

Network-Compute Co-Design for Distributed In-Memory Computing

THÈSE N° 8749 (2018)

PRÉSENTÉE LE 7 SEPTEMBRE 2018

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DE SYSTÈMES PARALLÈLES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Alexandros DAGLIS

acceptée sur proposition du jury:

Prof. P. lenne, président du jury
Prof. B. Falsafi, Prof. E. Bugnion, directeurs de thèse
Prof. G. Sohi, rapporteur
Dr P. Faraboschi, rapporteur
Prof. J. Larus, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

Τῆς παιδείας ἔφη τὰς μὲν ῥίζας εἶναι πικράς,
τὸν δὲ καρπὸν γλυκύν.
— Ἀριστοτέλης

The roots of education are bitter,
but the fruit is sweet.
— Aristotle

To my family

Acknowledgements

My PhD journey has undoubtedly been the most challenging and, at the same time, most rewarding so far in my life. A journey that transformed me into a better person and taught me the true value of perseverance, team work, empathy, and patience. I would not have been able to reach the finish line if it weren't for the wonderful people around me; people who were always there to magnify the joy of the best moments; people who would always support and help me push through the hardest times. To all of these people, I owe my deepest gratitude. It is therefore apposite to start this thesis by thanking them.

First and foremost, I am grateful to my advisors, Babak and Ed. Babak has been a constant source of stimulation to keep pushing myself out of my comfort zone, a practice instrumental to my success. He taught me the importance of asking the right questions and taking a step back to look at the big picture. Babak's attention to detail and quest for perfection confirms that the "the path of virtues goes through toils". I am grateful for the great group culture Babak has inculcated in the PARSA group, which forms strong ties between students; I truly hope to succeed in achieving the same with my future research group. Finally, I can't help but blame Babak for spoiling me regarding coffee. He made sure that PARSA always had arguably the best espresso at (at least) EPFL. As a consequence, it has become a real challenge to find passable coffee outside the lab... On the bright side, if everything else fails, I've acquired sufficient knowledge about great espresso to become a half-decent barista.

It was a true honor to have Ed as a co-advisor. Thanks to Ed, I was involved in a great research project of much wider breadth than my initial research direction plans, which were limited to a narrower classic computer architecture scope. His astonishingly strong networking and systems background and his immense industry experience have been truly inspiring and have played a

Acknowledgements

key role in broadening my research horizons. I want to thank Ed for that, and for always being exceptionally empathetic, generous, and pragmatic.

Next, I would like to thank Jim Larus, Guri Sohi, Paolo Faraboschi, and Paolo Ienne for the honor of serving in my PhD thesis committee. Hadi Esmaeilzadeh and Abhishek Bhattacharjee have been selflessly offering me invaluable advice and support in academic matters. Boris Grot has been a very close collaborator and mentor, from whom I have learned a great lot. In several ways, he has been like a third advisor to me. Stanko Novakovic and Dmitrii Ustiugov have been close collaborators in most of the work I did as part of my thesis and other exciting research projects, but have also been great friends. Thank you for making my PhD journey more fruitful and exciting!

An integral part of academia is continuous learning and bequeathing that acquired knowledge through teaching. For my academic inclination I am greatly indebted to the great teachers I've had throughout my life. I have been fortunate enough to have had several great teachers, who constantly inspired me to pursue knowledge and excellence. I would like to explicitly thank two of them here. First, my Computer Architecture professor and mentor throughout my undergraduate studies at NTUA, Nectarios Koziris. In addition to giving me invaluable advice on how to start building a successful career early on, he is also an excellent teacher whose enthusiasm and positivity inspired my passion for Computer Architecture. Second, I would like to deeply thank Giorgos Despotidis. Being an excellent violin teacher was the least of his qualities; a man of exemplary kindness and dignity, with a deep love for his discipline, he has been a role model for me in several aspects. Maestro, may you rest in peace. You are deeply missed...

The next group of people that deserves my gratitude is the PARSA lab. The strong collaborative group culture was among the best experiences during this PhD. My daily interactions with my peers have been a great source of learning. First, I'd like to thank Javier Picorel, my office mate for many years, with whom we've been through a lot: occasions good and bad, funny and sad. His positivity and support helped me keep my sanity. We had a really good run; I often reminisce about all the jokes and fun times we shared. Next, I'd like to thank Mario Drumond and Arash Pourhabibi for being not only great colleagues, but also awesome friends and neighbors. They have been a lot of fun to hang out and argue with. I'm grateful to Sotiria Fytraki and the "fantastic

four"—Stavros Volos, Onur Kocberber, Cansu Kaynak, and Djordje Jevdjic—who were senior PhD students when I first joined PARSA. I bugged them a lot, but also learned *a lot* from them. I will always very fondly remember our PARSA ski trips together and the legendary Las Vegas excursion after ASPLOS 2014. Finally, I would like to thank Mark Sutherland—on whom befell the demanding task of replacing Javier as my office mate, Sid Gupta—also my gym buddy who made sure I never skipped leg day, Hussein Kassir—our official FPGA prototype-er, Nooshin Mirzadeh, and Zilu Tian.

I want to thank PARSA's administrative and technical staff, for always offering top-quality support above and beyond their duty. Stéphanie has always been extremely friendly and helpful with every minor or major headache related to bureaucracy, event organization, French translation, etc., making sure the rest of us can focus on our academic goals and duties. Rodolphe, our remarkable sysadmin, made sure all lab compute infrastructure was running like clockwork, and was always available and responsive in the most critical situations (e.g., dealing with the joys of a whole cluster going down on a Saturday night, just a few days before a deadline).

I've been fortunate to have an amazing group of people to spend my limited leisure time with in Lausanne: enjoying good beer, watching movies, attending the long-established Burger Nights, or, of course, having our infamous philosophical discussions of critical importance regarding what constitutes a computer or a root or whether infinite sentences are a thing (yes, this is as confusing as it sounds). I want to thank Manos, Eleni, Pavlos, Nathalie, and Christos for making Lausanne feel like home. Of course, nothing would have been the same without the jolly Greek and EPFL gang: Stefanos, Natassa, Iraklis, Vasilis, Panagiotis, Stella, Matt, Onur, Jean, Farah, Apostolis, Thodoris, Myrsini, Katerina, Loukia, Iliana.

I am very grateful for to my friends from the good old times back in Greece, who are now spread out all over the world. Our emotional proximity makes up for the petty inconvenience of physical distance; the memories of all the great times we've had together are lifelong companions and sources of joy. I want to thank all of of these cherished friends. My dear high-school friends, Orestis, Vilma, Andreas, and Thodoris. My childhood friends Xenofon and Vasilis. My close friends and once-upon-a-time neighbors, Nicholas and Despina (a.k.a. Cuervo, and also the closest I've had to a sister) . The "tsouvlia" team from the Rosarte choir: Thanos, Sofia, Sofia

Acknowledgements

Jr., Miranda. The NTUA gang: Leonidas, Mary, Nikos, Ignatios, Ersi. Rea, for her friendship, wisdom, support, and invaluable advice over the past decade. I am looking forward to happy get-togethers around the globe for years to come!

Last but not least, I want to thank my family, from the bottom of my heart, for their endless love, support, and encouragement, during my PhD and my whole life. My parents, Ioannis and Anna, for eagerly offering me more than I could ever ask for; for bringing me up in a loving environment; for teaching me all things important: principles, ethics, justice, empathy, gratitude, honesty... and so much more. I could not have asked for better parents, and for that I am extremely fortunate and grateful. My dearest brothers, Thanasis and Dimitris, have been the best company to grow up with. I am very proud of you both and feel blessed to have you. My grandparents—Alexandros, Anthi, Thanasis, Vasiliki—for living a hard life to provide a better future for their children and grandchildren. My dear uncles, Stathis, Fotis, and Dimitris; my aunt Maria, who has been like second mother, and my cousins Eleni and Theofanis for their unconditional love. Finally, my very own person and partner in crime, Kyveli. You have been my safe haven, inspiration and joy for the past decade, and I am looking forward to spending a lifetime with you. I love you.



This thesis would not have been possible without numerous funding sources. I am thankful to Babak for making sure I never had to worry about funding. My PhD research has been partially supported by an EPFL Fellowship, a Microsoft Research Fellowship, the *EuroCloud* project of the 7th Framework Program of the European Commission, the *Workloads and Server Architecture for Green Datacenters* project of the Swiss National Science Foundation, the Nano-Tera *YINS* project, the *Scale-Out NUMA* project of the Microsoft-EPFL Joint Research Center, and the CHIST-ERA *DIVIDEND* project.

Lausanne, August 1, 2018

A. D.

Abstract

The booming popularity of online services is rapidly raising the demands for modern datacenters. In order to cope with data deluge, growing user bases, and tight quality of service constraints, service providers deploy massive datacenters with tens to hundreds of thousands of servers, keeping petabytes of latency-critical data memory resident. Such data distribution and the multi-tiered nature of the software used by feature-rich services results in frequent inter-server communication and remote memory access over the network. Hence, networking takes center stage in datacenters.

In response to growing internal datacenter network traffic, networking technology is rapidly evolving. Lean user-level protocols, like RDMA, and high-performance fabrics have started making their appearance, dramatically reducing datacenter-wide network latency and offering unprecedented per-server bandwidth. At the same time, the end of Dennard scaling is grinding processor performance improvements to a halt. The net result is a growing mismatch between the per-server network and compute capabilities: it will soon be difficult for a server processor to utilize all of its available network bandwidth.

Restoring balance between network and compute capabilities requires tighter co-design of the two. The network interface (NI) is of particular interest, as it lies on the boundary of network and compute. In this thesis, we focus on the design of an NI for a lightweight RDMA-like protocol and its full integration with modern manycore server processors. The NI capabilities scale with both the increasing network bandwidth and the growing number of cores on modern server processors.

Leveraging our architecture's integrated NI logic, we introduce new functionality at the network endpoints that yields performance improvements for distributed systems. Such additions include

Abstract

new network operations with stronger semantics tailored to common application requirements and integrated logic for balancing network load across a modern processor's multiple cores. We make the case that exposing richer, end-to-end semantics to the NI is a unique enabler for optimizations that can reduce software complexity and remove significant load from the processor, contributing towards maintaining balance between the two valuable resources of network and compute. Overall, network-compute co-design is an approach that addresses challenges associated with the emerging technological mismatch of compute and networking capabilities, yielding significant performance improvements for distributed memory systems.

Key words: datacenters, servers, network interface, network protocol, integration, co-design, one-sided operations, RDMA, distributed memory, remote memory

Zusammenfassung

Die wachsende Popularität von Online-Diensten erhöht die Nachfrage nach modernen Rechenzentren rasant. Um mit Datenflut, wachsenden Benutzerzahlen und strikten Servicequalität-Einschränkungen zurechtzukommen, stellen Service-Provider massive Rechenzentren mit zehntausenden bis hunderttausenden von Servern bereit, die Petabytes von latenzkritischen Datenspeichern resident halten. Eine solche Datenverteilung und die mehrstufige Natur der Software, die von funktionsreichen Diensten verwendet wird, führt zu einer häufigen Inter-Server-Kommunikation und einem Fernspeicherzugriff über das Netzwerk. Daher steht das Netzwerk in Rechenzentren im Mittelpunkt.

Als Reaktion auf den wachsenden internen Datenverkehr in Rechenzentren entwickelt sich die Netzwerktechnologie rasant. Leichte Benutzerebene Protokolle, wie RDMA, und High-Performance-Fabrics haben Einzug gehalten, wodurch die Rechenzentrum-weite Netzwerklatenz dramatisch reduziert und eine noch nie dagewesene Datenübertragungsrate pro Server geboten wird. Gleichzeitig bringt das Ende der Dennard-Skalierung die Prozessorleistungverbesserungen zum Stillstand. Das Endergebnis ist eine wachsende Diskrepanz zwischen den pro-Server Netzwerkfähigkeiten und Rechenfähigkeiten: Es wird für einen Server-Prozessor bald schwierig sein, die gesamte verfügbare Datenübertragungsrate des Netzwerks zu nutzen.

Die Wiederherstellung des Gleichgewichts zwischen Netzwerk- und Rechnerleistung erfordert ein engeres Co-Design der beiden. Die Netzwerkschnittstelle (NS) ist von besonderem Interesse, da sie auf der Grenze von Netzwerk und Rechner liegt. In dieser Arbeit konzentrieren wir uns auf das Design einer NS für ein leichtes RDMA-ähnliches Protokoll und dessen vollständige Integration in moderne Mehrkern-Server-Prozessoren. Die NS-Funktionen skalieren sowohl mit der steigenden Netzwerk-Datenübertragungsrate, als auch mit der wachsenden Kernanzahl

Zusammenfassung

moderner Server-Prozessoren.

Den Vorteil der NS-Integration unserer Architektur ziehend, führen wir neue Funktionalität an den Netzwerkendpunkten ein, die für verteilte Systeme Leistungsverbesserungen bringt. Solche Ergänzungen beinhalten neue Netzwerkoperationen mit stärkerer Semantik, die auf allgemeine Anwendungsanforderungen zugeschnitten sind, sowie auch integrierte Logik zum Ausgleichen der Netzwerklast über die mehreren Kerne moderner Prozessoren. Wir zeigen, dass die Bereitstellung umfassenderer End-to-End-Semantiken für NS ein einzigartiger Ermöglicher für Optimierungen ist, der die Softwarekomplexität reduzieren kann und erhebliche Lasten aus dem Prozessor entfernen kann. Dadurch kann sich das Gleichgewicht zwischen den beiden wertvollen Ressourcen Netzwerk und Rechenleistung aufrechterhalten. Das Co-Design von Netzwerk-Computing ist ein Ansatz, der die Herausforderungen im Zusammenhang mit der sich abzeichnenden technologischen Diskrepanz zwischen Rechen- und Netzwerkfähigkeiten anspricht und zu erheblichen Verbesserungen der Rechenzentrumsleistung führt.

Stichwörter: Rechenzentren, Netzwerkschnittstelle, Netzwerkprotokoll, Integration, Co-Design, einseitige Operationen, RDMA, verteilter Speicher, Remotespeicher

Contents

| | |
|---|------------|
| Acknowledgements | i |
| Abstract (English/Deutsch) | v |
| List of figures | xv |
| List of tables | xix |
| 1 Introduction | 1 |
| 1.1 Forms of Inter-Server Communication | 3 |
| 1.2 Thesis Goals | 4 |
| 1.3 Thesis Contributions | 5 |
| 1.4 Thesis Organization | 7 |
| 1.4.1 Bibliographic Notes | 8 |
| 2 Application and Technology Trends | 9 |
| 2.1 Datacenter Services | 9 |
| 2.2 Server Architectures | 10 |
| 2.3 RDMA and Lossless Fabrics | 11 |
| 2.4 Rack-Scale Computing | 12 |
| 2.5 The Emerging Network–Compute Mismatch | 14 |
| 2.6 One-Sided Operations Versus RPCs | 15 |
| | ix |

| | | |
|----------|---|-----------|
| I | ICONIC Architecture Features | 19 |
| 3 | The Scale-Out NUMA Architecture | 21 |
| 3.1 | Obstacles to Fast Remote Memory | 23 |
| 3.2 | Scale-Out NUMA Overview | 24 |
| 3.3 | Remote Memory Controller | 26 |
| 3.3.1 | Hardware/Software Interface | 26 |
| 3.3.2 | RMC Overview | 27 |
| 3.4 | Communication Protocol | 30 |
| 3.4.1 | Handling Packet Loss | 31 |
| 3.5 | Software Support | 33 |
| 3.5.1 | Device Driver | 33 |
| 3.5.2 | Access Library | 34 |
| 3.5.3 | Synchronization Library | 36 |
| 3.5.4 | Messaging over One-Sided Operations | 36 |
| 3.6 | RMC Microarchitecture | 38 |
| 3.6.1 | Shared SRAM Structures | 41 |
| 3.6.2 | Request Generation Pipeline | 42 |
| 3.6.3 | Request Completion Pipeline | 42 |
| 3.6.4 | Remote Request Processing Pipeline | 44 |
| 3.6.5 | RMC Area and Power Estimation | 45 |
| 3.7 | Chapter Summary | 48 |
| 4 | Remote Memory Operations with Richer Semantics | 49 |
| 4.1 | Atomic Remote Object Reads | 52 |
| 4.1.1 | In-Memory Object Stores | 52 |
| 4.1.2 | Atomic One-Sided Operations | 53 |
| 4.1.3 | Implications of Faster Networking | 55 |
| 4.1.4 | The Case for SABRe | 56 |
| 4.2 | SABRe Design Space | 57 |

| | | |
|-----------|--|-----------|
| 4.2.1 | Destination-Side Concurrency Control | 57 |
| 4.2.2 | Design Goals | 59 |
| 4.2.3 | Safely Overlapping Lock and Data Access | 61 |
| 4.3 | LightSABRe | 62 |
| 4.3.1 | Address Range Snooping Implementation | 63 |
| 4.3.2 | System Integration | 65 |
| 4.4 | LightSABRe on Scale-Out NUMA | 68 |
| 4.4.1 | Integration with RRPP | 69 |
| 4.4.2 | Other Protocol and Hardware Modifications | 70 |
| 4.5 | Chapter Summary | 71 |
| 5 | Integrated Tail-aware Load Balancing | 73 |
| 5.1 | Theoretical Load Balancing Implications | 76 |
| 5.2 | Load Balancing in Practice | 79 |
| 5.3 | Towards Dynamic Load Balancing | 80 |
| 5.4 | Native Messaging | 82 |
| 5.4.1 | Additional Benefits of Native Messaging | 87 |
| 5.5 | Dynamic Load Balancing Design | 88 |
| 5.6 | soNUMA Extensions for Dynamic Load Balancing | 89 |
| 5.7 | Chapter Summary | 91 |
| II | Implementation and Evaluation of an ICONIC Architecture | 93 |
| 6 | Manycore Chip Design | 95 |
| 6.1 | Key Design Considerations | 98 |
| 6.1.1 | Application Requirements and Technology Trends | 98 |
| 6.1.2 | QP-Based Interface for Remote Memory Access | 99 |
| 6.2 | Manycore Network Interfaces | 100 |
| 6.2.1 | Conventional Edge-Based NI | 101 |
| 6.2.2 | Per-Tile NI | 104 |

Contents

| | | |
|----------|---|------------|
| 6.2.3 | Split NI | 105 |
| 6.2.4 | NI Cache | 105 |
| 6.3 | A Case Study with Scale-Out NUMA | 108 |
| 6.3.1 | soNUMA NI Scaling and Placement | 108 |
| 6.3.2 | Other Design Issues | 110 |
| 6.4 | Methodology | 112 |
| 6.5 | Evaluation | 114 |
| 6.5.1 | Latency Characterization | 114 |
| 6.5.2 | Bandwidth Characterization | 118 |
| 6.5.3 | Effect of Latency-Optimized Topology | 120 |
| 6.6 | Chapter Summary | 123 |
| 7 | LightSABRe in Action | 125 |
| 7.1 | Manycore NI Architecture Implications on LightSABRe | 125 |
| 7.2 | Methodology | 126 |
| 7.3 | Evaluation | 129 |
| 7.3.1 | Latency and Throughput Characterization | 129 |
| 7.3.2 | Conflict Sensitivity | 130 |
| 7.3.3 | FaRM Key-Value Store | 131 |
| 7.4 | Chapter Summary | 134 |
| 8 | Tail-Aware Balancing of μS-Scale RPCs | 135 |
| 8.1 | Manycore NI Design Implications | 135 |
| 8.1.1 | 4 \times 4 Queuing System | 136 |
| 8.1.2 | 1 \times 16 Queuing System | 137 |
| 8.2 | Methodology | 139 |
| 8.2.1 | Load Balancing | 139 |
| 8.2.2 | Messaging | 141 |
| 8.3 | Evaluation | 142 |
| 8.3.1 | Load Balancing: Hardware Queuing Systems | 142 |

| | | |
|--------------------------------------|---|------------|
| 8.3.2 | Comparison to Queuing Model | 143 |
| 8.3.3 | Hardware Versus Software Load Balancing | 145 |
| 8.3.4 | Messaging Performance | 146 |
| 8.3.5 | Messaging Memory Requirements | 147 |
| 8.4 | Chapter Summary | 149 |
| III Related and Future Work | | 151 |
| 9 Related Work | | 153 |
| 9.1 | soNUMA and NI Integration | 153 |
| 9.1.1 | Partitioned Global Address Space | 153 |
| 9.1.2 | Software Distributed Shared Memory | 154 |
| 9.1.3 | Cache-Coherent NUMA | 154 |
| 9.1.4 | User-Level Messaging | 155 |
| 9.1.5 | Remote Memory Access | 156 |
| 9.1.6 | Coherent NI Integration | 157 |
| 9.2 | Hardware Support for Atomic Remote Object Reads | 158 |
| 9.2.1 | Hardware-Software Contract | 158 |
| 9.2.2 | Atomic Chunk Operations | 159 |
| 9.2.3 | Memory Subsystem Support | 159 |
| 9.2.4 | SABRe: One-Sided Operation or RPC? | 160 |
| 9.2.5 | Destination-Side Concurrency Control | 160 |
| 9.3 | Load Balancing | 161 |
| 9.3.1 | Load Distribution and Imbalance | 161 |
| 9.3.2 | Load Balancing Policies | 162 |
| 9.3.3 | Programmable NIs | 164 |
| 10 Future Research Directions | | 167 |
| 10.1 | Hardware Heterogeneity in Datacenters | 167 |
| 10.2 | Dynamic Load Balancing Extensions | 169 |

