and network speed). Second, having to divert data to the off-path PEs can introduce additional latency [121].

2.3.3 In-path

The second strategy for avoiding processing deadlines – and hence enable offloaded general-purpose compute on the NIC – is to reuse on-path PEs. Such an in-path architecture is represented in Figure 2.3: data passes through the PEs as it travels from A to B, and has the option of being ‘recycled’ back into the pipeline for additional processing.

In-path architectures avoid deadlines by adjusting the scope of what constitutes program execution: rather than view execution as something that happens within the pipeline, program execution occurs over that pipeline. That is, while the execution of individual ‘instructions’ is subject to the deadline, overall program execution is not.

One consequence is that the architecture has the benefit of leveraging existing on-path PEs to provide general-purpose compute capability. Second, code developed for one generation is, in principle, portable to future generations. Third, we can assume that existing on-path PEs will adapt to increasing network speeds. Therefore, as network speeds increase, so will program execution speeds. Fourth, as discussed in depth in Chapter 6, because network speeds (i.e., message rates) are currently relatively slow in comparison to CPU speeds (MHz versus GHz), in-path code execution will likewise be slow. This is somewhat mitigated by projected increases in network speeds, but also suggests additional hardware for accelerating kernel execution will be useful. Finally,
like off-path architectures, in-path architectures gain independence from deadlines at the cost of increased state, e.g., for caching data during program execution.

2.4 Discussion: The path forward

In this chapter we’ve considered intelligent network interfaces from three perspectives: the hardware underwriting intelligence, the types of offloaded applications, and the general architecture of SmartNICs.

Regarding hardware, we’ve highlighted three broad categories: ASICs, CPUs, and FPGAs. While we emphasize some types of hardware over others while providing examples of each, it is safe to say that in most (if not all) cases, SmartNICs include some combination of the three, a trend that has a long history. For example, the influential Intel IXP family of network processors includes modified RISC cores for fast packet processing, and an ARM processor for executing arbitrary code [85], [130].

The speed of ASICs or ASIPs makes them good candidates for traditional packet processing tasks that at this point in the evolution of network technology are unlikely to change. However, as this survey reveals, flexibility has become the most important feature for determining whether a NIC is ‘smart’. For example, it is currently not uncommon to find SmartNICs defined as NICs including CPUs or FPGAs [120], [122]. Moreover, there currently exist options that, from a hardware perspective, include ‘everything but the kitchen sink’, e.g., Xilinx’s Versal NICs include an FPGA, realtime and standard CPUs, and configurable ASICs dedicated to artificial intelligence processing [131]. Consequently, while we began the chapter with a broad definition of a SmartNIC as a network interface that offloads core network applications traditionally performed by the host CPU, from a contemporary perspective, a SmartNIC is one that not only offloads, but also does so using hardware (CPUs, FPGAs) that affords the execution of arbitrary (or nearly arbitrary) programs.

The emphasis on flexibility is further highlighted by trends in offloaded applications,
<table>
<thead>
<tr>
<th>Application type</th>
<th>Characteristics</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core network</td>
<td>Offloads L2-L4 protocol processing, collectives, message demultiplexing, etc.</td>
<td>Packet forwarding, TCP segmentation, virtualization (Section 2.2.1.1) TOEs (Section 2.2.1.2) Security (Section 2.2.1.3) Collectives (Section 2.2.1.4) Message matching (Section 2.2.1.5)</td>
</tr>
<tr>
<td>Host</td>
<td>Offloads components of host applications</td>
<td>Key-value stores, memcached, web caching, consensus algorithms, data packing/unpacking, AI (Section 2.2.2)</td>
</tr>
<tr>
<td>Independent</td>
<td>Offloads applications executing autonomously of host applications</td>
<td>Arbitrary jobs (Section 2.2.3)</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of survey of SmartNIC applications (Section 2.2).

summarized in Table 2.1. Initially, an intelligent network interface was simply one that offloaded core network applications. This situation was encouraged by slow NIC processing speeds relative to host CPUs: why bother offloading complex tasks when there is capacity to spare on the host? [76], [95] However, as the limits of Moore’s law approach, support for offloaded applications has climbed up the network stack, increasing in the complexity of computation and the amount of state required. This shift in focus is reflected by recent research that looks beyond core network functions to offloading parts of host applications onto the NIC, or even to treating the NIC as a platform in its own right, i.e., as a compute resource running applications that are independent of any traditional host CPU. Perhaps unsurprisingly, trends towards diversity in the types of offloaded applications parallel those towards flexibility in available hardware.

Finally, we surveyed the abstract architectural approaches that have been pursued for making NICs intelligent. The primary conclusions of this survey are summarized in Table 2.2. On-path architectures situate processing elements on the packet processing pipeline, meaning there are deadlines on the amount of work they can do, imposed by the network speed. Off-path architectures avoid these deadlines by moving processing
On processing pipeline?

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Examples</th>
<th>On processing pipeline?</th>
<th>Deadline-based?</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-path</td>
<td>Quadrics [45]</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Netronome Agilio [36]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Marvell LiquidIOII [53]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Myrinet [43]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPiN [54]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-path</td>
<td>Broadcom Stingray [52]</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>NVIDIA Mellanox Bluefield [50], [51]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In-path</td>
<td>INCA [132]</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of survey of SmartNIC architectures (Section 2.3).

elements off the packet processing pipeline, e.g., by equipping the NIC with an on-board CPU. In-path approaches – of which the design proposed below (Chapter 4) is, to our knowledge, the sole representative – synthesize the on- and off-path approaches by leveraging on-path processing elements to nonetheless provide deadline-free kernel execution.

Many HPC NICs are smart in the traditional sense of offloading core network applications, e.g., collective communication or message demultiplexing. In the next chapter, we begin our inquiry into what SmartNICs can do by identifying message processing overheads incurred by emerging multithreaded approaches to inter-process communication, and assessing the ability of a state-of-the-art NIC offering offloaded support for MPI message matching to deal with these overheads.
Chapter 3

Message matching

Many scientific applications rely on the Message Passing Interface (MPI) for managing data exchange between processes. As noted in Chapter 2, the importance of MPI to these applications has led the offloading of parts of MPI to the NIC or host adapter, e.g., message matching. NICs providing such offloaded capabilities are thus ‘smart’ in the traditional sense of the term.\footnote{Results reported in this chapter are drawn from [133], [134], and [135].}

These smart capabilities may increase application performance given present state of practice, but – as noted in Chapter 1 – we are also concerned with assessing how such capabilities apply to future scenarios. In this chapter, we explore one such future paradigm – multithreaded communication in scientific applications – and assess the ability of a contemporary SmartNIC offering offloaded support for MPI message matching to handle the overheads incurred by multithreaded communication.

MPI includes support for allowing user-level threads to concurrently call into an MPI library [69]. While this level of thread support (\texttt{MPI\_THREAD\_MULTIPLE}) is rarely used in current practice, a recent survey of application developers in the United States’ Department of Energy Exascale Computing Project indicates a majority (86\%) are interested in taking advantage of the opportunities it affords [136]. Specifically, trends towards larger core counts and faster-yet-lower-capacity memory (e.g., high-bandwidth memory
(HBM) [1]) discourage traditional ‘MPI-everywhere’ approaches, where each MPI process instantiates its own set of communication resources. Instead, memory requirements can be reduced by adopting a multithreaded approach, where a single set of resources is used. Therefore, understanding the implications of adding multithreading to MPI codes is important for future application development.

Since under `MPI_THREAD_MULTIPLE`, individual threads can issue sends and receives, this mode of operation may lead to significant increases in the number of messages exchanged. Furthermore, since many threads in a process may be issuing receives or sends concurrently, multithreaded communication may result in deviations to the expected ordering of messages, making the standard strategy of posting receives in the order incoming messages are anticipated to arrive ineffective. This suggests that utilizing multithreaded communication can result in more time spent in MPI message matching, with potential consequences for application performance. This calls for methods for assessing the performance impacts of pursuing multithreaded communication under MPI, including for hardware offloading.

One such method is to emulate a multithreaded halo exchange. This approach has the benefit of being ‘low-cost’ in the sense it requires fewer resources to execute, e.g., two nodes rather than the 9 or 27 required by full exchanges. Consequently, the benchmark can be deployed on systems where acquiring a full allocation is difficult (e.g., because it is under high utilization) or even impossible (e.g., because it lacks the necessary number of nodes).

An alternative is to provide a benchmark implementing the complete, ‘real-world’ multithreaded halo exchange. In comparison to the low-cost approach, this strategy has the benefit of imposing more realistic demands on MPI and the underlying network, but also incurs additional resource costs.

In this chapter we make the following contributions:

- A formal analysis of thread and message counts under multithreaded halo commu-
The design and implementation of a ‘low-cost’ benchmark for assessing MPI communication performance under typical multithreaded halo exchanges;

• The design and implementation of a ‘real-world’ benchmark for assessing MPI communication performance under those same communication patterns;

• An evaluation of the low-cost benchmark relative to the real-world baseline, across multiple architectures; and

• A case study using the low-cost benchmark to examine the impact of message matching offloading on multithreaded halo exchange.

The low-cost and real-world benchmarks will be included in the next release version of the Sandia MPI Micro-Benchmark Suite (SMB) [137].

The remainder of the chapter is structured as follows. In Section 3.1, we provide summary of MPI message matching and how multithreaded communication might affect it. Section 3.2 provides a brief survey of related work, and in Section 3.3, we provide a formal analysis of features of multithreaded halo exchanges such as number of threads participating in inter-process communication and number of messages exchanged. In Section 3.4, we describe the benchmarks and the systems upon which they were run. Results from both benchmarks are presented and discussed in Section 3.5, and in Section 3.6, we consider results from running the low-cost benchmark on a system with hardware offload support for message matching.

3.1 Background

Message matching is MPI’s receiver-side data-placement mechanic, used primarily to support point-to-point communication. To send a message (e.g., through \texttt{MPI\_Send} or \texttt{MPI\_Isend}), an MPI process specifies a buffer containing data to be sent, a destination
ID (‘rank’), and a placement identifier (‘tag’). The receiving MPI process posts a corresponding receive (e.g., MPI_Recv or MPI_Irecv) specifying a buffer where data will be placed, the rank of the sender, and the tag of the expected message. The communication is completed when the receiver matches the sending rank and tag of an incoming message to that of a posted receive, and the payload delivered to the specified buffer.

The MPI specification imposes several constraints on receiver-side message matching. First, messages with the same matching fields must be matched in the order their receives are posted. Second, the matching mechanism must allow wildcards for both rank and tag. To handle these requirements, traditional implementations use two linked lists: a list of outstanding receive requests in a posted receive queue (PRQ), and a list of messages that have arrived and failed to match any receive request in the unexpected message queue (UMQ).

![Figure 3.1: MPI message matching queues. Given an incoming message, the receive requests posted in the PRQ are searched, and if no match is found, the message is enqueued at the tail of the UMQ. Likewise, given a new posted request, the UMQ is searched to determine whether the matching message has already arrived, and if not, the request is appended to the PRQ.](image-url)
As shown in Figure 3.1, when an MPI process posts a receive, its UMQ is traversed to determine whether a message with the desired sending rank and tag has already arrived, and if not, the receive is appended to the PRQ. When a message arrives at that process, the PRQ is traversed to determine whether a receive with the required rank and tag has already been posted, and if not, the information is appended to the UMQ. MPI ordering and wildcard semantics are guaranteed by initiating searches from queue heads and appending to their tails. For the purposes of this paper, we use a traditional dual-queue model for message matching, based on the model used by MPICH [138] and its derivatives. Some other implementations have opted for different models. For example, Open MPI [139] utilizes an array of lists, indexed by sending rank, which can reduce average search depth at the cost of increased memory. The benchmarks and results presented in this chapter can provide a better understanding on how these optimized models will impact next-generation applications.

An obvious concern for this approach are situations where (i) the PRQ or UMQ grow large, and (ii) the order in which messages arrive or requests are posted is such that either no matches are found, or that when matches are found, they tend to occur near the end of a list. In these cases, message processing latencies increase due to time spent searching, and this can disrupt application performance. The introduction of multithreading to communication via MPI\_THREAD\_MULTIPLE prima facie encourages this problem in two ways. First, because individual threads are now participating in communication, there is a corresponding increase in the number of messages exchanged, and queue sizes grow. Second, a rule of thumb for efficient MPI programming is to post sends in the same order as receives so that even if there are many messages to be processed (and the PRQ or UMQ grows large), matches are always found at or near the head of the list, keeping time spent searching to a minimum. With multithreaded communication, however, threads are issuing sends or receives concurrently, introducing nondeterminism into this ordering, and rendering the standard strategy ineffective. One goal of this research is to ascertain
how just how disruptive this nondeterminism is.

### 3.2 Related work

The possibility that message matching overheads may negatively affect application performance has led to research on both understanding MPI message matching behavior, and into strategies for mitigating the impact of performing queue searches. For example, initial work by Underwood et al. [140] explored the performance impact of long queues. Balaji et al. [141] look at search overheads on BlueGene/P, and further studies by Barrett et al. [142] showed the impact of match list length on a variety of system architectures. Brightwell et al. [143] measure queue characteristics of several applications, and Dosanjh et al. [144] considers the impacts of temporal and spatial locality on match engine performance. Ferreira et al. [145] and Levy et al. [134] use trace-based simulations to gain insights into message matching costs for a variety of scientific applications, noting that in most (single-threaded) cases queue lengths tend to remain small. Finally, Bridges et al. [146] present a model for assessing MPI queue performance on many-core systems.

Strategies for mitigating message matching costs include both software refinements and, more recently, hardware offloading. Proposed software-side optimizations include using a dedicated thread for message processing [147], incorporating data structures such as hash tables [148]–[152], leveraging underlying InfiniBand queue pairs to satisfy MPI semantics [153], using vector units to check multiple items in parallel [154], or enabling more fine-grained locks on queue data structures [155].

Regarding hardware-based approaches, Ferreira et al. [156] use simulation results to identify requirements on hardware solutions (e.g., amount of memory). Tourancheau and Westrelin [97] address rendezvous communication by proposing message matching and associated queues be offloaded to an on-NIC CPU. Underwood et al. [32] propose a TCAM-based architecture for offloading message matching, Tanabe et al. [104] explore an offloading strategy based on separating headers and payloads, Klenk et al. [105] consider
how GPUs may accelerate matching if current MPI semantics can be relaxed, and current Atos Bull and NVIDIA Mellanox ConnectX-5 NICs provide offloaded MPI message matching functionality [2], [35].

Several works have addressed multithreading support in MPI by improving implementation internals [157]–[159], and proposing new interfaces [160], [161]. In addition to traditional send/receive multithreading and matching overheads, work has also examined multithreading in the context of MPI one-sided communication [162], [163]. Other MPI multi-threaded benchmarks have been proposed for basic MPI functionality and performance testing [137], [164], [165].

In the research described in remainder of this chapter, we contribute to the general understanding of message processing overheads by investigating a natural extension of halo exchange communication to multithreaded contexts. Moreover, as illustrated by the case study provided in Section 3.6, the benchmarks developed here allow one to assess how hardware offload solutions perform.

3.3 Analysis

As noted in Section 3.1, multithreaded communication, when applied to typical halo exchange communication patterns, increases the number of messages exchanged. To emulate such a pattern, then, we need to establish both the number of threads and the number of messages exchanged, for a variety of common 2D and 3D stencils. In this section, we present this analysis.

The halo exchange communication pattern is common amongst scientific applications [166], [167]. Figure 3.2a illustrates a simple single-threaded, 9-point, two-dimensional halo exchange between process $p_4$ and its neighbors. In a typical bulk-synchronous processing (BSP) application using this stencil, after each process completes its assigned work, it exchanges data with its eight nearest neighbors. Once this exchange is complete, it continues into a new work phase.
(a) A singlethreaded, nine-point halo exchange.

(b) A multithreaded, nine-point halo exchange.

Figure 3.2: Singlethreaded (a) and multithreaded (b) nine-point halo exchanges.

Figure 3.2b represents a straightforward multithreaded version using the same stencil communication pattern. In this scenario, work on $p_4$ is divided between 16 threads ($t_0$ through $t_{15}$). During the communication phase, each thread is responsible for exchanging messages with its neighboring threads, based on the same 9-point stencil pattern shown in Figure 3.2a.

Assuming that inter-thread communication within the same process is handled outside of the MPI matching engine (e.g., via shared memory), one can calculate the number of sending and receiving threads, and the total number of messages exchanged between processes. Specifically, a decomposition can be viewed as a collection of ‘faces’ of different dimensions. For example, a square (2D) decomposition comprises 4 faces of dimension 0 (corners)\(^2\), 4 faces of dimension 1 (edges)\(^3\) and 1 face of dimension 2 (interior)\(^4\). If we assume, for purposes of exposition, that all dimensions have the same length, $x$, then the total number of threads, $t(k)$, on faces of dimension $k$ can be expressed as a variation on:

\[^2\]Each corner comprises a face containing a single thread (e.g., $t_0$, $t_3$, $t_{12}$, and $t_{15}$ in Figure 3.2b).
\[^3\]Each edge comprises a face containing a one-dimensional line of threads (e.g., $t_1$-$t_2$ in Figure 3.2b).
\[^4\]The interior comprises a face containing a two-dimensional grid of threads (e.g., $t_5$, $t_6$, $t_9$, and $t_{10}$ in Figure 3.2b).
the standard equation for calculating the faces of a hypercube of dimension $d$:

$$ t(k) = (x - 2)^k 2^{d-k} \binom{d}{k} $$

(3.1)

for $x \geq 1$. In this equation, the first factor $((x - 2)^k)$ is the volume of the $k$-dimensional face, expressed as the number of threads it contains. The remainder of the equation computes the number of $k$-dimensional faces. The total number of threads is simply the sum of the number of threads for each possible dimension:

$$ \sum_{k=0}^{d} t(k) $$

(3.2)

The number of threads involved in inter-process communication depends on the dimensionality of the stencil used to define the communication. For this work, we limit our analysis to standard 5 and 9-point 2D stencils, and 7 and 27-point 3D stencils. If the dimensionality of the stencil ($d_s$) is greater than the dimension of the thread decomposition ($d$), then the set of participating threads includes every thread in the decomposition. Otherwise, the number of participating threads is equal to Equation 3.2, where the upper bound of the summation is reduced to $d - 1$, i.e., the single $d$-dimensional face is excluded. The case where $d_s < d$ is beyond the scope of this work.

The number of messages exchanged by each thread depends on the face it belongs to and the stencil used. For communication limited to the Von Neumann neighborhood (5 and 7-point stencils), the number of messages is given by Equation 3.3. For communication within the Moore neighborhood (9 and 27-point stencils) the number of messages is given by Equation 3.4.

$$ m(k) = 2d_s - (d + k) $$

(3.3)

$$ m(k) = 3^{d_s} - 3^k 2^{d-k} $$

(3.4)
That is, the number of messages exchanged by a thread is equal to the total number of messages \((2d_s\text{ for the Von Neumann neighborhood, and } 3d_s\text{ for the Moore neighborhood})\) minus the number of intra-process messages. The total number of messages exchanged is therefore the sum of the messages processed by each thread on each face:

\[
\sum_{k=0}^{d} t(k)m(k) \tag{3.5}
\]

If \(d_s = d\), the upper bound of the summation is reduced to \(d - 1\).

This analysis can be straightforwardly (if tediously) extended to handle the case where the number of threads in each dimension is not equal, e.g., a rectangular (rather than square) decomposition in two dimensions. For present purposes, Table 3.1 summarizes the number of messages processed by the receiver for each of the decompositions and stencils considered in the experiments reported in this chapter, as calculated using the analysis just provided. These data illustrate how multithreading can significantly increase the number of messages exchanged.

<table>
<thead>
<tr>
<th></th>
<th>1x1</th>
<th>2x1</th>
<th>2x2</th>
<th>4x2</th>
<th>4x4</th>
<th>8x4</th>
<th>8x8</th>
<th>16x8</th>
<th>16x16</th>
</tr>
</thead>
<tbody>
<tr>
<td>5pt</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>9pt</td>
<td>8</td>
<td>14</td>
<td>20</td>
<td>32</td>
<td>44</td>
<td>68</td>
<td>92</td>
<td>140</td>
<td>188</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1x1x1</th>
<th>2x1x1</th>
<th>2x2x1</th>
<th>2x2x2</th>
<th>4x2x2</th>
<th>4x4x2</th>
<th>4x4x4</th>
<th>8x4x4</th>
<th>8x8x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pt</td>
<td>6</td>
<td>10</td>
<td>16</td>
<td>24</td>
<td>40</td>
<td>64</td>
<td>96</td>
<td>160</td>
<td>256</td>
</tr>
<tr>
<td>27pt</td>
<td>26</td>
<td>50</td>
<td>92</td>
<td>152</td>
<td>272</td>
<td>464</td>
<td>728</td>
<td>1256</td>
<td>2072</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1x1x1</th>
<th>1x1x2</th>
<th>1x1x4</th>
<th>1x1x8</th>
<th>1x1x16</th>
<th>1x1x32</th>
<th>1x1x64</th>
<th>1x1x128</th>
<th>1x1x256</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pt</td>
<td>6</td>
<td>10</td>
<td>18</td>
<td>34</td>
<td>66</td>
<td>130</td>
<td>258</td>
<td>514</td>
<td>1026</td>
</tr>
<tr>
<td>27pt</td>
<td>26</td>
<td>50</td>
<td>98</td>
<td>194</td>
<td>386</td>
<td>770</td>
<td>1538</td>
<td>3074</td>
<td>6146</td>
</tr>
</tbody>
</table>

Table 3.1: Number of messages processed by receiver under different thread-level decompositions and stencils.