Contents

| | | |
|-----------|---|------------|
| 10.2.1 | Advanced Load Balancing | 170 |
| 10.2.2 | Proactive Versus Reactive Load Balancing | 171 |
| 10.2.3 | Scaling to CPUs with Hundreds of Cores | 172 |
| 10.2.4 | Load Balancing Opportunities with PCIe-Attached NIs | 173 |
| 11 | Conclusion | 175 |
| | Bibliography | 177 |
| | Curriculum Vitae | 201 |

List of Figures

| | | |
|------|--|----|
| 2.1 | Overview of a rack-scale computer. | 13 |
| 3.1 | Netpipe benchmark on a Calxeda microserver. | 23 |
| 3.2 | soNUMA overview. | 25 |
| 3.3 | QP interactions and memory access on the remote end for a remote read. | 27 |
| 3.4 | Functionality of the RMC pipelines. The notation ‘ <i>L</i> ’ next to a stage indicates local processing in combinational logic; ‘ <i>T</i> ’ indicates a TLB access; the rest of the states access memory via an MMU. | 28 |
| 3.5 | Communication protocol for a remote read. | 31 |
| 3.6 | Computation of a PageRank superstep in soNUMA through a combination of remote memory accesses (via the asynchronous API) and local shared memory. | 35 |
| 3.7 | Messaging emulation using one-sided operations. | 37 |
| 3.8 | RMC interface to the on-chip and off-chip network. | 39 |
| 3.9 | The Request Generation Pipeline (RGP). SRAM structures appear shaded. | 43 |
| 3.10 | The Request Completion Pipeline (RCP). SRAM structures appear shaded. | 44 |
| 4.1 | End-to-end remote object read latency using the per-cache-line-version software atomicity check mechanism on FaRM over soNUMA. | 57 |
| 4.2 | Reader–Writer race example. | 60 |
| 4.3 | LightSABRe: Leveraging stream buffers to safely overlap lock & data access. | 64 |
| 4.4 | Block diagram of a LightSABRe-enhanced NI. | 66 |
| 4.5 | soNUMA overview. | 68 |

List of Figures

| | | |
|------|---|-----|
| 5.1 | Different queuing models for 16 serving units (CPU cores). $P(\lambda)$ stands for Poisson arrival distribution. | 76 |
| 5.2 | Service time distributions. | 77 |
| 5.3 | Throughput vs. tail latency for different queuing systems and service time distributions. | 78 |
| 5.4 | Inter-packet interleavings of arriving multi-packet messages. | 82 |
| 5.5 | Messaging illustration: Sender. Boxes marked as c_i represent CPU cores. . . . | 84 |
| 5.6 | Messaging illustration: Receiver. Boxes marked as c_i represent CPU cores. . . | 85 |
| 5.7 | Messaging mechanism extensions for load balancing. | 88 |
| 5.8 | Extensions of soNUMA's RRPP for load balancing support. | 90 |
| 6.1 | QP-based remote read. | 100 |
| 6.2 | Core and NI WQ interactions on an NI_{edge} design. | 101 |
| 6.3 | $NI_{per-tile}$ design. | 104 |
| 6.4 | NI_{split} design. | 105 |
| 6.5 | NI cache coherence state diagram. | 107 |
| 6.6 | Logical separation of soNUMA's RGP and RCP into a frontend and a backend. . . | 109 |
| 6.7 | Projection of the end-to-end latency of a cache-block remote read operation for multiple intra-rack network hops. Bars map to the right y-axis, lines to the left y-axis. | 116 |
| 6.8 | End-to-end latency for synchronous remote reads. | 117 |
| 6.9 | Application bandwidth for asynchronous remote reads. | 119 |
| 6.10 | NI design space for NOC-Out-based manycore CMPs. Striped rectangles represent LLC tiles. | 120 |
| 6.11 | Latency for synchronous remote reads on NOC-Out. | 122 |
| 6.12 | Application bandwidth for asynchronous remote reads on NOC-Out. | 123 |
| 7.1 | Multicore chip layout based on NI_{split} architecture. | 126 |
| 7.2 | Microbenchmark with one-sided operations. | 129 |
| 7.3 | Application throughput with increasing conflict rate. | 131 |

| | |
|---|-----|
| 7.4 FaRM KV store: baseline versus LightSABRe. | 132 |
| 7.5 FaRM local reads throughput comparison. | 134 |
| 8.1 Message dispatch groups on a multicore chip. | 137 |
| 8.2 Modeled service time distributions. | 140 |
| 8.3 Load balancing with three different queuing system implementations in hardware. | 142 |
| 8.4 Performance of three hardware load balancing implementations (1×16 , 4×4 , 1×16) as compared to a theoretical queuing model, for four service time distributions: fixed, uniform, exponential, GEV. Tail latency shown as a multiple of the average service time S | 144 |
| 8.5 Load balancing performance of a 1×16 queuing system: Hardware vs. software implementation. | 146 |
| 8.6 Messaging latency. | 147 |
| 8.7 Total memory footprint of messaging mechanism for different cluster and message sizes (64B, 256B, 1KB, 4KB). | 149 |

List of Tables

| | | |
|-----|--|-----|
| 3.1 | Estimated area and power for RMC SRAM structures. | 45 |
| 4.1 | Design space for one-sided atomic object reads. | 58 |
| 5.1 | Throughput loss of different queuing systems at target 99th percentile tail latency as compared to 1×16 , for different service time distributions. | 79 |
| 6.1 | Latency comparison of a QP-based model and a pure load/store interface. | 103 |
| 6.2 | System parameters for simulation on Flexus. | 112 |
| 6.3 | Zero-load latency breakdown of a single-block remote read. | 115 |
| 6.4 | Zero-load latency breakdown of a single-block remote read (NOC-Out). | 121 |
| 7.1 | Flexus simulation parameters for LightSABRe on soNUMA. | 127 |
| 8.1 | Estimation of buffering slots required for a peak target messaging throughput of 100Gbps, as a function of message size. | 148 |

1 Introduction

Modern online services have gradually become an integral part of everyday life for billions of users. Web search, email, social networking, and e-commerce are a few examples of such popular massive-scale services. At the time of writing, Google claims 1 billion Gmail users and 1.5 billion search engine users, servicing over 3.5 billion search queries per day [42, 84]. Facebook has over 1.4 billion daily and 2.1 billion monthly active users [58], while massive-scale online retailers Amazon and Alibaba receive online orders corresponding to 3 and 12 million daily item shipments, respectively [172]. With every user constantly generating data and each user request probing data services handling petabytes of data, data access demands are growing dramatically. To cope with such data and userbase deluge, online service providers deploy several massive-scale datacenters, each populated with tens of thousands of servers.

In addition to the challenge of immense volume, online services have to be *interactive*, delivering seamless high-quality experience to all users; failing to do so may result in customer loss. Prior work has shown that users are sensitive to response latencies in the orders of hundreds of milliseconds [86, 131]. Experiences from real-life commercial settings corroborate this observation and highlight the dramatic impact of latency in company revenue: every 100ms of latency costs Amazon 1% in sales, while an extra 500ms in search page generation time drops traffic to Google by 20% [77]. It is therefore common for online service providers to set strict latency boundaries for servicing user requests as part of their service's quality metric, commonly

Chapter 1. Introduction

referred to as *Service Level Objectives* (SLO). To deal with such demands for low latency, it has become common practice for service providers to distribute the data across the memory of the datacenter's servers.

Keeping data memory resident removes the bottleneck of disk accesses, accelerating data access by up to five orders of magnitude (100ns versus 10ms). However, data distribution across thousands of servers unavoidably results in accesses to data residing in the memory of remote servers, thus requiring inter-server communication. Using commodity networking technology, servers and operating systems, communication delays can exceed $100\mu\text{s}$ [149]; hence, accessing data in remote memory is $1000\times$ more expensive than accessing local memory ($100\mu\text{s}$ versus 100ns). For the most challenging applications traversing large data structures that cannot be easily partitioned (e.g., graphs) or accessing many disparate pieces of data (e.g., key-value stores), distributed computation results in frequent inter-server communication, which may easily dominate the total time required to process a user request. Therefore, inter-server communication within the datacenter becomes a first-order performance concern.

The importance of communication has resulted in fast datacenter network infrastructure evolution. Advanced networking technologies such as high-performance lossless fabrics (e.g., InfiniBand) and Remote Direct Memory Access (RDMA) [124] that would typically only appear in High-Performance Computing environments have started penetrating the datacenter space as well, promising dramatic improvements in network bandwidth and latency. Modern fabrics continue improving network bandwidth, in contrast to silicon, whose seamless density scaling met an abrupt slowdown with the end of Dennard scaling. Datacenters already feature 10Gbps Ethernet, with 40Gbps already ramping up and 100Gbps just around the corner. InfiniBand, while still more expensive than Ethernet, already offers up to 300Gbps (InfiniBand EDR) and will soon double that (InfiniBand HDR) [81]. On the latency front, the evolution of optics and introduction of cut-through switches has enabled datacenter traversals in just a few tens of microseconds. Further down the line, advancements in silicon photonics foreshadow end-to-end optic communication, which could enable datacenter-wide communication in just a couple of microseconds, ultimately approaching fundamental bounds set by the speed of light. Overall, the dramatic improvement of

raw network performance capabilities lay the groundwork for large-scale distributed memory systems of unprecedented performance. However, reaping these network capabilities requires a major rethink of software, network protocols, and hardware architectures. In this thesis, we focus on protocol and architecture redesign for communication-intensive distributed memory systems.

1.1 Forms of Inter-Server Communication

The majority of modern large-scale distributed memory systems, such as datacenters, is deployed in a scale-out fashion. The size of the system grows with the addition of more servers that tap into the system’s network, and each server deploys its own OS instance managing its local resources (e.g., CPU, memory, storage). The most typical form of inter-server communication in such scale-out deployments is Remote Procedure Calls (RPCs), invoked over the network. RPCs are a very versatile form of inter-server communication, which has established them as the lingua franca of datacenters; all the internal services in modern datacenters communicate via RPCs. For instance, every user request for a Google service triggers more than 1000 RPCs within the datacenter [18, 91].

RDMA technology that has recently started appearing in the datacenter space, introduces an additional form of communication. As the name implies, RDMA—Remote Direct Memory Access—enables a server to directly read a remote server’s memory. Unlike RPCs, this is a *one-sided operation*, i.e., it does not involve the remote end’s CPU. One-sided operations come with simple memory access semantics and provide the opportunity to expose the aggregate memory resources of a scale-out deployment as a single global memory pool. The capability of direct access to a global memory pool brings back to scale-out architectures some of the features of scale-up architectures, without the drawbacks associated with the latter (e.g., cost, single-OS limitations, hurdles of verification and fault containment). Memory pooling enables faster access to remote data, lower memory overprovisioning requirements, and stronger resilience to load imbalance arising from skewed data popularity distributions [133, 136].

Despite their strengths, one-sided operations are semantically limited to simple remote memory

access. Hence, they cannot generally replace the versatile RPCs as the sole form of inter-server communication. Each of the two communication forms has its own merits and drawbacks, which we discuss in further detail in Section 2.6. We expect that future distributed systems will eventually deploy an appropriate combination of one-sided operations and RPCs, leveraging the strengths of each.

1.2 Thesis Goals

The primary goal of this thesis is the drastic acceleration of inter-server communication in distributed memory environments. We aim to offer substantial improvements for both major communication models, one-sided operations and RPCs. To that end, we investigate the limits of inter-server communication latency and the impact of network evolution on the design of future server chips and the network stack itself, from the protocol layer down to hardware.

We start by focusing on one-sided operations and pursue a holistic system design to approach the lower latency bounds of remote memory access. We find that the evolution of networks has shifted the bottlenecks of inter-server communication from the physical network itself to the higher layers of the stack that comprises networking. Particularly, we identify conventional deep network stacks and the slow PCIe bus connecting the CPU to the Network Interface (NI) logic as the last major obstacles to low-latency inter-server communication. To overcome the first latency obstacle, we design a lightweight user-level and hardware-terminated protocol. In turn, the protocol's simplicity enables the design of a simple enough protocol controller that allows full on-chip of the NI logic, enabling rapid CPU-NI interaction. We show that NI integration not only accelerates existing forms of communication, but also opens up new opportunities for network-compute co-design that improves the efficiency of both one-sided- and RPC-based inter-server communication.

Thesis statement:

Network interface integration and co-design with compute logic enables network endpoint operations with richer functionality and stronger semantics, resulting in significant performance improvements for distributed memory systems.

In this thesis, we advocate architectures leveraging **I**ntegration and **CO**-design of **N**etwork **I**nterface and **C**ompute logic (**ICONIC** architectures) as new building blocks capable of significantly boosting the performance of communication-intensive distributed memory systems.

1.3 Thesis Contributions

This thesis introduces network-compute co-design and network interface integration as key design aspects to drastically improve the performance and versatility of communication-intensive distributed memory systems. We introduce basic design guidelines for an **ICONIC** architecture and demonstrate a number of new features such an architecture can deliver. We then implement a proof-of-concept instance of such an architecture and demonstrate its benefits.

First, we propose Scale-Out NUMA (soNUMA), a new architecture, programming model, and communication protocol that enables fast remote memory access by eliminating the last remaining major obstacles to low latency, namely the deep network stack and the slow interface between the CPU and the network. The heart of soNUMA is its on-chip integrated NI implementing soNUMA's protocol controller logic. The NI is not only integrated on chip, but also taps into its local CPU's coherence domain, which serves as a mechanism for rapid CPU-NI interaction. soNUMA is a representative of an **ICONIC** architecture—featuring an NI tightly coupled with compute logic—that serves as an appropriate baseline to demonstrate the new opportunities arising from network-compute integration and co-design.

Second, motivated by the vast semantic gap between one-sided operations and RPCs, we advocate the introduction of new one-sided primitives with richer semantics. As a concrete proposal of such a primitive, we identify an operation that is ubiquitously used by modern distributed object

Chapter 1. Introduction

stores, yet performed in a surprisingly inefficient manner in existing systems: atomic object reads from remote memory. We introduce *SABRe*, a new one-sided operation with the semantics of an *atomic remote object read*, and detail all the protocol and hardware additions required to support it. We demonstrate that the new SABRe operation yields significant performance gains and software simplification for distributed object stores.

Third, we show that the proximity of NI and compute logic in ICONIC architectures opens opportunities for dynamic load balancing mechanisms integrated as part of the NI logic. Such hardware support at the NI delivers significant throughput improvements under tight response time tail latency constraints for the most challenging, short-lived RPCs, where existing software solutions are unable to react to load imbalances in a timely manner. We show that such a dynamic load balancing solution outperforms pre-existing adaptive software-based or static hardware-based load balancing mechanisms, by being the only solution that breaks the tradeoff between load imbalance resulting from static load distribution decisions and synchronization overheads of software-based load balancing practices.

Fourth, we address practical chip design challenges that arise when considering practical implementations of ICONIC architectures, which have to accommodate for the modern technological realities of growing CPU core counts and network bandwidth per server. We find that obvious approaches to scaling and integrating the NI in a manycore chip significantly hurt either latency or bandwidth. In contrast, careful splitting of NI functionality into core-NI interaction and data transfer, and independent scaling and placement of these two components enables an NI design that optimizes for both latency and bandwidth. Based on that insight, we propose NI_{split} , a novel scalable NI design that outperforms alternative NI designs in both latency and bandwidth. Importantly, our NI design study demonstrates that the performance of remote memory access is primarily dictated by chip design choices rather than the hardware/software interface used to initiate remote data transfers. A specialized load-store interface for direct remote memory accesses is *neither necessary nor sufficient* for high performance. A less intrusive hardware/software interface based on a set of memory-mapped queues, when combined with proper chip and NI design, is equally competitive.

Finally, we describe a concrete implementation of an ICONIC architecture based on the soNUMA protocol and our scalable chip design with NI_{split} , featuring our new SABRes primitive and our integrated dynamic load balancing mechanism. Our evaluation of the system demonstrates significant performance improvements, in terms of both latency and throughput, showcasing the strengths of tight NI integration and network-compute integration and co-design.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides background on key application and technology trends that necessitate a rethink in the way we design large-scale communication-intensive distributed memory systems, motivating a network-centric design approach. The rest of the thesis is organized in three parts:

- **Part I** introduces the key design principles of an ICONIC architecture, and a set of new features they offer. Chapter 3 presents the Scale-Out NUMA protocol and its specialized on-chip integrated NI. Chapter 4 propose SABRes, a new one-sided operation with rich semantics offering the capability reading objects from remote memory atomically. Chapter 5 introduces a novel dynamic load balancing mechanism of incoming network messages to CPU cores, integrated in the NI logic, demonstrating unique benefits of ICONIC architectures in RPCs handling.
- **Part II** is focused on the implementation and evaluation of an ICONIC architecture, based on the design presented in Part I. Chapter 6 introduces a novel chip design that addresses the practical challenges of scaling the performance of an on-chip integrated NI with the evolving capabilities of modern servers, in terms of CPU core counts and network bandwidth. Building on top of that chip design, Chapters 7 and 8 implement and evaluate the performance benefits of our proposed SABRes primitive and dynamic load balancing mechanism, respectively.
- **Part III** discusses related work (Chapter 9) and future research directions (Chapter 10).

Finally, Chapter 11 concludes the thesis.

1.4.1 Bibliographic Notes

This thesis was conducted under the supervision of my advisors, Babak Falsafi and Edouard Bugnion. Portions of it are product of collaboration with three colleagues: Boris Grot, Stanko Novakovic, and Dmitrii Ustiugov. Chapter 3 is partially based on a conference paper published in the *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* in 2014 [134] and a patent granted by the US Patent & Trademark Office in 2017 [137]. Chapters 4 and 7 are based on a conference paper published in the *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* in 2016 [40]. Finally, Chapter 6 is based on a conference paper published in the *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)* in 2015 [39].

2 Application and Technology Trends

Modern datacenters are evolving rapidly, being shaped by growing demands for online services. From a single server's design to the overall datacenter network architecture, the deployed platforms evolve in a scale-out fashion to meet the high volume demand at tight latency constraints. This chapter provides an overview of key software and hardware trends that highlight the role of networking in the datacenter, and motivate the need for tighter network-compute integration.

2.1 Datacenter Services

Today's massive web-scale services, such as web search, social networking, e-commerce or analytics, require tens of thousands of servers and petabytes of storage [175]. Increasingly, the trend has been toward deeper analysis and understanding of data in response to real-time queries. To minimize the latency, datacenter operators have shifted hot datasets from disk to DRAM, necessitating terabytes, if not petabytes, of DRAM distributed across a large number of servers.

Online services typically comprise several software layers, resulting in multi-tiered architectures. In the most basic model, datacenter traffic patterns are *north-south*, as every user request propagates through the different service tiers. As services are gradually offering richer features, inter-server communication patterns in the datacenter become *east-west*, which are more complex and unpredictable [168]. Typically, while the amount of north-south traffic is a function of

incoming user requests, east-west traffic increases as a function of the rapidly increasing offered functionality per request, causing internal datacenter network bandwidth demands to double every 12–15 months [57, 157]. Every incoming user request triggers multiple software layer interactions, involving hundreds of servers. For instance, Amazon reports that the rendering of a single page typically requires access to over 150 internal services [48], while a single Google search query uses 1000 servers to retrieve an answer [46]. Latency considerations force Facebook to restrict the number of sequential data accesses to fewer than 150 per rendered web page [149].

Related work examining sources of network latency overhead in datacenters found that a typical deployment based on commodity technologies may incur over $100\mu s$ in round-trip latency between a pair of servers [149]. According to the study, principal sources of latency overhead include the operating system stack, NIC, and intermediate network switches. While $100\mu s$ may seem insignificant, communication time can end up dominating the overall latency of a user request, mainly for two reasons. First, every request results in long sequences of inter-server communication, as it goes through several internal datacenter service layers. Second many of these service layers mainly involve data retrieval with minimal computation per data item loaded (e.g., key-value stores). For example, read operations dominate key-value store traffic, and simply return the object in memory. With $1000\times$ difference in data access latency between local DRAM (100ns) and remote memory ($100\mu s$), distributing the dataset, although necessary, incurs a dramatic performance overhead. In conclusion, inter-server communication is taking center stage as a major performance determinant of online services.

2.2 Server Architectures

Datacenters employ commodity technologies due to their favorable cost-performance characteristics. The end result is a *scale-out* architecture characterized by a large number of commodity servers connected via commodity networking equipment. Two architectural trends are emerging in scale-out designs.

First, System-on-Chips (SoC) provide high chip-level integration and are a major trend in servers.

Current server SoCs combine many processing cores, memory interfaces, and I/O to reduce cost and improve overall efficiency by eliminating extra system components. More recently, some SoCs went as far as integrating the network endpoints on the chip. For instance, Calxeda (now defunct) integrated the Ethernet controller on chip [44], a practice that Intel recently also started following with its Xeon D SoCs [13]. AppliedMicro’s X-Gene2 server SoC [107] and Oracle’s Sonoma [109] integrated an RDMA controller directly on chip. While the controller still communicates with the chip’s memory hierarchy over DMA transfers, this is a clear effort to bridge the gap between the compute and the network.

Second, there is a growing trend for manycore server chips, motivated by the nature of online services, which operate on massive datasets, exhibiting little data locality and immense request-level parallelism. These characteristics result in CPU cores processing short-lived independent requests and spending most time waiting for data retrieval from memory [62, 91, 119]. The net result is that servers are gradually featuring more and more—potentially leaner—cores. Emerging server processors, such as Cavium’s ThunderX series [110, 111], AppliedMicro’s X-Gene 3 [115], Phytium’s FT-2000/64 [112], Qualcomm’s Centriq [113], and EZChip’s TILE-Mx [56], already feature from several dozens to 100 ARM cores. Even the latest Intel and AMD x86 CPUs, which typically feature brawnier cores, are hitting the 30s range [114, 116]. For example, the latest Skylake Xeons offer up to 28 cores.

While both trends are beneficial for online services running on modern datacenters, they currently seem to evolve independently; it is, however, important to reconcile the manycore and network integration trends. Both are key to server efficiency, thus the two should be co-designed.

2.3 RDMA and Lossless Fabrics

RDMA [124] enables memory-to-memory data transfers across the network without processor involvement on the destination side. These direct data transfers from remote memory are also commonly referred to as *one-sided operations*. By exposing remote memory and reliable connections directly to user-level applications, RDMA eliminates all kernel overheads. Furthermore,

one-sided remote memory operations are handled entirely by the adapter without interrupting the destination core. RDMA is supported on lossless fabrics such as InfiniBand [80] and Converged Ethernet [79] that scale to thousands of nodes and can offer remote memory read latency as low as a couple of μs .

Although historically associated with the High-Performance Computing market, RDMA is now making inroads into web-scale datacenters, such as Microsoft's and Google's [155]. Latency-sensitive key-value stores such as RAMCloud [138], Pilaf [127], FaRM [53, 54], HERD [89], and DrTM [171] use RDMA fabrics to achieve key-value lookups from remote memory at latencies as low as $5\mu\text{s}$.

There are a number of limitations that are currently blocking the adoption of full datacenter-scale RDMA, such as the lack of integrated congestion management in the protocol and the difficulty of scaling lossless fabrics to networks of tens of thousands of nodes, a guarantee RDMA relies on to achieve high performance. The strong interest of massive online service providers, who also own the largest datacenters, in RDMA technology, is driving significant resources into research to address these challenges. However, it should be noted that even if these challenges remain unsolved, RDMA and RDMA-like solutions, such as the soNUMA architecture proposed in this thesis (Chapter 3), will remain highly relevant to datacenter architectures, as they can provide significant performance benefits. We discuss in the following section how such technologies can be leveraged to offer more powerful building blocks for datacenters.

2.4 Rack-Scale Computing

Rack-scale computing is a young field that recently started gaining traction from both industry and academia [5]. Rack-scale computing identifies the *rack* as an architectural block—instead of the typical *server* architectural block in today's datacenters—within which, all components are tightly integrated to deliver significant compute capacity at high efficiency in a contained scale.

Figure 2.1 represents a high-level view of such a rack-scale computer. In addition to the traditional

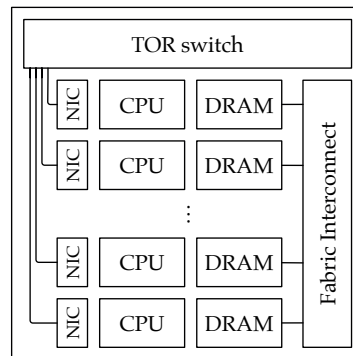


Figure 2.1 – Overview of a rack-scale computer.

Ethernet connections to a top-of-the-rack (TOR) switch for conventional networking with the outside world, all SoCs in the rack are directly attached to a secondary high-performance fabric interconnect. The key strength of rack-scale computers is that they offer high density of cores and memory that can rapidly communicate via integrated high-performance fabrics. In the near future, rack-scale computers will feature 1000s of cores and terabytes of memory in a rack form factor, with glueless fabrics offering high-bandwidth, low-latency interconnection. Such systems will be capable of replacing large-scale NUMA machines, as they will offer comparable performance at a fraction of the cost. Early examples of rack-scale systems are AMD SeaMicro [50], Boston Viridis [24], HP’s The Machine [76], and Oracle’s ExaLogic and ExaData [139]. The aggregate vast memory pool of a rack-scale computer can be seamlessly accessed using fast one-sided operations over the fabric, making these systems a great fit for computation on massive datasets that cannot be easily partitioned, hence remote memory access is unavoidable and frequent.

A rack-scale computer can offer enough resources to be used as a standalone solution for a family of medium-sized problems, or as even as a building block for future datacenters. For example, the RackOut architecture [136] demonstrates that organizing the datacenter as collection of rack-scale computers rather than a collection of servers can yield substantial utilization improvements under tight response time tail latency guarantees. Thus, even if systems with fully integrated networking and lightweight protocols will not outgrow current rack-scale solutions, they still represent excellent building blocks for future datacenter-scale deployments.

Our main focus on this thesis is on systems of contained scale, such as these rack-scale computers, where network-compute co-design and tighter integration will have the most dramatic impact. In such systems, inter-server communication performance is largely determined by the endpoints: the protocol executed on the CPU and network controller, and the flow of information between the CPU and the network controller. Furthermore, any software overhead added to the bare remote memory access latency imposed by the underlying hardware perceptibly increases the end-to-end latency. We take a vertical system redesign approach to holistically tackle these major sources of inter-server communication overheads in rack-scale computers.

At datacenter scale, there are other factors that significantly affect inter-server communication performance, such as multi-hop topologies with several switches that add a measurable latency, oversubscribed network tiers, and long distances. However, these additional overheads of datacenter-scale communication will be gradually ameliorated. Rapid advancements in datacenter networking equipment (e.g., adoption of high-performance fabrics like InfiniBand) already offer datacenter-wide network roundtrips faster than $20\mu\text{s}$ [28]. End-to-end optic networks at full datacenter scale are expected in the near future; such networks will allow datacenter traversals in a single μs , dictated by the speed of light. Thus, whether the performance of datacenter inter-server communication will converge with that of rack-scale computers, or rack-scale computers will emerge as building blocks in the datacenter space, we expect the contributions of this thesis to eventually be highly relevant even at datacenter scale.

2.5 The Emerging Network–Compute Mismatch

We are entering an era of intense technological turmoil, which, among others, will significantly affect the design of communication-intensive systems. Historically, compute logic has been dramatically faster than network communication and systems were designed around this basic assumption. However, technology advancements are bound to disrupt long-established balances. The slowdown of Moore’s Law and the end of Dennard Scaling are leading to stagnating performance of general-purpose logic. At the same time, networks are evolving rapidly. On the

latency front, in-datacenter propagation delays approaching the fundamental limit imposed by the speed of light, enabling full datacenter traversals in just a couple of μs . On the bandwidth front, we continuous improvements that are expected to continue in the foreseeable future; the InfiniBand Trade Association's roadmap predicts a quadrupling of network bandwidth [81]. The net result is in the near future, balanced communication-intensive systems will have to be increasingly more frugal in the amount of computation spent per network message.

The emerging imbalance between CPU processing capacity and network capabilities has already started surfacing. To illustrate, at the time of writing, a high-end Mellanox InfiniBand NIC delivers 200Gb/s and 200M IOPS; this leaves even the highest-end server CPUs with fewer than 1000 cycles to complete a request associated with a single network packet. For example, the Xeon Platinum 8176 features 28 cores at 2.1GHz. In a 2-socket configuration, utilizing all the available network bandwidth using small messages would require spending as few as 600 CPU cycles per message. It is becoming increasingly more challenging to utilize the growing network bandwidth, especially with small messages. Therefore, any achieved reduction in computational resources spent per network message directly contributes towards building more balanced systems. In this thesis, we put actively put effort towards this direction. Network-compute integration and co-design opens a range of opportunities to alleviate the emerging network-compute imbalance.

2.6 One-Sided Operations Versus RPCs

Computation in distributed memory systems requires inter-server interaction, which generally takes one of two forms: either data is *pulled* from a remote server to a local server, or computation is *pushed* from the local server to the remote server, where it is executed on the target data. We refer to the former interaction form as *remote memory access* and to the latter as *Remote Procedure Call (RPC)*.

An RPC is inherently a *two-sided* operation: RPC-based inter-server interaction implies CPU involvement of both communicating servers. A remote memory access may of course be performed over an RPC, but can also be performed over a *one-sided* operation, i.e., an operation

Chapter 2. Application and Technology Trends

that does not involve the CPU of the remote end. For most scale-out system deployments that rely on conventional network stacks such as TCP/IP, one-sided remote memory access is not possible. However, hardware-terminated protocols like InfiniBand not only offload the bulk of protocol processing to hardware, leaving a lean user-level protocol to be executed by the CPU, but also offer such operations. Such technology is commonly referred to as RDMA (Remote Direct Memory Access). An RDMA NIC has the capability of directly accessing application memory without CPU involvement at the remote end, offering the fastest path to remote memory. An additional benefit of one-sided operations over two-sided operations is tighter response latency distribution, as bypassing software interaction at the remote end removes a major source of unpredictability [161]. Hardware delivers more predictable latencies than software, resulting in tighter tails, as also demonstrated by Microsoft's Catapult project [28]. Several software frameworks have recently been built to leverage the strengths of these one-sided operations offered by RDMA technology [53, 54, 89, 127].

However, one-sided operations also have disadvantages, with first and foremost their lack of flexibility, because each operation is limited to reading/writing a single remote memory location. The requesting server has to specify exactly the remote memory location to be accessed. Such requirement is not trivial, as it involves software and use of data structures (e.g., FaRM's Hopscotch [53]) specially designed to facilitate location of remote data from the requesting side. Consequently, legacy software cannot make use of such operations without a major rewrite. Second, even with specially designed software, locating the target remote data can result in multiple roundtrips over the network. Adding an extra roundtrip voids the benefit of using a one-sided operation, as the overall remote data access time ultimately exceeds that of a conventional two-sided operation.

As a result, conventional wisdom dictates resorting to a two-sided operation (a.k.a. an RPC—Remote Procedure Call) to conduct any remote memory operation more complicated than a simple remote memory location read. To illustrate, even systems that are specifically designed to heavily rely on one-sided operations for fast remote memory reads (e.g., Pilaf [127] and Microsoft's FaRM [53]), resort to two-sided operations for writes. The reason for that design

2.6. One-Sided Operations Versus RPCs

choice is that writes can trigger complex side-effects that can only be dealt with in software, such as data structure rebalances, memory allocation, etc. Of course, the flexibility of two-sided operations does not come for free; it involves CPU involvement at the remote end, which is a valuable resource (see Section 2.5): a message triggers some arbitrary code to be executed on a general-purpose CPU core. The involvement of software at the remote end also implies increased response latency and unpredictability.

Datacenter services still rely mostly on RPCs partially because of their great flexibility of RPCs and partially because they have been a well-established model for inter-server communication for decades; RDMA technology that enables one-sided operations started appearing in the datacenter space only recently. We expect that software layers that deliver very simple but latency-critical functionality, such as key-value stores, will gradually be restructured to leverage the low latency of one-sided operations. While the jury is still out as to when each of the two operation types is preferable, the community seems to be gradually reaching consensus that future systems using high-performance networking solutions should judiciously use both one- and two-sided operations, leveraging the merits of each [52]. Several recent well-engineered software stacks for distributed memory systems are a good example of that direction, as they combine the strengths of both operation types to maximize performance (e.g., [34, 171, 177]).

The net result is that any high-performance distributed memory architecture should offer support for rapid one-sided operations and efficient two-sided communication. In this thesis, we propose a new architecture with co-designed compute and network interface logic, aiming to significantly improve the performance and scalability of hardware-terminated protocols. We demonstrate that tightly integrated ICONIC architectures deliver superior performance, scalability, and flexibility for one-sided operations, and new opportunities for more efficient RPC invocations. Finally, in an attempt to bridge the semantic gap between one-sided operations and RPCs, we also investigate new one-operations with richer semantics and propose such a new primitive.

ICONIC Architecture Features Part I

3 The Scale-Out NUMA Architecture

The rising demand for real-time online services has made it common practice for service providers to keep all data memory resident, distributed across millions of servers in a datacenter. While memory residency eliminates disk accesses, shrinking data access latency from 10s of milliseconds to 100s of nanoseconds, data distribution across machines results in frequent inter-server communication. As modern datacenters are built with commodity networking technology running on top of commodity servers and operating systems, inter-server communication delays can exceed $100\mu\text{s}$ [149], a $1000\times$ overhead over the desired memory access latency.

The reasons for the high communication latency are well known and include deep network stacks, complex network interface cards (NIC), and slow chip-to-NIC interfaces [149, 63]. RDMA reduces end-to-end latency by enabling memory-to-memory data transfers over InfiniBand [80] and Converged Ethernet [79] fabrics. By exposing remote memory at user-level and offloading network processing to the adapter, RDMA enables remote memory read latencies in the range of a couple of μs ; however, that still represents a $>10\times$ latency increase over local DRAM.

To mitigate the performance gap between local and remote memory, we introduce Scale-Out NUMA (soNUMA): an architecture, programming model, and communication protocol for distributed, in-memory applications that reduces remote memory access latency to within a small factor ($\sim 3\text{--}4\times$) of local memory. soNUMA leverages two simple ideas to minimize latency. The

Chapter 3. The Scale-Out NUMA Architecture

first is to use a stateless request/reply protocol running over a NUMA memory fabric to drastically reduce or eliminate the network stack, complex NIC, and switch gear delays. The second is to integrate the protocol controller into the node's local coherence hierarchy, thus avoiding state replication and data movement across the slow PCI Express (PCIe) interface.

soNUMA exposes the abstraction of a partitioned global virtual address space, which is useful for big-data applications with irregular data structures such as graphs. The programming model is inspired by RDMA [124], with application threads making explicit remote memory read and write requests with copy semantics. The model is supported by an architecturally-exposed hardware block, called the *remote memory controller* (RMC), that safely exposes the global address space to applications. The RMC is integrated into each node's coherence hierarchy, providing for a frictionless, low-latency interface between the processor, memory, and the interconnect fabric. This chapter describes the soNUMA architecture, programming model and protocol, with a particular focus on the RMC design.

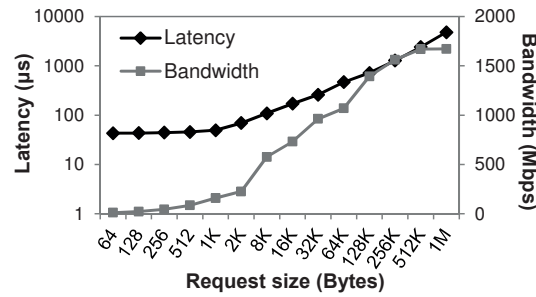


Figure 3.1 – Netpipe benchmark on a Calxeda microserver.

3.1 Obstacles to Fast Remote Memory

As datasets grow, the trend is toward more sophisticated algorithms at ever-tightening latency bounds. While SoCs, glueless fabrics, and RDMA technologies help lower network latencies, the network delay per byte loaded remains high. Here, we discuss principal reasons behind the difficulty of further reducing the latency for in-memory applications in modern scale-out deployments.

Deep network stacks are costly. Distributed systems rely on networks to communicate. Unfortunately, today’s deep network stacks require a significant amount of processing per network packet which factors considerably into end-to-end latency. Figure 3.1 shows the network performance between two directly-connected Calxeda EnergyCore ECX-1000 SoCs, measured using the standard netpipe benchmark [159]. The fabric and the integrated NICs provide 10Gbps worth of bandwidth. Despite the immediate proximity of the nodes, the integrated NICs and the lack of intermediate switches, we observe high latency (in excess of $40\mu\text{s}$) for small packet sizes and poor bandwidth scalability (under 2 Gbps) with large packets. These bottlenecks exist due to the high processing requirements of TCP/IP and are aggravated by the limited performance offered by Calxeda’s wimpy ARM Cortex-A9 cores.

PCIe/DMA latencies limit performance. I/O bypass architectures have successfully removed most sources of latency except the PCIe bus. Studies have shown that it takes 400–500ns to communicate short bursts over the PCIe bus [63], making such transfers $7\text{--}8\times$ more expensive,

in terms of latency, than local direct DRAM accesses. Furthermore, PCIe does not allow for the cache-coherent sharing of control structures between the system and the I/O device, leading to the need of replicating system state such as page tables into the device and system memory. In the latter case, the device memory serves as a cache, resulting in additional DMA transactions to access the state. SoC integration alone (e.g., integrated RDMA controller in X-Gene 2 and Oracle's Sonoma) does not eliminate these overheads, since IP blocks often use DMA internally to communicate with the main processor [20].

3.2 Scale-Out NUMA Overview

soNUMA is an architecture and programming model for low-latency distributed memory, designed to address each of the obstacles to low-latency described in Section 3.1. soNUMA goes after a scale-out model with physically distributed processing and memory: (i) it replaces deep network stacks with a lean user-level, hardware-terminated protocol; (ii) eschews system-wide coherence in favor of a global partitioned virtual address space accessible via RMDA-like remote memory operations with copy semantics; (iii) replaces transfers over the slow PCIe bus with fast cache-to-cache transfers; and (iv) is optimized for rack-scale deployments, where physical distance (i.e., propagation delays) is minuscule. In effect, our design goal is to borrow the desirable qualities of ccNUMA and RDMA without their respective drawbacks.

Figure 3.2 identifies the essential components of soNUMA. At a high level, soNUMA combines a lean memory fabric with an RDMA-like programming model in a rack-scale system. Applications access remote portions of the global virtual address space through remote memory operations. A new architecturally-exposed block, the *remote memory controller* (RMC), converts these operations into network transactions and directly performs the memory accesses. Applications directly communicate with the RMC, bypassing the operating system, which gets involved only in setting up the necessary in-memory control data structures.

Unlike traditional implementations of RDMA, which operate over PCIe, the RMC benefits from a tight integration into the processor's cache coherence hierarchy. In particular, the processor and

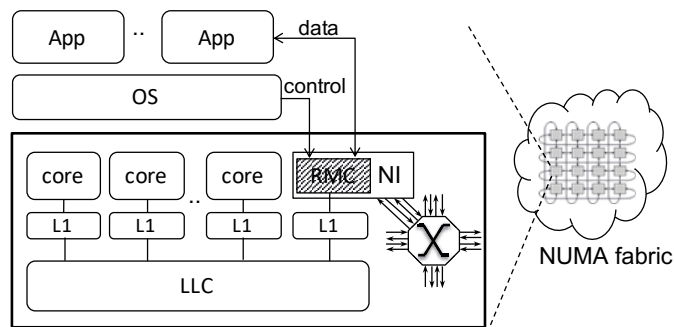


Figure 3.2 – soNUMA overview.

the RMC share all data structures via the cache hierarchy. The implementation of the RMC is further simplified by limiting the architectural support to one-sided remote memory read, write, and atomic operations, and by unrolling multi-line requests at the source RMC. As a result, the protocol can be implemented in a stateless manner by the destination node.

The RMC converts application commands into remote requests that are sent to the *network interface* (NI). The NI is connected to an on-chip low-radix router with reliable, point-to-point links to other soNUMA nodes. The notion of fast low-radix routers borrows from supercomputer interconnects; for instance, the mesh fabric of the Alpha 21364 connected 128 nodes in a 2D torus using an on-chip router with a pin-to-pin delay of just 11 ns [128].

soNUMA's memory fabric bears semblance (at the link and network layer, but not at the protocol layer) to the QPI and HTX solutions that interconnect sockets together into multiple NUMA domains. In such fabrics, parallel transfers over traces minimize pin-to-pin delays, short messages (header + a payload of a single cache line) minimize buffering requirements, topology-based routing eliminates costly CAM or TCAM lookups, and virtual lanes ensure deadlock freedom. Although Figure 3.2 illustrates a 2D-torus, the design is not restricted to any particular topology.

Terminology note: We introduced the RMC as a protocol controller terminating the soNUMA protocol and directly interfacing the NI. The term *NI* is usually referred to the entity handling the network layer. However, in favor of simplicity, we will be using the term *NI* as an umbrella term for the hardware entity handling both the protocol and network layers. Hence, in the context of

soNUMA, the term *NI* encompasses the RMC, as depicted in Figure 3.2.

3.3 Remote Memory Controller

The foundational component of soNUMA is the RMC, an architectural block that services remote memory accesses originating at the local node, as well as incoming requests from remote nodes. The RMC integrates into the processor's coherence hierarchy via a private L1 cache and communicates with the application threads via memory-mapped queues. This section describes the hardware/software interface that is used by the applications to interact with the RMC and provides a functional overview of the RMC. We then proceed to provide a complete picture of the soNUMA architecture by presenting the soNUMA communication protocol and required software support in Sections 3.4 and 3.5, respectively. We shift our focus back to the RMC in Section 3.6, with an analysis of its microarchitecture and its area/power cost.

3.3.1 Hardware/Software Interface

soNUMA provides application nodes with the abstraction of globally addressable, virtual address spaces that can be accessed via explicit memory operations. The RMC exposes this abstraction to applications, allowing them to safely and directly copy data to/from global memory into a local buffer using remote write, read, and atomic operations, without kernel intervention. The interface offers atomicity guarantees at the cache-line granularity, and no ordering guarantees within or across requests.

soNUMA's hardware/software interface is centered around four main abstractions directly exposed by the RMC: (i) the context identifier (`ctx_id`), which is used by all nodes participating in the same application to create a global address space; (ii) the context segment, a range of the node's address space which is globally accessible by others; (iii) the queue pair (QP), used by applications to schedule remote memory operations and get notified of their completion; and (iv) local buffers, which can be used as the source or destination of remote operations.

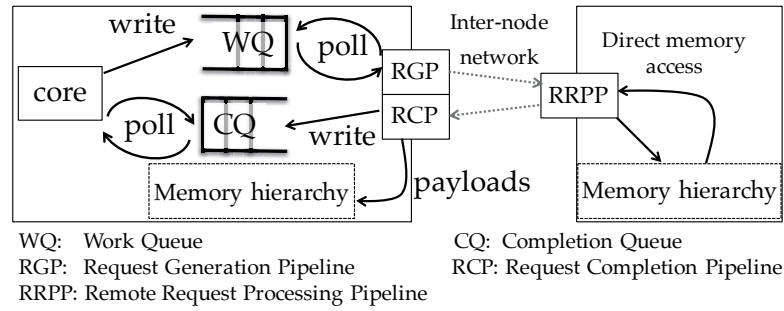


Figure 3.3 – QP interactions and memory access on the remote end for a remote read.