### 3.4 Benchmarks and methodology

To assess the impact of multithreading on MPI message matching performance, we designed and implemented two multithreaded halo exchange benchmarks. Each benchmark
follows a BSP pattern, where receives are pre-posted in anticipation of incoming messages.

Recent versions of Open MPI include an option to utilize a traditional dual-queue matching engine of the sort described in Section 3.1, optimized for SIMD operations (see [154]). To collect data on number of items searched and time spent searching, we instrumented this traditional dual-queue matching engine, without enabling these optimizations. Our instrumented Open MPI reports: (i) the number of items searched in the PRQ before a match is found for each incoming message; and (ii) the total amount of time spent searching the PRQ. Since all of the receives are pre-posted in these benchmarks, all incoming messages will have a match in the PRQ.

The first benchmark is *real-world* in the sense it performs an actual multithreaded halo exchange. That is, nine processes are used for 2D stencils, and 27 processes are used for 3D stencils. Depending on the dimensionality of the process-level decomposition, Each process is partitioned into \( x \times y \) or \( x \times y \times z \) threads. Using `MPI_THREAD_MULTIPLE`, threads on the perimeter of the decomposition participate in inter-process message exchanges with those of neighboring processes, posting receives and issuing sends. Communication between processes is non-periodic (i.e., non-toroidal). Because the goal is to assess MPI matching engine performance, and this engine can be bypassed when MPI processes reside on the same node, each process is situated on a distinct node. At the thread level, all receives are pre-posted in the same order as the corresponding sends, and data is collected from the center process via the modified Open MPI described above.

Based on the analysis presented in Section 3.3, the *low-cost* benchmark *emulates* an actual multithreaded halo exchange using only two MPI processes: a sender and a receiver (cf. [133]). Both use multithreaded MPI for communication, but their thread counts differ. The number of receiver-side threads is determined by a version of Equation 3.2, and is equal to the number of threads used by the center process in the real-world

---

5Open MPI hash dd74e6252f8947e213e83f470024f3a4ce78b10b
benchmark. The number of threads used by the sender is the total number of threads issuing sends in the real-world case, as calculated using minor variations on Equations 3.1 and 3.2. In other words, while the receiver corresponds to the center process in an actual halo exchange (e.g., \textit{p}_4 in Figure 3.2a), in the low-cost benchmark, the threads from surrounding processes (e.g., \textit{p}_0-\textit{p}_3 and \textit{p}_5-\textit{p}_8 in Figure 3.2a) are collected onto a single sending process.

There are multiple ways that an application developer can decompose their problem with multiple threads. We consider three possible decompositions: two-dimensional ‘square’ (e.g., 1×1, 2×1, 2×2, 4×2, …), three-dimensional ‘cube’ (e.g., 1×1×1, 2×1×1, 2×2×1, 2×2×2, …), and three-dimensional ‘linear’, which scales only along the \textit{z} axis (e.g., 1×1×1, 1×1×2, 1×1×4, …). Square and cube decompositions represent typical ways that applications decompose problems. The linear decomposition represents a less efficient corner case.

To compare the real-world and low-cost benchmarks, we ran both on a Cray XC40. This system comprises two compute node partitions. In the first, each compute node has two sockets of Intel Xeon ES-2698 v3 (Haswell) processors operating at 2.3GHz. Each processor has 16 cores with 2 hardware threads per core, for a total of 64 thread contexts. In the second partition, each node has a single socket containing a 68 core Intel Xeon Phi 7250 (Knights Landing) processor operating at 1.4GHz. Each core has 4 hardware threads, for a total of 272 thread contexts. The nodes are connected via a Cray Aries network. All tests were conducted on a single cabinet from one of the Cray XC40’s partitions. Therefore the results are for a fully connected network (not a dragonfly, as would be the case for communication that spanned multiple cabinets).

The low-cost and real-world benchmarks were executed on each partition, using one MPI process per node, with each of the three types of decomposition. Each decomposition was scaled up until the point oversubscription would occur on the receiver side. For the low-cost benchmark, this means sender-side oversubscription is permitted. For the real-
world benchmark, this means that oversubscription is not permitted, since all processes are both senders and receivers. For the square decomposition, data was collected for 5 and 9-point stencils; for the cube and linear decompositions, data was collected for 7 and 27-point stencils. Because internal nodes in a regular halo exchange have an equal communication overhead, and assessing this overhead is the goal of the benchmark, scaling beyond 9 nodes (for 2D decompositions) or 27 nodes (for 3D decompositions) is not necessary. For each configuration, the benchmarks were run 50 times, with each run executing two trials, i.e., two halo exchanges. All reported results are for the second of the two trials, and all messages contained 8-byte payloads.

For the case study on offloaded message matching (Section 3.6), we ran the low-cost benchmark on an ARM-based testbed, with offloaded matching disabled and enabled. On this system, each compute node has two sockets. Each socket contains a 28-core Cavium ThunderX2 CN9775 processor operating at 2GHz. The ARM testbed’s network utilizes Mellanox ConnectX-5 interfaces, which provide dedicated hardware assistance for MPI message matching [168]. This capability is accessed via OpenUCX [169]. For the experiments in this paper, we used UCX 1.7.0, configured with optimizations (--enable-optimizations) and multithreaded support (--enable-mt) enabled. We used the same version of MPI as in the previous experiments. However, because Open MPI’s message matching is bypassed by UCX, our MPI instrumentation was also bypassed. Therefore, for these results, we report the time spent processing messages as measured in the benchmark code. This includes a barrier (to ensure the sender cannot begin issuing messages before the timer is started) and a call to MPI_Waitall. In addition to supporting hardware offloading, UCX also includes message matching optimizations in software (in the form of binning messages into 1021 bins according to tag and source), and these optimizations are reflected in the reported times. Hardware offloading of message matching is controlled via the UCX_RC_MLX5_TM_ENABLE environment variable. The default threshold recommended by Mellanox for engaging offloading
is 1024 bytes. For our experiments, we kept this default value, meaning that even when offloading is enabled, it is not used for messages whose size is less than 1KiB.

3.5 Results

In this section, we compare results derived from the two benchmarks, confirming that the low-cost benchmark provides a reasonable approximation of a genuine multithreaded halo exchange while requiring fewer resources.

3.5.1 Resource usage

![Figure 3.3: Average core hours for the real-world and low-cost multithreaded halo exchange benchmarks executing a 27-point 3D halo exchange.](image)

Figure 3.3 compares the average (over 30 runs) number of core hours for the real-world and low-cost benchmarks executing a multithreaded 27-point, 3D halo exchange. Note that results were obtained on 2.3GHz Intel Xeon ES-2698 v3 processors using Cray MPICH v7.7.6. As the size of the thread decomposition increases, the low-cost benchmark remains under 0.25 core hours, while the real-world benchmark approaches 2 core hours. The reduction in resource usage afforded by the low-cost benchmark ranges from 9.1x to 16.1x relative to the real-world benchmark. This reduction in resource usage decreases as the number of threads increases, because the amount of work each process in the low-cost benchmark has to do increases faster than the work done by each of the processes in the
real-world benchmark. However, we expect the low-cost benchmark to be used at small scale, mostly by MPI researchers and for MPI performance regression testing. MPI halo exchange performance is often tested/monitored on supercomputers, which can now use a lower-cost benchmark to achieve this goal.

### 3.5.2 Items searched

The total number of items searched during a halo exchange, as well as search depths for the processing of individual messages, are useful for understanding the overheads are
due to the non-deterministic behavior that is introduced by multi-threaded communication [133]. Consequently, we begin by contrasting total items searched as reported by the low-cost and real-world benchmarks. Figure 3.4 summarizes this data for square, cube, and linear decompositions for KNL and Haswell architectures. Data points represent median search depth (over 50 runs), and error bars are first and third quartiles. Note that the $y$-axis is logarithmic, and differs between KNL and Haswell architectures. Furthermore, the decomposition for the KNL architecture has two extra data points over Haswell due to the greater number of available execution contexts.

Both benchmarks confirm the hypothesis that the non-deterministic ordering introduced by multithreaded communication leads to increased search depths [133]. For instance, under the real-world benchmark on Haswell, a 4x4x4 cube decomposition with 27-point stencil communication results in a total number of items searched that, on average, is 170.5 times larger than the ideal, where each search matches on the first item. Comparing the two benchmarks, the mean absolute error across all stencils and decompositions, expressed as a ratio to the median reported by the real-world benchmark, is 16.4% ($\sigma = 11.1$). We observe that disagreement between the low-cost and real-world benchmarks occurs under larger (9 and 27-point) stencils, and this typically involves the low-cost benchmark underestimating the total number of items searched. For instance, the average error of 5-point stencils versus 9-point stencils for square decompositions, across both architectures, is 7.6% and 16.6%, respectively. Likewise, for 7-point stencils versus 27-point stencils and cube decompositions, these values are 6.3% and 27.7%.

Figure 3.5 offers a more detailed view of the data for three different cube decompositions on the Haswell architecture, selected because they clearly manifest this trend. In these histograms, the $x$-axis specifies bins of numbers of items searched during the processing of each incoming message during an exchange, and the $y$-axis the number of searches of that depth, averaged across all 50 trials. In all three cases, there is strong agreement between low-cost and real-world benchmarks for 7-point stencil communica-
Figure 3.5: Histograms showing average number of searches at each bin of items searched.
Figure 3.6: Median queue drain times for 2D (5 and 9-point stencils) and 3D (7 and 27-point stencils) decompositions, for KNL (left) and Haswell (right).

In contrast, for 27-point stencils, the low-cost benchmarks exhibits greater numbers of shallower searches, and correspondingly fewer deep searches, in comparison to the real-world benchmark. Furthermore, the maximum search depths for the real-world benchmark consistently exceed that of the low-cost benchmark.

### 3.5.3 Matching overheads

Determining search times can give a user an idea of how expensive MPI overhead is for multiple decomposition strategies. Even close agreement between low-cost and real-world benchmarks with respect to mean or median number of items searched does not
guarantee that the temporal overhead of searching the match lists is similar, because
the temporal costs of searching may not be linear. Consequently, to assess how the low-
cost benchmark relates to the real-world version regarding processing times, we compare
time to process all items in the posted receive queue (i.e., ‘queue drain times’) for each
architecture, stencil, and decomposition. Results are summarized in Figure 3.6. As
before, data points represent medians, and error bars are first and third quartiles. The
$y$-axis is logarithmic and differs between architectures, and KNL has two additional data
points.

Previous work has observed that the costs of multi-threaded communication can be
prohibitively expensive, requiring more time for message processing than currently al-
located by current scientific applications for an entire compute-plus-communication cy-
cle [133]. The results reported here confirm this observation across both benchmarks
and architectures. For example, on Haswell, 27-point communication on a 4x4x4 cube
decomposition exceeds 1 millisecond for queue processing.

Comparing the benchmarks, the mean absolute error across all stencils and decom-
positions (calculated as in Section 3.5.2) is 24.9% ($\sigma = 18.6$). However, as shown in
the subfigures, discrepancies between benchmarks are more pronounced for decomposi-
tions and stencils with fewer numbers of messages; as the number of messages increases,
disagreement decreases. For example, whereas the average error across 5-point stencils
is 36.0%, this decreases to 17.6% for 27-point cube decompositions, and to 16.7% for
27-point linear decompositions.

Figure 3.7 shows results from a square and two cube decompositions, all executed on
Haswell, chosen because they are representative of the trend towards agreement. As these
figures illustrate, a notable contributor to discrepancies between low-cost and real-world
results are outliers: on all stencils, for lower message counts, the low-cost benchmark has
more extreme slow outliers than the real-world benchmark, and this gap closes as the
number of messages increases. For example, for the 4x4 decomposition, the maximum
Figure 3.7: Histograms showing average time per search for selected decompositions. Annotations show the minimum and maximum search times for real-world (rw) and low-cost (lc) benchmarks, for each stencil.
real-world search times under 5 and 9-point stencils are 436ns and 828ns, respectively, while the corresponding low-cost values are 12686ns and 9367ns, exhibiting gaps of one or two magnitudes. For the 4x2x2 cube decomposition, the maximum real-world 7-point stencil is 909ns while the low-cost is 7213ns, and for 27-point these are 11051ns and 12941ns. Finally, for the 4x4x4 cube decomposition, the maximums are 6251ns vs. 7817ns (7-point) and 23080ns vs. 24061ns (27-point), indicating that by this point, the gaps have closed significantly.

3.5.4 Discussion

Previous work has studied the impact of multithreading on message queues and processing times on a KNL system [133]. As part of the present work, we’ve reproduced that study. While our results support the conclusion that multithreaded communication leads to increased search depths and potentially problematic message processing times, observed message processing times trend higher than in that earlier work. This is likely due to the Spectre/Meltdown exploit, a variant of which affects look-ahead pointer dereferencing [170]. In contrast to the data presented here, results gathered for the earlier work were obtained prior to the exploit being patched. Unfortunately we were unable to test this hypothesis due to the unavailability of unpatched nodes.

As noted above, the results indicate a user of the low-cost benchmark should be aware that the benchmark may, under larger message counts, underestimate the number of items searched during message processing. This is not unexpected: whereas in the low-cost benchmark, the non-deterministic order of message arrivals at the receiver is due entirely to thread-level competition at the single sender, the real-world benchmark adds network-induced non-determinism, i.e., variation in message arrivals due to taking different routes through the network. We hypothesize that as adaptive routing gains traction in HPC, this divergence between benchmarks will decrease, assuming processes participating in the low-cost benchmark are sufficiently far removed in the network topology.
Finally, the user should also keep in mind that the low-cost benchmark can overestimate message processing time for smaller message counts in comparison to the real-world benchmark. Again, this is not unexpected: because there is only a single sender, in the low-cost benchmark messages arrive at a higher rate than in the real-world benchmark, where arrival times are diluted by having multiple senders. This means in the low-cost benchmark, more memory accesses are occurring in a smaller window of time, introducing contention and leading to the outliers noted in Figure 3.7. As the number of messages increases, the real-world benchmark trends towards the behavior exhibited by the low-cost benchmark.

3.6 Case study: assessing hardware offloaded message matching

As described in Section 3.4, we ran the low-cost benchmark on a ConnectX-5 system with hardware matching enabled and disabled. Because this system bypasses our instrumented MPI, we record the time spent processing incoming messages, within the benchmark rather than within MPI. Furthermore, since offloading is only active above a certain threshold of message size (1024 B, the default), we also vary message sizes to span this transition point (from 8 B to 1 MiB). For each set of parameters, the benchmark was executed 50 times, with 11 emulated halo exchanges per run; the data presented here discards the 1st exchange from each run, for a total of 500 trials.

Results for three message sizes – small (512 B), medium (16 KiB), and large (1 MiB) – are presented in Figure 3.8. The left column shows results for square decompositions, and the right for cube decompositions. Plotted values are medians, and error bars represent the 1st and 3rd quartiles.

As expected, when message sizes are below the threshold for engaging offloaded matching (<1024 B), processing time is not affected by hardware matching being enabled.
Figure 3.8: Median time spent processing messages with hardware matching enabled (HWM+) and disabled (HWM−), for square and cube decompositions and small (512B), medium (16KiB), and large (1MiB) messages.
Figure 3.9: Median time spent processing messages by message size, square and cube decompositions, with hardware matching enabled (HWM+) and disabled (HWM−).

(Figure 3.8, row 1). When message sizes are modestly above the threshold (e.g., 16 KiB), hardware matching accelerates processing time across most square and cube decompositions and stencils (row 2). Within each type of decomposition, speedup is more pronounced for larger stencil sizes (larger numbers of messages). For square decompositions, the average speedup for the 5-point stencil is 1.13×, and for the 9-point stencil it is 1.23×. Similarly, for cube decompositions, the average speedup for the 7-point stencil is 1.21× while that for the 27-point is 1.39×.

However, our benchmark also reveals (Figure 3.8, row 3) that when messages become large (e.g., 1 MiB), hardware-assisted message match may actually slow down message processing relative to software message matching. For 1 MiB messages, the average slowdown is remarkably consistent across types of decomposition and stencils: 1.85× for 5-point square, 1.87× for 9-point square, 1.87× for 7-point cube, and 1.88× for 27-point cube. These results suggest that, on this system, there exists a ‘window of effectiveness’ of hardware offloading as regards handling the overheads incurred by multithreaded communication patterns.

To better characterize this window, we plot processing time across all message sizes for selected square and cube decompositions in Figure 3.9 (note the gap between 8 B and 512 B). We observe that offloaded matching typically reduces processing time for message
sizes beginning near the default threshold of 1024 B, but this benefit disappears between 32 KiB and 64 KiB, at which point offloaded matching incurs additional overheads in comparison to not using offloading. This effect threshold is observed regardless of the number of messages being processed.

### 3.6.1 Discussion

These results confirm the potential benefits of hardware-assisted message matching in handling the increased overhead of multithreaded applications. However, they also suggest that there are situations where offloading may be detrimental to application performance (e.g., when message sizes exceed 32 KiB).

Previous work by Marts et al. [168] has also considered the impact of ConnectX-5 message matching offloading. The results presented in here are consistent with this earlier study. Marts et al. saw similar performance benefits that were limited to a window of message sizes between 1 KiB and 16 KiB. For messages larger than 16 KiB, they showed that hardware-assisted message matching was slower than software message matching. The same effect is seen in Figure 3.9. These results are also consistent with Mellanox’s default threshold which limits hardware-assisted message matching to messages that are larger than 1024 B.

Marts et al. also observed that even for message sizes within this window, UCX tag binning collisions can decrease the performance of hardware message matching. Therefore, the fact that we observe a speedup for these medium-sized messages suggests that our benchmark is balancing its tags across the bins effectively. Provided that a user sends messages in this ideal size window and ensures that their tag use is consistent with low binning collisions, Mellanox’s hardware offloading has the potential to alleviate some of the aforementioned concerns in MPI matching overhead.
3.7 Conclusion

Because of the centrality of MPI to HPC, vendors are now offering NICs that are ‘smart’ in the traditional sense of providing hardware support for MPI message matching. A goal of the present work is to assess what these sorts of offloaded capabilities can do, especially with respect to future programming paradigms.

To this end, in this chapter we’ve presented the design and implementation of two benchmarks for assessing the potential performance impact of multithreaded MPI communication (using `MPI_THREAD_MULTIPLE`) under common 2D and 3D halo exchanges. The ‘real-world’ benchmark implements a full halo exchange using 9 or 27 nodes, while the ‘low-cost’ version builds on the analysis of thread and message counts provided in Section 3.3 to emulate the exchange using only two processes. Results from both benchmarks executed on multiple architectures show the two are comparable with respect to number of items searched and time spent searching.

The benchmark results confirm and extend earlier work suggesting that multithreaded halo exchanges can incur unacceptable latencies, severely impacting application performance. For example, Schonbein et al. [133] note that on current systems, molecular dynamics applications (e.g., [171]) typically budget approximately $10^5$ to less than $10^3$ µs per simulated femtosecond; this includes a complete halo exchange. The results reported above reveal multiple scenarios where message matching overhead alone can exceed the iteration’s entire time budget. Taken together with other known inefficiencies in implementations of multithreaded MPI (e.g., [160]), these results provide additional motivation for exploring other uses for multithreading in MPI, such as partitioned communication [161].

Finally, the offloading case study indicates that hardware-based matching can be effective at reducing message processing latencies, even under the increased message counts incurred by multithreaded communication. However, message processing times remain high, and (at least in the case of the specific hardware considered in this study)
larger message sizes may actually incur additional overheads when hardware offloading
is enabled.

These results provide a more nuanced understanding of SmartNIC capabilities as
regards offloaded message matching. In the following chapters, we show that hardware-
based message matching has another potential future life in HPC, besides coping with
multithreaded communication: when bundled with other offloaded network applications
typical of HPC, message matching enables NICs whose offloading capabilities were not
general in the sense invoked by the contemporary notion of a SmartNIC can be ‘made
intelligent’, i.e., made capable of executing arbitrary programs to assist host applications,
or even execute independent programs.
Chapter 4

INCA: In-Network Compute Assistance

In the previous chapter, we explored what a particular offloaded ‘smart’ capability – message matching – can contribute to the emerging paradigm of multithreaded communication in scientific computing. In this chapter, we consider what offloaded message matching can accomplish when coupled with other hardware-assisted smart capabilities.

State-of-the-art NICs deployed in HPC systems provide offloaded support for several network applications of critical importance to scientific workflows. These include support for MPI message matching (as explored in Chapter 3), and for collective operations such as barriers, broadcasts, gathers, and reductions. NICs supporting the OpenFabrics standard [172], such those based on the Portals network programming API [3], [35], provide this offloaded support through three basic capabilities. The first is direct support for message matching, e.g, via TCAMs [32]. Support for collectives, including reductions and barriers, is provided through two additional offloaded capabilities: triggered operations and atomic operations. A NIC supporting triggered operations can autonomously generate outgoing messages in response to events (e.g., buffer updates) caused by incoming traffic. Triggered operations thus provide primitives for constructing offloaded collectives and rendezvous messaging [102], [103]. A NIC supporting atomic operations can perform basic arithmetic or logical operations (e.g., summation, compare-and-swap) on message payloads. In conjunction with triggered operations, atomic operations facilitate...
the offloading of reductions, distributed mutexes, and so on.

Considered individually, each of these offloaded capabilities renders the NIC ‘smart’ in the traditional sense of the term (Chapter 2). However, in this chapter, we show how these three task-specific capabilities – message matching, atomic operations, and triggered operations – can be leveraged to make a NIC intelligent in the contemporary sense, i.e., being capable of executing arbitrary, user-defined programs. That is, offloaded general-purpose compute capabilities do not require appeals to CPUs or other general-purpose hardware; existing task-specific capabilities common to HPC NICs may be sufficient.

The demonstration proffered in this chapter provides the formal basis for INCA (In-Network Compute Assistance), a framework for SmartNIC offloading that is further developed in subsequent chapters. INCA is unique in the SmartNIC landscape in that it offers deadline-free kernel execution while nonetheless utilizing on-path processing elements. In the terminology of Chapter 2, INCA is in-path. To our knowledge, while the strategy of recirculating data through an on-path architecture makes rare appearances in the literature (e.g., [126]), INCA is the first contemporary SmartNIC design to explicitly adopt an in-path architecture\(^1\).

We begin with an overview of the Portals message processing pipeline in Section 4.1. In Section 4.2 we make the strategy for securing general-purpose compute capabilities concrete by presenting the Triggered Operation Machine (TOM), a formal model of computation based on the primitive capabilities of message demultiplexing, triggered operations, and atomic operations. We prove the TOM is Turing complete by reducing the well-known universal register machine model of computation [173] to the TOM in Section 4.3. The practical outcome of this proof is a clear specification of a set of basic changes to existing Portals NICs sufficient to secure general-purpose compute capabilities; these are discussed in Section 4.4.

\(^1\)Some of the results reported in Chapters 4, 5, 6, and 7 appear in [132].
4.1 The big picture

NICs supporting the OpenFabrics standard [172], such as those adopting the Portals network programming API [3], typically provide hardware support for three primitive network applications: (1) message demultiplexing (cf. Chapter 3), (2) triggered operations, and (3) atomic operations.

Figure 4.1: The Portals message processing pipeline. Numbers indicate steps in the processing pipeline, as described in the main text.

Figure 4.1 provides a data-structure view of the Portals message processing pipeline, including these functions. (1) An incoming message matches against a list of Portals table entries (PTEs), and then (2) against a list of matching elements (MEs) attached to that PTE. In this illustration, the incoming message matches the second ME. Each ME specifies a local buffer that is the destination for the payload of the incoming message. When the incoming message matches the ME, an optional atomic operation (3) is performed on the message payload and (in the case of a binary operation) the contents of the local buffer specified by the ME, and the result stored in that buffer. (4) The act of writing to the buffer increments a counter attached to that buffer. Finally, (5) a table stores a list of operations along with the counter thresholds at which point they are triggered. If the counter is greater than or equal to a given operation’s threshold, the
operation is triggered, and an outgoing message generated.

![Figure 4.2: Program execution under INCA.](image)

The basic insight behind INCA is that the three primitive operations of triggering messages, demultiplexing incoming messages by matching tags, and applying atomic operations, when taken together, can be interpreted as executing an instruction. Figure 4.2 illustrates how this is the case. First, a triggered operation generates a message, perhaps containing a payload. Second, the message matches against a PTE/ME. When the message is matched, we have (i) an operand, i.e., the contents of the buffer specified by the ME; (ii) an (optional) second operand, contained in the payload of the triggered message, and (iii) an operation to apply to one or both those payloads, which (in a departure from current Portals semantics) we assume is also specified by the ME. The atomic unit then executes the operation, writing the results to the buffer indicated by the ME. In this way, the sequence of generating a message, matching, and performing an atomic operation can be viewed as the execution of an instruction.

The counter provides the means to sequence instructions into programs by serving as an analog to a traditional program counter. Let a sequence of instructions be ordered by the triggering threshold of each, and suppose all of these instructions share the same counter. Then, as each instruction executes, its completion triggers the next instruction in the program.

To sum, the basic strategy behind INCA is to treat triples of triggered messages, matching elements, and atomic operations, as instructions. By sharing a common counter, these instructions can be sequenced to execute a program.
4.2 The triggered operation machine

In this section we present a formal model capturing the intuitive notion of program execution given above: the Triggered Operation Machine (TOM). The TOM stands to INCA as, e.g., the traditional register machine stands to typical RISC-based program execution. Specifically, first, it demonstrates that when properly organized, these primitives are indeed capable of general-purpose computing (i.e., are Turing complete); and second, by specifying the basic capabilities required of any practical realization, the model serves as a guide for what is required of any hardware implementation (Section 4.4).

Formally, a TOM comprises two components. First, a finite set of atomic unit (AU) entries. Second, a finite set of triggering unit (TU) entries. These sets are then associated through messages (which link TU entries to AU entries) and triggers (which link AU entries to TU entries). Figure 4.3 shows the model. An incoming message \( m_j \) is interpreted (‘matched’) by the Atomic Unit, and an atomic operation applied. This event updates the contents of a trigger, which, when interpreted by the Triggering Unit, causes a new message \( m_k \) to be generated.

![Message processing flow for the triggered operation machine.](image)

Figure 4.3: Message processing flow for the triggered operation machine. An incoming message \( m_j \) carrying an optional operand is interpreted (‘matched’) by the Atomic Unit, and an atomic operation applied. This event updates the contents of a trigger, which, when interpreted by the Triggering Unit, causes a new message \( m_k \) to be generated.

An incoming message \( m_j \) is a pair of an operand \( o_j \) and a tag \( \mu_j \). Each member of the AU set is a 4-tuple comprising a tag \( \mu_i \), an operand \( o_i \), an atomic operation \( \alpha_i \), and a trigger \( R_i \). Finally, each member of the TU set is also a 4-tuple, each comprising a trigger \( R_k \), a threshold \( \theta_k \), an operand \( o_k \), and a tag \( \mu_k \). This notation is
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_i$</td>
<td>Operand location</td>
</tr>
<tr>
<td>$O$</td>
<td>Set of operand locations $o_0, o_1, \ldots$</td>
</tr>
<tr>
<td>$\langle o_i \rangle$</td>
<td>Contents of operand $o$</td>
</tr>
<tr>
<td>$\mu_i$</td>
<td>Tag</td>
</tr>
<tr>
<td>$m_j = (o_j, \mu_j)$</td>
<td>Message</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Atomic operation (unary or binary)</td>
</tr>
<tr>
<td>$R_i$</td>
<td>Trigger</td>
</tr>
<tr>
<td>$a_i = (o_i, \mu_i, \alpha, R_i)$</td>
<td>Atomic unit (AU) tuple</td>
</tr>
<tr>
<td>$A$</td>
<td>Set of AU tuples</td>
</tr>
<tr>
<td>$\theta_k$</td>
<td>Trigger threshold</td>
</tr>
<tr>
<td>$t_k = (\theta_k, R_k, o_k, \mu_k)$</td>
<td>Triggering unit (TU) tuple</td>
</tr>
<tr>
<td>$T$</td>
<td>Set of TU tuples</td>
</tr>
</tbody>
</table>

Table 4.1: Triggered operation machine (TOM) notation.

summarized in Table 4.1.

In this notation, operands and triggers are locations, and angled brackets designate the contents of those locations. Let the set of tuples of the AU be $A$. Given an incoming message $m_j = (o_j, \mu_j)$, the operation of the AU can be represented as updating the state of all operands $o_i$ referenced by the members of $A$:

$$\forall a_i \in A : \langle o_i \rangle \leftarrow \begin{cases} 
\alpha(\langle o_i \rangle, \langle o_j \rangle) & \mu_i = \mu_j \\
\langle o_i \rangle & \text{otherwise}
\end{cases} \quad (4.1)$$

Unary operations are accommodated by operating only on $o_i$, ignoring $o_j$. Note that ‘matching’ on a tag or set of match bits (e.g., on a PTE and ME in Portals as in Figure 4.1) is integrated into this piecewise function as the condition $\mu_i = \mu_j$.

The contents of the trigger specified in the AU tuple are updated according to some function; since the present goal is to have a trigger emulate a program counter that is updated after each instruction, we use the successor function to count the number of
atomics initiated using the same trigger, where \( R \) is the set of triggers:

\[
\forall R_i \in R : \langle R_i \rangle \leftarrow \begin{cases} 
\langle R_i \rangle + 1 & \mu_i = \mu_j \\
\langle R_i \rangle & \text{otherwise}
\end{cases} \tag{4.2}
\]

Finally, letting \( T \) be the set of 4-tuples of the TU, the operation of the TU can then be represented as updating the state of \( \text{OUT} \) based on the status of the triggers identified by each tuple:

\[
\forall t_k \in T : \langle \text{OUT} \rangle \leftarrow \begin{cases} 
(o_k, \mu_k) & \theta_k = \langle R_k \rangle \\
\epsilon & \text{otherwise}
\end{cases} \tag{4.3}
\]

Note that these equations are slightly underconstrained; this is a deliberate attempt to anticipate future variations on the model. For present purposes, let us further constrain the definition of a TOM as follows. First, all tags \( \mu_i \) in the members of the AU set are unique (i.e., multiple atomic operations cannot be induced by the same incoming message). Second, the set of triggers \( R \) is unary, containing only a single trigger, the ‘program counter’. Third (as is implied by the previous restriction), all members of the AU set specify the sole member of \( R \). Finally, all thresholds \( \theta_k \) in the members of the TU set are also unique (i.e., multiple messages cannot be generated off of the same trigger update).

Given these definitions, a token TOM is a set of AU tuples, a set of TU tuples, and a rule for updating the sole trigger (namely, succession).

### 4.3 The Turing completeness of TOM

The definition of the TOM model given above does not specify the set of available atomic operations. Here, we establish the Turing completeness of the TOM model, and in doing so, simultaneously determine a sufficient set of atomics. Moreover, as discussed in Section 4.4, the proof has implications for how current hardware can be extended to
secure arbitrary program execution.

Our demonstration proceeds by reducing the well-known universal register machine (URM) model to TOM [173]. A simple URM is defined as follows. First, let \( R = \{ r_0, r_1, \ldots \} \) be an unbounded set of registers, each capable of storing a member of \( \mathbb{N}_0 \). Second, the behavior of the machine is determined by a finite list of indexed instructions \( I = \{ i_1, i_2, \ldots, i_k \} \), each of which is one of the following four possibilities, where \( r_i, r_j \in R \):

1. Zero: \( Z(r_i): \langle r_i \rangle \leftarrow 0 \).
2. Successor: \( S(r_i): \langle r_i \rangle \leftarrow \langle r_i \rangle + 1 \).
3. Transfer: \( T(r_i, r_j): \langle r_i \rangle \leftarrow \langle r_j \rangle \).
4. Jump: \( J(r_i, r_j, q) \), where \( i_q \in I \): letting \( p \) be the index of the current instruction, if \( \langle r_i \rangle = \langle r_j \rangle \), jump to instruction \( i_q \); otherwise, proceed to instruction \( i_{p+1} \).

For all instructions but jump, the instruction index is incremented by one after the instruction is executed. The machine begins at instruction \( i_1 \), and continues until there is no instruction with the current index to execute. The result of the computation is the contents of the registers.

URMs are reduced to the TOM as follows. First, for each URM register \( r_0, r_1, \ldots \) designate a corresponding TOM operand location \( o_0, o_1, \ldots \). Second, let all AU and TU tuples utilize the same trigger, \( R_{pc} \). \( \langle R_{pc} \rangle \) can be viewed as a program counter storing the index of the current URM instruction in the list of instructions comprising a URM program. Third, each TOM operation has two parts: an entry in the TU and a matching entry in the AU (connected by the message generated by the TU). A TOM instruction \( i_j \) is therefore a pair \(( t_k, a_i \) = \(( (\theta_k, R_{pc}, o_k, \mu_k), (o_i, \mu_i, \alpha, R_{pc}) \)). This can be simplified as follows. First, since the tag for a message cannot vary between the AU tuple and the TU tuple, \( \mu_i = \mu_k \), so can be dropped. Second, since all instructions use the same trigger, it is also omitted. Finally, rearranging terms for clarity and dropping superfluous parentheses gives a simplified definition: \( i_j = (\theta_k, o_i, o_k, \alpha) \). Fourth, a TOM program is
defined as a list of \( n \geq 1 \) TOM instructions \( i_1, i_2, \ldots, i_n \) beginning with \( \theta_k = 1 \), and concluding with \( \theta_k = n \). Since under this definition, \( \langle \theta_k \rangle = j \), the TU threshold can be omitted, and instruction \( j \) of a TOM program – corresponding to instruction \( j \) of a URM program – is represented simply as: \( i_j = (o_i, o_k, \alpha) \).

The remainder of the reduction involves the selection of an appropriate set of primitive operations for possible values of \( \alpha \). Recall incoming messages are \texttt{PUT}s from location \( o_j \) to location \( o_i \), where the contents of the latter are replaced with the contents of some atomic operation performed on one or both operands. Consequently, if \( \alpha \) is identity, the result is merely \( \langle o_i \rangle \leftarrow \langle o_j \rangle \). Therefore, for each URM instruction of the form \( Z(r_i) \), let the corresponding TOM instruction be \( (o_i, 0, =) \); and for each URM instruction of the form \( T(r_i, r_k) \), let the corresponding TOM instruction be \( (o_i, o_k, =) \).

Addition is a standard atomic operation in current hardware, so the URM successor function can be simulated by letting \( \alpha \) be addition, and setting \( \langle o_k \rangle = 1 \). Each URM instruction of the form \( S(r_i) \) thus has a corresponding TOM instruction \( (o_i, 1, +) \).

To reduce the URM jump instruction, first note that compare-and-swap is a typical atomic operation, so testing whether the contents of two operand locations are equal is a reasonable atomic operation for the TOM. Consequently, the equality test of a URM jump can be simulated by a TOM instruction of the form \( (o_i, o_k, ==) \), where \( \langle o_i \rangle == \langle o_k \rangle \) is defined as \( \langle o_i \rangle \leftarrow \begin{cases} 1 & \langle o_i \rangle = \langle o_k \rangle \\ 0 & \text{otherwise} \end{cases} \). Second, since triggers are ‘registers’ in the same sense as operand locations, we define a special atomic operation that takes advantage of this fact: \([o_i, q, > 0]\), interpreted as follows:

\[
\langle R_{pc} \rangle \leftarrow \begin{cases} q & \langle o_i \rangle > 0 \\ \langle R_{pc} \rangle + 1 & \text{otherwise} \end{cases}
\]  

(4.4)

Note that since each URM jump results in two TOM instructions, the values of \( q \) and \( j \) for all instructions must be adjusted appropriately. To simplify exposition, we exclude
Table 4.2: Summary of the reduction of URMs to TOM.

<table>
<thead>
<tr>
<th>URM instruction</th>
<th>⇒</th>
<th>TOM instruction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero $Z(r_i)$</td>
<td>⇒</td>
<td>$(o_i, 0, =)$</td>
</tr>
<tr>
<td>Successor $S(r_i)$</td>
<td>⇒</td>
<td>$(o_i, 1, +)$</td>
</tr>
<tr>
<td>Transfer $T(r_i, r_k)$</td>
<td>⇒</td>
<td>$(o_i, o_k, =)$</td>
</tr>
<tr>
<td>Jump $J(r_i, r_k, q)$</td>
<td>⇒</td>
<td>$(o_i, o_k, ==)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[o_i, q, &gt; 0]$</td>
</tr>
</tbody>
</table>

these details.

Table 4.2 summarizes the reduction of URMs to TOMs. This reduction tells us the TOM model can be made Turing complete if, at the minimum, it supplies atomic operations for identity (i.e, PUT), addition, and equivalence, as well as the capacity to conditionally modify the contents of a location (the trigger) depending on whether the contents of another are greater than zero. In the next section, we use these requirements to guide the construction of INCA, a concrete implementation of the TOM model within the Portals networking API.

### 4.4 Implementing INCA

INCA is an implementation of the TOM model on top of the Portals network programming API. In other words, INCA shows how, with relatively modest modifications to existing offloaded capabilities, current Portals-compliant network interfaces can provide general-purpose compute capabilities of the sort offered by existing on- and off-path SmartNIC architectures.

Portals-compliant NICs offer message matching, atomic operations, and triggered operation functionality (Section 4.1). Consequently, much of the TOM model maps straightforwardly onto Portals. A TOM instruction corresponds to a pair of a Portals triggered operation ($\text{PtlTriggeredPut}$) and an ME, linked by unique match bits. All
instructions (i.e., all MEs and triggered operations) use the same trigger, i.e., a program counter. One TOM operand \( o_i \) is specified as the ME buffer, and the other \( o_k \) is given by the message generated by the triggered operation. TOM transfers and the zero operation are thus standard Portals PUT operations.

In a slight modification to current Portals semantics (which has the sender of a message specify an optional atomic operation \( \alpha \) to be performed at the receiver), we stipulate that the ME can specify the atomic. The reason for this change is to maintain consistency with how program execution is typically understood, namely, the local PEs store and execute the program, not remote hosts.

The atomics included as part of the Portals standard include addition, so it is trivial to simulate the URM successor function; however, rather than limit ourselves to URM operations, our implementation allows \( \alpha \) to be any of the atomics provided by the Portals specification. In addition to arithmetic operations (addition, subtraction, multiplication), these include bitwise, logical, and comparison operators.

Implementing the remainder of the TOM model on top of Portals can be accomplished with three additional modifications to the Portals specification. First, under the current specification, a Portals triggered operation is discharged when its threshold is \( \leq \) the counter value. This is problematic from the perspective of program execution, because if these entries are persistent (i.e., they do not disappear when triggered), incrementing the program counter would result in all prior instructions being ‘re-executed’ each time the counter is incremented. Therefore, we modify this behavior to allow ‘strict’ indexing (i.e., \( == \)), as required by Equation 4.3 of the TOM model.

Second, while the current Portals standard allows MEs to be used multiple times, triggered operations are by default ‘use once’, and are removed from the set of registered operations when discharged. Since TOM supports backwards jumps, we extend Portals to include persistent triggered operations (Portals already allows for persistent MEs).

Finally, to realize jumps, the TOM requires a method for testing for equality, and
modifying a counter value conditional on the outcome of this test (Equation 4.4). While Portals provides an atomic operation for testing equality, it does not support the required counter modification. To address this, we introduce a new atomic operation, ‘branch if \( \leq \) zero’ (BLEZ). An ME specifying this atomic also provides an instruction index. If the value contained in the operand identified by the ME is zero or less, the counter is updated to this instruction index; otherwise, it is incremented by one. Obviously, any equivalent branching atomic could be substituted for BLEZ.

To sum, the majority of the TOM model can be realized within Portals with no modification to the current specification. The remaining proposed changes represent a relatively small delta from that specification: adding strict indexing for triggered operations, making those operations persistent, and adding a single new atomic operation. As a proof of concept, we’ve modified the current Portals reference implementation [174] to include these changes, although at the time of writing, this modified version is not yet publicly available.

### 4.5 Conclusion

In this chapter, we have shown (via the Triggered Operation Machine model) that when effectively coordinated, standard task-specific offloaded capabilities – namely, message demultiplexing, atomic operations, and triggered operations – are capable of enabling Turing complete, deadline-free, on-NIC program execution. That is, many standard HPC NICs already provide the basic building blocks required to be ‘smart’ in the contemporary sense of a SmartNIC, without the need for additional processors.

Moreover, the demonstration given in this chapter serves to identify a set of modest changes to the current Portals standard sufficient for securing Turing completeness. These changes include the strict indexing of triggered operations, persistent triggered operations, and a single additional atomic operation whose result is written to a counter.

The demonstration offered in this chapter introduces the core concepts underwriting
In-Network Compute Assistance. In the next chapter, we build on this foundation to design and implement an ‘ecosystem’ for exploiting the expanded smart capabilities of an INCA-enabled NIC.
Chapter 5

The INCA Ecosystem

The triggered operation machine provides a formal foundation for INCA, and, with modest modifications, Portals-compliant NICs can implement this model (Chapter 4). However, just as ‘programming’ a Turing machine or working directly in assembly language can be difficult, so can working at the level of the TOM. Indeed, programming with triggered operations can itself be counterintuitive, and require significant specialized knowledge regarding the underlying hardware.

To facilitate the development of INCA kernels, we designed and implemented an ‘ecosystem’ comprising various languages for expressing these kernels, a compiler, and an interpreter. In this chapter we provide a broad overview of these components; more details can be found in Appendices A and B.

5.1 The INCA ecosystem

Figure 5.1 is an overview of the INCA ecosystem. INCA-Q is a high-level language based on C that provides a familiar means to express INCA kernels. The ‘Q-compiler’ compiles INCA-Q code into a lower-level INCA ‘assembly’ language – INCA-A – that directly mirrors the structure of the TOM. INCA-A code, in turn, can be executed through an interpreter. INCA-A code can also be executed through a modified version of the Portals reference implementation; however, we do not discuss the reference implementation in
In the remainder of this chapter, we take a closer look at these components.

5.2 INCA-Q

INCA-Q is a high-level language for expressing INCA kernels. As noted in the previous chapter, registering an INCA program with the NIC involves setting up a collection of Portals matching elements (each of which specify buffers), atomic operations, and triggered operations that generate messages containing payloads. INCA kernels then execute using the resources specified in this ‘program’.

Since INCA does not itself allocate memory, the INCA-Q language divides programs into two parts. The first is a list of ‘directives’ establishing the execution environment, i.e., the available buffers and variables, their types, and (if needed), their values. The second part is the code that executes in that environment, i.e., the INCA kernel itself.