The QP model consists of a work queue (WQ), a bounded buffer written exclusively by the application, and a completion queue (CQ), a bounded buffer of the same size written exclusively by the RMC. The CQ entry contains the index of the completed WQ request. Both are stored in main memory and coherently cached by the cores and the RMC alike. In each operation, the remote address is specified by the combination of `node_id`, `ctx_id`, `offset`. Other parameters include the length and the local buffer address.

3.3.2 RMC Overview

The RMC consists of three hardwired pipelines that interact with the queues exposed by the hardware/software interface and with the NI. These pipelines are responsible for request generation, remote request processing, and request completion, respectively. Figure 3.3 is a high-level illustration of a CPU core’s interaction with the three RMC pipelines through a QP. New remote memory access requests are scheduled from the core by writing into a WQ. The Request Generation Pipeline (RGP) identifies the new request by polling the head of the WQ. After some local processing, it sends the request to the target node over the network, where the Remote Request Processing Pipeline (RRPP) will process the request, access its local memory accordingly and send a reply message back to the requesting node. Finally, the requesting node’s Request Completion Pipeline (RCP) processes the reply, matches it to the original request, and notifies the core of the request’s completion by writing an identifying entry in the CQ. The core identifies the request’s completion by polling on the CQ’s head.

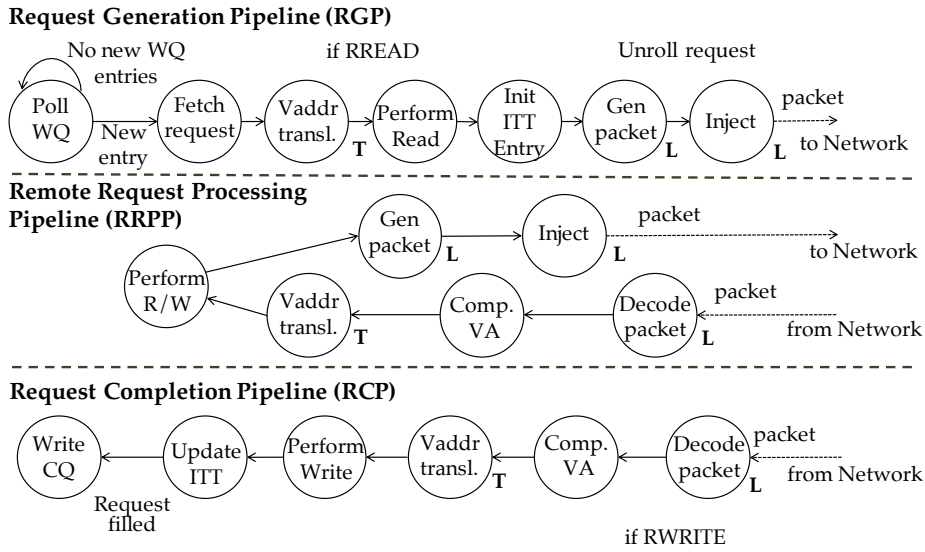


Figure 3.4 – Functionality of the RMC pipelines. The notation ‘L’ next to a stage indicates local processing in combinational logic; ‘T’ indicates a TLB access; the rest of the states access memory via an MMU.

The RMC pipelines are controlled by a configuration data structure, the *Context Table (CT)*, and leverage an internal structure, the *Inflight Transaction Table (ITT)*. The CT is maintained in memory and is initialized by system software. The CT keeps track of all registered context segments, queue pairs, and page table root addresses. Each CT entry, indexed by its `ctx_id`, specifies the address space and a list of registered QPs (WQ, CQ) for that context. Multi-threaded processes can register multiple QPs for the same address space and `ctx_id`. Meanwhile, the ITT is used exclusively by the RMC and keeps track of the progress of each WQ request.

Figure 3.4 highlights the main states and transitions for the three independent pipelines. Each pipeline can have multiple transactions in flight. Most transitions require an MMU access, which may be retired in any order. Therefore, transactions will be reordered as they flow through a pipeline.

Request Generation Pipeline (RGP). The RMC initiates remote memory access transactions in response to an application’s remote memory requests (reads, writes, atomics). To detect such requests, the RMC polls on each registered WQ. Upon a new WQ request, the RMC generates

one or more network packets using the information in the WQ entry. For remote writes and atomic operations, the RMC accesses the local node's memory to read the required data, which it then encapsulates into the generated packet(s). For each request, the RMC generates a transfer identifier (`tid`) that allows the source RMC to associate replies with requests.

Remote transactions in soNUMA operate at cache line granularity. Coarser granularities, in cache-line-sized multiples, can be specified by the application via the `length` field in the WQ request. The RMC unrolls multi-line requests in hardware, generating a sequence of line-sized read or write transactions. To perform unrolling, the RMC uses the ITT, which tracks the number of completed cache-line transactions for each WQ request and is indexed by the request's `tid`.

Remote Request Processing Pipeline (RRPP). This pipeline handles incoming requests originating from remote RMCs. The soNUMA protocol is stateless, which means that the RRPP can process remote requests using only the values in the header and the local configuration state. Specifically, the RRPP uses the `ctx_id` to access the CT, computes the virtual address, translates it to the corresponding physical address, and then performs a read, write, or atomic operation as specified in the request. The RRPP always completes by generating a reply message, which is sent to the source. Virtual addresses that fall outside of the range of the specified security context are signaled through an error message, which is propagated to the offending thread in a special reply packet and delivered to the application via the CQ.

Request Completion Pipeline (RCP). This pipeline handles incoming message replies. The RMC extracts the `tid` and uses it to identify the originating WQ entry. For reads and atomics, the RMC then stores the payload into the application's memory at the virtual address specified in the request's WQ entry. For multi-line requests, the RMC computes the target virtual address based on the buffer base address specified in the WQ entry and the offset specified in the reply message.

The ITT keeps track of the number of completed cache-line requests. Once the last reply is processed, the RMC signals the request's completion by writing the index of the completed

WQ entry into the corresponding CQ and moving the CQ head pointer. Requests can therefore complete out of order and, when they do, are processed out of order by the application. Remote write acknowledgments are processed similarly to read completions, although remote writes naturally do not require an update of the application's memory at the source node.

3.4 Communication Protocol

soNUMA's communication protocol naturally follows the design choices of the three RMC pipelines at the protocol layer. At the link and routing layers, our design borrows from existing memory fabric architectures (e.g., QPI or HTX) to minimize pin-to-pin delays.

Link layer. The memory fabric delivers messages reliably over high-speed point-to-point links with credit-based flow control. The message MTU is large enough to support a fixed-size header and an optional cache-line-sized payload. Each point-to-point physical link has two virtual lanes to support deadlock-free request/reply protocols.

Routing layer. The routing-layer header contains the destination and source address of the nodes in the fabric (`dst_nid`, `src_nid`). `dst_nid` is used for routing, and `src_nid` to generate the reply packet. The router's forwarding logic directly maps destination addresses to outgoing router ports, eliminating expensive CAM or TCAM lookups found in networking fabrics.

Protocol layer. The RMC protocol is a simple request-reply protocol, with exactly one reply message generated for each request. The WQ entry specifies the `dst_nid`, the command (e.g., `read`, `write`, or `atomic`), the `offset`, the `length` and the local buffer address. The RMC copies the `dst_nid` into the routing header, determines the `ctx_id` associated with the WQ, and generates the `tid`. The `tid` serves as an index into the ITT and allows the source RMC to map each reply message to a WQ and the corresponding WQ entry. The `tid` is opaque to the destination node, but is transferred from the request to the associated reply packet.

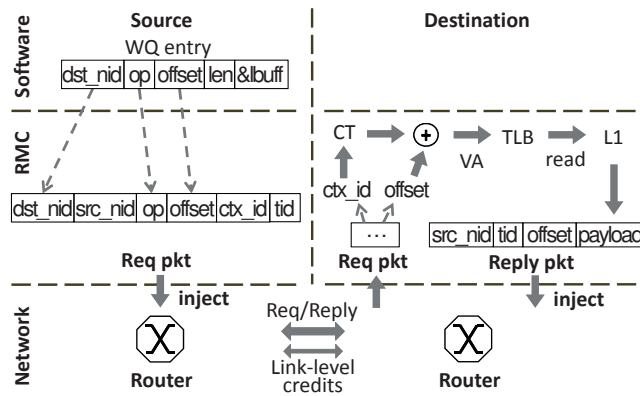


Figure 3.5 – Communication protocol for a remote read.

Figure 3.5 illustrates the actions taken by the RMCs for a remote read of a single cache line. The RGP in the requesting side’s RMC first assigns a `tid` for the WQ entry and the `ctx_id` corresponding to that WQ. The RMC specifies the destination node via a `dst_nid` field. The request packet is then injected into the fabric and the packet is delivered to the target node’s RMC. The receiving RMC’s RRPP decodes the packet, computes the local virtual address using the `ctx_id` and the `offset` found in it and translates that virtual address to a physical address. This stateless handling does not require any software interaction on the destination node. As soon as the request is completed in the remote node’s memory hierarchy, its RMC creates a reply packet and sends it back to the requesting node. Once the reply arrives to the original requester, the RMC’s RCP completes the transaction by writing the payload into the corresponding local buffer and by notifying the application via a CQ entry (not shown in Figure 3.5).

3.4.1 Handling Packet Loss

Reliable delivery of packets is commonly enforced at the link and transport layers of the network stack. In TCP/IP networks, the TCP layer is responsible for flow control, congestion management, retransmissions in case of packet loss, connection management, etc. Most of TCP processing is onloaded to the CPU, to which the high latency of network protocol processing is partially attributed.

The soNUMA protocol is much closer to InfiniBand, where the network protocol is terminated

Chapter 3. The Scale-Out NUMA Architecture

by the NIC hardware (known as an HCA). InfiniBand assumes a lossless link layer, namely no packet can be lost because of buffer overflows. Despite that, packet losses are still possible for various other reasons, such as FCS errors. Therefore, InfiniBand implements a trivial back-to-0 retransmission mechanism in hardware, for packet retransmission at the transport layer, in the rare occasion the link layer fails to deliver. Prior work investigating datacenter-scale deployment of RDMA technology, which relies on lossless fabrics such as InfiniBand or Converged Ethernet, showed that packet drops indeed occur, and retransmission policies as simple as back-to-0 can result in severe performance degradation [68]. The authors of [68] conclude that, at datacenter scale, smarter retransmission mechanisms are required and suggest that doing so, along with the addition of hardware for better forward error correction, is a technologically sound choice that can significantly relax the strict requirements for lossless packet delivery placed on the network fabric.

The soNUMA architecture is built on top of a memory fabric with credit-based link-layer flow control. The fabric thus guarantees lossless delivery at the link layer, similar to InfiniBand. To keep the design of the RMC as simple as possible, when a packet drop is detected, the RMC exposes the failure to the application directly, which then decides what action to take when a packet sent to a remote node is not matched by a response. The RMC notifies the application framework via the CQ and passes an error code signaling the packet loss. It should be trivial to extend the RMC hardware to implement a back-to-0 retransmission, as such mechanism does not require additional state. However, given that our solution is mainly targeting rack-scale deployments, such retransmissions are expected to be extremely rare [90], making the software fallback choice attractive. Finally, the same trade-offs as in InfiniBand apply for soNUMA: better lossless fabrics versus additional hardware support at the endpoints for improved retransmission mechanisms and forward error correction. The balance of this tradeoff, however, is different for soNUMA and InfiniBand, as all RMC logic is integrated on the CPU chip rather than on a discrete network card.

3.5 Software Support

Making the RMC hardware accessible and the exported hardware/software interface usable by programmers, a significant effort is required on the software front. We now briefly describe the system and application software support required to expose the RMC to applications and enable the soNUMA programming model. Further details on system and software support for soNUMA are available in [133].

3.5.1 Device Driver

The role of the operating system on an soNUMA node is to establish the global virtual address spaces. This includes the management of the context namespace, virtual memory, QP registration, etc. The RMC device driver manages the RMC itself, responds to application requests, and interacts with the virtual memory subsystem to allocate and pin pages in physical memory. The RMC device driver is also responsible for allocating the CT and ITT on behalf of the RMC.

Unlike a traditional RDMA NIC, the RMC has direct access to the page tables managed by the operating system, leveraging the ability to share cache-coherent data structures. As a result, the RMC and the application both operate using virtual addresses of the application's process once the data structures have been initialized.

The RMC device driver implements a simple security model in which access control is granted on a per `ctx_id` basis. To join a global address space `<ctx_id>`, a process first opens the device `/dev/rmc_contexts/<ctx_id>`, which requires the user to have appropriate permissions. All subsequent interactions with the operating system are done by issuing `ioctl` calls via the previously-opened file descriptor. In effect, soNUMA relies on the built-in operating system mechanism for access control when opening the context, and further assumes that all operating system instances of an soNUMA fabric are under a single administrative domain.

Finally, the RMC notifies the driver of failures within the soNUMA fabric, including the loss of links and nodes. Such transitions typically require a reset of the RMC's state, and may require a

restart of the applications.

3.5.2 Access Library

The QPs are accessed via a lightweight API, a set of C/C++ inline functions that issue remote memory commands and synchronize by polling the completion queue. We expose a synchronous (blocking) and an asynchronous (non-blocking) set of functions for both reads and writes. The asynchronous API is comparable in terms of functionality to the Split-C programming model [38].

Fig. 3.6 illustrates the use of the asynchronous API for the implementation of the classic PageRank graph algorithm [141]. `rmc_wait_for_slot` processes CQ events (calling `pagerank_async` for all completed slots) until the head of the WQ is free. It then returns the freed slot where the next entry will be scheduled. `rmc_read_async` (similar to Split-C's `get`) requests a copy of a remote vertex into a local buffer. Finally, `rmc_drain_cq` waits until all outstanding remote operations have completed while performing the remaining callbacks.

This programming model is efficient as: (i) the callback (`pagerank_async`) does not require a dedicated execution context, but instead is called directly within the main thread; (ii) when the callback is an inline function, it is passed as an argument to another inline function (`rmc_wait_for_slot`), thereby enabling compilers to generate optimized code without any function calls in the inner loop; (iii) when the algorithm has no read dependencies (as is the case here), asynchronous remote memory accesses can be fully pipelined to hide their latency.

To summarize, soNUMA's programming model combines true shared memory (by the threads running within a cache-coherent node) with explicit remote memory operations (when accessing data across nodes). In the PageRank example, the `is_local` flag determines the appropriate course of action to separate intra-node accesses (where the memory hierarchy ensures cache coherence) from inter-node accesses (which are explicit).

Finally, the RMC access library exposes atomic operations such as compare-and-swap and fetch-and-add as inline functions. These operations are executed atomically within the local cache

```

float *async_dest_addr[MAX_WQ_SIZE];
Vertex lbuf[MAX_WQ_SIZE];

inline void pagerank_async(int slot, void *arg) {
    *async_dest_addr[slot] += 0.85 * lbuf[slot].rank[superstep%2] / lbuf[slot].out_degree;
}

void pagerank_superstep(QP *qp) {
    int evenodd = (superstep+1) % 2;
    for(int v=first_vertex; v<=last_vertex; v++) {
        vertices[v].rank[evenodd] = 0.15 / total_num_vertices;
        for(int e=vertices[v].start; e<vertices[v].end; e++) {
            if(edges[e].is_local) {
                // shared memory model
                Vertex *v2 = (Vertex *)edges[e].v;
                vertices[v].rank[evenodd] += 0.85 * v2->rank[superstep%2] / v2->out_degree;
            } else {
                // flow control
                int slot = rmc_wait_for_slot(qp, pagerank_async);
                // setup callback arguments
                async_dest_addr[slot] = &vertices[v].rank[evenodd];
                // issue split operation
                rmc_read_async(qp, slot,
                    edges[e].nid, //remote node ID
                    edges[e].offset, //offset
                    &lbuf[slot], //local buffer
                    sizeof(Vertex)); //len
            }
        }
    }
    rmc_drain_cq(qp, pagerank_async);
    superstep++;
}

```

Figure 3.6 – Computation of a PageRank superstep in soNUMA through a combination of remote memory accesses (via the asynchronous API) and local shared memory.

coherence hierarchy of the destination node.

3.5.3 Synchronization Library

By providing architectural support for only read, write and atomic operations, soNUMA reduces hardware cost and complexity. The minimal set of architecturally-supported operations is not a fundamental limitation, however, as standard communication and synchronization primitives can be built in software on top of these three basic primitives.

For instance, we have implemented a simple barrier primitive such that nodes sharing a `ctx_id` can synchronize. Each participating node broadcasts the arrival at a barrier by issuing a `write` to an agreed upon offset on each of its peers. The nodes then poll locally until all of them reach the barrier. Other types of synchronization primitives can also be implemented by leveraging this trivial mechanism.

3.5.4 Messaging over One-Sided Operations

The soNUMA protocol [134] only offers native support for one-sided remote memory access, in favor of simplicity. While one-sided operations offer the fastest path to remote memory, they only represent one form of inter-node communication on distributed memory systems. The most widespread form of communication is messaging—i.e., two-sided communication—which involves the participation of the remote end’s CPU. soNUMA’s protocol design can implement `send` and `receive` operations for messaging on top of one-sided operations. To communicate using `send` and `receive` operations, two application instances must first each allocate a bounded buffer from their own portion of the global virtual address space. The sender always writes to the peer’s buffer using `rmc_write` operations, and the content is read locally from cached memory by the receiver. Each buffer is an array of cache-line sized structures that contain header information (such as the length, memory location, and flow-control acknowledgments), as well as an optional payload. Flow-control is implemented via a credit scheme that piggybacks existing communication.

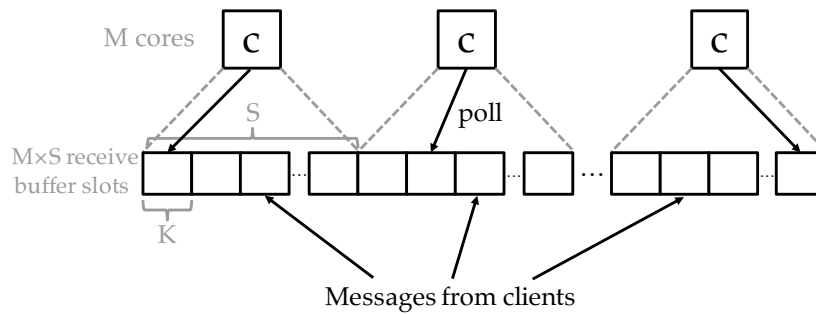


Figure 3.7 – Messaging emulation using one-sided operations.

For small messages, the sender creates packets of predefined size, each carrying a portion of the message content as part of the payload. It then *pushes* the packets into the peer’s buffer. To receive a message, the receiver polls on the local buffer. In the common case, the send operation requires a single `rmc_write`, and it returns without requiring any implicit synchronization between the peers. In our implementation, each 64B messaging packet comprises 4B of header/metadata, and 60B of payload (assuming a typical cache line size of 64B). For example, transferring 64B of data requires packetization into two soNUMA packets: the 1st packet contains a 4B header and 60B of data; the 2nd packet contains a 4B header, 4B of data, and 60B of padding, as all network packets in soNUMA are cache-block-sized (64B).

For large messages stored within a registered global address space, the sender only provides the base address and size to the receiver’s bounded buffer. The receiver then *pulls* the content using a single `rmc_read` and acknowledges the completion by writing a zero-length message into the sender’s bounded buffer. This approach delivers a direct memory-to-memory communication solution, but requires synchronization between the peers.

At compile time, the user can define the boundary between the two mechanisms by setting a minimal message-size threshold: push has lower latency since small messages complete through a single `rmc_write` operation and also allows for decoupled operations. The pull mechanism leads to higher bandwidth since it eliminates the intermediate packetization and copy step.

Chapter 3. The Scale-Out NUMA Architecture

A similar messaging mechanism was deployed by HERD [89], a software framework that offers fast RPCs over one-sided RDMA operations. Figure 3.7 illustrates an example deployment of such a messaging mechanism. Each server registers a receive buffer comprising $M \times S$ slots of size K , where:

- M is the number of the server's cores.
- S is the number of received buffer slots dedicated to each core.
- K is the receive buffer slot's size, which defines the maximum acceptable message size.

After the receive buffer is set up, each core periodically polls all of its corresponding buffer slots for new incoming messages. For a client, sending a message to the server involves explicitly picking a location within the receive buffer to write to, using a one-sided write operation. An implicit effect is that the location of the write also dictates which specific core of the server will service that message. Each client can only write to a statically predefined subset of the receive buffer slots, to prevent clients from squashing each others' messages. Consequently, $M \times S$ has to be sufficiently larger than the total number of clients in order to allow multiple outstanding requests per client. The received message triggers an RPC, which produces the result corresponding to the incoming request. Sending the result back to the client involves the same process, with the roles of the server and client inversed.

3.6 RMC Microarchitecture

Section 3.3 introduced the high-level organization and functionality of the RMC. In this section, we dive into the RMC's microarchitectural details, and then proceed to evaluate its cost in terms of area and power.