For example, Algorithm 1 is INCA-Q code for performing dot product on two 10-element vectors. Lines 1-4 are prefixed with ‘%’, and define the execution environment. For example, i is a 16-bit integer, c a float, and the two arrays are arrays of floats. These directives do not specify what the contents of these variables and arrays are. Lines 5-11 comprise the INCA kernel itself, defining the operations to be performed within the
Algorithm 1 INCA-Q Dot Product

1: % i16 i
2: % f c
3: % f A[10]
4: % f B[10]
5: i = 0
6: c = 0.0
7: while i < 10 do {
8:   c = c + (A[i] * B[i])
9:   i = i + 1
10: }
11: end

environment defined by the directives. Lines 5 and 6 initialize two of the variables, and the loop calculates the dot product. At the end of execution, c contains the result.

The full INCA-Q grammar is given in Appendix B. INCA-Q supports while loops and conditionals. At present, it does not include the capability to define function calls; all code is inline.

5.3 INCA-A

INCA-A is a low-level ‘assembly’ language for expressing INCA kernels. Like INCA-Q, INCA-A does not allocate memory, so code is divided into two sections, the directives defining the execution environment, and the code specifying the algorithm to execute within that environment. Moreover, while INCA-Q allows the use of literals, all values in INCA-A are stored in locations in memory; therefore, INCA-A must define, as part of its directives, locations containing any literals used in INCA-Q, designated with an _C prefix.

Finally, note that atomic operations as defined in the Portals specification are clobbering in the sense that the value of one of the operands in the operation will be overwritten with the result. Consequently, INCA-A code may contain ‘registers’, designated with a _R prefix, that specify locations used to temporarily store values to avoid clobbering.
Algorithm 2 INCA-A Dot Product

1: d i16 _C0
2: d _C0 = 0
3: d i16 _C1
4: d _C1 = 10
5: d i16 _C2
6: d _C2 = 1
7: d f A[10]
8: d f B[10]
9: d f c
10: d i16 i
11: 1 PUTL i, _C0, i16
12: 2 PUTL _R0, i, i16
13: 3 LT _R0, _R0, _C1, i16
14: 4 BLEZ _R0, 10
15: 5 PUTL _R1, A[i], f
16: 6 MUL _R1, _R1, B[i], f
17: 7 ADD c, c, _R1, f
18: 8 ADD i, i, _C2, i16
19: 9 JMP 2
20: 10 END

Algorithm 2 is INCA-A code that corresponds to the INCA-Q code given in Algorithm 1, and illustrates the points just raised. Lines 1-9 are directives defining the execution environment for the code specified on lines 10-20. The INCA-Q kernel appeals to three literals – 0, 10, and 1 – so the INCA-A directives declare and initialize three constants, _C0, _C1, and _C2. The INCA-A kernel proper appears on lines 11-20.

To illustrate the syntax of an INCA-A instruction, consider the multiplication (MUL) on line 16. From left to right, the first item (6) is the threshold for triggering the instruction. The second item is the instruction (MUL) to be executed, followed by the destination where the result of the operation should be placed (_R1), the location containing the first operand (also _R1), and the location containing the second operand (B[i]). The final item is the type, in this case float.

Because the destination location and the location containing the first operand are the same, the first operand is clobbered, i.e., this example illustrates the clobbering
semantics of Portals atomic operations. Note that all of the binary operations appearing in the kernel share this feature. To avoid losing the overwritten data, the INCA-A code introduces temporary storage locations (‘registers’), e.g., _R0 and _R1, and data to be preserved is first copied to one of these locations before it is operated on. For example, line 15 copies an element of array A to register _R1, so that the multiplication performed on line 16 does not destroy the data. In some cases, clobbering is permissible, e.g., in accumulating the results of the ongoing multiplication on line 17.

We note that an alternative, non-clobbering semantics for INCA instructions presents an obvious opportunity for optimization: if instructions allowed the destination of an operation to be a location besides the first operand, the additional memory would not be necessary, and the copying of data could be avoided. While all of the work presented in this document retains the standard clobbering semantics, INCA-A syntax is designed to allow for such a distinction in the future.

The current version of INCA-A includes directives for defining the execution environment (declarations, initializations), core operations for data manipulation (arithmetic, logical, bitwise), program control operations (branch, jump, end), and data movement operations (put). A list of INCA-A instructions is included in Appendix A.

5.4 Compiler and interpreter

To translate kernels written in INCA-Q to INCA-A, we designed and implemented a ‘Q-compiler’. The Q-compiler is written in Python using the Lark library [175], which handles the tasks of tokenization and parsing.

The compiler makes an initial pass over the INCA-Q code to assemble tables for variable types (for type checking), and for arrays and their dimensions (for bounds checking). Next, an AST is generated for the program, and traversed to do type and bounds checking, and to generate INCA code. During code generation, the compiler generates locations to store any literals, as described in the previous section. Furthermore, the
compiler automatically generates temporary registers and code for copying data to and from those locations, to address the clobbering issue discussed above.

At the time of writing, the Q-compiler is optimized in that effort has been made to avoid superfluous data movement. For example, in the expression $x = x + y$, the Q-compiler recognizes the final destination of the sum is $x$, so there is no need to avoid clobbering $x$ (compare: $z = x + y$). Similarly, the compiler recognizes statements such as $x = x$ do not require data movement.

In addition to the compiler, we also developed two methods for executing INCA-A kernels. First, we modified the Portals reference implementation to support INCA. As noted in Chapter 4, this required extending Portals to support strict indexing, persistent triggered operations, and an additional atomic to enable program branching. Since the Portals reference implementation is written in C, we also developed a tool to translate INCA-A kernels into C code that uses the extended version of the Portals API. The resulting code can be compiled and executed over an InfiniBand or Ethernet network.

The second method is an INCA-A interpreter. This interpreter is implemented in Python, again with assistance from the Lark library to lex and parse the INCA-A code. The resulting collection of ASTs is traversed to generate an representation of the initial state of the variables at the start of program execution. The ASTs are then traversed again, and the operations executed in the context of the interpreter. The interpreter reports the initial state of all variables, and the final state of those variables. Furthermore, the interpreter records the INCA-A instructions appearing in the kernel as well as how many times each is executed; this information is then used for performance modeling, as described in Chapter 6.

5.5 Conclusion

In Chapter 4, we demonstrated how traditional, task-specific smart capabilities can be leveraged to enable a Portals-compliant NIC to be intelligent in the contemporary sense.
of the term. In this chapter, we’ve provided a brief overview of a set of tools – languages, a compiler, and an interpreter – designed to facilitate the deployment of these general-purpose compute capabilities.

The tools in the INCA ecosystem include high- and low-level languages for expressing INCA kernels, a compiler, and an interpreter. An example kernel written in the high-level language (INCA-Q) was discussed, as was a corresponding kernel written in the low-level language (INCA-A). In the process, we highlighted some potential opportunities for optimization, e.g., modifying Portals semantics to allow results of an operation to be placed in a location distinct from that of the operands. We also briefly described the Q-compiler for compiling INCA-Q code to INCA-A code, and the methods developed to execute INCA-A code.

In the following chapters, we use this ecosystem to develop a variety of INCA kernels. By supplementing the INCA-A interpreter with a simulator, we provide runtime estimates for these kernels (Chapter 6). Moreover, we investigate the impact such offloaded kernels may have for host application performance (Chapter 7), and explore how INCA affords the offloading of entirely independent applications to the network, e.g., for purposes of predicting network traffic (Chapter 8).
Chapter 6

INCA: Kernel Performance

Traditional offloaded core network applications – message demultiplexing, atomic operations, and triggered operations – can be leveraged to provide Turing complete and deadline-free on-NIC compute capabilities (Chapter 4). In this chapter, we utilize the INCA ecosystem tools described in Chapter 5 to refine our understanding of what an INCA-enabled SmartNIC can do by implementing a selection of representative kernels, and assessing their runtimes under different configurations of network speeds, memory access times, and other hardware optimizations. To achieve this evaluation, we designed and implemented a simulator – INCAsim – that interfaces with the INCA-A interpreter described in the previous chapter.

The results of this simulation study show that ‘vanilla’ INCA – i.e., the result of making the bare minimum modifications to contemporary NICs to make them INCA-compliant – is, not surprisingly, slow relative to standard CPUs. However, by taking advantage of currently-unused on-NIC silicon area to include hardware optimizations (e.g., SIMD units), and through intelligent data staging, the execution speeds can be made comparable to CPUs, and in certain cases, exceed them.

In Section 6.1 we describe the simulator used to derive performance estimates. In Section 6.2 we introduce the set of representative kernels selected for consideration, as well as the various INCAsim configurations used for evaluation. The results of the performance
evaluation are discussed in Section 6.3.

6.1 INCASim

INCASim is based on the LogGP model of parallel computation [176], which itself is an
extension of the original LogP model [177]. Under LogGP, communication performance
is modeled by five parameters:

- $L$: message latency;
- $o$: message processing overhead;
- $g$: the inter-message gap, i.e., the inverse of bandwidth;
- $G$: the inter-byte gap, i.e., the inverse of bandwidth calculated at the byte level;
- $P$: the number of processes.

The total time to send a small message is estimated as the sum of latency and send-
and receive-side overheads: $L + 2o$. If $m$ messages are sent, then the total time spent in
communication is $m(L + 2o) + (m - 1)g$. The inter-byte gap $G$ is used to characterize
communication time for larger messages; the time to send a large message of $b$ bytes is
$L + 2o + (b - 1)G$.

Because the phases in executing INCA instructions are precisely those involved in
communication, the LogGP model maps naturally to INCA program execution. Fig-
ure 6.1 shows this relationship by annotating the original INCA instruction execution
illustration given in Chapter 4 (Figure 4.2) with the parameters of the LogGP model.
Latency $L$ is the time it takes the message generated by the triggering unit to travel
to the matching unit. Overhead $o$ is the time spent matching, performing the atomic,
and triggering any further message. And the gap ($g$ or $G$) characterizes how quickly the
messages (or bytes) can leave the triggering unit. Note that $o$ is bounded above by $g$,
Figure 6.1: The extended LogGP model used by INCAsim.

because the NIC hardware is itself designed to operate at line rate, i.e., it should take no more time than $g$ to match, apply an atomic, and trigger a new message.

To capture interactions between the execution pipeline and memory, we extend the LogGP model to include a parameter for memory access, $m$. In principle, an INCA instruction could involve zero, one, or two fetches per instruction depending on the instruction. Figure 6.1 illustrates a two-fetch scenario. Suppose the instruction being executed is addition. We assume one operand is contained in the incoming message, so no fetch is required. Suppose, however, the second operand, as specified by the ME, is not present in cache; the result is a fetch prior to addition being applied. Furthermore, if the data to be sent in a triggered message is not already present – e.g., if the outgoing data is not the result of the operation or one of the two operands – then the triggering unit must also fetch. We assume writing results out to local memory or to host through the DMA engine can proceed independently of program execution. Therefore, in the example illustrated in Figure 6.1, the time to execute the instruction is $L + 2o + 2m + g$, and an upper bound on the time to execute a program of $m$ instructions is $m(L + 2o + 2m) + (m - 1)g$.

The INCA-A interpreter records which instructions are executed, and how many times each is executed. The INCAsim simulator takes as inputs (1) a configuration file specifying network speed (GB/s) or message rate (messages/s), PCIe configuration or
local memory access speed, cable lengths, switch latencies, etc., and (2) the instruction counts provided by the interpreter, and applies the extended LogGP model to generate an estimate of total kernel runtimes.

This model is deployed conservatively by INCAsim. For example, we assume that the triggered unit must always perform a fetch. Likewise, we assume there is no pre-fetching of operands specified by MEs registered in the matching unit. Consequently, every binary operation (for instance) always involves two fetches. This situation represents a clear opportunity for future optimization: since MEs are associated with instructions, and instructions are ordered, at sequences of operands could be pre-fetched into the atomic unit concurrently with the actions of other units.

6.2 Kernels and INCAsim configurations

In this section we describe the set of representative kernels selected for performance evaluation, and the set of INCAsim configurations used to generate runtime estimates.

6.2.1 Kernels

The kernels we study are listed in Table 6.1 in order of complexity. Given an incoming array of data, the filter kernel replaces each item in host memory if the new value is within a user-defined neighborhood surrounding the current value; incoming data outside the neighborhood is considered invalid. The matrix-unpack kernel unpacks an incoming array containing data from an eastern neighbor participating in a three-dimensional halo exchange. The linear-interpolation kernel performs a linear interpolation on an incoming array of data, interpolating two intermediate points between each adjacent pair of points in the incoming data. The convolution kernel applies a standard 3x3 edge detection filter to an incoming matrix, with wraparound for edge pixels. hadamard-product is the entrywise product of two matrices. The remaining kernels, vector-dot-product,
<table>
<thead>
<tr>
<th>Kernel</th>
<th>Payload size</th>
<th>128B</th>
<th>256B</th>
<th>512B</th>
<th>1024B</th>
<th>2048B</th>
<th>4096B</th>
<th>8192B</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector-dot-product</td>
<td></td>
<td>132</td>
<td>260</td>
<td>516</td>
<td>1028</td>
<td>2052</td>
<td>4100</td>
<td>8196</td>
</tr>
<tr>
<td>matrix-transpose</td>
<td></td>
<td>136</td>
<td>268</td>
<td>460</td>
<td>916</td>
<td>1684</td>
<td>3364</td>
<td>6436</td>
</tr>
<tr>
<td>hadamard-product</td>
<td></td>
<td>168</td>
<td>332</td>
<td>588</td>
<td>1172</td>
<td>2196</td>
<td>4388</td>
<td>8484</td>
</tr>
<tr>
<td>filter</td>
<td></td>
<td>208</td>
<td>406</td>
<td>808</td>
<td>1647</td>
<td>3271</td>
<td>6555</td>
<td>13112</td>
</tr>
<tr>
<td>matrix-unpack</td>
<td></td>
<td>246</td>
<td>486</td>
<td>966</td>
<td>1926</td>
<td>3846</td>
<td>7686</td>
<td>15366</td>
</tr>
<tr>
<td>matrix-multiplication</td>
<td></td>
<td>696</td>
<td>2700</td>
<td>4748</td>
<td>18836</td>
<td>35220</td>
<td>140580</td>
<td>271652</td>
</tr>
<tr>
<td>convolution</td>
<td></td>
<td>808</td>
<td>1616</td>
<td>3088</td>
<td>6176</td>
<td>12064</td>
<td>24128</td>
<td>47680</td>
</tr>
<tr>
<td>linear-interpolation</td>
<td></td>
<td>826</td>
<td>1690</td>
<td>3418</td>
<td>6874</td>
<td>13786</td>
<td>27610</td>
<td>55258</td>
</tr>
</tbody>
</table>

Table 6.1: Number of instructions executed for each kernel by payload size. Shaded cells indicate instruction counts that exceed the deadline-imposed limit on a 32 core, 2.5GHz, 200Gb/s, 128B packet deadline-based SmartNIC.

matrix-transpose and matrix-multiplication, are self-explanatory.

Table 6.1 also reports instruction counts, as reported by the INCA-A interpreter, for payloads ranging from 128 to 8192 bytes. The grey-shaded entries indicate instruction counts that exceed the processing deadline for on-path architectures on a 32 core, 2.5GHz, 200Gb/s, 128B packet system; most of these kernels cannot be executed on such a system.

### 6.2.2 INCAsim configurations

For the studies presented here, we consider a variety of hardware configurations, beginning with a ‘baseline’ configuration, ‘base’. The baseline configuration represents the smallest delta from existing architectures; i.e., it reflects a scenario where current state-of-the-art Portals NIC hardware is modified to achieve Turing completeness, as detailed above, with no further optimizations.

Figure 6.2 depicts the architecture of a Portals-compliant NIC, modified to support the base INCA configuration. The Portals Unit (center of diagram, shaded) contains the matching, atomic, and trigger units comprising the foundation of INCA program
In accordance with the results given in Chapter 4, the trigger unit now includes strict indexing, MEs and triggered operations are persistent, and the atomic unit provides the capacity to modify the contents of a trigger if those of another buffer are $\leq 0$. Otherwise, the base configuration works with the capabilities already supplied by the NIC. This has two implications for execution speeds. First, triggered messages must bounce off or loopback from the local switch, a process that increases latency ($L$ in the extended LogGP model). Second, because there is no NIC-local memory dedicated to INCA kernel execution, all operands must be fetched from main memory via PCIe, and memory access times ($m$ in the extended LogGP model) will therefore be significant. Note we assume access to main memory leverages the Tx DMA engine.

The second model configuration, ‘scratchpad’, accounts for several optimizations, diagrammed in Figure 6.3. First, a fast path is available for message loopback, eliminating the trip through the switch. This fast path includes a low-priority queue for the pending operation, allowing incoming data to take precedence over INCA programs. Second, we assume the Portals system is extended with a scratchpad memory similar to fast cache (SRAM) as employed in contemporary stream models [52], [54]. This memory provides
a low-latency, NIC-local space for caching current data, and operands specified by an INCA program can be pre-loaded to this local memory, e.g., by traversing the matching list.

In both configurations, Portals overhead and gap are bounded by network speed, because a Portals-enabled NIC is expected to process incoming requests and issue outgoing data at least at message rate [102]. Figure 6.4 shows the ideal processing speeds at network bandwidths targeted by the InfiniBand roadmap [4]. The three message sizes
correspond to 8 B operands, 44 B messages containing the minimum Portals header information (according to the current Portals reference implementation) and an operand, and the common 64 B cache line. Message rates reported by Intel (100 Gb/s OmniPath) and Mellanox (100 Gb/s EDR and 200 Gb/s HDR) are shown in colored triangles. For current hardware, Portals overhead can therefore be expected to fall somewhere between 5 ns (gap for 200 million messages/s) and 1.76 ns (gap for 44 B messages). For the present evaluation, we use the conservative larger overhead value.

Configuration parameters are summarized in Table 6.2. Loopback latency for base reflects the time to traverse the local switch, and Portals overhead is assessed as just described. Memory latency for base estimates the cost of retrieving an operand via generation 5 x32 PCIe [178], while scratchpad memory latency is based on the value adopted in [54].

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Scratchpad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loopback latency</td>
<td>50 ns</td>
<td>0</td>
</tr>
<tr>
<td>Portals overhead</td>
<td>5 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>Memory latency</td>
<td>250 ns</td>
<td>1 ns</td>
</tr>
<tr>
<td>gap</td>
<td>5 ns</td>
<td>5 ns</td>
</tr>
</tbody>
</table>

Table 6.2: Parameters used for INCAsim.

### 6.3 Performance Evaluation

Using INCA sim with the parameters described in the previous section, we first calculated INCA runtimes for the base and the scratchpad configurations. We then calculated runtimes for a series of optimizations. These optimizations are presented incrementally so that benefits of each can be assessed independently, e.g., as regards potential implementation costs. Results for all configurations and optimizations are collected in Table 6.3.
<table>
<thead>
<tr>
<th>Kernel</th>
<th>Optimization</th>
<th>Payload size</th>
<th>Avg spdup wrt base</th>
<th>Avg spdup wrt spad</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-trans</td>
<td>base</td>
<td>42.16</td>
<td>283.96</td>
<td>1042.84</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>1.52</td>
<td>522.04</td>
<td>37.58</td>
</tr>
<tr>
<td>filter</td>
<td>base</td>
<td>65.98</td>
<td>525.57</td>
<td>1046.01</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>2.34</td>
<td>18.81</td>
<td>3022.91</td>
</tr>
<tr>
<td>m-unpack</td>
<td>base</td>
<td>96.51</td>
<td>757.31</td>
<td>87.02</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>2.80</td>
<td>21.96</td>
<td>6043.71</td>
</tr>
<tr>
<td>convolution</td>
<td>base</td>
<td>328.48</td>
<td>7501.84</td>
<td>30.30</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>9.26</td>
<td>70.91</td>
<td>548.09</td>
</tr>
<tr>
<td>lin-inter</td>
<td>base</td>
<td>301.31</td>
<td>12512.19</td>
<td>627.28</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>9.38</td>
<td>78.03</td>
<td>7208.41</td>
</tr>
<tr>
<td>hadamard</td>
<td>base</td>
<td>56.08</td>
<td>47.48</td>
<td>3488.28</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>1.89</td>
<td>24.79</td>
<td>95.44</td>
</tr>
<tr>
<td></td>
<td>clobber</td>
<td>1.54</td>
<td>19.07</td>
<td>72.91</td>
</tr>
<tr>
<td></td>
<td>parallel</td>
<td>0.02</td>
<td>0.16</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>para-clobber</td>
<td>0.01</td>
<td>0.05</td>
<td>0.18</td>
</tr>
<tr>
<td>dot-prod</td>
<td>base</td>
<td>48.92</td>
<td>4863.16</td>
<td>3052.76</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>1.50</td>
<td>213.88</td>
<td>93.23</td>
</tr>
<tr>
<td></td>
<td>clobber</td>
<td>1.32</td>
<td>8.84</td>
<td>37.18×</td>
</tr>
<tr>
<td></td>
<td>parallel</td>
<td>1.16</td>
<td>17.63</td>
<td>43.09×</td>
</tr>
<tr>
<td></td>
<td>para-clobber</td>
<td>1.15</td>
<td>35.04</td>
<td>43.33×</td>
</tr>
<tr>
<td></td>
<td>adv-parallel</td>
<td>0.04</td>
<td>7208.41×</td>
<td></td>
</tr>
<tr>
<td>m-mult</td>
<td>base</td>
<td>247.76</td>
<td>51771.8</td>
<td>100596.12</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>7.89</td>
<td>400.25</td>
<td>3088.59</td>
</tr>
<tr>
<td></td>
<td>parallel</td>
<td>1.13</td>
<td>417.68</td>
<td>246.12×</td>
</tr>
<tr>
<td></td>
<td>para-clobber</td>
<td>1.08</td>
<td>406.43</td>
<td>7.68×</td>
</tr>
<tr>
<td></td>
<td>adv-parallel</td>
<td>0.29</td>
<td>1534.69×</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Kernel runtimes in µs, under different optimizations, and speedups with respect to the base and scratchpad (‘spad’) configurations. Kernel abbreviations are as follows. m-trans: matrix transposition; m-unpack: matrix unpack; lin-inter: linear interpolation; hadamard: hadamard product; dot-prod: dot product; and m-mult: matrix multiplication. Optimization abbreviations are: para-clobber: parallel operations with clobbering; adv-parallel: advanced parallel optimizations. See main text for description of kernels and optimizations.
6.3.1 Base and scratchpad performance

The base configuration reflects INCA performance if current generation HPC networks were extended to support the TOM model as described in Chapter 4. Perhaps unsurprisingly, the base runtimes range from slow (42 µs for 128 B matrix transpose) to very slow (100 ms for 8192 B matrix multiplication). As a point of comparison, 8 KiB matrix multiplication on a 2.3 GHz Intel Haswell CPU takes between 10.56 µs and 139.49 µs, depending on gcc compiler optimization level.

While below we present a series of significant optimizations, we observe that even the performance of the base configuration may not be problematic. An application running on the host with sufficient latent parallelism still can take advantage of relatively slow compute assistance. Moreover, since INCA programs are intrinsically preemptible, a host application can always assume control over NIC compute resources. For instance, Portals includes an event notification queue that can be polled by host applications to determine the status of posted requests. This event queue can be used to provide information regarding the current state of an executing INCA program. Consequently, given a mechanism for signaling the NIC to halt INCA program execution, an idle host could steal partially-completed work from the NIC, picking up where the INCA program left off.

The scratchpad configuration assumes an INCA NIC is equipped with a local scratchpad memory to minimize memory access penalties, and a fast loopback path to avoid routing instructions through the switch. Unsurprisingly, enabling these features has a significant performance impact, reducing runtimes by one or two orders of magnitude across all kernels considered.
6.3.2 Network speeds

As illustrated in Figure 6.4, INCA execution times are expected to decrease as network speeds increase. To investigate the impact of network bandwidth on INCA kernel runtimes, we modified model parameters to execute kernels on the scratchpad configuration with 64B messages at 400 Gb/s and 1000 Gb/s network speeds; gap and Portals overhead becomes 1.28 ns and 0.512 ns, respectively. These parameters reflect the assumption that loopback for INCA program execution is optimized to avoid overhead. Table 6.4 shows results for kernel execution on an initial 8 KiB payload of 8 B operands. Note that, at 1000 Gb/s, the 1 ns scratchpad memory latency starts to become a bottleneck (although at these network speeds, we also expect cache latency to be less).

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Network Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>400Gb/s</td>
</tr>
<tr>
<td>dot-product</td>
<td>32.2</td>
</tr>
<tr>
<td>matrix-transpose</td>
<td>24.0</td>
</tr>
<tr>
<td>hadamard-product</td>
<td>32.32</td>
</tr>
<tr>
<td>filter</td>
<td>50.26</td>
</tr>
<tr>
<td>matrix-unpack</td>
<td>60.85</td>
</tr>
<tr>
<td>linear-interpolation</td>
<td>216.16</td>
</tr>
<tr>
<td>matrix-multiplication</td>
<td>1067.5</td>
</tr>
<tr>
<td>convolution</td>
<td>193.35</td>
</tr>
</tbody>
</table>

Table 6.4: Runtimes (µs) for kernels with scratchpad configuration, 64 B messages at 400 Gb/s and 1000 Gb/s, and initial payload of 8 KiB.

6.3.3 Reducing data movement

Atomic operations overwrite the contents of the first operand, so, to preserve standard program semantics, INCA programs copy operands before operating on them. However, this data copy may introduce unnecessary overhead. When possible, we modified kernels
Algorithm 3 INCA-A Dot Product w/ clobbering

1: PUTL i, _C0, i16
2: PUTL _R0, i, i16
3: LT _R0, _R0, _C1, i16
4: BLEZ _R0, 9
5: MUL A[i], A[i], B[i], f
6: ADD c, c, A[i], f
7: ADD i, i, _C2, i16
8: JMP 2
9: END

Contents of A are clobbered

to work in place under traditional atomic semantics. For example, the dot product code given in Chapter 5 is modified to allow clobbering of array A, as shown in Algorithm 3. Some copying is still necessary since overwriting the contents of the index during the loop condition evaluation would invalidate the loop. Note that the INCA-A directives are not shown in Algorithm 3.

Table 6.3 shows runtimes for the kernels that can be straightforwardly modified to allow clobbering: dot-product and hadamard-product (rows labeled ‘clobber’). In comparison to scratchpad, which shows an average speedup relative to base of 30.0×, and 32.7× for the two kernels, respectively, clobber brings this speedup to 38.13× and 37.18× (1.27× and 1.14× speedup with respect to scratchpad), demonstrating that when possible, avoiding data movement through clobbering is an effective optimization. These results also provide an initial indication of the speedups available if a non-clobbering semantics were available, i.e., one that allowed the destination location for a result to differ from that of the operand(s).

6.3.4 Hardware optimizations

While integrating scratchpad memory and allowing data to be overwritten when possible promise significant boosts to INCA kernel performance, reasonable hardware enhancements are also promising. In particular, faster NICs require more pins, so as process technologies scale to smaller sizes, the overall size of a NIC chip may not scale; instead,
the NIC circuity simply occupies less of the overall area. So, for instance, ALUs of
the sort utilized by a Portals NIC occupy approximately \(1/8^{th}\) of the die space typically
occupied by the compute logic circuitry of a modern CPU core.

Consequently, there is space to include additional hardware for accelerating INCA
kernel execution. Here, we consider the impact of general purpose acceleration hardware,
namely ALUs used in parallel to provide SIMD or simple MIMD functionality.

Algorithm 4 INCA-A Parallel Dot Product \((n \leq 256)\)

```
1: 1 PUTLM T[0], A[0], f, 256  \(\triangleright\) Bulk copy.
2: 2 MULM T[0], T[0], B[0], f, 256 \(\triangleright\) SIMD multiply.
3: 3 PUTL i, _C0, i16
4: 4 PUTL _R0, i, i16
5: 5 LT _R0, _R0, _C1, i16
6: 6 BLEZ _R0, 10
7: 7 ADD c, c, T[i], f
8: 8 ADD i, i, _C2, i16
9: 9 JMP 2
10: 10 END
```

As an illustrative example, consider the INCA microcode for dot product in Algo-

rithm 2 (Chapter 5). With SIMD multiplication available, we can move the multiplication
out of the inner loop, substituting \(\lceil k/A \rceil\) SIMD multiplications – where \(A\) is the width
of the SIMD instruction – for the original \(k\) single-operand instructions. The innermost
addition cannot be SIMD-parallelized easily because it accumulates to a single memory
location, so the loop is not entirely eliminated. The resulting parallel algorithm for inputs
with 256 or fewer operands is shown in Algorithm 4. The contents of array A are copied
to a temporary location (line 1), \(T\), which is then clobbered (line 2) by a new SIMD INCA
instruction, \texttt{MULM} (‘multiply multiple’). The remaining code accumulates the result.

We parallelized \texttt{dot-product} and \texttt{hadamard-product} in the manner just described.
We also parallelized \texttt{matrix-multiplication} under the assumption data is provided in
a format that facilitates SIMD operations, e.g., the second input matrix is transposed,
and distinct matrices corresponding to rows of the first input matrix are provided (so
that the Hadamard product can be used to concurrently multiply all columns by a row). For comparison (see below), we also implemented parallel versions with clobbering for each of these three kernels, which eliminates the need to copy vectors or matrices to temporary locations.

The other kernels were excluded from parallelization considerations: matrix-transpose and matrix-unpack are primarily data-movement kernels. filter is parallelizable assuming a ternary compare-and-swap operation; currently this is not an available atomic operation under Portals. linear-interpolation contains a backwards dependency, and convolution invokes non-trivial logic for handling edge data.

To evaluate the impact of parallelization, we extended the INCA interpreter to include SIMD versions of arithmetic instructions (MULM, ADDM, etc.). To estimate execution time, we incorporated inter-byte gap for ‘multiple’ instructions (G in the extended LogGP model described in Section 6.1), because those instructions involve larger payloads. Runtime calculations use whichever parameter (G or g) gives a longer execution time.

We posit 256 parallel ALUs, corresponding to approximately half the die space of a common 32-core CPU, leaving enough die space for NIC logic, processing pipelines and other miscellaneous circuitry. For payloads with more than 256 operands, multiple instances of the INCA SIMD instructions are used, unrolled rather than iterated. A diagram of the proposed SIMD parallel ALU design is shown in Figure 6.5. It should be noted that due to the reactive nature of computation used by INCA, data for the local operands can be staged to scratchpad or even local ALU cache (when predictable) prior to the remote operands arriving in network messages. Local operand caches can be small (a few operands) up to a size of a few KiB. For example, a reasonable design may use 4 KiB caches. This would typically require at least a 1 MiB scratchpad to feed 256 ALU 4KiB caches at a reasonable rate. It is possible to design a scratchpad within the timing requirements at 1 MiB up to several MiB as this matches size and speed requirements on modern CPU cache design.
Figure 6.5: INCA SIMD Unit. Rx FIFO inputs are those from the incoming data in the message. Local operands are staged well in advance of the operations in the scratchpad memory and pipelined into the local ALU operand caches on demand.

Revisiting Table 6.3, for the parallel-clobber kernels we observe average speedups (compared to base) of 254.01× for matrix_multiplication, 43.33× for dot_product, and 7208.41× for hadamard_product. Compared to scratchpad, the respective speedups are 7.92×, 1.33×, and 240.11×. The large speedup for Hadamard product is due to the fact the product can often be calculated in a single operation.

We can also compare (for dot_product and hadamard_product) the clobber and parallel optimizations, and the combination thereof. We observe that while using hardware SIMD parallelization offers benefits over simply minimizing data copies, the combined impact of parallelization and clobbering renders only a marginal benefit over parallelization alone. This is because the use of SIMD instructions itself eliminates the data movement associated with executing the arithmetic instructions serially.

Our basic parallelizations for dot_product and matrix_multiplication are limited
by the fact their summation phases remain serialized. While this phase cannot be fully parallelized, it can be restructured as a binary tree reducing $n$ operands to an accumulated value in $\lceil \log_2 n \rceil$ steps. To investigate the potential impact of specialized ALU hardware for affecting this process, we extended the INCA interpreter to include a logarithmic addition instruction. We assume that each stage in the reduction incurs the same Portals overhead cost as for the scratchpad configuration (5ns), but that these stages occur in a cascade, without requiring re-injection of new instructions (so the gap and memory latency costs are only incurred once, at the initiation of the instruction). Again referring to Table 6.3, these advanced parallel dot-product and matrix-multiplication kernels achieve average speedups over base of 2807.45× and 1354.69×, respectively (85.81× and 42.09× over scratchpad).

6.4 Conclusion

In this chapter, we have continued to explore the capabilities of SmartNICs, in particular, NICs supporting INCA. To this end, we presented INCA sim, a simulator that works with the INCA-A interpreter to estimate INCA kernel runtimes using a version of the LogGP model, extended to include memory access costs. Using the INCA ecosystem (INCA-Q, Q-compiler, INCA-A, interpreter), we developed a set of representative kernels and evaluated their runtimes under different workloads, software optimizations, and hardware optimizations. As a result of this exploration, we highlight the following conclusions.

First, it is clearly desirable to have on-NIC memory dedicated to INCA program execution. While runtimes will decrease as network speeds increase (Section 6.3.2), and it may be possible to reduce the costs of accessing main memory by aggregating accesses or pre-fetching, given the current runtimes, it is not clear these strategies are worth pursuing. Instead, since low-latency on-NIC memory is already standard within SmartNICs, including a scratchpad is a reasonable solution.

Second, avoiding unnecessary data movement significantly reduces runtimes. For
example, with clobbering, 8 KiB Hadamard product is approximately 20% faster than
the non-clobbering version, and dot product about 10% faster. Adjusting hardware to
allow ternary INCA-A instructions – e.g., placing the result of a binary operation in
a location distinct from that of either of the two operands – is even more effective at
avoiding data movement. For example, avoiding data movement to temporary locations
in the 8 KiB scratchpad dot product kernel reduces runtimes from 93.23 µs to 70.8 µs,
making the optimization comparable to introducing SIMD instructions.

The third conclusion is that it is reasonable to take advantage of available on-chip silica-
con area to include additional hardware for accelerating INCA kernel execution. Through
a combination of hardware acceleration and intelligent data staging, INCA performance
can, in at least some cases, be made comparable with contemporary CPUs, e.g., 8 KiB
matrix multiplication is 7.42 µs on INCA versus 10.56 to 139.49 µs on a contemporary
CPU.

The results presented in this chapter shed light on what one can expect from INCA
kernels with respect to their runtimes. In the following chapter, we explore their potential
impact on host application performance.
Chapter 7

INCA: Host Applications

In previous chapters, we investigated how one particular ‘smart’ capability – message matching – copes with overheads incurred by emerging multithreaded communication paradigms (Chapter 3), demonstrated that when coordinated with other task-specific offloaded capabilities, message matching enables general-purpose, deadline-free computation on the NIC (Chapter 4), and showed that a SmartNIC based on this model – i.e., an INCA SmartNIC – can execute some kernels with runtimes comparable to those of contemporary CPUs (Chapter 6). In this chapter, we continue this progression of inquiry by considering what INCA offloading affords with respect to speedups of host applications, namely, by offloading components of those applications to the NIC.

There are at least two scenarios where utilizing INCA to offload parts of host applications has the potential to benefit those applications. The first is by offloading ‘near-network’ parts of host applications, i.e., work that the host application does on data immediately before sending or immediately after receiving, e.g., data packing and unpacking, performing an initial matrix operation, etc. Since the data being operated on is passing through the NIC, it is reasonable to offload these near-network components and have the NIC deliver the transformed results. If this offloaded work can be overlapped with other work done on the host CPU, the result is an accelerated runtime.

The second scenario involves applications with idle networks. For example, an appli-
cation processing sensor data (e.g., from satellites) may periodically receive an influx of new data. However, outside of that relatively short burst, the network is idle; applications executing on host CPUs are compute-bound, attempting to produce useful results before the next influx. In cases such as these, it may be possible to utilize NIC resources as a co-processor by loading a kernel to the NIC and assigning it some chunk of the in-memory data to process. This could speed up the application by allowing it to complete earlier, or it could free up CPU resources to do other work that leads to better results.

In this chapter we explore both of these scenarios. In Secton 7.1, we evaluate the potential speedups afforded by INCA for a selection of applications and miniapps when near-network functions are offloaded. In Section 7.2, we present a preliminary investigation into the potential for INCA to accelerate an application with mostly-idle networks.

7.1 Offloading near-network functions

By providing the capacity to execute arbitrary kernels, INCA offers opportunities for accelerating applications by overlapping host application compute with more than mere core network applications. To assess the potential for the acceleration of applications, we studied a set of proxy applications identified by the Exascale project [179] and a full application.

MiniAMR [180] is an adaptive mesh refinement code designed to represent a range of applications. It should be noted that a general solver is used in this application, as the target of the proxy application is to capture the behavior of an adaptive mesh, rather than the computational kernels. MiniAMR uses pack/unpack and interpolation routines similar to the INCA kernels described above.

MiniMD [180] is a molecular dynamics code that is a subset of the application LAMMPS [171]. This code runs the Lennard Jones Liquid solver from LAMMPS that is based on a halo exchange pattern. This application was selected because it runs a common simulation mode of a major application. To analyze the potential for INCA,
we adapted the computation to separate internal data dependencies, the overlap target, from external dependencies, the INCA kernel target. This process also allows for communication and computation to be overlapped.

MiniFE [180] is a finite elements code that solves a conjugate gradient. This application was selected for its broad applicability to major production applications. Particularly, MiniFE’s solver features a matrix vector multiply that can be split into two separate computation phases, one handles intra-rank data dependencies and the other handles inter-rank data dependencies. As these split computation phases don’t directly depend on the other, the inter-rank computation can be directly translated into an INCA kernel without restructuring the application. MiniFE spends an observable amount of time in its setup phase that would otherwise be negligible in a production application. Therefore, to examine the impact on real applications we only consider the major time consuming section for production codes, the solver time.

LAMMPS [171] is a full molecular dynamics application which MiniMD is based on. For this test, we ran the same problem as with miniMD. The adaptations here mirror those of MiniMD discussed above. Many of the solvers in LAMMPS share the same structure and can be converted to INCA in a similar manner.

For each application, we identified code that could potentially be offloaded as an INCA kernel, subject to several criteria. First, these potential “INCA targets” should come directly before or after a communication region, i.e., they are ‘near-network’. Second, the target needs to take a significant portion of the application’s runtime. Finally, the target has to be separable from other computation. For MiniFE, MiniMD, and LAMMPS, there were computational regions that could easily be separated to internally dependent (intrinode) and externally dependent (internode) computation. As an initial evaluation, we measured the target kernel as just the external computation. Additionally, for these applications we analyzed the maximum possible speedup if we could also refactor the code to better split the computation between the INCA processor and the CPU. This analysis
has the caveat that it doesn’t account for variations in the throughput between an INCA processor and a CPU. In contrast to the other three apps, MiniAMR does not have easily separable regions, so MiniAMR required a refactor to stagger data dependencies to allow INCA to run in parallel with the CPU.

For code modification, we set a limit of no more than 10 lines of source code (not including the INCA kernel) to meet our definition of not refactoring the code. Non-refactored code represents a ‘first pass’ INCA implementation expected to take a few hours to several days of developer time. This requires modifications to allow for communication overlap with internal computation, to prevent external computation from occurring, and implementing the INCA kernel. The effort required to address the first two is relatively minimal, e.g., MiniFE already addresses communication overlap, and required removing nine lines to address the external computation. Similarly, MiniMD required under 10 lines to affect communication overlap, and two additional lines to remove externally dependent computation from the CPU code. Converting one of the LAMMPS solvers requires a similar effort to MiniMD.

For each application, we timed the INCA target kernel, communication, and the computation phase the target kernel would overlap. Table 7.1 shows the results of profiling these applications averaged over 10 runs. For non-refactored code, we observe a potential runtime improvement of up to 2.98% in MiniFE, up to 11.0% in MiniMD, and up to 11.5% in the LJ_cut solver for LAMMPS. With a structural refactoring, we can leverage the INCA’s processing power to further parallelize these applications. With MiniFE and MiniMD our refactor predictions are the result of moving some of the internal computation to the INCA kernel. For MiniAMR the process is more complex, where computation starts and continues as an unpack INCA kernel is making incoming data available and a pack INCA kernel is preparing completed data for transfer. With refactoring, speedups 26%, 37.2%, 25.7%, and 28.9% for MiniAMR, MiniMD, MiniFE, and LAMMPS respectively.
There are complexities that this analysis doesn’t account for, namely performance variation from laggard processes and network congestion. With delays caused in data arrival, the amount of time available to run INCA kernels before the host needs the data may be reduced such that the data must be handed off to the host before processing is complete, necessitating a lengthened overall runtime due to reduced overlap potential. However, even in the unlikely case where extreme performance variation halved the performance improvement shown in this analysis, the impacts of leveraging INCA kernels could have significant impact on the runtime of applications at scale.

### 7.2 NIC as co-processor

The applications considered in the preceding section follow a bulk synchronous processing pattern where work phases alternate with communication phases. Consequently, we chose targets for offloading that are ‘near-network’ in the sense they involve working on data that is either incoming or departing the host.

However, some applications have predominantly idle networks because they only engage in communication for a small portion of their overall runtime. For example, satellite data may only be delivered once an hour, with the ingress phase lasting minutes, and
the remaining time dedicated to processing by the host application. During the non-
communication phase of these applications, the network is idle. Moreover, the resolution
of processing done on incoming data is indirectly constrained by the requirements of
finishing the processing prior to the next influx of data, so gains on the order of micro-
seconds can be high impact. Since INCA harvests idle network resources, this type of
application presents a *prima facie* opportunity for an INCA NIC to serve as a coprocessor
(rather than near-network processor) to which work could be offloaded.

As a preliminary investigation into this NIC-as-coprocessor scenario, we profiled an
application tasked with processing satellite data for purposes of enabling over-the-horizon
radar. Data arrives hourly, ingress lasts approximately four minutes, and the system is
expected to finish processing before the next batch arrives approximately 56 minutes
later. Incoming data is subject to some initial complex processing (which, given the results reported above, we do not currently consider as candidates for INCA
offloading), at which point it is placed in memory. Once in memory, the data is traversed
to perform a version of standard ray tracing. This portion of the processing pipeline
was profiled to identify bottlenecks. On the basis of this profiling, we identified the
*range_doppler* function as a potential bottleneck (763.70 ms/call). Within this func-
tion, *solve_normal_dist* was called over two million times, at 0.0393 µs/call, for total
time spent in the function at greater than 78600 µs. The *convolve_normal* function was
also identified as time-consuming, at 16.63 µs per call. The question, then, is whether
INCA can help alleviate these bottlenecks.

For these simulations, we assumed a scratchpad configuration with local loopback, as
described in Chapter 6. Network speed is InfiniBand HDR (400Gb/s) and packets are
64B.