Figure 3.8 illustrates the implementation of the RMC as a set of three completely decoupled pipelines, affording concurrency in the handling of different functions at low area and design cost. The RMC features two separate interfaces: a coherent memory interface to a private cache and a

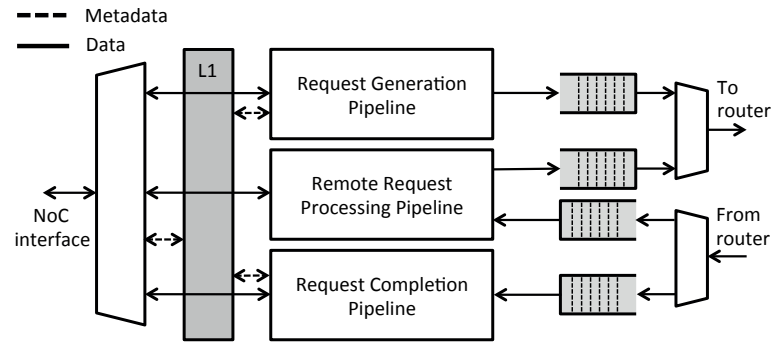


Figure 3.8 – RMC interface to the on-chip and off-chip network.

network interface to the on-die router providing system-level connectivity. On the network side, the three RMC pipelines are connected to distinct queues that interface a low-radix router block with support for two virtual lanes. The RMC’s integration into the node’s coherence hierarchy is a critical feature of soNUMA that eliminates wasteful data copying of control structures, and of page tables in particular. It also reduces the latency of the application/RMC interface by eliminating the need to set up DMA transfers of ring buffer fragments.

The RMC acts both as a protocol controller and a data manipulator and, as such, handles three distinct classes of data: system state, application metadata, and application memory (data). Each of these has different characteristics, which justifies designing the handling of each separately.

System state includes essential protocol-specific metadata that are maintained by the RMC. First, the RMC needs to maintain the protocol state, i.e., the information related to the registered QPs. Second, the RMC needs to maintain some additional internal state to keep track of each soNUMA operation. While the soNUMA protocol is connectionless, a minimal amount of state is still required per operation, as the protocol supports variable sized requests, which are unrolled at the source. As a result, the RMC that initiates a multi-block request needs to keep track of the pending packets that comprise the request in order to track its progress and completion (ITT).

Application metadata includes information that is frequently communicated between the application and the RMC: the WQ and CQ entries. The application writes WQ entries to send new requests to the RMC, while the RMC writes CQ entries to notify the application of request completions. As the queues are memory-mapped, this information exchange is enabled by the

Chapter 3. The Scale-Out NUMA Architecture

combination of a polling-based mechanism and the default coherence protocol cache block transfers, which guarantees that the most recent version of a requested block is being read.

The third and last category is application memory. While the RMC is handling the movement of a large amount of application data, it never is the final consumer. The real data consumers are the applications which send the remote data access requests to the RMC. Therefore, keeping data close to the RMC provides no benefit—on the contrary, it may unnecessarily increase the access latency for the original data requester.

For the above reasons, the RMC handles these three classes differently:

1. System state is relatively small, is constantly accessed by the RMC, and is not touched by the outside system in the steady state. Thus, it should be stored in specialized, dedicated SRAM structures.
2. Application metadata leverages cache coherence for efficient data transfer between the RMC and the application. Furthermore, as the minimum transfer unit in a conventional memory subsystem is a cache block, more than one valid control queue entries can be fetched in a single transfer, resulting in spatial locality. Thus, the RMC can benefit from a small private cache, integrated into the coherence domain.
3. Application memory is never used by the RMC, so it should not be stored in any caching layer private to the RMC. Thus, data corresponding to application memory always bypass the RMC's cache.

While each of the three RMC pipelines implements its own datapath and control logic, some data structures and hardware components are shared across them. The RGP and RCP handle the same requests at different phases of their lifetime (initiation and completion), hence they share some state. In contrast, the RRPP is completely independent, as it statelessly processes incoming requests from remote nodes. We first describe the structures that are shared by the RGP and RCP, followed by a description of each of the three pipelines.

3.6.1 Shared SRAM Structures

RMC Cache. The RGP and RCP share a small private cache, which is used for their communication with the cores that run the application code, via memory-mapped operations. The RMC cache has a 16-byte interface, optimized for the WQ entry size. Each cache block can fit four WQ entries or 64 CQ entries.

QP Table. A QP is registered to an RMC by allocating an entry in the RMC's QP table. A QP table entry consists of the base physical addresses of the WQ and the CQ (note that, as in RDMA, all pages involved in soNUMA operations are pinned in memory). In the same QP entry, the RMC pipelines store their current index to each of the two queues. The RGP accesses the information related to the WQ, while the RCP accesses the information related to CQ. The QP table sizing depends on the target maximum number of active QPs in the system.

ITT Table. This structure is used to keep track of each outstanding request's progress. Its existence is essential for large requests that get unrolled, as responses may arrive at any order. For every new request, a new ITT entry is allocated by the RGP, and the allocated entry's index is used as the transaction's id. An ITT entry contains a counter that indicates the number of replies required for that request, as well as the QP id (the index to the corresponding QP table entry) and WQ index, required to identify the origin of a request, once it's completed. The local buffer's address is also kept in the ITT, so that every reply packet that carries payload can directly find the location in which it has to be written. The alternative would have been reading the request's corresponding WQ entry from the memory subsystem again, to retrieve the associated buffer address that is embedded in the request entry. The RCP accesses the ITT table upon every reply packet arrival, to decrement the counter that tracks the number of block-sized transfers associated to a request. The ITT table should be sized based on the expected round-trip time of remote memory accesses so that a new transaction id is always available to avoid RGP stalls.

3.6.2 Request Generation Pipeline

Figure 3.9 shows the RGP, which initiates soNUMA remote operations. It periodically polls all the WQs that are registered in the QP table to check for new enqueued requests. A new entry is identified by a valid bit, set by the application. A unique transaction id is assigned to every soNUMA request generated by an RGP, in order to identify the originating request, once a reply arrives. Requests that are larger than a cache block need to be unrolled, which is done in the unroll stage and requires as many cycles as the request size in terms of cache blocks. In cases of remote write requests, data needs to be read from a local buffer and attached as payload in the outgoing packets, before they are forwarded to the network router. In addition to the QP and ITT table, the RGP uses the following SRAM structures:

- **Load Queue for WQ entries.** WQ entries are read from remote L1 caches of the same chip. A remote L1 cache may incur a latency of several tens or hundreds of cycles [43], depending on the chip size and interconnect. In order to avoid stalling the RGP upon every remote block read, we provision the RGP with a Load Queue to allow multiple parallel outstanding WQ reads.
- **Data Load Queue.** Used exclusively for remote write requests, the Data Load Queue stores packet headers while it waits for the payload from the local memory hierarchy, which has to be attached to the packet header before the packet is injected in the network. This structure is sized according to the expected average memory access latency, to avoid stalling the RGP.

3.6.3 Request Completion Pipeline

The RCP, shown in Figure 3.10, receives replies from the network, matches them to the original requests, and notifies the application upon each request's completion. Upon a reply packet arrival, the RCP uses the packet's transaction id field to index the ITT table. The counter in the ITT is decremented, and the associated QP id, WQ index, and base local buffer address are read out.

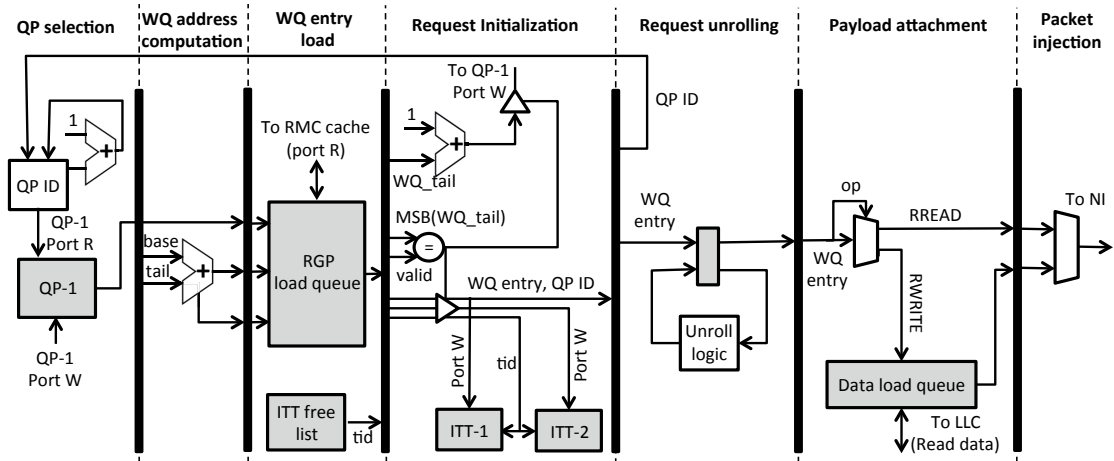


Figure 3.9 – The Request Generation Pipeline (RGP). SRAM structures appear shaded.

If the reply carries payload, it needs to be written in a local buffer. The target virtual address is computed by adding the packet’s offset field to the buffer base address, that address is translated, and the data block is sent out to the LLC. If the counter reaches zero, i.e., that packet was the last of a request, the RCP reads the CQ base address and its current head from the QP table, writes the request’s corresponding WQ index to the CQ’s head, and increments the CQ head. The application that polls at the CQ’s tail, will eventually identify the new valid CQ entry, which indicates the completion of the WQ entry with index “WQ index”. In addition to the QP and ITT tables, the RCP uses the following SRAM structures:

- **Data Store Queue.** Used exclusively for packets that contain payload (i.e., remote read replies), the Data Store Queue temporarily holds data to be written in the LLC. Its depth should be enough to avoid filling up in the steady operation state, which would result in pipeline stalls.
- **CQ Compaction Buffers.** These buffers in the last stage of the RCP are an optimization to reduce on-chip traffic. Every CQ entry is 1 byte, as it only contains the WQ index of the WQ entry the request originated from. To amortize the cost of ping-ponging a block between the RCP and the application core’s L1, we can compact up to 16 CQ entries per CQ (as the RMC cache has a 16-byte interface) before we flush them to the RMC’s cache. This is a batching optimization that can be dynamically enabled under high load.

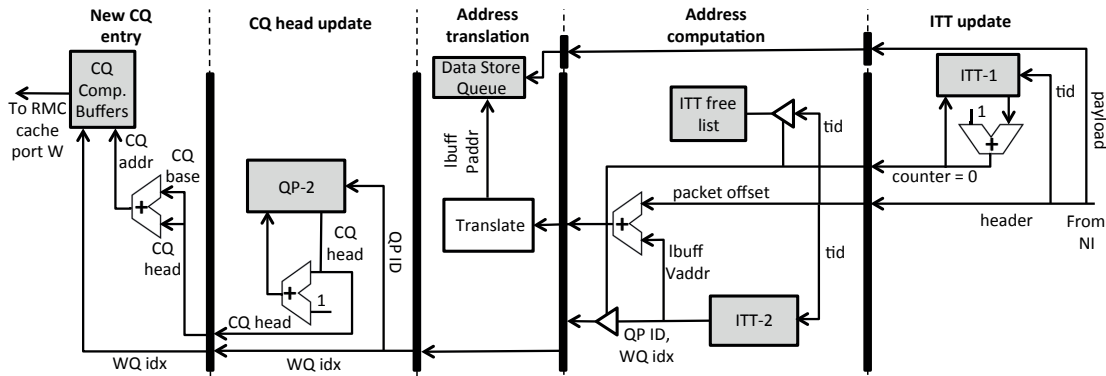


Figure 3.10 – The Request Completion Pipeline (RCP). SRAM structures appear shaded.

3.6.4 Remote Request Processing Pipeline

The RRPP is the simplest pipeline in terms of protocol processing complexity. It services incoming remote requests by reading or writing local memory and responding with the appropriate reply packet. The RRPP only features a single private SRAM structure, the *RRPP Memory Queue*. Similar to the RGP’s Data Load Queue, this structure keeps a packet header, while the corresponding local memory access is in progress. Because soNUMA does not provide ordering guarantees, the RRPP memory queue is a simple SRAM structure that does not need to be probed by younger requests. This is in contrast to a processor’s load-store queue, which is organized as a power-hungry CAM that must be probed on each memory access to handle ordering and memory reference dependencies.

RRPP address translation. soNUMA provides the abstraction of globally addressable, virtual address spaces. By adding an extra layer of indirection, a global context is built, and every node participating in the context exposes a part of its virtual memory as part of the global address space. The RRPP receives remote requests that address these exposed memory regions of the local node by specifying (context id, offset) pair. Thus, before the RRPP accesses memory to serve the incoming request, it needs to a) translate this pair into a virtual address, and b) translate the resulting virtual address to a physical address. We use context ids as address space identifiers (ASID) are used in modern TLBs, to directly translate the (context id, offset) pair into a physical address, thus avoiding a separate first translation stage.

3.6. RMC Microarchitecture

| Structure Name | # of Entries | Entry Size | Total Size | Area (mm ²) | Power (mW) | Description |
|------------------------|--------------|------------|------------|-------------------------|------------|---|
| QP table | 80 | 11B | 840B | 0.004 | 2.8 | Holds QP (WQ and CQ) information. |
| ITT table | 120 | 78b | 1.2KB | 0.008 | 1.7 | Tracks in-flight transfers. |
| RGP - Load Queue | 8 | 22B | 176B | 0.008 | 4.4 | Tracks issued WQ entry reads. |
| RGP - Data Load Queue | 23 | 81B | 1.8KB | 0.08 | 53 | Tracks outstanding data loads, for remote write operations. |
| RCP - Data Store Queue | 23 | 70B | 1.6KB | 0.06 | 40 | Tracks outstanding data writes, for remote read replies. |
| RCP - CQ Buffers | 8 | 16B | 128B | 0.005 | 2.5 | Compacts CQ entries before writing them back to cache. |
| RRPP - Memory Queue | 23 | 81B | 1.8KB | 0.08 | 53 | Tracks outstanding reads & writes, to send back response upon completion. |
| Total | | | 7.5KB | 0.245 | 157 | |

Table 3.1 – Estimated area and power for RMC SRAM structures.

3.6.5 RMC Area and Power Estimation

The RMC is a small and simple IP block, purposefully designed so to facilitate its on-chip integration. Its logic is minimal; the RMC’s area and power dissipation are dominated by its SRAM structures. In the following SRAM structure sizing analysis, we assume an RMC design where each pipeline is capable of handling a data transfer bandwidth of 16GB/s. This number roughly corresponds to the effective bandwidth that can be delivered by a high-end DDR4 channel, and we provision each pipeline to be capable of handling such bandwidth, to cover all data transfer scenarios: remote reads only (stress on RCP), remote writes only (stress on RGP), or processing incoming requests only (stress on RRPP).

We use McPAT [100] to estimate the area and power of the RMC’s SRAM structures assuming a 32nm process technology and a frequency of 2GHz. Table 3.1 summarizes the complete set of structures, and their capacity, area, and power consumption. In the rest of this section, we describe an instantiation of the SRAM structures for the RMC under a set of deployment assumptions, and explain the sizing strategy for each structure, to facilitate recalculation under different assumptions.

Chapter 3. The Scale-Out NUMA Architecture

QP Table. Each QP table entry contains the base physical addresses of a WQ and a CQ, the WQ's tail index, and the CQ's head index. In our current design, a queue index is 8 bits. With a 48-bit physical address space and a 4KB-page alignment of the queues, each QP table entry is 88 bits. Based on the soNUMA protocol, one QP per context per core is required. Provisioning for 8 cores per RMC and 10 concurrently used contexts, each QP table holds 80 entries. The logically single QP table consists of two physical structures, in order to reduce the structures' porting requirements. Note that the QP table sizing does not limit the maximum number of usable QPs, as it is possible to add a mechanism to spill QP entries to the chip's last level cache and memory. While this can have a negative impact on performance, the effects can be ameliorated by implementing smart mechanisms that cycle through the QP entries and prefetch them from memory to the QP table in a timely manner.

ITT Table. Each ITT table entry is 78 bits and contains a 19-bit counter, as well as the QP id, the WQ index, and the local buffer base address of the request to which the ITT entry is assigned. To avoid RGP stalls, the number of ITT entries needs to match the expected number of in-flight transactions. Assuming a 256-node 3D torus topology (10-hop diameter), 35ns latency per hop [165], 120ns of RRPP servicing at the remote node, we account for an average round-trip of 470ns. Using Little's Law, for 470ns average latency and a target peak bandwidth of 16GB/s, we provision the RMC with 120 ITT entries. The ITT table is also divided into two physical structures, for the same reasons described before. The first structure contains the counter, while the second the rest of the fields.

RGP Load Queue for WQ Entries. As a cache block fits four WQ entries, we provision for one remote L1 access every four WQ reads. Each entry of the Load Queue is 22 bytes, accounting for a 48-bit physical address and a 16-byte WQ entry, and we allocate 8 entries in total to hide the latency of on-chip data transfer latencies and allow polling multiple WQs concurrently.

RGP Data Load Queue. Each entry contains an address, a packet header, and a field for the block-sized payload, for a total of 81 bytes. Provisioning for an average latency of 90ns (DRAM latency + on-chip interconnect), the structure has 23 entries.

RCP Data Store Queue. Similar to the RGP Data Load Queue, we provision the structure with 23 entries. Each entry is 70 bytes (address + payload).

RRPP Memory Queue. Similar to the RGP Data Load Queue, this structure has 23 entries and each entry is 81 bytes.

Overall, an RMC sized to deliver a peak bandwidth of 16GB/s and serve 8 cores with 10 concurrently active QPs each, comes at an area cost of $\sim 0.25\text{mm}^2$ and a peak power dissipation of 156mW. In comparison, the size and TDP of a low-end server core such as the ARM Cortex-A57 (64-bit 3-way OoO) estimated at a 32nm technology (projected from reported numbers at 20nm [11, 108]) is 5.3mm^2 and 2.4W, $\sim 20\times$ larger and $\sim 15\times$ more power hungry than the RMC. Note that our estimations for the RMC only include its SRAM structures, which should dominate the area and power cost as the RMC's logic is very simple and only performs trivial computations.

As part of his MSc thesis at EPFL, Hussein Kassir prototyped the soNUMA architecture on the Intel HARP hybrid CPU-FPGA platform. Synthesis tools reported his RMC implementation's area to be 0.35mm^2 at a 40nm technology. This roughly corresponds to an area of 0.22mm^2 at a projected 32nm technology, being very close to the above first-order approximation.

In conclusion, soNUMA's protocol simplicity enables the design of a lean protocol controller that can easily be integrated on chip. On-chip integration of the RMC removes one of the last obstacles to fast remote memory access, namely the high PCIe/DMA latency modern systems incur on CPU-NI interaction (see Section 3.1).

3.7 Chapter Summary

Scale-Out NUMA (soNUMA) is an architecture, programming model, and communication protocol for low-latency big-data processing. soNUMA eliminates kernel, network stack, and I/O bus overheads by exposing a new hardware block—the remote memory controller (RMC)—within the cache coherent hierarchy of the processor. The RMC is directly accessible by applications and is a small hardware structure that is connected directly into a NUMA fabric. In the second part of this thesis (Chapter 6), we describe a concrete implementation of the soNUMA architecture and demonstrate remote memory access latencies within a small factor of local DRAM access (3–4×). By bringing shrinking the latency gap between local and remote memory access, soNUMA enables distributed memory systems with the abstraction of a global memory pool, where the aggregate memory resources are seamlessly accessible.

4 Remote Memory Operations with Richer Semantics

One-sided operations, offered by soNUMA and modern RDMA technology, offer the fastest path to remote memory by avoiding CPU interaction at the remote end. Several modern software frameworks for in-memory distributed computing have started making use of these operations offered by RDMA, showing dramatic performance improvements. Existing RDMA technologies such as InfiniBand can deliver remote memory access at a latency as low as 10–20× of local memory access (1–2μs versus 60–100ns). soNUMA further shrinks this gap through tight integration and a leaner protocol, bringing remote memory access latency just within 3–4× of local memory access, providing even stronger motivation for the use of one-sided operations.

While fast, existing one-sided operations provided by current RDMA technology have scant semantics, offering read, write, and limited atomic operations. For any remote memory access more complicated than that (e.g., simple pointer traversal at the remote end), software either has to employ a sequence of one-sided operations, or completely give up on the capability of direct remote memory access and resort to conventional RPCs. The former is rarely a good option; for the vast majority of remote memory access that cannot complete with a single one-sided operation, RPC is usually the option of choice. As a result, the usability of one-sided operations is severely limited.

We argue that one-sided operations need not be as semantically limited as they currently are.

Chapter 4. Remote Memory Operations with Richer Semantics

Extending their semantics to better accommodate common memory operations performed by software stacks deployed over distributed memory systems can yield significant performance improvement, software simplification, and reduction in required CPU cycles per operation (an endeavor strongly motivated by ongoing technology trends, as argued in Section 2.5). Network-compute co-design, as advocated in this thesis, not only facilitates the introduction of new one-sided operations with richer semantics, but also enables functionality that would otherwise not be possible with conventional PCIe-attached NI logic.

A key requirement to justify the introduction of any new one-sided operation, along with the NI hardware extensions it entails, is the identification of functionality that is ubiquitously needed by applications, yet is contained and simple enough to implement in hardware. As a case study for richer remote memory operations, we study the concurrency control aspect of distributed memory software and identify such a candidate operation. In particular, we study distributed object stores and identify that a very common operation, reading a data object from remote memory *atomically*, is surprisingly inefficient in modern distributed memory systems.

The inefficiency stems from a mismatch between what software needs and what current RDMA hardware provides. While software data objects have an arbitrary size, often larger than a single cache block (typically 64B), one-sided operations cannot guarantee atomicity for any memory access straddling multiple cache blocks—a direct consequence of their DMA-based implementation. To overcome this limitation, data management systems leveraging one-sided operations employ software mechanisms such as locks or optimistic concurrency control to enforce atomic remote object accesses [53, 127, 171].

Providing object atomicity in software in current systems incurs a performance penalty, though currently acceptable. However, software-provided atomicity will gradually become a performance limiter as modern fabrics and new architectures such as soNUMA drastically improve inter-server communication’s latency and bandwidth. Indeed, our study shows that the state-of-the-art software mechanism delivering atomic object accesses in FaRM [53] accounts for up to 50% of the end-to-end remote memory access latency for large objects (8KB) on soNUMA. Consequently,

providing atomic remote object access becomes a first-order performance concern calling for architectural support to replace the costly software mechanisms.