Algorithm 5 is a pseudoalgorithmic depiction of the original *solve_normal_dist*
code identified by the profile. While the logic is simple, the exponential function (*exp*) is
non-trivial. To accommodate *exp* under the INCA simulator, we inspected the standard
Algorithm 5 Solve Normal Distribution

1: procedure SOLVENORMALDIST\(x, \text{sigma}, x0, \text{mag}\)
2: \(\text{result} \leftarrow 0.0\)
3: \(xoffset \leftarrow x - x0\)
4: \(\text{twosiqsq} \leftarrow 2.0 * \text{sigma} * \text{sigma}\)
5: if \(\text{twosiqsq} \neq 0.0\) then
6: \(\text{result} \leftarrow \text{mag} \cdot \exp\left(-\frac{xoffset \cdot xoffset}{\text{twosiqsq}}\right)\)
7: end if
8: return \(\text{result}\)
9: end procedure

C math library using \texttt{objdump}, and given those results, modified INCA sim to charge for 50 binary instructions. In comparison to the roughly 40ns/call reported by the profiling, the INCA implementation requires 300ns/call. So, assuming offloaded INCA kernels overlap with the same function executed on the host, using INCA could trim off 9104 \(\mu s\) from the execution time of the application by offloading \texttt{solve_normal_dist}, i.e., approximately 11\% of the current time spent in the function.

In the INCA-A implementation of \texttt{solve_normal_dist}, the exponential function call is by far the most expensive: 231.56 ns, or 77.2\% of the total kernel execution time. In Chapter 6 we noted that on-chip real estate is available for additional acceleration hardware. Furthermore, because of its importance to scientific computing, the exponential function has been the target of acceleration attempts, usually involving FPGAs.

<table>
<thead>
<tr>
<th>Publication</th>
<th>Precision</th>
<th>Freq (MHz)</th>
<th>Latency (cycles)</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detry &amp; Dinechin [181]</td>
<td>Single</td>
<td>100</td>
<td>-</td>
<td>85</td>
</tr>
<tr>
<td>Jamro et al. [108]</td>
<td>Double</td>
<td>161</td>
<td>27</td>
<td>167.7</td>
</tr>
<tr>
<td>Pottathuparabil &amp; Sass [182]</td>
<td>Double</td>
<td>100</td>
<td>258</td>
<td>2580</td>
</tr>
<tr>
<td>Wielgosz et al. [183]</td>
<td>Double</td>
<td>200</td>
<td>30</td>
<td>150</td>
</tr>
<tr>
<td>De Dinechin &amp; Pasca [107]</td>
<td>-</td>
<td>310</td>
<td>35</td>
<td>112.9</td>
</tr>
<tr>
<td>Alachiotis et al. [106]</td>
<td>Double</td>
<td>252</td>
<td>224</td>
<td>888.9</td>
</tr>
<tr>
<td>Yuan &amp; Xu [184]</td>
<td>Double</td>
<td>230</td>
<td>23</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 7.2: Latencies for FPGA offloaded exponential function.

Table 7.2 summarizes a selection of research on offloading the exponential function

98
to FPGA. As this survey indicates, execution times for FPGA-accelerated exponential functions can be near 100ns. Assuming, then, the NIC is equipped with hardware acceleration support for the exponential function, the kernel runtime is reduced to 172ns. Under this scenario, total time spent in the function is reduced to 63981 µs, saving over 14 ms, a 19% improvement.

Algorithm 6 Convolve Normal

1: procedure CONVOLVE_NORMAL(imsize, imp, in, numpts)
2: maxindex = numpts − 1
3: midpoint = imp/2
4: for i ← 0, maxindex do
5: sumval ← 0.0
6: sumdat ← 0.0
7: for j ← 0, imsize do
8: if i ≤ midpoint then
9: if midpoint − i − j > 0 then
10: val = in[midpoint − i − j]
11: else
12: val = in[j − midpoint + i]
13: end if
14: else if i ≥ maxindex − midpoint then
15: if i − midpoint + j > maxindex then
16: val ← in[2 * maxindex − index + midpoint − j]
17: else
18: val ← in[j − midpoint + i]
19: end if
20: else
21: val = in[i − midpoint + j]
22: end if
23: sumval+ = imp[j] * val
24: sumdat+ = val
25: end for
26: result[i] = sumval − (sumdat/imsize)
27: end for
28: return result
29: end procedure

Algorithm 6 is a pseudoalgorithmic representation of the original convolve_normal C++ code. As can be seen, the logic for this function is considerably more complicated than for solve_normal_dist. According to the profile, the convolve_normal function
<table>
<thead>
<tr>
<th>Function</th>
<th>Time in function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>solve_normal_dist</td>
<td>78600 µs</td>
</tr>
<tr>
<td>convolve_normal</td>
<td>28487 µs</td>
</tr>
</tbody>
</table>

Table 7.3: Time spent in function call without and with offloading to INCA under the scratchpad configuration (‘INCA scratchpad’) and INCA under the scratchpad configuration with additional hardware acceleration support (‘INCA hardware’). Speedups relative to the original time are given in parentheses.

is called 1713 times at 16.63 µs/call, for a total time of 28487.19 µs.

We converted the `convolve_normal` code into INCA-Q and compiled an INCA-A kernel. The estimated execution time of this kernel is 25962.06 µs. While significantly slower than the CPU-side code, a single INCA call could be overlapped with the others, reducing the time spent in the function by 16.63 µs.

Motivated in part by the rise of deep neural networks, there has been considerable research into accelerating varieties of convolution using FPGAs or other hardware. Such hardware is capable of outperforming GPUs on, e.g., convolutional neural network processing [109]. Mohammad et al. [185] report a 19.298 ns latency for their FPGA implementation of a 4x4 convolution, and the parallel FPGA approach developed by Strom [186] shows 47000 µs latency for processing an 852x480 image. `convolve_normal` operates over 512 elements, so roughly extrapolating from the per-element timings, we project an FPGA-based convolution accelerator applied to this particular task would incur a latency between 700 ns and 60 µs. Since this form of acceleration would effectively transform the majority of the INCA-A kernel into a single use of a specialized INCA-A instruction (`CONV` or similar), any remaining overhead is due to data staging or movement. Suppose for sake of discussion, then, that the total accelerated INCA kernel runtime is 80 µs. Under this scenario, execution for the 1713 calls to `convolve_normal` is reduced to about 23628 µs.

Table 7.3 summarizes the results of this preliminary study. In this table, speedups
are relative to the amount of time spent in the function call in the original code. All INCA use cases offer modest reductions of time spent in the bottleneck function relative to the original. However, as noted above, even modest speedups can be high impact by enabling higher-resolution results.

This is only a preliminary study. For example, INCA kernel execution times do not take into account the costs of ‘calling’ the INCA kernel. Moreover, speedups are calculated under the assumption that calls to the same function can be overlapped. However, in the actual application, it may be possible to offload more calls to these functions because they can be overlapped with other work to be done by the host. Identifying these opportunities will require a more thorough examination of the target application, a project left to future work.

7.3 Conclusion

In this chapter, we’ve extended our inquiry into what SmartNICs can do by exploring two scenarios where INCA may accelerate application performance: offloading ‘near-network’ aspects of host applications under BSP communication patterns, and treating the NIC as a co-processor for addressing bottlenecks in applications where the network is more often idle than not. In the former case, we found that under ideal conditions, INCA offers 2% to 37% speedups. In the latter, we presented preliminary results suggesting that by harvesting idle network resources, INCA can offer speedups up to $1.23\times$, depending on available hardware acceleration support.

Both scenarios considered in this chapter involve offloading parts of host applications for purposes of accelerating those applications. However, INCA also affords the opportunity to offload applications that execute autonomously from host applications, entirely within the network. In Chapter 8, we explore this opportunity in more depth.
Chapter 8

Independent Applications for Adaptive Networks

Variations in network workloads can impact overall quality of service. For example, large (‘elephant’) flows can disrupt applications by consuming available network resources for extended periods of time, large numbers of unexpected messages arriving in a short time period may exhaust receive-side buffers (the ‘ingress problem’), or incoming RDMA traffic may create contention across the receiver’s memory bus, decreasing application performance [187]. Likewise, distinct jobs executing on the same network can interfere with each other, making scheduling and resource allocation (e.g., job placement on the network topology) important considerations [188], [189].

Given their location between host application and network, as well as their advanced processing capabilities, in-path, deadline-free NICs such as INCA make attractive candidates for addressing some of these issues. Specifically, applying techniques from machine learning (ML) and artificial intelligence (AI) may enable adaptive or self-learning networks, i.e., networks that anticipate and dynamically adapt to changing network configurations. For example, the capacity to predict a large amount of data will arrive over a short period of time may allow the NIC to intelligently schedule DMA transfers (to address memory contention), preemptively adjust credits assigned to senders (to avoid congestion), or schedule processors (in a sPIN-type NIC [54]) to allow additional time.
for packet processing.

In this scenario, offloaded ML kernels are independent applications in the sense defined in Chapter 2: they neither service traditional network applications, nor are they parts of host applications. Rather, they are applications that execute autonomously, within the network. Moreover, the deadline-free nature of INCA kernel execution is essential. What ought to drive such applications is not the need to operate within the gap defined by the network speed, but rather the need to generate predictions that are useful at whatever time scale is required, and this requirement is independent of gap. That is, constraining a solution by network speed is an artificial constraint on processing. By providing deadline-free kernel execution, INCA avoids this unnecessary constraint.

These considerations raise three questions. First, is it feasible to offload ML kernels for purposes of traffic prediction to an INCA-enabled SmartNIC? Second, do these ML kernels generate reasonably accurate predictions? And third, how well do these kernels address one or more of the issues raised above, e.g., in terms of increasing QoS or decreasing application runtimes? In this chapter, we take an important first step towards enabling adaptive networks by addressing the first and second of these questions. We design and implement a series of INCA kernels that accurately and effectively predict local network traffic. These kernels reside entirely in the NIC, operating independently of any host applications. To our knowledge, this is the first study demonstrating the feasibility of fully offloading ML kernels into the network for purposes of enabling adaptive networks.

In Section 8.1, we identify constraints on acceptable ML kernels imposed by INCA and the task demands, broadly construed. In Section 8.2 we list the applications whose traffic patterns we chose to profile. The system used to execute the applications, the data collected, and the means by which it was collected is described in Section 8.3. In Section 8.4 we identify two candidate ML kernels – simple linear regression and variations on rolling linear regression – selected on the basis of a consideration of features of the raw
data as well as the constraints established previously. The results of each ML method are presented in Sections 8.5 and 8.6. For each, we discuss the accuracy of the predictions generated by the ML method, followed by an analysis of the resource requirements of the corresponding INCA kernel(s).

8.1 Constraints on ML kernels

Since INCA’s compute capabilities are Turing complete (Chapter 4), an INCA SmartNIC can run any ML algorithm. Moreover, because the architecture is in-path, kernel execution is not subject to deadlines imposed by network speeds (Chapter 2). In practice, however, there are practical constraints on the types of ML kernels that can be executed. First, as noted in Chapter 6, we assume on-NIC memory is limited (e.g., \( \leq 1\text{MiB} \)). Second, predictions must be generated sufficiently quickly to be useful for whatever task they are to be used for. For example, adjusting credits in anticipation of a possible ingress event requires sufficient time to accomplish that adjustment prior to the event occurring, and this includes sending messages to senders with updated credit information. InfiniBand EDR (100 Gb/s) point-to-point latency is roughly 1 \( \mu s \) for small messages, so any prediction must allow for at least 2 \( \mu s \) before the predicted event.

Third, there are interactions with sampling rates that must be taken into consideration, especially if the model is dynamic, i.e., updated to take into account incoming data. With dynamic methods, the ML kernel must both update the model and generate predictions with sufficient time to handle the next incoming data point. Alternatively, incoming data could be buffered and processed in batches, and predictions generated between batches. Finally, a third possibility is to have two distinct kernels executing, one to update the model, and another to generate predictions using the most recently available parameters. In this study, we choose the simplest solution, namely, to keep the ML kernel execution time within the sampling rate.

For the purposes of this study, these considerations imply two constraints on ML
kernel selection. First, a suitable kernel should have a relatively small memory footprint. Second, kernels with short (e.g., less than 50 ms) runtimes are desirable.

8.2 Applications

For this study, we investigated the network traffic of seven applications, proxy applications, and miniapps (henceforth collectively referred to as ‘applications’). The HPCG (High Performance Conjugate Gradients) benchmark is a widely-used I/O bound benchmark that models data access patterns of common scientific workloads [190]. LAMMPS is a classical molecular dynamics code [191]. For this study, we ran the Lennard-Jones (atomic fluid) and Rhodo (rhodospin protein) benchmarks. Lulesh is a shock hydrodynamics proxy application [192]. MILC is a quantum chromodynamics code [193]. The final three miniapps are taken from the Mantevo miniapps suite [194]. MiniAMR performs adaptive mesh refinement, MiniAMR MiniFE is an unstructured implicit finite solver, and MiniMD solves a molecular dynamics problem.

8.3 Data collection

All applications were executed on an ARM-based system, where each node has two sockets, and each socket contains a 32-core Cavium ThunderX2 ARM CPU operating at 2GHz. The network is 4x EDR (100 Gb/s) InfiniBand using NVIDIA Mellanox ConnectX-5 NICs. All applications were executed on eight nodes with one MPI process per node.

CPU hardware performance counters are a popular source of input data for ML tasks in HPC and networking, appearing in studies involving memory contention, resource allocation, and intrusion detection [195]–[198]. Like CPUs, ConnectX-5 NICs also include performance counters. These counters track features such as number of bytes and packets sent and received, number of dropped packets, number of buffer overruns, packet sequence
errors, and so on. For this study, we focus solely on predicting the amount of received data.

One way to access the ConnectX-5 hardware counters is through a dedicated system call, `perfquery`, which requires root access. Counters are also exposed through the filesystem, so can be accessed without root privileges using `cat`. Based on a simple benchmark that queries each of these access methods, we found that the maximum sampling rate using the system call method is 22.22 Hz (period = 45.01 ms), and through the filesystem the maximum is 105.93 Hz (period = 9.5 ms). Of course, any kernel operating on-NIC will not be constrained by either of these methods, and will be capable of much higher sampling frequencies.

For the results reported here, we use the filesystem access method with a sampling frequency of 20.0 Hz. We focus on bytes received, as reported by the NIC’s `port_rcv_data` counter. Our data gathering tool is pinned to one core on one socket, and does not itself engage in any communication through the IB port. The MPI processes of the applications are pinned to cores on the other CPU. The hardware counters used for this study only collect data for bytes received by processes residing on the CPU used by the application. Each application was executed 11 times, providing 11 data sets for each rank of each application.

### 8.4 ML kernels

Figure 8.1 shows raw counter data (bytes received) for a single run of each application. Since the applications are executed with one MPI process per host, the eight hosts shown in each graph show the data received by each MPI rank in the application (plus any additional network traffic).

Most of the applications exhibit similar gross behavior: after an initial startup phase, data arrives at a more-or-less steady rate, and then levels off as the application enters a teardown phase. The duration of the startup phase before sustained communication
Figure 8.1: Examples of raw data (bytes received) acquired from the NIC port_recv_data hardware counter for the first run of each application. Byte totals are adjusted to the start of the application run.
Table 8.1: Mean probability of data arriving at any given point during the main work/communication phase, calculated across all participating NICs for all runs. Standard deviation was $\leq 0.02$ for all means. Shaded cells show which category has higher probability.

<table>
<thead>
<tr>
<th>Application</th>
<th>$Pr(data)$</th>
<th>$Pr(\neg data)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPCG</td>
<td>0.15</td>
<td>0.85</td>
</tr>
<tr>
<td>LAMMPS-lj</td>
<td>0.03</td>
<td>0.97</td>
</tr>
<tr>
<td>LAMMPS-rhodo</td>
<td>0.09</td>
<td>0.91</td>
</tr>
<tr>
<td>Lulesh</td>
<td>0.93</td>
<td>0.07</td>
</tr>
<tr>
<td>MILC</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>MiniAMR</td>
<td>0.79</td>
<td>0.21</td>
</tr>
<tr>
<td>MiniFE</td>
<td>0.14</td>
<td>0.86</td>
</tr>
<tr>
<td>MiniMD</td>
<td>0.06</td>
<td>0.94</td>
</tr>
</tbody>
</table>

begins varies from around 5 samples (250 ms) in the case of Lulesh to over 4000 samples (200 s) for MiniFE. This period likely includes samples from before the application launch simply because the tool for gathering data must be launched before the application. While the startup period may appear flat, it can involve the reception of small amounts of data. For instance, in the run of HPCG shown in Figure 8.1, sustained communication begins at 596 samples (29.8 s), but by 200 samples nearly all nodes have received multiple influxes of small amounts of data ($< 600$ B). Likewise, by 200 samples half of the MiniFE nodes have received more than one influx of data ($< 4$ KiB). As with the startup phase, the teardown phase may differ in length across applications, and in the raw data, includes any additional samples that might occur between exiting the application and terminating the data gathering tool.

During the main work and communication loop, nodes for some applications exhibit a ‘stairway-with-landings’ pattern one expects from bulk synchronous processing applications (LAMMPS-lj, LAMMPS-rhodo, MiniMD). Likewise, the pattern of data received across MiniMD nodes reflects the fact the miniapp performs adaptive mesh refinement during execution, so workloads are adjusted across nodes.

To gain a better understanding of the data, we calculated the probability of data
arriving at any given point in the application execution (not including startup and tear-down phases). The results of this analysis are shown in Table 8.1. As this analysis shows, the applications fall into two categories: those that maintain a more-or-less steady influx of data (Lulesh, MILC, and to a lesser extent, MiniAMR), and those where data arrives infrequently (HPCG, LAMMPS-lj, LAMMPS-rhodo, MiniFE, and MiniMD). In what follows, we refer to the former class as ‘dense’ applications, and the latter as ‘sparse’. This distinction will play a role in understanding some of the ML techniques explored below.

The main work and communication loop is of primary importance for a traffic prediction task, since that is where the vast majority of data arrives. Focusing only on this phase, given the gross behavior of the data as shown in Figure 8.1, and the constraints on memory and execution time given in Section 8.1, we determined a reasonable candidate ML method is ordinary linear regression (LR) and variations thereof. That is, given the characteristics of the data and the fact we are looking for methods with small memory and runtime footprints, methods based on linear regression are reasonable candidates. In the remainder of this section, we describe the regression methods used in the study.

8.4.1 Ordinary LR

The first LR method we consider is ordinary LR, i.e., simple linear regression using the ordinary least squares method. This is a static method in the sense the model is not updated during application execution. Instead, a possible use case is to have a library of pre-trained models (which are nothing more than sets of pairs of intercepts and slopes), one of which is loaded at application launch time to the NIC along with the INCA kernel that performs predictions when requested.

As previously noted, the raw data includes a main phase of sustained communication where the majority of data arrives, with (perhaps optional) plateaux on either side representing startup and teardown phases. Since our target is the sustained
work/communication phase, our training script pre-processes the raw data to remove these plateaux by trimming the starting phase up until the first sample at which data is received, and trimming the teardown phase until the last sample at which data is received.

For each application, for each rank in each of the 11 runs, the model is trained on the data from that rank, and then tested against the same rank data from the remaining 10 runs. For each rank, root mean squared error (RMSE) is calculated for both the 11 training instances and the 110 testing instances.

### 8.4.2 Rolling LR

In addition to the static method described above, we also considered two *dynamic* LR techniques: rolling (ordinary) LR, and rolling weighted LR. These methods attempt to reduce error by adapting the model to changes in the amount of received data as they occur, and hence may provide more accurate prediction, e.g., when in plateaux phases such as those seen in LAAMPS-lj and MiniMD (Figure 8.1).

Both types of rolling LR methods share the same basic strategy, shown in Figure 8.2. A window of size $k$ samples is ‘rolled’ along the sequence of samples as they arrive. At each sample, the model is trained on the samples in the training window, and prediction
generated for one or more points $p_i$ in the future.

In the ordinary rolling LR condition, the LR performed at each step is the same as in the static LR case. In the weighted condition, the error at each point in the training window is adjusted by a vector of weights, affecting their influence on the result. For example, a vector of weights that reduces the importance of the most recent sample relative to the others will be less responsive to changes and may ‘smooth’ the estimation given transient perturbations. For weighted rolling LR, we considered two types of weights: exponentially increasing and exponentially decreasing. In the former, older samples are discounted while the most recent sample is amplified, and in the latter, the oldest sample is given the greatest weight. Weights are calculated using a standard exponential function, scaled so that the maximum is 1.0.

These methods raise the issue of preferred window sizes: which training window sizes perform best for predictions at which point ($p_i$ in Figure 8.2) in the future? One possibility is to have the kernel itself dynamically adapt the training window size to minimize error for a given prediction point as the application is running. However, for this study we adopt a use case similar to that proposed for static, ordinary LR. Through some initial analysis, we determine performant window sizes for two classes of predictions: short-term and long-term. Then, these window sizes (and their weights, in the case of rolling weighted LR) are loaded from a library at the launch of the application. Given the two window sizes, we then evaluate the methods by calculate the average RMSE across all ranks of all 11 runs of each application. Results from both the initial analysis of window sizes, and the performance given windows selected on the basis of that analysis, are given below. As in the case of static ordinary LR, the raw data is trimmed to remove startup and teardown plateaux.
8.5 Results: Ordinary linear regression

In this section we present and discuss the results of the ordinary (static) linear regression experiments described above. We first consider (1) how well the model captures the data, and (2) the resource requirements (time and memory) for an INCA kernel that implements the inference phase.