Since remote object reads represent the most frequent remote memory operation, introducing a one-sided hardware primitive with the semantics of an *atomic remote object read* is critical to the performance distributed memory systems. Motivated by the need to provide hardware support for the ubiquitous operation of accessing objects from remote memory atomically, we perform a design space exploration to identify the best approach to offload this functionality to the NI logic. As a result of this exploration, we introduce *SABRe* (Single-site Atomic Bulk Read), a new one-sided primitive with stronger semantics than any existing one-sided primitive. We then present *LightSABRe*, a lightweight and high-performance implementation of *SABRe* that leverages coherent NI integration for eager detection of object atomicity violations. Our evaluation of *LightSABRe* on an instance of an *ICONIC* system in Chapter 7 shows that the introduction of hardware support for atomic remote object reads completely removes the software overhead associated with providing the desired atomicity trait.

4.1 Atomic Remote Object Reads

In this section, we establish the importance of atomically accessing objects from remote memory for distributed memory systems, and advocate the introduction of a new one-sided operation with the semantics of an atomic remote object read. We start with some background on modern in-memory object stores, underline the inability of existing one-sided operations to accommodate the need for atomic remote object access, and demonstrate why the negative effect of this limitation will be amplified on future systems with faster networking.

4.1.1 In-Memory Object Stores

In-memory object stores (or key-value stores) are critical components of many modern cloud systems. Several large-scale services are powered by well-engineered object store software stacks, which are designed to scale to thousands of servers and petabytes of data and serve billions of requests per second [16, 26, 48, 132]. There are several well-known representatives such as Memcached [2], Redis [4], Dynamo [48], TAO [26], and Voldemort [3], which are deployed in production environments of large service providers such as Facebook, Amazon, Twitter, Zynga, and LinkedIn [9, 105, 132, 167]. The popularity of these systems has resulted in considerable research and development efforts, including open-source implementations [1], research prototypes [14, 140] and a wide range of sophisticated, highly tuned frameworks that aspire to become the state-of-the-art solution [53, 101, 103].

An emerging category of software frameworks for in-memory object stores is designed to leverage the low communication offered by RDMA technology, which recently started penetrating the datacenters. These object stores take advantage of RDMA one-sided operations to deliver fast access to remote objects and dramatically improve system performance [53, 89, 127, 171].

For applications that operate on structured data, the granularity of an operation (and also the minimum unit of transfer when accessing remote memory) is the object. The size of these objects is application-specific, and can range from a few bytes to several kilobytes [104]. Unfortunately,

RDMA technology relies on PCIe DMA to transfer data between the memory and the network, and therefore its remote memory access semantics are limited to read, write, and cache-block-sized atomic operations, such as remote CAS. The latter only provide atomic access to a memory region not exceeding a single cache block in size. No existing hardware mechanism can provide atomic access to larger memory regions; thus, the challenge of accessing objects atomically falls on the software.

4.1.2 Atomic One-Sided Operations

Several modern frameworks for in-memory distributed computing rely on one-sided RDMA operations (e.g., Pilaf [127], FaRM [53], DrTM [171]). One-sided operations deliver fast access to remote memory by completely avoiding remote CPU involvement, but offer limited semantics. In most cases, one-sided operations are only used for reads, while writes are sent to the data owner over an RPC. This common design choice simplifies software design and is motivated by the read-dominated nature of most applications.

To the best of our knowledge, the only system using one-sided operations for both reads and writes is DrTM [171]. DrTM uses HTM as an enabler for one-sided writes, relying on it to detect local conflicts with incoming remote writes and abort conflicting local reads. While DrTM introduces an interesting design point, we focus on the common case of *one-sided read operations*. Because HTM functionality is bounded to its local node's coherence domain, it cannot be directly used for atomic multi-cache-block remote reads.

Modern frameworks rely on software techniques to complement the limited semantics of one-sided operations, which only offer cache-block-sized atomicity. A defining characteristic for these techniques is the employed concurrency control method: locking versus optimistic concurrency control. Combining locking with one-sided reads is simple. Each object in the data store has an associated lock. When a node requires atomic access to a remote object, it issues a first one-sided (cache-block atomic) RDMA CAS operation to acquire the remote object's lock, followed by another one-sided operation to access the object atomically—locking prevents any conflicts.

Chapter 4. Remote Memory Operations with Richer Semantics

However, remote lock acquisition comes with two drawbacks. First, it increases the latency of remote memory access by an additional network roundtrip. Second, it introduces fault-tolerance concerns, as a node's failure may result in deployment-wide deadlocks, turning the RDMA cluster into a single failure domain and thus jeopardizing the traditional high resilience of scale-out deployments. The latter concern can be addressed for reads by replacing conventional locks with lease locks, as illustrated by DrTM. Unfortunately, lease locks are sensitive to clock skew across the deployment's machines, and their duration can significantly impact concurrency and abort rates.

Optimistic concurrency control addresses the shortcomings of remote locking. Driven by the observation that most workloads are read-dominated, and hence the probability of a conflict is low, optimistic concurrency control relies on conflict detection rather than conflict prevention for high performance (Pilaf [127], FaRM [53]). Since hardware only provides cache-block-sized atomicity, remote reads are paired with ad hoc software-based mechanisms for conflict detection, which do not come for free. For instance, Pilaf [127] embeds a checksum in each object's header as additional metadata. The checksum is recomputed after every update, and remote readers compute the checksum of the object's data to compare it to the object's checksum—a mismatch indicates an atomicity violation. Unfortunately, while conceptually simple, the checksum mechanism is expensive, as the cost of CRC64 is about a dozen CPU cycles per checksummed byte [127]. For KB-sized objects, this overhead can grow to tens of thousands of CPU cycles (i.e., several microseconds) per object transfer.

FaRM [53] introduces the more efficient approach of per-cache-line versions: every object has a 64-bit version in its header, and a number of that header's least significant bits are replicated in a per-cache-line header. Writers update all versions upon an object update, and readers compare all cache-line versions to detect atomicity violations before consuming the data. While computationally cheaper than checksums, per-cache-line versions still introduce measurable CPU overhead for both readers and writers. More importantly, per-cache-line versions prevent zero-copy object transfers: before the application can use the object, the CPU has to extract the clean data into a buffer by stripping off the embedded per-cache-line versions. This overhead

applies to all types of read/write accesses, both local and remote. Despite the overhead, FaRM's per-cache-line versions mechanism is the state-of-the-art approach to provide *optimistic* and *atomic* one-sided reads from remote memory.

4.1.3 Implications of Faster Networking

While RDMA is the leading product in providing fast inter-node communication and remote memory access, its performance is ultimately capped by the latency overhead of the PCIe interface [134]. With single-cache-line RDMA reads exceeding $1\mu\text{s}$ in latency, the latency of accessing remote memory alone dwarfs the latency of consequent local memory operations, such as the post-transfer data extraction and version checks required when using FaRM's per-cache-line versions technique, which may only account for a few hundred nanoseconds. Thus, FaRM's design choice does not effectively impact the end-to-end latency of one-sided RDMA reads.

However, RDMA technology is evolving, moving away from PCIe and towards tightly integrated solutions. For instance, AppliedMicro's X-Gene 2 [107] and Oracle's Sonoma [109] integrate an RDMA controller on chip. The trend towards tight integration is not limited to the chip level. In fact, recent technological advancements have led to the emergence of tightly integrated chassis- and rack-scale systems, such as HP's Moonshot [75] and The Machine [76], Oracle Exadata [139], and AMD SeaMicro [50] (see Section 2.4).. These systems interconnect a large number of servers, each with an on-chip NI, using a supercomputer-like lossless fabric. NI integration and short intra-rack communication distances help reduce communication delays. At the same time, research proposals (e.g., Firebox [15], soNUMA [134]) show how sub- μs remote memory access is achievable through the combination of lean network protocols, tight integration, and contained physical scale. We envision that emerging rack-scale systems will soon adopt such lightweight network stacks, which, combined with tightly integrated SoCs, will significantly improve the performance of remote memory access in terms of both latency and bandwidth as compared to existing RDMA solutions. We expect our observations to also apply to larger systems, such as datacenters, in the near future, when high-performance networking solutions start getting deployed at large scale.

4.1.4 The Case for SABRe

In the context of emerging tightly integrated rack-scale systems, we evaluate the performance impact of software-based atomicity mechanisms. As a case study, we use soNUMA and run a key-value store on top of FaRM [53]. We simulate two directly connected soNUMA nodes to measure the latency breakdown of one-sided remote reads. Object atomicity is achieved through FaRM’s per-cache-line versions mechanism. Methodology and simulation details can be found in Section 7.2.

Figure 4.1 shows the end-to-end latency breakdown of an atomic remote object read. For every object size, we break down the latency into three components: the soNUMA transfer time, the time spent in the FaRM framework and application code, and the time spent by the core extracting useful data from the transferred object, by stripping off and comparing the per-cache-line versions to check for atomicity violation. We observe that the latency of one-sided reads over soNUMA starts at just 3–4× of local memory access and scales sublinearly with object size, due to soNUMA’s high-bandwidth fabric. In contrast, while the software atomicity check latency is negligible for small objects (~10% for 128B objects), it scales almost linearly with object size and thus quickly outgrows the soNUMA transfer latency, accounting for 50% of the end-to-end latency for 8KB objects. Furthermore, a fraction of the latency goes to FaRM buffer management, which is necessary for storing the transferred data, before it is cleaned up and moved to the application’s buffer. Importantly, the latency overhead of software-provided atomicity directly corresponds to wasted CPU cycles, which is an increasingly more precious resource, as we argued in Section 2.5.

We introduce a new *Single-site Atomic Bulk Read (SABRe)* one-sided primitive in hardware that removes the atomicity-associated software overhead and enables zero-copy transfers, by obviating the need for intermediate buffering.

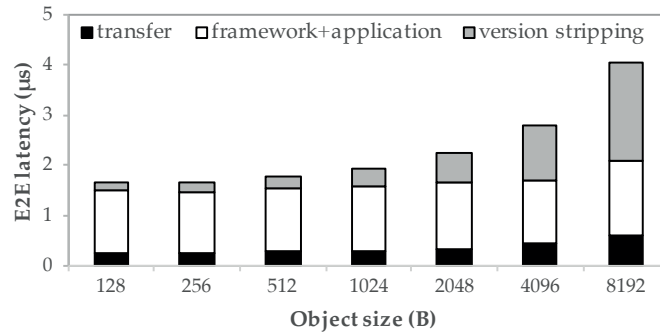


Figure 4.1 – End-to-end remote object read latency using the per-cache-line-version software atomicity check mechanism on FaRM over soNUMA.

4.2 SABRe Design Space

In this section, we perform a design space exploration to identify the best design practices for the realization of SABRe, a new one-sided primitive with the semantics of an atomic remote object read. We consider architectures that feature an on-chip integrated NI with a protocol controller supporting one-sided remote memory operations, such as soNUMA.

4.2.1 Destination-Side Concurrency Control

Table 4.1 summarizes the design space for atomic remote object access, with or without hardware support. In our taxonomy, the terms *source* and *destination* refer to the origin of a request rather than the location of the requested data. Under that definition, all software-based approaches leveraging one-sided operations essentially implement *source-side* concurrency control since the destination side’s CPU is not involved. DrTM relies on acquiring remote locks, with locking explicitly controlled at the source prior to accessing the remote object’s data. FaRM and Pilaf implement different optimistic concurrency control mechanisms to enforce atomicity, but as the source has to perform post-transfer atomicity checks, both are source-side mechanisms.

Introducing hardware support expands the design space, with possible source-side or destination-side accelerators. For example, one can easily envision source-side hardware accelerators that deal with hardware checksums or per-cache-line versions. However, such an approach has a number

Chapter 4. Remote Memory Operations with Richer Semantics

| | Source | Destination |
|---------|------------------------|-------------|
| Locking | DrTM [171] | SABRe |
| OCC | FaRM [53], Pilaf [127] | |

Table 4.1 – Design space for one-sided atomic object reads.

of weaknesses. First, cache-block-sized replies with payloads can arrive out of order. Depending on the mechanism, these replies might need to first be reordered, requiring intermediate buffering (e.g., in the case of checksums). Second, the application’s whole data store needs restructuring just to embed the necessary per-object metadata that enable atomicity checks for one-sided remote operations. Such restructuring also affects the performance of all local operations (reads & writes), as they have to comply with the modified data layout’s rules: readers might need to unpack data before consumption, writers need to always update corresponding metadata as well. Ultimately, the weakness of source-side mechanisms is that they are limited to post-transfer atomicity checks and thus require additional metadata embedded in—and always transferred with—the requested remote object.

In contrast, destination-side hardware support offers more appealing opportunities. Providing concurrency control directly at the destination is a natural option; this is where the target data is located and, thus, where synchronization between concurrent accesses to that data occurs. Therefore, destination-side concurrency control offers higher flexibility and efficiency, such as leveraging local coherence for online atomicity violation detection and obviating the need to maintain and transfer any additional metadata for post-transfer validation at the source. For instance, locking directly at the destination alleviates both drawbacks of remote locking (i.e., increased latency and fault-tolerance concerns). Similarly, reading data optimistically while actively monitoring atomicity at the destination obviates the need for restructuring the data store to embed special metadata required by optimistic concurrency control mechanisms (like the ones discussed in Section 4.1.2), and also allows for early conflict detection.

Overall, destination-side concurrency control comes with many desirable properties, which trump source-side alternatives. Therefore, our hardware extensions for SABRe target Table 4.1’s

rightmost column, representing the first destination-side concurrency control solution solely based on one-sided operations.

4.2.2 Design Goals

Given the advantages of destination-side concurrency control, we now define the three design goals (DG) necessary for an efficient SABRe hardware design:

- [DG1] *Minimal single-SABRe latency.*
- [DG2] *High inter-SABRe concurrency.* The mechanism should be able to utilize all the available bandwidth even with a multitude of small SABRes.
- [DG3] *Low hardware complexity/cost* (e.g., no modifications to the chip's coherence protocol).

A straightforward and efficient approach to implement SABRe is lock acquisition at the destination. Since objects typically have a header with a lock for synchronization between local threads, the controller can acquire the lock as any other local thread. To support high reader concurrency, shared reader locks are essential, yet only add minimal complexity to the locking logic.

For read-dominated applications, optimistic concurrency control is typically preferable to locking. For that reason, many modern software frameworks, such as key-value stores and in-memory DBMSs, do not employ reader locks, but rely on optimistic reads for high reader throughput (e.g., [53, 103, 122, 166, 170]). To enable optimistic reads, objects have a version in their header, which is incremented at the beginning and at the end of each update. To determine a read's atomicity, the controller simply compares the version's value before and after the read. Enhancing the protocol controller at the destination for optimistic reads is also quite simple: instead of acquiring an object's lock, the controller can at any time assess the object's state by reading the object's version.

The biggest drawback of a naive implementation of either mechanism for hardware SABRe (locking, or optimistic concurrency control using version checks) is the requirement for a serialized first access to read the version or acquire the lock prior to any data access. In the general case when the

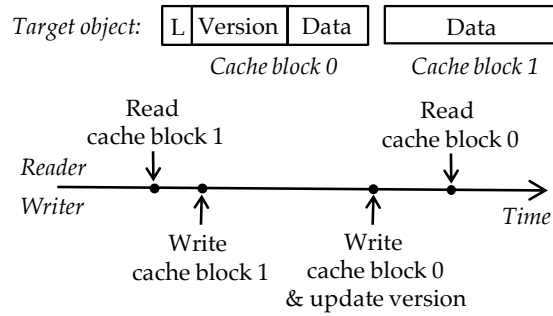


Figure 4.2 – Reader–Writer race example.

target object is in memory, this requirement can significantly increase the object read latency by exposing the full latency of that first memory access (i.e., ~60–100ns), incurring a considerable latency overhead especially for small objects. To illustrate, on a tightly integrated system such as soNUMA, this serialization can increase the end-to-end latency of a two-cache-block SABRe by up to 40% (details in Section 7.3.1). In the case of version comparison, an additional serialized load to re-read the object’s version after all data has been read is also required. However, the latency overhead of this second load is less critical, as it will likely hit in an on-chip cache.

Violating the *read-version-then-data* (or *acquire-lock-then-read-data*) serialization to avoid exposing that latency penalty can result in undetected atomicity violations. Figure 4.2 illustrates a potential race condition that may arise if we overlap the version read with data read. In this example, we assume that the protocol controller receives a remote read request for an object that spans two cache blocks and that it implements optimistic concurrency control using the object’s header version (the example equally holds in the case of locking). Cache block 0 contains the object’s header, with the corresponding lock and version, and a writer currently holds the object’s lock. If the controller issues a read for cache blocks 0 and 1 concurrently, the read for cache block 1 may complete first, as any reordering can occur in the memory subsystem and on-chip network. Then, the writer modifies cache block 1, updates the object’s version and frees the lock (cache block 0). After this intervention, the controller’s read for cache block 0 also completes, finding a free lock. At this point, the controller has no means of detecting the writer’s intervention and wrongly assumes that the object has been read atomically, while in practice it has retrieved the latest value of cache block 0 and an old value of cache block 1. Reading the object’s version

before issuing any other read operation, though, guarantees that such races causing transparent atomicity violations cannot occur.

A careful implementation of the simple hardware SABRe mechanisms mentioned above can satisfy DG2 and DG3 (i.e., high inter-SABRe concurrency and low hardware complexity), but not DG1 (i.e., minimal single-SABRe latency), because of the serialization limitation. We can break the *read-version-then-data* problem by leveraging speculation techniques. The tight integration of the protocol controllers with the chip also implies integration into the chip's coherence domain. This integration enables a variety of options regarding atomicity enforcement mechanisms. Speculation techniques proposed for relaxing memory ordering (e.g., fence speculation) [22, 67, 130], or conflict detection and resolution mechanisms employed by HTM could be directly applicable to register and guard a SABRe's address range during its lifetime. However, those mechanisms are unnecessarily complex and contradict DG3.

Our key insight is that a SABRe implementation requires considerably simpler functionality than HTM or other sophisticated speculative structures employed by aggressive cores to relax memory order. First, a SABRe only involves reads and no writes. Second, SABRes naturally come with software-provided characteristics that can simplify hardware requirements; that is, a SABRe is by definition accesses to structured data that comprise objects in a data store rather than accesses to arbitrary memory locations. Every object typically features a header with associated metadata, such as a lock and/or a version, and a range of sequential addresses containing data. Writers update this header accordingly upon each write to the object. We can thus expose these semantics to the hardware, and rely on a hardware-software contract to simplify the hardware.

4.2.3 Safely Overlapping Lock and Data Access

We now leverage our insights from Section 4.2.2 to design a lightweight hardware mechanism that safely overlaps an object's lock/version access and data read, meeting DG1. It is possible to hide the serialization latency and read all data in parallel instead, thus extracting maximum memory-level parallelism (MLP), as long as we provide a mechanism to detect any data atomicity

Chapter 4. Remote Memory Operations with Richer Semantics

violation that may occur before the completion of the object's first version read or lock acquisition. Since memory accesses can be reordered by the memory subsystem, requested data may return in any order. We define the time between issuing an access to the SABRe's first cache block, which contains the object's version or lock, and its completion, as that SABRe's *window of vulnerability*. Within that window, all data are speculatively read, as it is unknown whether the read operation is racing against a concurrent write to the same object, risking a transparent atomicity violation as in Figure 4.2's example.

We rely on the integration of the protocol controller in the chip's coherence domain to detect atomicity violations during this window of vulnerability. Given that a SABRe comprises a sequence of reads to consecutive addresses, the mechanism only needs to snoop coherence traffic for an *address range* rather than a set of independent addresses. At the high level, such range tracking can be trivially implemented by a structure that just keeps track of a SABRe's starting address and length, allowing for simple indexed lookups through simple base-and-offset arithmetic. Using this structure, the loads comprising a SABRe can be performed in parallel, exploiting maximum MLP. The critical addresses are trivially captured as an address range and are snooped upon each reception of a coherence invalidation message during the window of vulnerability. An invalidation matching the address of an already read block triggers an abort of the corresponding SABRe.

Implementing such address range snooping structure in hardware is much simpler than an out-of-order processor's load-store queue, or an address resolution buffer [64, 65, 154]: no dynamic memory disambiguation or associative searches, within or across different address range snooping structures are required. We provide an implementation of the proposed mechanism in the following section.

4.3 LightSABRe

In this section we describe *LightSABRe*, an implementation of a destination-side concurrency control mechanism for SABRe that performs address range snooping using stream buffers.

We describe an implementation based on a generic on-chip protocol controller for one-sided operations integrated into the chip's coherence domain.

4.3.1 Address Range Snooping Implementation

We implement address range snooping by leveraging an adaptation of stream buffers [87], illustrated in Figure 4.3. Every inbound SABRe request is associated with a stream buffer; starting from the SABRe's *base physical address*, each SABRe cache block is mapped to an entry of the associated stream buffer. Since all blocks comprising a SABRe are consecutive, issued loads for the same SABRe map to consecutive stream buffer slots (with the exception of SABRes spanning two non-consecutive physical pages). The stream buffer holds the range of addresses touched by the controller during the window of vulnerability.

Integration of such stream buffers with the protocol controller allows overlapping the object's lock/version access and data read, thus enabling maximum MLP for a single SABRe even during the window of vulnerability. The controller can keep pushing consecutive cache-block-sized read requests to the memory hierarchy as long as (i) the SABRe's associated stream buffer is deep enough to contain all the outstanding loads; and (ii) there is no boundary crossing between two non-consecutive physical pages. If the controller hits any of these two limitations while issuing loads for a SABRe, that SABRe simply needs to stall, without any correctness implications. Once the window of vulnerability is over (i.e., the version/lock is accessed), the stream buffer is not useful anymore and reading the object's data can seamlessly continue without the previous two limitations. Page boundary crossing during the window of vulnerability is an infrequent event that does not raise performance concerns, especially given the common RDMA/soNUMA practice of using superpages for the memory regions exposed to the global address space (e.g., [53]).

A stream buffer's entries represent a sequence of *loads to consecutive physical memory addresses*. With the exception of the head entry, stream buffer entries do not store an address. Instead, each entry's corresponding address is deduced as a simple addition of the stream buffer's associated base address and its location offset. This property provides a cheap lookup mechanism through

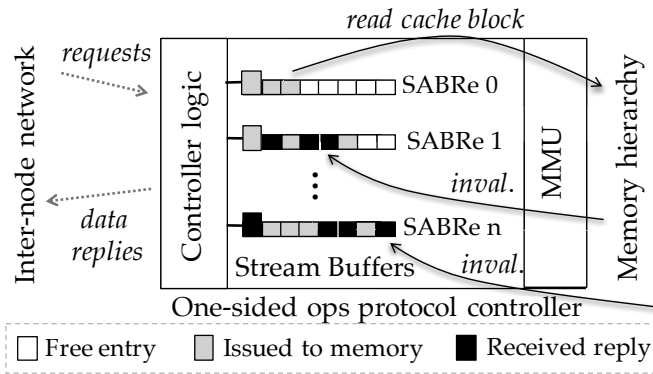


Figure 4.3 – LightSABRe: Leveraging stream buffers to safely overlap lock & data access.

simple indexing rather than associative search. Data replies arriving from the memory hierarchy are not stored in the stream buffer either, but are directly sent back to the requester by the protocol controller.

In Figure 4.3’s example, white stream buffer entries are currently unused, gray entries denote cache-block accesses that have been issued to the memory hierarchy and await a reply, while black entries have already received a reply and the payload has already been sent back to the requester. The controller issues the third cache-block read request for *SABRe 0*. At the same time, a coherence invalidation message is received for *SABRe 1*’s fifth cache block. Since *SABRe 1*’s head cache block has not yet been accessed, this invalidation indicates a possible race condition with a writer, so *SABRe 1* will abort. In contrast, *SABRe n* does not abort upon reception of an invalidation for its last stream buffer entry, as it has already accessed the head entry’s block; this invalidation must have been triggered by an eviction.

The key insight regarding stream buffer provisioning that makes our mechanism lightweight and scalable is that both the number and depth of required stream buffers is orthogonal to the SABRe’s length. Sizing is only a function of the memory hierarchy and the target peak bandwidth of the controller that is enhanced with LightSABRe. The number of stream buffers defines the maximum number of concurrent SABRes the controller can handle; there should be enough stream buffers to allow the controller to utilize its full aggregate bandwidth even for the smallest SABRes (i.e., two-cache-block SABRes). The depth of the stream buffers affects the latency of

each SABRe. The controller can keep pushing cache block load requests for a SABRe, as long as there are available slots in that SABRe's corresponding stream buffer *or* the target object's version has been read (or, in the case of locking, the object's lock has been acquired). Thus, the stream buffer's depth should be sufficient to allow pushing data load requests to the memory subsystem at the controller's maximum bandwidth, until the first request to access the object's version/lock completes.

4.3.2 System Integration

As discussed in Section 4.2.2, several modern software frameworks rely on optimistic concurrency control, allowing readers to optimistically proceed without acquiring any locks, as conflicts are expected to be rare and retries are cheap. Without loss of generality, we will focus the implementation description of LightSABRe on an optimistic concurrency control mechanism. The same principles are applicable to locking; in fact, the same implementation with minimal modifications can be used for both locking and optimistic concurrency control.

We assume that the software maintains versions for concurrency control similar in philosophy to Masstree's [122] object versions. Each object has a version in its header. Writers increment the version to acquire exclusive access to an object, and increment it again once they are done with their changes. Thus, an odd version indicates a locked object, and an even version indicates a free object. This is functionally equivalent to having a lock acquired before updating an object, and a version incremented before the lock is freed again. Therefore, without loss of generality, we assume that writers use the odd/even version mechanism for updates.

Figure 4.4 shows a protocol controller pipeline of an NI, enhanced with LightSABRe. The key entity driving a SABRe's progress is an SRAM structure, dubbed Active Transfers Table (ATT). An ATT entry represents a SABRe during its lifetime. Every ATT entry controls an associated stream buffer, and every stream buffer holds a base address, a length field, and a bitvector representing a range of consecutive cache blocks following the base address, with each bit representing a cache block. A set bit indicates that the cache block has been read from the

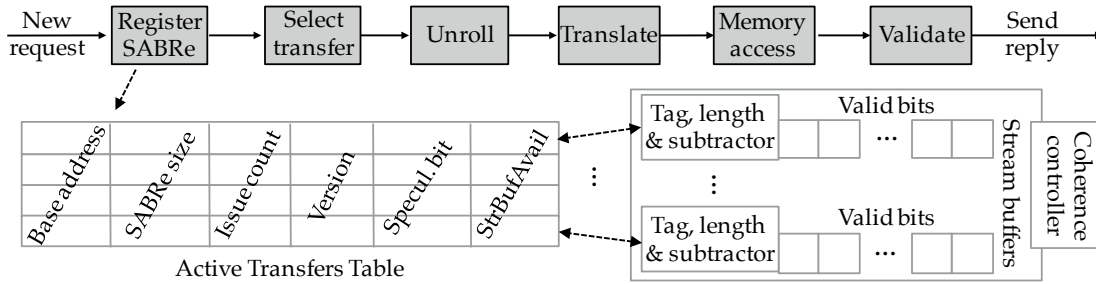


Figure 4.4 – Block diagram of a LightSABRe-enhanced NI.

memory subsystem. Each stream buffer also features a subtractor, used to determine whether a message from the memory hierarchy (reply or snoop) matches an entry in the stream buffer by subtracting the stream buffer’s base address from the received address to index the bitvector. This simple lookup mechanism eliminates associative searches within each stream buffer.

Upon the reception of a new SABRe request, the *register SABRe* stage allocates a new entry in the ATT; the request carries the *SABRe size* and *base address*. The *select transfer* stage is a simple SABRe scheduler that selects one of the active SABRes in the ATT and starts unrolling it. The *unroll* stage issues load requests for the registered SABRe and increments the *issue count* while (i) *issue count* \leq *SABRe size*, and (ii) there is a free slot in the associated stream buffer (*StrBufAvail*), or the object’s version has already been read, so the SABRe is past its window of vulnerability (*speculation bit* cleared). If condition (ii) is not met, the serviced SABRe gets descheduled and the *select transfer* stage starts servicing another active SABRe.

For every reply that arrives to the protocol controller, all stream buffers are snooped to check for an address match in their tracked address range; upon a match, the corresponding bit of the bitvector is set. A similar match is triggered by received invalidation messages; if the invalidated address matches a valid entry in a stream buffer (entry’s bit set), the invalidation is propagated to the stream buffer’s corresponding ATT entry. If the version for that SABRe hasn’t yet been read (*speculation bit* set), this event implies a race condition with a writer, and therefore the SABRe aborts. Otherwise, if the version has already been read (*speculation bit* cleared), the invalidation is ignored, as it has to be triggered by a cache block’s eviction from the chip.

The only ambiguous event is the reception of an invalidation for a stream buffer's base address, which represents the block that holds the target object's version. Such an invalidation message may be triggered by a real conflict from a writer concurrently writing the same object, or may be a false alarm triggered by the block's eviction from the chip. To avoid false conflicts, an invalidation for the SABRe's base address does not automatically abort the SABRe. Instead, we deploy the following mechanism: every cache block read from the memory hierarchy is directly sent back to the requester, and, after all payload replies for a SABRe have been sent back, the protocol controller sends a final payload-free packet indicating the transfer's atomicity success or failure. Whenever a SABRe's data accesses finish and the base address entry is still valid in the corresponding stream buffer, the NI *immediately* confirms the SABRe's success. In the uncommon event of an invalidation reception for the SABRe's base block, the NI must verify whether there was a true atomicity violation: after all data blocks for the SABRe have been read, the NI's *Validate* stage reads the object's header again and checks if the newly read version matches the ATT entry's *version* field (initialized the first time the object's header was read). A version match guarantees atomicity, while a mismatch implies atomicity violation and causes a SABRe abort.

The relative location of the lock/version in each object's header with respect to its base address is fixed for a given data store, but may vary across data stores. While LightSABRe require this information, a device driver can trivially specify that at initialization time, when it registers the data store's memory to the protocol controller, thus associating that metadata with the registered memory chunk.

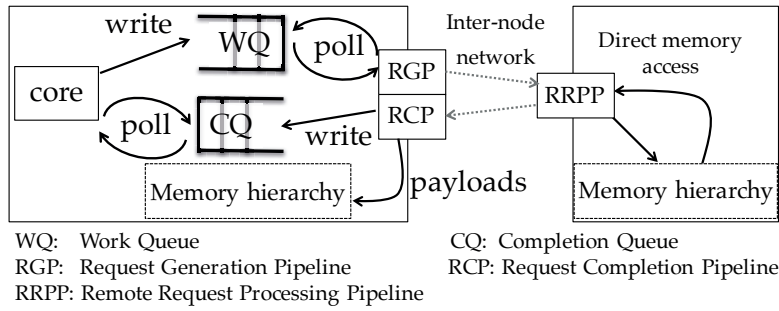


Figure 4.5 – soNUMA overview.

4.4 LightSABRe on Scale-Out NUMA

Extending the soNUMA architecture with LightSABRe is straightforward, as the required protocol and hardware modifications are limited. soNUMA’s specialized NI, the Remote Memory Controller (RMC), is directly integrated into the chip’s coherence domain, so the basic premise of the LightSABRe mechanism, the ability to snoop on coherence messages, is inherent in the architecture. soNUMA has two additional important characteristics to be taken into account when extending the protocol and architecture with the new SABRe primitive:

- (i) Remote accesses spanning multiple cache blocks are *unrolled into cache-block-sized requests* at the source node.
- (ii) soNUMA’s flow control requires a strict one-to-one request-reply protocol: every request packet has to be matched by a reply packet.

As detailed in Chapter 3, each soNUMA request is generated at the requesting node, serviced at a remote node, and completed once it returns to the requesting node. These three logical stages are handled by the RMC’s three independent pipelines, as illustrated at a high level in Figure 4.5. Since LightSABRe only involve destination-side processing, we only focus on the remote end’s pipeline, namely the *Remote Request Processing Pipeline (RRPP)*, which statelessly services incoming remote requests by reading or writing local memory.

4.4.1 Integration with RRPP

The LightSABRe-enhanced protocol controller described in Section 4.3.2 subsumes soNUMA's baseline RRPP, as the RRPP normally handles independent cache-block-sized requests and just performs address translation and direct memory access. In the case of SABRes, the independence property does not hold, as request packets belonging to the same SABRe are related. In the extended version of the RRPP, the pipeline gradually folds the received request packets belonging to the same SABRe into a single entry. For that purpose, we add two more fields to the ATT: the *SABRe id* and the *request counter*. A new SABRe is registered in the ATT by a special *SABRe registration* packet, with a *SABRe id* uniquely defined by the set of source node id, Request Generation Pipeline id, and transfer id, all of which are carried in each request packet. A registered SABRe's *request counter* is incremented for every consequent request packet belonging to the same SABRe (matching *SABRe id*). An additional limitation to the *unroll* stage is that requests to the memory hierarchy can be issued only if *issue count request counter* as well, to guarantee that the number of generated replies never exceeds the number of received requests.

Upon a SABRe abort, the RRPP could transparently retry the failed SABRe. However, we consciously opt out of this approach for two reasons. First, retrying a failed SABRe in hardware will directly increase the occupancy of the RRPP and also transparently increase the remote read's completion latency for an arbitrary amount of time from the application's perspective. Second, unless a conflict is detected on the first data block read, retrying a failed SABRe at the remote end will result in repeating some reply packets, thus breaking the request-reply flow control invariant of soNUMA. We choose to make the common case fast and expose the uncommon case of atomicity violation to software, to provide end-to-end control and flexibility. The application decides whether to retry an optimistic read after a backoff, or read the object over an RPC. Such policies are hard to implement solely in hardware, and the expected low abort rates do not justify the complexity and effort.

Properly sizing the ATT and the stream buffers, both in terms of number and depth, is key to LightSABRe's performance. As detailed in Section 4.3.1, sizing is solely determined by the

chip's memory hierarchy and the RRPP's target peak bandwidth. For example, assuming a 16-core system with an average memory access latency of 90ns and a target per-RRPP peak bandwidth of 20GBps, LightSABRe only require 16 stream buffers (one per ATT entry) with a depth (bitvector width) of 32, numbers simply derived by our target bandwidth-delay product (Little's Law). With 24 bytes per ATT entry and 11 bytes per stream buffer, the total additional per-RRPP hardware requirement is 560 bytes of SRAM storage, plus a 42-bit subtractor per stream buffer.

4.4.2 Other Protocol and Hardware Modifications

Enhancing soNUMA with SABRe operations requires a few modifications to the protocol and the RMC's two remaining pipelines, namely the Request Generation and Request Completion Pipeline (RGP and RCP). The hardware-software interface is enhanced with a new SABRe operation type and an additional *success* field in the Completion Queue entry. This field is used by the RCP in the Completion Queue entry to expose SABRe atomicity violations to the application. At the transport layer, we add two new packet types. The first is the *SABRe registration* packet, which precedes the SABRe's data request packets and contains the SABRe's total size; this is essential for the SABRe's registration at the destination node's RRPP ATT. We assume a network that guarantees in-order packet delivery, but the mechanism can be easily extended to unordered networks, by carrying that information in every request packet. The second new packet type is the *SABRe validation*, which is the last reply sent by the RRPP to indicate a SABRe's atomicity success or failure.

The RGP and RCP need to comply with the aforementioned protocol changes. The RGP is extended to recognize the new SABRe request type and send a first *SABRe registration* packet to the destination before unrolling the data request packets. The RCP is extended to recognize the *SABRe validation* packets carrying the success/failure information for a SABRe, and to encode the SABRe's success in the corresponding field of the Completion Queue entry upon reception of the SABRe's last reply packet.

4.5 Chapter Summary

The emergence of highly integrated rack-scale systems employing lightweight communication protocols, high-performance fabrics, and integrated NIs brings the remote memory access latency down to a bare minimum, within a small factor of local memory memory access. In such systems, any software overheads added on top of the hardware latency for remote memory access are on the critical path and directly impact the performed operation's end-to-end latency. This is the case for modern software mechanisms that provide atomic access to remote objects, which is a ubiquitous operation. To address this inefficiency, we introduce SABRe, a new one-sided operation with richer semantics than any pre-existing one-sided operation, that provides atomic object reads in hardware. Our implementation, LightSABRe, leverages coherent on-chip integration to completely remove the software overhead for atomicity enforcement, with modest hardware requirements. We evaluate LightSABRe on an instance of an ICONIC architecture in Chapter 7 and report remote object read throughput improvements of up to 97% for a microbenchmark and up to 60% for a key-value lookup application running on top of the full software stack of a modern distributed object store.

5 Integrated Tail-aware Load Balancing

So far, we focused on protocol and architecture design to improve the performance of direct remote memory access via one-sided operations. We also proposed a new one-sided operation with richer semantics than pre-existing ones, to improve the utility and flexibility of the direct remote memory access form of inter-server communication. While one-sided operations offer the fastest path to remote memory, their key limitation is lack of flexibility, which will remain a fundamental limiter even after the introduction of additional commonly used operations in hardware, such as SABRe. This is where two-sided communication, or Remote Procedure Calls (RPCs), come into play, as they allow the invocation of arbitrary logic at the end. In fact, due to their flexibility, two-sided communication is the most widespread communication model used in modern distributed systems, including datacenters. We therefore now shift our focus from one-sided to two-sided communication, and show that ICONIC architectures introduce new opportunities for improved RPC performance as well.

To identify a specific RPC-related optimization target, we consider software deployments on large-scale distributed systems, the most prominent candidates of which are modern datacenters. Datacenters deploy a software architecture of distributed multi-tiered services on thousands of servers. Every software tier exposes a set of APIs, and a software tier's services are invoked through these APIs in the form of RPCs. Inter-server communication in the form of RPCs is very frequent, as every incoming user request fans out to 100s to 1000s of servers to get serviced

Chapter 5. Integrated Tail-aware Load Balancing

[18, 48, 91, 149]. Such a high fan-out is a direct consequence of data growth and the demand for low latency, which requires data to remain resident in distributed memory.

The decomposition of large services into multiple layers facilitates modularity and scalability, but raises some implications. A first side-effect of the high sub-request fan-out every incoming user request results in, is the increased vulnerability to stragglers. At a high level, the total latency of a user request is directly affected by the latency variability of its sub-requests. While every *individual* sub-request has a small probability of experiencing a latency spike because of a wide range of hard-to-predict reasons, the probability of *one* sub-request experiencing such a delay quickly grows with the system size, ultimately affecting the overarching user request¹. This well-known challenge, commonly referred to as *the tail at scale* [47], has led service providers to optimize for worst-case response latencies; it is common practice to evaluate systems based on their 99th or even 99.9th tail latency. Tail-tolerant computing is one of the major ongoing challenges in the datacenter space.

A second side-effect of software decomposition is that the service time for several important software layers is short, in the range of a few microseconds. Distributed in-memory object stores are a prominent example of a ubiquitously deployed software layer, with service times of just a couple of μs . For example, the average service time for Memcached [2] is about $2\mu s$ [144]. Even software layers that offer richer functionality than simple data retrieval exhibit μs -scale service times; for example, the average TPC-C query service time on the Silo in-memory database [166] is only $33\mu s$ [144]. The additional challenge with such short-lived service invocations is that any small latency disruption significantly affects their service time latency, exacerbating the challenge of tightly bounding tail latency.

In this chapter, we focus on the challenge of improving the throughput of the most challenging short-lived services invoked on modern servers from the network (i.e., in the form of an RPC), under tight response time tail latency goals. As discussed in Section 2.2, modern server processors are featuring more and more cores, currently in the range of a few tens and approaching 100.

¹For example, assuming a 0.1% chance of a latency spike and a fan-out of 1000, the probability of the latency spike affecting one sub-query and, consequently, its originating user request is $1 - 0.999^{1000} \approx 63\%$.

With all this available parallelism on chip, distributing the incoming network load across cores is a growing challenge that has a direct effect on load balancing, which also implicitly affects tail latency. In an effort to address this challenge, modern network adapters implement mechanisms that spread the incoming network load across multiple cores, like Receive Side Scaling (RSS) [126] and Flow Direction [82]. However, these mechanisms achieve *load distribution* rather than *load balancing*: they are oblivious to the actual load each core ends up with, and instead statically apply a set of application-agnostic rules to split incoming network packets into multiple receive queues based on packet headers.

The higher the core count, the higher the fragmentation of the aggregate on-chip compute resources. As we demonstrated in a related study [135, 136], the probability for load imbalance across work partitions grows as a function of the number of partitions. In turn, load imbalance negatively impacts a service’s tail latency. It is therefore important to either provide a secondary mechanism for load rebalancing across cores, or a smarter, more adaptive dynamic load balancing mechanism that dispatches load from the NI to cores adaptively. Prior work has demonstrated this problem in the case of partitioned dataplanes and tackled the inter-core load imbalance problem by introducing an intermediate layer of work stealing [144]. However, even though work stealing alleviates the load imbalance problem, it still leaves significant room for throughput improvements under tight tail latency goals, especially in the case of services with very short service times, such as Memcached.

We find that ICONIC architectures, such as soNUMA, offer a unique opportunity for implementing dynamic load balancing mechanisms in their tightly coupled NI logic. Unlike existing load distribution mechanisms in modern NICs (e.g., RSS and Flow Direction), integrated NIs can take load dispatch decisions dynamically by using live CPU core occupancy information, instead of applying static rules on network message headers. The key enabler for that is the possibility of fine-grained, nanosecond-scale communication between an on-chip NI and the CPU cores, which is not possible with conventional PCIe-attached NICs.

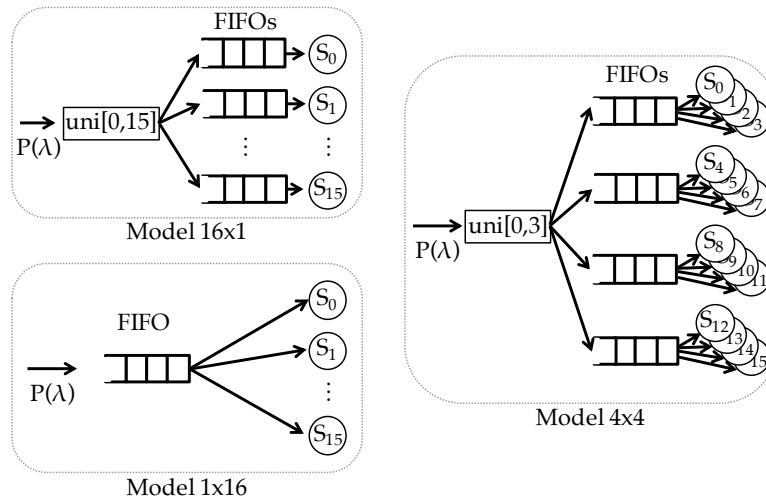


Figure 5.1 – Different queuing models for 16 serving units (CPU cores). $P(\lambda)$ stands for Poisson arrival distribution.