8.5.1 Method performance

Figure 8.3 shows normalized RMSE (NRMSE) for all ranks of all applications. Results are averaged across 11 runs for the training results, and across 110 runs for the test results. Error bars are standard deviation, although this may be sufficiently low as to not be visible.

We see that training and testing normalized RMSE never exceeds 2.5% (MiniAMR), and for some apps remains close to 0.5% (LAAMPS-rhodo, Lulesh). Test NRMSE is always higher than training, and for some applications this suggests overfitting (HPCG, MILC, MiniFE, and some ranks of MiniMD). One possible contributor to this outcome may be the method for trimming startup plateaux described in Section 8.4. The raw data is trimmed to remove startup plateaux by removing samples up to the first positive change. However, in some cases this method is insufficient. For example, Figure 8.4 shows the result of training on rank 0 data from a specific run of MiniFE (left), and the application of the result of that training to data from the same rank on a different run (right). As the figure makes clear, despite the relatively good fit on the training data, and the *prima facie* good fit to the test data, the lack of the startup plateaux on the testing data offsets the predictions, resulting in increased error. One possible method for addressing this issue is to have the application signal to the NIC when the model should be active, i.e., after any startup phase.
Figure 8.3: Results from the static, ordinary LR study.
Figure 8.4: An illustration of how the startup plateau can offset the learned model. The training data (left) has a long startup plateau, so despite the relatively good fit, when applied to another data set with no such plateau (right), there is significant error.

8.5.2 INCA kernel resources

The static, ordinary LR method was chosen not only because the data suggested it, but also because it promises to have a light resource footprint. To confirm this, we implemented the inference phase as an INCA kernel, and estimated memory requirements and runtimes using INCAsim.

Algorithm 7 INCA-A Ordinary Linear Regression

1: PUTL output, b0, f
2: PUTL _R1, b1, f
3: MUL _R1, _R1, x, f
4: ADD output, output, _R1, f
5: END

The INCA-A kernel is shown in Algorithm 7. The kernel takes advantage of the fact the output can be clobbered, so only requires four instruction to complete. The memory required is extremely modest. Assuming 64 bit operands, the total amount of memory required is less than 64 B. Regarding runtimes, on a 200 Gb/s NIC with local scratchpad memory and 64B packets, the estimated runtime of this kernel is 26.48 ns. When network speeds increase to 400 Gb/s [4], all else being equal, the runtime is reduced to 16.24 ns.
8.6 Results: Rolling linear regression

One of the limitations of a static LR approach is it does not adjust to changes in the incoming data stream. The ‘offset problem’ illustrated in Figure 8.4 highlights this drawback. If a process is significantly delayed in sending data to a receiver, the result may be a plateau, and lacking any mechanism for adapting to the change, subsequent predictions are shifted.

Rolling LR methods avoid these issues by dynamically updating the model as new data is acquired. Prediction accuracy is a function of both the size of the training window and the future point to be predicted. To assess training window sizes with respect to prediction points, for each rank in each application run, we execute the prediction method while varying training window size.

Figure 8.5 shows the results of this process for a single rank (rank 3) from each application, averaged across 11 runs, for prediction points ranging from $t_0 + 1 (p_0)$ to $t_0 + 10 (p_9)$, and window sizes ranging from 2 to 250. Error bars show standard deviation, and for each prediction point, a red ‘×’ indicates the best performing window size. For six of the eight applications – HPCG, LAMMPS-lj, LAMMPS-rhodo, MiniFE, and MiniMD – best performing window sizes fall into two groups: smaller window sizes for short-term predictions, and larger window sizes for long-term predictions. This is to be expected given the raw data shown in Figure 8.1 and the analysis provided in Table 8.1. NICs for applications such as HPCG, LAMMPS-lj, LAMMPS-rhodo, MiniFE, and MiniMD have are mostly quiescent with respect to data arrival. Consequently, smaller window sizes allow the regression to react to sudden increases in amounts of incoming data, and to return to quiescence (i.e., zero slope), while larger window sizes allow the regression to ignore plateaux where no data arrives, capturing the longer-term trends. In contrast, in cases where data is nearly constantly arriving (Lulesh, MILC, MiniAMR), there is not a huge distinction between small and large window sizes, because the slope of the data
Figure 8.5: Error as a function of window size, for predictions ranging from $t_0 + 1$ ($p_0$) to $t_0 + 10$ ($p_9$) samples and window sizes ranging from 2 to 250. Results are averaged across 11 runs, and error bars show standard deviation. The best performing window size for each prediction point is indicated by a red ‘×’.
We executed the window size search (depicted in Figure 8.5 for a single MPI rank from each application using ordinary LR) for all MPI ranks of all runs of all applications, and determined the best performing window size for each prediction point under the three types of rolling LR. The results of this analysis are given in Table 8.2, which shows the median best performing window size for each prediction point for each application and rolling LR method. Again, the pattern observed in Figure 8.5 is evident insofar as applications either exhibit two distinct groupings of window sizes, one for short-term predictions and the other for long-term predictions, or do not exhibit such an obvious distinction (Lulesh, MILC, MiniAMR). Moreover, we see that weighted rolling LR with increasing (exp-inc) weights tends to increase the best performing window size, while decreasing weights (exp-dec) tends to decrease the best performing window size. As before, this is expected. For example, the increasing weights emphasize the most recent sample, so, on applications where data arrival is relatively infrequent, larger window sizes are required to compensate for the prima facie overemphasis on that sample. The opposite holds for the decreasing case.

The fact many applications have apparently distinct groupings of window sizes corresponding to short- and long-term prediction points suggests a strategy where an INCA kernel can use two different windows, one for each type of prediction. To determine these window sizes, we applied the Fisher-Jenks algorithm to each set of median window sizes given in Table 8.2 to provide guidance for establishing a boundary between the two classes of window. We then selected the median window size from each of those groups to establish a small and large window size for each app, for each type of rolling LR. The results of this analysis are given in Table 8.3. In one case – MiniFE – we overrode the results of the algorithm to move the boundary by one location to better correspond to the intuitive class divisions.\footnote{For example, in the MiniFE ord case, the algorithm placed 6 and 54 into the small category, and 55 through 61 into the large category; we manually adjusted the boundary to occur between 6 and 54.}
<table>
<thead>
<tr>
<th>Application</th>
<th>Weights</th>
<th>$p_0$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
<th>$p_6$</th>
<th>$p_7$</th>
<th>$p_8$</th>
<th>$p_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpcg</td>
<td>ord</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>13</td>
<td>151</td>
<td>150</td>
<td>149</td>
<td>148</td>
<td>148</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>17</td>
<td>19</td>
<td>247</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>117</td>
<td>148</td>
<td>147</td>
<td>146</td>
<td>145</td>
<td>145</td>
<td>144</td>
</tr>
<tr>
<td>lammps-lj</td>
<td>ord</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>26</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>lammps-rhodo</td>
<td>ord</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>107</td>
<td>106</td>
<td>106</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>248</td>
<td>250</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>99</td>
<td>98</td>
<td>98</td>
<td>97</td>
</tr>
<tr>
<td>lulesh</td>
<td>ord</td>
<td>45</td>
<td>15</td>
<td>5</td>
<td>48</td>
<td>12</td>
<td>11</td>
<td>56</td>
<td>15</td>
<td>28</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>138</td>
<td>68</td>
<td>11</td>
<td>143</td>
<td>53</td>
<td>25</td>
<td>148</td>
<td>41</td>
<td>66</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>13</td>
<td>15</td>
<td>14</td>
<td>16</td>
<td>32</td>
<td>31</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>milc</td>
<td>ord</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>minimamr</td>
<td>ord</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>39</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>minife</td>
<td>ord</td>
<td>4</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>58</td>
<td>57</td>
<td>57</td>
<td>56</td>
<td>55</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>8</td>
<td>131</td>
<td>153</td>
<td>174</td>
<td>174</td>
<td>173</td>
<td>173</td>
<td>171</td>
<td>170</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>60</td>
<td>59</td>
<td>58</td>
<td>57</td>
<td>56</td>
<td>55</td>
<td>54</td>
<td>53</td>
<td>76</td>
<td>75</td>
</tr>
<tr>
<td>minimd</td>
<td>ord</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td>105</td>
<td>105</td>
<td>104</td>
<td>104</td>
<td>103</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>exp-inc</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>22</td>
<td>233</td>
<td>238</td>
<td>241</td>
<td>244</td>
<td>245</td>
</tr>
<tr>
<td></td>
<td>exp-dec</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>99</td>
<td>98</td>
<td>97</td>
<td>96</td>
<td>95</td>
<td>94</td>
</tr>
</tbody>
</table>

Table 8.2: Best-performing window sizes for each prediction point, for ordinary (unweighted) linear regression (ord), exponentially increasing weights (exp-inc), and exponentially decreasing weights (exp-dec).
Table 8.3: Small and large window sizes selected on basis of the best-performing window size parameter sweep, for ordinary (unweighted) linear regression (ord), exponentially increasing weights (exp-inc), and exponentially decreasing weights (exp-dec). Values determined using the Fisher-Jenks algorithm; class boundaries for MiniME were slightly adjusted by hand.

<table>
<thead>
<tr>
<th>Application</th>
<th>ord</th>
<th>exp-inc</th>
<th>exp-dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small</td>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td>hpcg</td>
<td>6</td>
<td>149</td>
<td>14</td>
</tr>
<tr>
<td>lammps-lj</td>
<td>6</td>
<td>242</td>
<td>9</td>
</tr>
<tr>
<td>lammps-rhodo</td>
<td>5</td>
<td>106</td>
<td>7</td>
</tr>
<tr>
<td>lulesh</td>
<td>14</td>
<td>52</td>
<td>47</td>
</tr>
<tr>
<td>milc</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>minimar</td>
<td>6</td>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td>minife</td>
<td>4</td>
<td>57</td>
<td>8</td>
</tr>
<tr>
<td>minimd</td>
<td>3</td>
<td>104</td>
<td>11</td>
</tr>
</tbody>
</table>

8.6.1 Method performance

Given the small and large window sizes specified in Table 8.3, we executed each type of rolling LR on the data from each rank from each application, and calculated the NRMSE for each prediction point. To facilitate comparison, we started accumulating error only at sample 251, the first prediction available at the largest considered window size; the NRMSE reported for ordinary LR (Section 8.5) is also calculated in this way.

Figure 8.6 shows average NRMSE for small and large training window sizes, for each application, prediction point, and rolling LR type (ordinary (ord), exponentially increasing weighted (exp-inc) and exponentially decreasing weighted (exp-dec)). We note, first, that the crossover point between lines representing predictions made on the basis of LR on small and large training windows corresponds to the division between groups of window sizes given in Table 8.2. For instance, the crossover between small and large in the ord method applied to LAMMPS-rhodo is between \( p_5 \) and \( p_6 \). Second, for short-term predictions, exponentially increasing weighted LR tends to perform better than ord or exp-dec. Moreover, this holds regardless of whether the application is
Figure 8.6: Average error for ordinary (ord), exponentially increasing weighted (exp-inc), and exponentially decreasing weighted (exp-dec) rolling LR methods, for small and large training window sizes, and prediction points ranging from \( t_0 + 1 \) (\( p_0 \), 50 ms) to \( t_0 + 10 \) (\( p_9 \), 500 ms).
sparse or dense. This result is intuitive, because emphasizing the more recent samples in comparison to older samples improves the ability of the method to react to incoming data. Third, regarding longer-term predictions, for three of the eight applications (HPCG, LAMMPS-rhodo, and MiniMD), \textit{ord} performs better than either \textit{exp-inc} or \textit{exp-dec}. These are all sparse applications. In dense applications, \textit{exp-inc} performs better for the most part. Fourth, rarely does \textit{exp-dec} perform better than the others for either short- or long-term predictions, with MiniFE being the most notable exception. Finally, Lulesh’s oscillating errors are an obvious departure from those of the other applications. Unlike the other applications surveyed here, there are significant differences in the amount of data each rank receives by the time the application terminates (Figure 8.1). We hypothesize the unique behavior with respect to NRMSE is due in part to the current strategy of using a single pair of training window sizes (small and large) for all ranks; testing this hypothesis is left to future work.

In comparison to the static, ordinary LR method (Figure 8.3), the rolling LR methods represent a significant reduction in error. For example, with ordinary LR testing NRMSE for HPCG approached 1%; in comparison, the best performing rolling LR methods remain below 0.1%. Similarly, whereas NRMSE for ordinary LR for LAMMPS-lj can exceed 1%, the best performing rolling methods remain below 0.3%. In general, in comparison to static ordinary LR, the rolling LR methods secure reductions in NRMSE for all of the applications considered here. Because rolling methods – \textit{ord} and \textit{exp-dec} in particular – secure reduced prediction error, and by being dynamic address the issue of responding to unexpected events raised at the beginning of this section, we believe a rolling LR method is preferable to the standard method as an offloaded kernel. The question, then, is how the increased complexity of the methods impact resource requirements.
8.6.2 INCA kernel resources

Whereas the ordinary LR approach considered in Section 8.5 required the INCA kernel to do little more than perform a multiplication and addition, rolling LR methods require the kernel perform the actual regression. To assess the impact of this increased complexity, we implemented both ordinary and weighted rolling LR kernels in INCA, taking advantage of SIMD instructions (Chapter 6) when applicable.

<table>
<thead>
<tr>
<th>Rolling LR Method</th>
<th>Network Gb/s</th>
<th>5 Sample Window</th>
<th>250 Sample Window</th>
<th>Memory (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord</td>
<td>200</td>
<td>0.68</td>
<td>21.87</td>
<td>&lt; 4</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.41</td>
<td>13.46</td>
<td></td>
</tr>
<tr>
<td>ord-parallel</td>
<td>200</td>
<td>0.28</td>
<td>0.69</td>
<td>&lt; 4</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.17</td>
<td>0.34</td>
<td></td>
</tr>
<tr>
<td>weighted</td>
<td>200</td>
<td>1.17</td>
<td>38.10</td>
<td>&lt; 8</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.71</td>
<td>23.22</td>
<td></td>
</tr>
<tr>
<td>weighted-parallel</td>
<td>200</td>
<td>0.74</td>
<td>15.72</td>
<td>&lt; 12</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.45</td>
<td>9.65</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.4: Runtimes (μs) for INCA kernels implementing ordinary (ord) and weighted (weighted) rolling LR methods, for current and next-generation Infini-Band network speeds (64B packets). parallel versions use SIMD instructions.

Table 8.4 shows the simulation results for rolling LR INCA kernels using ordinary (ord) and weighted (weighted) linear regression, for current (200 Gb/s) network speeds, and the upcoming next generation (400 Gb/s). These kernels generate a single prediction. Runtimes are estimated for both a small training window size (5 samples) and a large window size (250 samples). The use of dot products in performing linear regression affords opportunities for parallelization through SIMD instructions (Chapter 6); the parallel versions take advantage of these opportunities.

These results show that INCA kernels generating predictions from small training win-
dow sizes have very modest runtimes, ranging from less than a microsecond to approximately seven microseconds. Runtimes for large training windows span tens of microseconds, but can be roughly halved by taking advantage of parallel (SIMD) instructions.

In the implementations considered here, the ordinary rolling LR requires less than 4 KiB of memory, weighted rolling LR requires approximately 8 KiB, and because of additional space required as destinations for SIMD instructions, the weighted parallel kernel requires less than 12 KiB. The memory requirements of these kernels are thus well within the 1 MiB limit proposed in Chapter 6.

Finally, we note that while the number of instructions executed for kernels using the small (5 samples) window size are in the low 100s, those for the large window size (250 samples) are in the thousands or tens of thousands. The larger number of instructions puts these kernels outside the capabilities of, e.g., a 32 core 2.5 GHz on-path SmartNIC, which (assuming 200 Gb/s and 64 B packets), have a deadline of approximately 205 instructions (Chapter 2). This outcome provides further motivation for preferring deadline-free SmartNIC architectures such as INCA for enabling adaptive networks.

8.7 Related work

A popular target for ML methods in HPC is job scheduling. Krishnaswamy et al. [199] use a collection of scripts with known runtimes and a similarity metric. Rodrigues et al. [200] use job submission features (user ID, resources requested, etc.) and consider k-means, SVMs, MLPs, and random forests. Wyatt et al. [201] convert job submission scripts into ‘pictures’, and then use a convolutional neural network to generate runtime predictions.

Closer to the sorts of network-side issues targeted in this chapter, Jain et al. [196] use random forests to predict which node allocation will result in better application performance. Dickov et al. [202] propose using n-grams to detect patterns in MPI calls issued by a host application to deactivate and reactivate NIC links to conserve power.
Kiran et al. [203] use a Gaussian mixture model to classify flows as elephant or mouse for purposes of segregating them. Groves et al. [195] show that random forests trained on CPU performance counter data can distinguish between situations where network-induced memory contention will and will not occur, potentially allowing the dynamic selection of solutions. None of these works propose offloading to the NIC.

Some of these works share a strategy of using hardware performance counter values as inputs [195], [196]. Performance counters have also been found to be useful for potentially detecting malicious attacks, e.g., Spectre or Rowhammer [197], [198]. The work described here is, as far as we know, the first to take advantage of on-NIC hardware counters.

8.8 Conclusion

Adaptive networks are networks that deploy techniques from ML or AI to dynamically adapt to changing network conditions. As ‘gatekeepers’ situated between the network and host applications, SmartNICs can play a pivotal role in enabling adaptive networks. In this chapter, we’ve demonstrated that INCA-enabled NICs can offload ML kernels for the purposes of predicting network traffic. Even without significant hardware acceleration support, these kernels can execute in sub-microsecond times, with modest (<1 MiB) memory requirements. Moreover, these kernels execute independently of any host application.
Chapter 9

Conclusion and future work

The past half-decade has witnessed a resurgence of interest in SmartNICs, i.e., network interfaces that not only move data, but also perform potentially complex work with or on that data. What sets this new wave of research apart from earlier attempts at diversifying network functionality is its scope. Where once making the network intelligent meant offloading characteristically network-oriented applications – packet forwarding, firewalls, segmentation, bits and pieces of specific protocols, collective communications, and so on – researchers are currently broadening the vision of what a ‘smart’ network can be expected to do. Rather than only perform core network functions, NICs and other network appliances are increasingly being asked to offload parts of host applications, or even execute applications that are entirely independent of those running on host CPUs.

While on-NIC capabilities have been expanding, what exactly they can do for high performance computing is not well-understood. For example, to what degree do current ‘smart’ capabilities address anticipated future paradigms such as multithreaded communication? Can existing on-NIC capabilities provide additional flexibility for supporting novel offloaded functionality, or must we appeal to general-purpose compute hardware such as CPUs? Finally, when such flexibility is secured, what can be done with it to service HPC applications?

In this work, we’ve explored answers to these questions. In Chapter 3, we showed
that offloaded message matching may help mitigate message processing overheads, but not alleviate them entirely. In Chapter 4 we established that offloaded message matching, when coordinated with other offloaded capabilities (namely, atomic and triggered operations), can elevate a NIC from being smart in the traditional sense of handling task-specific network applications, to the contemporary sense of supporting the execution of arbitrary programs. Finally, in Chapters 6 through 8, we explored what these general-purpose, deadline-free compute capabilities can offer as regards accelerating host applications or enabling the offloading of machine learning kernels.

To conclude, we briefly highlight some of the additional research opportunities afforded by INCA. First, INCA currently exists in theory and software. An obvious future direction of research is to secure a physical implementation of an INCA-capable NIC. To this end, we are currently collaborating with academic partners to realize this goal in the form of an FPGA implementation.

Second, we’ve presented INCA as an alternative to on-path or off-path approaches, but in fact these approaches are not mutually exclusive. Aside from mandating the primitive capabilities required to secure general-purpose compute capabilities, INCA is neutral with respect to the on- or off-path processing elements that are actually available. So, for instance, in Chapter 7, we proposed including FPGA or ASIC-based accelerators for performing certain functions, integrated into INCA as new INCA-A instructions. In the same way, INCA could schedule resources such as CPU cores in a sPIN-style system, making the resulting hybrid deadline-free. More generally, from this perspective, INCA can be viewed as an overarching system for coordinating the deployment of potentially heterogeneous on-NIC compute resources. Exploring how this type of hybrid ‘INCA+X’ SmartNIC can address the issues raised above is a potentially fruitful direction for future work.

Third, deadline-free SmartNICs such as INCA have potential applications to middleware runtimes, and these deserve close consideration. One example is tolerance to node
failures. For instance, it is possible that the host system panics while leaving the NIC still operational. In such a situation, the NIC is in an opportune position to both detect and mitigate the failure: noticing the host is defunct, an INCA kernel could intercept and forward incoming traffic to a replacement node while also alerting the sender of the new location for the destination process. If the CPU panics but leaves the memory system operational, the NIC may be able to handle the task of rescuing state through its DMA engine, moving it to the new location. Similarly, an INCA kernel may be able to take over the task of handling checkpoint I/O: rather than having the host application spend time pausing to write out a checkpoint, it could provide an INCA kernel with the memory location of the data to be written out, along with the size, and then continue doing work, letting the NIC (through DMA) stream the checkpoint to the filesystem. We explore additional examples of future directions in runtime offloading afforded by deadline-free approaches such as INCA in [204].

Fourth, in this work we’ve emphasized local INCA kernel execution, i.e., kernels that run on a single NIC. But the underlying compute model is not restricted in this way: rather than looping back to the origin, a message generated by a triggered operation could be sent to a distinct NIC. In principle, then, an INCA program could be distributed across a network, with different NICs responsible for different parts, and perhaps operating in parallel. One possible application for such a distributed execution model is the application of machine learning techniques to network monitoring. INCA kernels might periodically issue messages containing network status data, and as this data flows through the network, it is aggregated and processed by intermediate INCA kernels that classify the statuses. By the time it is delivered to its ultimate destination, the receiving application has actionable intelligence regarding network state.

Finally, the research on offloading machine learning kernels presented in Chapter 8 is just the first step towards adaptive, intelligent networks. There are many more techniques to investigate. While some are likely too computationally or memory intensive (n-grams,
random forests), an intriguing alternative is to treat network traffic as a ‘language’ – e.g. a sequence of symbols representing incoming data or no incoming data – and then applying techniques from natural language processing (NLP) to the task of predicting upcoming symbols. Such an approach need not be especially computationally expensive, e.g. traditional ‘simple recurrent networks’ of the sort used in pioneering research in neural network NLP [205]. Other obvious alternatives include Markov models and simple Bayesian networks, but clearly the space of possibilities is vast and potentially rich.
LIST OF APPENDICES

Appendix A. The INCA-A language ..................................................... 130
Appendix B. The INCA-Q Language .................................................. 136
Appendix A:
The INCA-A language

This appendix includes a description of the INCA-A language.

A.1 INCA-A: Instruction categories

Table 1 summarizes the basic categories of INCA-A instruction. The current implementation includes directives, core, and control instructions. Additional instructions under consideration for future versions are enclosed in square brackets.

A.2 INCA-A: Semantics

In this section we present the semantics for the current version of INCA-A. Instructions include those from the core, control, and data movement categories.

All variables are locations, i.e., \( L \)-values. Locations in memory are designated \( r_i, r_j, r_k \). Angled brackets refer to the contents of a memory location. \( t \) is a datatype; a list of supported types can be found in Appendix B. For remote operations, \( h_i \) is the destination host handle. item \( I \) is the set of valid instruction indices for a program, and \( p, q \in I \) are specific instruction indices. ‘Single’ instructions operate over single instances of a datatype. ‘Multiple’ instructions operate over multiple instances of a datatype, i.e., are SIMD instructions. For these instructions, \( s \) is the size or number of instances operated
<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>Use Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directives</td>
<td>Variable declarations</td>
<td>Required for interpretation</td>
</tr>
<tr>
<td></td>
<td>Variable initializations</td>
<td></td>
</tr>
<tr>
<td>Core</td>
<td>Arithmetic operations</td>
<td>Data manipulation</td>
</tr>
<tr>
<td></td>
<td>Logical operations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bitwise operations</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>Branch (BLEZ)</td>
<td>Turing completeness</td>
</tr>
<tr>
<td></td>
<td>Jump</td>
<td>Application interaction</td>
</tr>
<tr>
<td></td>
<td>Post END</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Post handoff]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Set counter]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Get counter]</td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>Put (local)</td>
<td>Data manipulation</td>
</tr>
<tr>
<td></td>
<td>[Put (remote)]</td>
<td>Distributed programs</td>
</tr>
<tr>
<td>Environment</td>
<td>[Get NIC id]</td>
<td>Collective communication</td>
</tr>
<tr>
<td></td>
<td>[Get NIC addressing mode]</td>
<td>Adaptive networks</td>
</tr>
<tr>
<td></td>
<td>[Get hardware counter X]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Get/set MTU]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Get time]</td>
<td></td>
</tr>
<tr>
<td>Header manipulation</td>
<td>[Get/set initiator]</td>
<td>Traffic control</td>
</tr>
<tr>
<td></td>
<td>[Get/set target]</td>
<td>Rendezvous communications</td>
</tr>
</tbody>
</table>

Table 1: INCA instruction categories. Items enclosed in square brackets are proposed for inclusion in future versions of INCA-A.

over. Finally, instructions marked with an asterisk represent atomic operations not in the current (4.2) Portals network programming API.

1. Arithmetic operations (single)

   (a) \textbf{ADD } r_i, r_j, r_k, t : \text{Add. } \langle r_i \rangle \leftarrow \langle r_j \rangle + \langle r_k \rangle

   (b) \textbf{SUB } r_i, r_j, r_k, t : \text{Subtract. } \langle r_i \rangle \leftarrow \langle r_j \rangle - \langle r_k \rangle

   (c) \textbf{MUL } r_i, r_j, r_k, t : \text{Multiply. } \langle r_i \rangle \leftarrow \langle r_j \rangle * \langle r_k \rangle

   (d) (* \textbf{DIV } r_i, r_j, r_k, t : \text{Divide. } \langle r_i \rangle \leftarrow \langle r_j \rangle / \langle r_k \rangle

   (e) (*) \textbf{EXP } r_i, r_j, t : \text{Exponentiation. } \langle r_i \rangle \leftarrow e^{(r_j)}

   (f) (*) \textbf{SIN } r_i, r_j, t : \text{Sine. } \langle r_i \rangle \leftarrow \sin(\langle r_j \rangle)
(g) (*) **COS** \( r_i, r_j, t \): Cosine. \( \langle r_i \rangle \leftarrow \cos(\langle r_j \rangle) \)

(h) (*) **TAN** \( r_i, r_j, t \): Tangent. \( \langle r_i \rangle \leftarrow \tan(\langle r_j \rangle) \)

2. Arithmetic operations (multiple)

(a) (*) **ADDM** \( r_i, r_j, r_k, t, s \): Add multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle + \langle r_k[m] \rangle
\]

(b) (*) **SUBM** \( r_i, r_j, r_k, t, s \): Subtract multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle - \langle r_k[m] \rangle
\]

(c) (*) **MULM** \( r_i, r_j, r_k, t, s \): Multiply multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle \times \langle r_k[m] \rangle
\]

(d) (*) **DIVM** \( r_i, r_j, r_k, t, s \): Divide multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle / \langle r_k[m] \rangle
\]

3. Bitwise operations (single)

(a) **AND** \( r_i, r_j, r_k, t \): Bitwise AND. \( \langle r_i \rangle \leftarrow \langle r_j \rangle \& \langle r_k \rangle \)

(b) **OR** \( r_i, r_j, r_k, t \): Bitwise OR. \( \langle r_i \rangle \leftarrow \langle r_j \rangle \mid \langle r_k \rangle \)

(c) **XOR** \( r_i, r_j, r_k, t \): Bitwise XOR. \( \langle r_i \rangle \leftarrow \langle r_j \rangle \hat{\&} \langle r_k \rangle \)

(d) (*) **NOT** \( r_i, r_j, t \): Bitwise NOT. \( \langle r_i \rangle \leftarrow \neg \langle r_j \rangle \)

4. Bitwise operations (multiple)

(a) (*) **ANDM** \( r_i, r_j, r_k, t, s \): Bitwise AND multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle \& \langle r_k[m] \rangle
\]

(b) (*) **ORM** \( r_i, r_j, r_k, t, s \): Bitwise OR multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle \mid \langle r_k[m] \rangle
\]

(c) (*) **XORM** \( r_i, r_j, r_k, t, s \): Bitwise XOR multiple \( s \) items.

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \langle r_j[m] \rangle \hat{\&} \langle r_k[m] \rangle
\]
(d) (*) NOTM \( r_i, r_j, t, s \) : Bitwise NOT multiple (s items).

\[ 0 \leq m < s : \langle r_i[m] \rangle \leftarrow \neg \langle r_j[m] \rangle \]

5. Logical operations (single)

(a) **LAND** \( r_i, r_j, r_k, t \) : Logical AND.

\[
\langle r_i \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j \rangle > 0 \text{ and } \langle r_k \rangle > 0 \\
0 & \text{otherwise}
\end{cases}
\]

(b) **LOR** \( r_i, r_j, r_k, t \) : Logical OR.

\[
\langle r_i \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j \rangle > 0 \text{ or } \langle r_k \rangle > 0 \\
0 & \text{otherwise}
\end{cases}
\]

(c) **LXOR** \( r_i, r_j, r_k, t \) : Logical XOR.

\[
\langle r_i \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j \rangle > 0 \text{ or } \langle r_k \rangle > 0 \text{ but not both.} \\
0 & \text{otherwise}
\end{cases}
\]

(d) (*) **LNOT** \( r_i, r_j, t \) : Logical NOT.

\[
\langle r_i \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j \rangle = 0 \\
0 & \text{otherwise}
\end{cases}
\]

6. Logical operations (multiple)

(a) (*) **LANDM** \( r_i, r_j, r_k, t, s \) : Logical AND multiple (s items).

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j[m] \rangle > 0 \text{ and } \langle r_k[m] \rangle > 0 \\
0 & \text{otherwise}
\end{cases}
\]

(b) (*) **LORM** \( r_i, r_j, r_k, t, s \) : Logical OR multiple (s items).

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j[m] \rangle > 0 \text{ or } \langle r_k[m] \rangle > 0 \\
0 & \text{otherwise}
\end{cases}
\]

(c) (*) **LXORM** \( r_i, r_j, r_k, t, s \) : Logical OR multiple (s items).

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \begin{cases} 
1 & \text{if } \langle r_j[m] \rangle > 0 \text{ or } \langle r_k[m] \rangle > 0 \text{ but not both.} \\
0 & \text{otherwise}
\end{cases}
\]
(d) (*) \textsc{LNOTM} \, r_i, r_j, r_k, t, s : Logical NOT multiple \,(s \text{ items}).

\[
0 \leq m < s : \langle r_i[m] \rangle \leftarrow \begin{cases} 1 & \text{if } \langle r_j[m] \rangle = 0 \\ 0 & \text{otherwise} \end{cases}
\]

7. Control flow operations

(a) (*) \textsc{BLEZ} \, r_i, q : Jump if less-than-or-equal-to zero. If \langle r_i \rangle \leq 0, \, pc = q, \, \text{else } \, pc = p + 1

(b) (*) \textsc{JMP} \, q : Jump to instruction index \, q.

8. Data movement operations

(a) \textsc{PUTL} \, r_i, r_j, t : Put local \,(1 \text{ item}). \, \langle r_i \rangle \leftarrow \langle r_j \rangle

(b) \textsc{PUTLM} \, r_i, r_j, t, s : Put local multiple \,(s \text{ items}). \, \langle r_i \rangle \leftarrow \langle r_j \rangle

(c) \textsc{PUTR} \, r_i, r_j, t, h_i : Put remote \,(1 \text{ item}). \, \langle r_i \rangle \leftarrow \langle r_j \rangle, \, \text{where } r_i \text{ is on host } h_i.

(d) \textsc{PUTRM} \, r_i, r_j, t, s, h_i : Put remote multiple \,(s \text{ items}). \, \langle r_i \rangle \leftarrow \langle r_j \rangle, \, \text{where } r_i \text{ is on host } h_i.

9. Comparison operations \,(single)

(a) \textsc{EQ} \, r_i, r_j, r_k, t : Equal local \,(1 \text{ item}). \, \text{If } \langle r_j \rangle = = \langle r_k \rangle > 0, \, \langle k_i \rangle = 1, \, \text{else } \, \langle r_i \rangle = 0.

(b) \textsc{NE} \, r_i, r_j, r_k, t : Not equal local \,(1 \text{ item}). \, \text{If } \langle r_j \rangle \neq \langle r_k \rangle > 0, \, \langle r_i \rangle = 1, \, \text{else } \, \langle r_i \rangle = 0.

(c) \textsc{GE} \, r_i, r_j, r_k, t : Greater than or equal local \,(1 \text{ item}). \, \text{If } \langle r_j \rangle \geq \langle r_k \rangle > 0, \, \langle r_i \rangle = 1, \, \text{else } \, \langle r_i \rangle = 0.

(d) \textsc{GT} \, r_i, r_j, r_k, t : Greater than local \,(1 \text{ item}). \, \text{If } \langle r_j \rangle > \langle r_k \rangle > 0, \, \langle r_i \rangle = 1, \, \text{else } \, \langle r_i \rangle = 0.

(e) \textsc{LE} \, r_i, r_j, r_k, t : Less than or equal local \,(1 \text{ item}). \, \text{If } \langle r_j \rangle \leq \langle r_k \rangle > 0, \, \langle r_i \rangle = 1, \, \text{else } \, \langle r_i \rangle = 0.
(f) LT $r_i, r_j, r_k, t$ : Less than local (1 item). If $\langle r_j \rangle < \langle r_k \rangle > 0$, $\langle r_i \rangle = 1$, else $\langle r_i \rangle = 0$.

(g) MIN $r_i, r_j, r_k, t$ : Minimum local (1 item). $\langle r_i \rangle \leftarrow \min(\langle r_j \rangle, \langle r_k \rangle)$.

(h) MAX $r_i, r_j, r_k, t$ : Maximum local (1 item). $\langle r_i \rangle \leftarrow \max(\langle r_j \rangle, \langle r_k \rangle)$.

10. Comparison operations (multiple)

(a) (*) MINM $r_i, r_j, r_k, t, s$ : Minimum local multiple ($s$ items).
\[ 0 \leq m < s : \langle r_i[m] \rangle \leftarrow \min(\langle r_j[m] \rangle, \langle r_k[m] \rangle) \]

(b) (*) MAXM $r_i, r_j, r_k, t, s$ : Maximum local multiple ($s$ items).
\[ 0 \leq m < s : \langle r_i[m] \rangle \leftarrow \min(\langle r_j[m] \rangle, \langle r_k[m] \rangle) \]

(c) (*) Multiple versions of EQ, NE, GE, GT, LE, and LT are also in INCA-A.

11. Finishing operations

(a) (*) END : Terminate program.

(b) (*) NOP : No operation.
Appendix B:

The INCA-Q Language

This appendix includes the INCA-Q language syntax.

B.1 INCA-Q: Program

An INCA-Q program has two parts: directives defining the execution environment of the kernel, and the kernel code itself. Every INCA-Q program must end with an end or fin keyword. Comments are any valid string, terminated with a newline.

\[
\langle \text{program} \rangle ::= \langle \text{instructions} \rangle \langle \text{end-instruction} \rangle \\
\langle \text{instructions} \rangle ::= \langle \text{instructions} \rangle \langle \text{instruction} \rangle \\
\quad | \langle \text{instruction} \rangle \\
\langle \text{instruction} \rangle ::= \langle \text{directive} \rangle \\
\quad | \langle \text{expression} \rangle \\
\quad | \langle \text{statement} \rangle \\
\quad | \langle \text{comment} \rangle \\
\quad | \epsilon \\
\langle \text{end-instruction} \rangle ::= \text{end} \\
\quad | \text{fin} \\
\langle \text{comment} \rangle ::= \# \ldots
\]
B.2 INCA-Q: Directives

INCA-Q directives define the execution environment of an INCA kernel, i.e., the variables and buffers it will access, as well as their types.

\[
\langle \text{directive} \rangle ::= \% \langle \text{directive-declaration} \rangle \\
| \% \langle \text{directive-array-declaration} \rangle \\
| \% \langle \text{directive-assignment} \rangle \\
| \% \langle \text{directive-wait} \rangle \\
\]

\[
\langle \text{directive-declaration} \rangle ::= \% \langle \text{type} \rangle \langle \text{variable} \rangle \\
\]

\[
\langle \text{directive-array-declaration} \rangle ::= \% \langle \text{type} \rangle \langle \text{variable} \rangle \langle \text{directive-array-dims} \rangle \\
\]

\[
\langle \text{directive-array-dims} \rangle ::= \langle \text{directive-array-dims} \rangle \langle \text{directive-array-dim} \rangle \\
| \langle \text{directive-array-dim} \rangle \\
\]

\[
\langle \text{directive-array-dim} \rangle ::= [ \langle \text{unsigned-integer} \rangle ] \\
\]

\[
\langle \text{directive-assignment} \rangle ::= \% \langle \text{variable} \rangle = \langle \text{integer} \rangle \\
| \% \langle \text{variable} \rangle = \langle \text{float} \rangle \\
\]

\[
\langle \text{directive-wait} \rangle ::= \% \text{wait} \langle \text{unsigned-integer} \rangle \\
\]

B.3 INCA-Q: Expressions

The INCA-Q expressions are given below. This is a compressed version of the grammar for expressions, leaving out the details that enforce operator precedence. Operator precedence, from highest to lowest, is

1. !, ~, exp, sin, cos, tan
2. \% \text{\textbackslash}, * \\
3. +, - \\
4. >, >=, <, <=, ==
5. &, ^, | 

6. &&, ||, min, max.

\[
\langle expression \rangle ::= \langle expression \rangle \mid \langle expression \rangle \\
| \langle expression \rangle \backslash \langle expression \rangle \\
| \langle expression \rangle \ast \langle expression \rangle \\
| \langle expression \rangle \cdot \langle expression \rangle \\
| \langle expression \rangle + \langle expression \rangle \\
| \langle expression \rangle + \langle expression \rangle \\
| \langle expression \rangle < \langle expression \rangle \\
| \langle expression \rangle <= \langle expression \rangle \\
| \langle expression \rangle >= \langle expression \rangle \\
| \langle expression \rangle <\langle expression \rangle \\
| \langle expression \rangle <=\langle expression \rangle \\
| \langle expression \rangle >=\langle expression \rangle \\
| \langle expression \rangle ==\langle expression \rangle \\
| \langle expression \rangle &\langle expression \rangle \\
| \langle expression \rangle ^\langle expression \rangle \\
| \langle expression \rangle |\langle expression \rangle \\
| \langle expression \rangle &&\langle expression \rangle \\
| \langle expression \rangle ||\langle expression \rangle \\
| \langle expression \rangle min\langle expression \rangle \\
| \langle expression \rangle max\langle expression \rangle \\
| ! \langle expression \rangle \\
| \langle expression \rangle \\
| exp( \langle expression \rangle ) \\
| ( \langle expression \rangle ) \\
| \langle variable \rangle \\
| \langle array-element \rangle \\
| \langle number \rangle 
\]
B.4 INCA-Q: Statements

\( \langle \text{statement} \rangle \quad ::= \quad \langle \text{if-statement} \rangle \)
\( \quad \quad \quad | \quad \langle \text{while-statement} \rangle \)
\( \quad \quad \quad | \quad \langle \text{assignment} \rangle \)

\( \langle \text{if-statement} \rangle \quad ::= \quad \text{if} \ \langle \text{expression} \rangle \ \langle \text{then-statement} \rangle \)

\( \langle \text{then-statement} \rangle \quad ::= \quad \text{then} \ \{ \ \langle \text{program-block} \rangle \ \} \)
\( \quad \quad \quad | \quad \text{then} \ \{ \ \langle \text{program-block} \rangle \ \} \ \text{else} \ \{ \ \langle \text{program-block} \rangle \ \} \)

\( \langle \text{while-statement} \rangle \quad ::= \quad \text{while} \ \langle \text{expression} \rangle \ \text{do} \ \{ \ \langle \text{program-block} \rangle \ \} \)

\( \langle \text{assignment} \rangle \quad ::= \quad \langle \text{variable} \rangle = \langle \text{expression} \rangle \)
\( \quad \quad \quad | \quad \langle \text{array-element} \rangle = \langle \text{expression} \rangle \)

\( \langle \text{variable} \rangle \quad ::= \quad [A-Za-z]+[0-9A-Za-z_]* \)

\( \langle \text{array-element} \rangle \quad ::= \quad \langle \text{variable} \rangle \ \langle \text{array-dims} \rangle \)

\( \langle \text{array-dims} \rangle \quad ::= \quad \langle \text{array-dims} \rangle \ \langle \text{array-dim} \rangle \)
\( \quad \quad \quad | \quad \langle \text{array-dim} \rangle \)

\( \langle \text{array-dim} \rangle \quad ::= \quad [ \ \langle \text{expression} \rangle \ \] \)

\( \langle \text{unsigned-integer} \rangle \quad ::= \quad [0-9]+ \)

\( \langle \text{integer} \rangle \quad ::= \quad [+|-] \ \langle \text{unsigned-integer} \rangle \)

\( \langle \text{unsigned-float} \rangle \quad ::= \quad \langle \text{unsigned-integer} \rangle \ . \ \langle \text{unsigned-integer} \rangle \)

\( \langle \text{float} \rangle \quad ::= \quad [+|-] \ \langle \text{unsigned-float} \rangle \)

B.5 INCA-Q: Additional information

INCA-Q includes a no-op function (\texttt{inca\_nop}) to facilitate placeholdering when developing INCA-A kernels that may use functionality not currently included in INCA-Q. In addition
to this function, the following strings are reserved and cannot be used as variable or array names:

\[ \text{min, max, exp, sin, cos, tan} \]

INCA-Q uses types as defined by the atomic operations appearing in the Portals specification, not including complex numbers. These are listed in Table 2.

\begin{center}
\begin{tabular}{ll}
i8 & 8-bit signed integer \\
ui8 & 8-bit unsigned integer \\
i16 & 16-bit signed integer \\
ui16 & 16-bit unsigned integer \\
i32 & 32-bit signed integer \\
ui32 & 32-bit unsigned integer \\
i64 & 64-bit signed integer \\
ui64 & 64-bit unsigned integer \\
f & 32-bit float \\
d & 64-bit float \\
ld & 64-bit float \\
\end{tabular}
\end{center}

Table 2: INCA-Q types.
References


145


[37] ——, (2018). Netronome 25gbe SmartNICs with Open vSwitch Hardware Offload Drive Unmatched Cloud and Data Center Infrastructure Performance, [Online].


[127] D. Sanvito, G. Siracusano, and R. Bifulco, “Can the Network Be the AI Accelerator?” In Proceedings of the 2018 Morning Workshop on In-Network Computing,