5.1 Theoretical Load Balancing Implications

To study the effect of load balancing on tail latency, we provide a first-order analysis using basic queuing theory. We model a hypothetical 16-core server after a queuing system that features a variable number of input queues and 16 serving units. Figure 5.1 shows three different queuing system organizations. The notation *Model* $X \times Y$ denotes a queuing system with X FIFOs where incoming messages are enqueued and Y serving units per FIFO. The invariant across the three illustrated models— 16×1 , 1×16 , 4×4 —is that $\# \text{ queues} \times \# \text{ serving units} = 16$. The 16×1 system is the least flexible one in terms of load balancing; incoming requests are uniformly distributed across 16 queues and each queue is solely serviced by a single server. 1×16 is at the other extreme, being the most flexible option that achieves the best load balancing: all serving units pull requests to service from a single FIFO. The 4×4 system represents a middle ground: Incoming messages are uniformly distributed across four FIFOs, and each FIFO is drained by four serving units. While only three configurations are shown on Figure 5.1, other combinations of X and Y with the same invariant are possible, such as 8×2 and 2×8 .

To evaluate the performance of the different queuing organizations, we perform discrete event simulations modeling Poisson arrivals and four different service time distributions, demonstrated

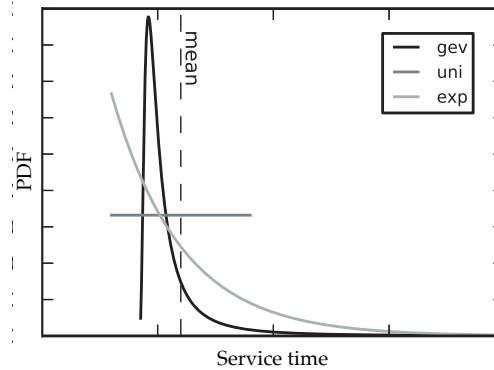
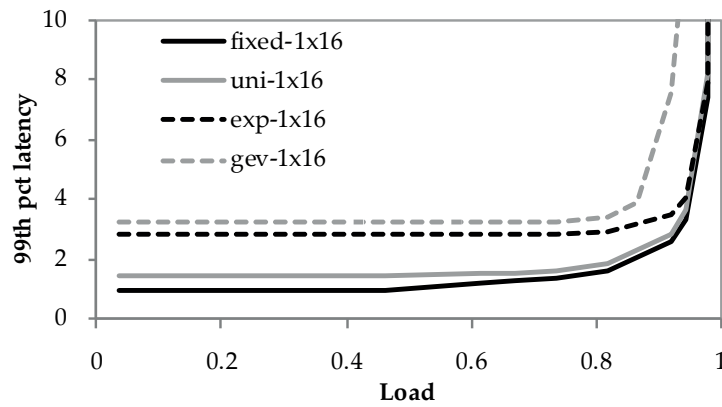


Figure 5.2 – Service time distributions.

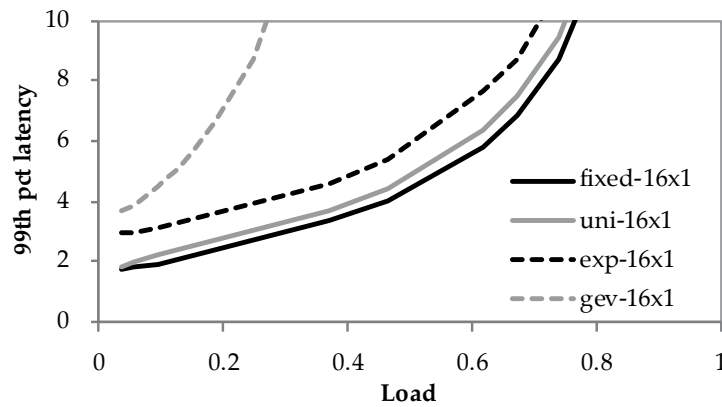
in Figure 5.2: fixed, uniform, exponential, and generalized extreme value (GEV).

Figures 5.3a and 5.3b show the relation of throughput and 99th percentile tail latency for the two extreme queuing system configurations, namely 1×16 and 16×1 . The y axis shows tail latency as a multiple of the mean service time, and we set the acceptable tail latency (upper bound) to $10 \times$ the mean service time. The first observation is that 1×16 significantly outperforms 16×1 , a well-known result from queuing theory. 16×1 's inflexibility of assigning requests to serving units results not only in significantly higher tail latencies, but also a peak throughput 25–73% lower than 1×16 at our tail latency target. The second observation is that the degree of performance degradation is affected by the service time distribution. For both queuing models, we observe that the higher a distribution's variance, the higher the tail latency (TL) before the saturation point is reached, hence $TL(\text{fixed}) < TL(\text{uni}) < TL(\text{exp}) < TL(\text{GEV})$. Also, the higher the distribution's spread, the more dramatic the performance difference between 1×16 and 16×1 , as is clearly seen for GEV.

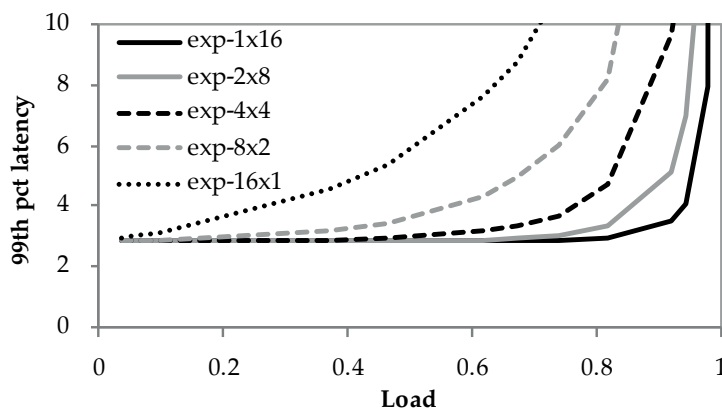
Unfortunately we cannot easily control the service time distribution, as it is affected by numerous software and hardware factors. However, we can control the queuing model that is implemented by the system. The results shown on Figures 5.3a and 5.3b suggest that systems should ideally always implement a 1×16 model, i.e., use a single queue from which all cores pull requests in order. The caveat is that in real system implementations, sharing resources requires synchronization, which introduces complexity and overhead that is not directly captured by the theoretical queuing



(a) Model 1 x 16.



(b) Model 16 x 1.



(c) Performance of different queuing models for exponential service time distribution.

Figure 5.3 – Throughput vs. tail latency for different queuing systems and service time distributions.

| | 2×8 | 4×4 | 8×2 | 16×1 |
|-------------|--------------|--------------|--------------|---------------|
| fixed | 4% | 6% | 16% | 25% |
| uniform | 4% | 6% | 16% | 25% |
| exponential | 4% | 6% | 16% | 31% |
| GEV | 6% | 20% | 46% | 73% |

Table 5.1 – Throughput loss of different queuing systems at target 99th percentile tail latency as compared to 1×16 , for different service time distributions.

model. Striking a middle ground can therefore be beneficial. We therefore also evaluate queuing model configurations between 1×16 and 16×1 .

Figure 5.3c shows the performance of five different queuing systems $X \times Y$ with $(X, Y) = (1, 16), (2, 8), (4, 4), (8, 2), (16, 1)$, assuming exponential service time distribution. As expected, performance is proportional to Y . 2×8 and even 4×4 are appealing system organizations, delivering performance within 4% and 6% of the ideal 1×16 , respectively.

Finally, Table 5.1 shows the throughput degradation of the same set of queuing systems as compared to the ideal 1×16 for different service time distributions. The *exponential* row corresponds to the results graphically demonstrated on Figure 5.3c. Qualitatively, these results indicate that even though the single-queue system is clearly superior to multi-queue systems, a modest fan-out degree per queue (e.g., 4×4) can significantly ameliorate the impact of load imbalance arising from Poisson arrival times and service time variability.

5.2 Load Balancing in Practice

Based on the previous queuing analysis, all systems should implement a single-queue load distribution mechanism. However, an implication the theoretical queuing models fail to encompass is the practical overheads associated with sharing a resource, i.e., the input queue. Allowing all the cores of a manycore CPU to pull incoming network messages from a single queue requires a synchronization mechanism. Especially for applications that exchange messages that trigger very

Chapter 5. Integrated Tail-aware Load Balancing

short-lived RPCs, with service times in the order of a few microseconds, such synchronization can be very expensive.

An alternative approach for load distribution to multiple cores that recent research efforts have advocated, is the dedication of a private queue of incoming network messages per core (e.g., [19, 142]). While this design choice, from a theoretical queuing perspective, corresponds to the weakest possible multi-queue system organization, it completely eschews overheads related to sharing (i.e., synchronization and coherence traffic), delivering significant throughput gains. To distribute incoming load to multiple input queues, modern NICs offer hardware support, which takes static decisions at message arrival time: the NIC applies a static hash function to specific fields of the network message header and that value determines to which network queue that message will be steered. As this load distribution decision takes no other system information into account, such as current per-queue occupancy, it can result in arbitrary load imbalance across queues, which conversely results in reduced throughput and increased tail latency. The pure effect of this imbalance, as compared to a system with perfectly balanced queues, is accurately represented by the queuing simulation in the previous section.

In conclusion, the two aforementioned approaches of RPC load distribution to cores introduces a tradeoff between synchronization overhead and load imbalance. In the rest of this chapter, we introduce a new load distribution approach that aims to break that tradeoff.

5.3 Towards Dynamic Load Balancing

We advocate that an ICONIC architecture introduces the opportunity to break this tradeoff between load imbalance and synchronization overheads. By leveraging the fact that on-chip NI integration enables fine-grained real-time (nanosecond-scale) information of per-core load, we envision NI functionality for *dynamic* load balancing decisions. Cores can send periodic information to the NIs in the form of heartbeats, indicating their current load. Transferring this information from the cores to the on-chip NIs is fast, as it only involves a few hops on the on-chip interconnect (10s of nanoseconds) rather than slow microsecond-scale PCIe crossings.

5.3. Towards Dynamic Load Balancing

Balancing load adaptively based on real-time information is particularly important, because accurately predicting an incoming request's processing time remains an open challenge [72]. Messages can trigger different RPCs, which is a prime reason for execution time variability across different requests. Even though the NI could potentially "learn" the expected duration of each RPC and examine each incoming message to deduce which RPC the message will trigger, there are many unpredictable events that can occur during the RPC's execution and affect its execution time, such as caching effects, TLB misses, interrupts, page faults, and context switches. Therefore, dynamic monitoring of the load and reactive adjustment to it represents the most flexible and generally applicable load balancing approach.

At a high level, dynamic load balancing decisions by the NI involves three steps: the NI (i) receives a message that carries an RPC invocation from the network; (ii) determines which core should handle the received message; and (iii) notifies the core about the message reception. Unfortunately, ICONIC architectures such as soNUMA that only offer native support for one-sided operations hinder the introduction of such a mechanism. Messaging emulation over one-sided operations is possible, as we described in Section 3.5.4, but this emulation comes with the drawback that the notion of messaging is completely transparent to the NI logic; an NI cannot distinguish between a conventional one-sided operation and a *one-sided operation that is used as a trigger for two-sided communication*. With emulated messaging, clients effectively determine which particular CPU core at the server will service each message, precluding post-message-reception load balancing by the NI. Because of this subtle issue, implementing the aforementioned generic dynamic load balancing mechanism over emulated messaging is very challenging.

To enable dynamic load balancing of incoming messaging at the NI logic, we need the hardware-terminated network protocol to support a form of native messaging. In the case of soNUMA, such native support was not provided in favor of simplicity. In the following section, we investigate the introduction of native messaging support in a lean hardware-terminated protocol such as soNUMA, with the ultimate goal of enabling NI-controlled dynamic load balancing.

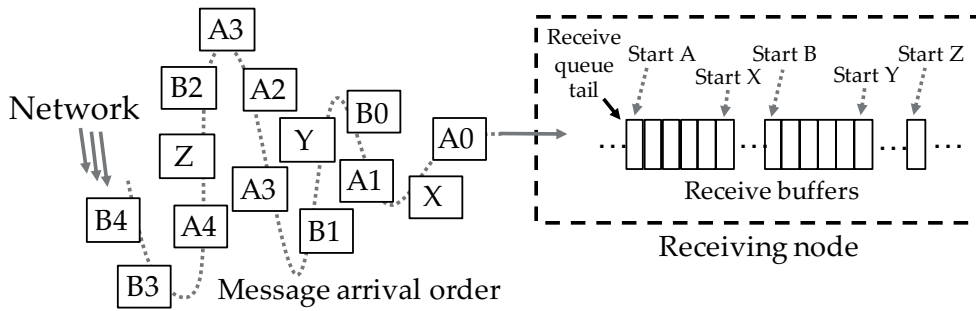


Figure 5.4 – Inter-packet interleavings of arriving multi-packet messages.

5.4 Native Messaging

To address the shortcomings of emulated messaging and provide the required building block for dynamic load balancing decisions at the NI, we devise a lightweight implementation of native messaging. We show that adding native messaging support does not require significant additional complexity, hence the simplicity of a protocol that only supports a minimalistic set of one-sided operations—an enabler for on-chip NI integration—is maintained.

We design a native messaging mechanism with two main goals: enable explicit message dispatch from the NI to a selected core, and keep hardware additions minimal. A benefit of the emulated messaging mechanism over one-sided operations is that no hardware support for reassembly of multi-packet messages is required, as all packets are directly written to a memory location pre-determined by the sender. The following example illustrates why multi-packet messages are challenging.

Let's consider two multi-packet messages, A and B, each comprised of five network packets, arriving concurrently at the same node. Their packets can arbitrarily interleave with each other and with other arriving messages (e.g., messages X, Y, Z). Figure 5.4 graphically demonstrates this example. The first packet of message A that arrives allocates the next available slot at the tail of the receive queue, and the queue's tail advances. The same happens when the first packet of any of the messages B, X, Y, or Z arrives. As packets of different messages can arbitrarily interleave in the network, an arbitrary number of other messages' packets can intervene between

two consecutive packets of the same message A. Determining the appropriate location for each subsequent incoming packet of a given message in the receive queue can be expensive, as several receive queue slots have to be examined. The depth at which this search has to be performed can be limited by implementing a form of sliding windows in hardware; however, that still requires dedicated hardware proportional to the number of concurrently operating receive queues.

One workaround to avoid message reassembly complications would be to limit the maximum message size to the network MTU. Such an approach has been used in related work on an RDMA/InfiniBand-based system [89] (a design choice motivated by different reasons, however). Even though a compromise, such a design choice may be an acceptable limitation for InfiniBand systems, which has a relatively big MTU of 4KB. For fully integrated solutions that will likely feature small MTUs (e.g., single cache line as in the case of soNUMA), limiting the maximum message size to MTU is not an option.

We take a different approach to avoid any hardware overhead associated with message reassembly. We keep the buffer provisioning of the emulated messaging approach, namely the send request's source determines the memory location at the destination where the message will be written. However, we expose the notion of multi-packet messages to the NI, which keeps track of packet receptions and deduces when a whole message has been received and is ready to be handed off to a core for processing. To achieve that, we define a new pair of *send* and *receive* operations, which transfer data between dedicated memory-resident send/receive buffers. We detail the workflow below, using in parallel Figures 5.5 and 5.6 for illustration purposes. Figures 5.5 and 5.6 demonstrate the steps required to complete a message delivery from Node 0 to Node 1. Completing the message delivery requires the execution of a *send* operation on Node 0 and a *receive* operation on Node 1.

Buffer provisioning. We introduce the concept of a *messaging domain*, which includes a number of nodes that can exchange messages and is defined by a pair of buffers allocated in each node's memory, the *send* buffer and the *receive* buffer. The *send* buffer comprises N sets of S send slots, where:

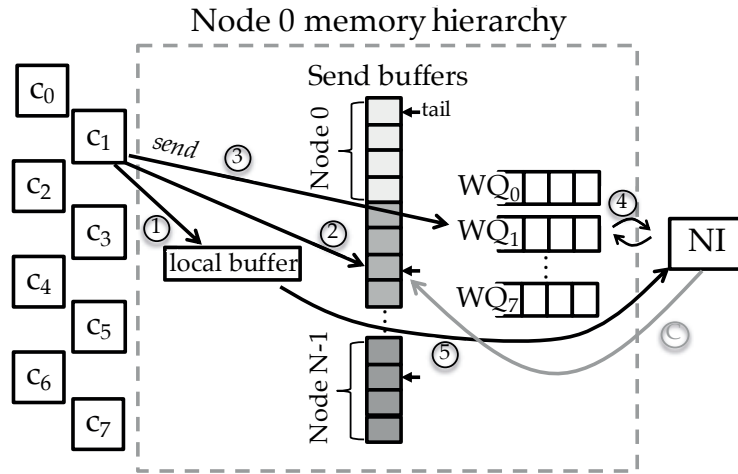


Figure 5.5 – Messaging illustration: Sender. Boxes marked as c_i represent CPU cores.

- N is the total number of nodes participating in the established messaging domain.
- S is the maximum number of concurrently outstanding requests any pair of nodes can maintain at any point in time.

Figure 5.5 illustrates a *send* buffer with $S=4$ and different shades of gray distinguishing the send slots per participating node. Each send slot contains bookkeeping information for the local cores to keep track of their outstanding messages. It contains a *valid* bit, indicating whether the send slot is currently being used, a pointer to a buffer in local memory containing the message’s payload, and a field indicating the size of the payload to be sent. A separate in-memory data structure maintains the head pointer for each of the N sets of send slots, which the cores use to atomically enqueue new send requests (not shown).

The *receive* buffer, illustrated on Figure 5.6, is the dual of the *send* buffer, where incoming send messages from remote nodes end up, hence it is sized similarly (N sets of S receive slots). Unlike send slots, receive slots are sized to accommodate message payloads. Each receive slot also contains a counter field, used to determine whether all of a message’s packets have arrived. Even though the counter field should provide just enough bits to represent the number of cache blocks comprising the largest message, we provision a full cache block (64B), to avoid unaligned accesses for incoming payloads.

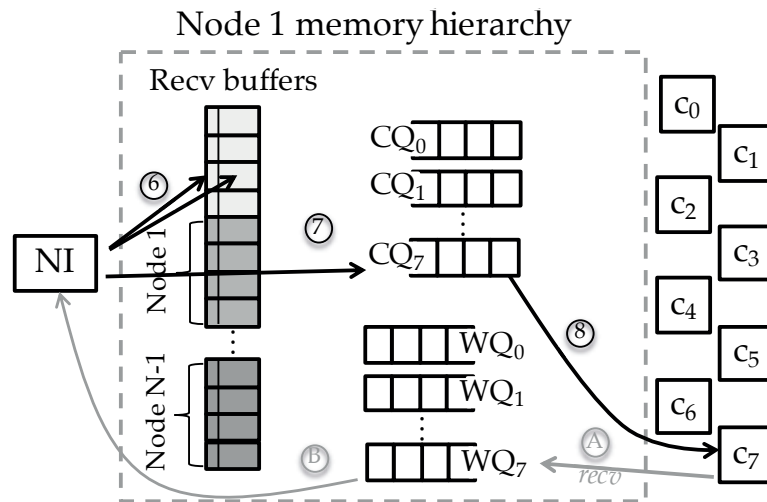


Figure 5.6 – Messaging illustration: Receiver. Boxes marked as c_i represent CPU cores.

Overall, the messaging mechanism’s memory footprint is $32 \times N \times S + (max_msg_size + 64) \times N \times S$ bytes. We expect that for most applications and system deployments, that number should not exceed a few tens of MBs. Systems that will most likely adopt fully integrated solutions will be of contained scale, featuring a few hundreds of nodes, hence bounding the N parameter. In addition, most communication-intensive latency-sensitive applications send small messages, bounding the max_msg_size parameter. For instance, the vast majority of objects in object stores like Memcached are $<500B$ [16], while 90% of all packets sent within Facebook’s datacenters are smaller than 1KB [148]. Finally, given the extremely low network latencies fully integrated solutions, such as soNUMA, deliver, the number of concurrent outstanding requests S required to sustain peak throughput per pair of nodes would be modest (a few tens). We provide a detailed analysis of the messaging mechanism’s memory requirements in Section 8.3.5.

Importantly, the choice of max_msg_size does not preclude the exchange of larger messages altogether. A *rendezvous* mechanism [164] can be used for larger messages, where the sending node sends a small message specifying the location and size of the data to be sent, and the receiving node uses a one-sided read operation to directly pull the message’s payload from the sending node’s memory.

Send operation. Sending a message to a remote node involves the following steps: First, the core writes the message in a local buffer (Figure 5.5, step ①), then updates the tail entry of the send buffer set corresponding to the target node (e.g., Node 1) ② and enqueues a *send* operation in its private WQ ③. The *send* operation specifies a messaging domain, the target node id, the address of the remote end's target receive buffer slot, a pointer to a local buffer containing the message to be sent, and the message's size. The address of the target receive buffer slot can be trivially computed, as the number of participating nodes in the messaging domain, the number of send/receive slots per node, and the receive buffer slot size are all defined at the messaging domain's deployment time. The NI polls on the core's WQ ④, parses the command, reads the message from its local memory's buffer ⑤ and sends it to the destination node. At the destination, the NI writes each *send* packet directly into the specified receive slot and increments that receive slot's counter (Figure 5.6, step ⑥). When the counter matches the send operation's total packet count (contained in each packet's network header), the NI picks a core² and notifies it by writing the receive buffer's index into that core's corresponding CQ ⑦. The core, which is polling on its private CQ's head, receives the new *send* request ⑧, then directly reads the message from the receive buffer and processes it.

Receive operation. A *receive* operation always follows the receipt of a *send* operation, with the purpose of notifying the *send* operation's initiating node that the request has been processed, and hence its corresponding send buffer slot is free and can be reused. In Figure 5.6's example, when core 7 is done processing the incoming *send* request, it enqueues a *receive* request in its private WQ ⑨. The request only contains the target node and the target send buffer slot's address, trivially deduced from the receive buffer index the serviced message was retrieved from. The NI, which is polling at the head of the WQ, receives the new *receive* request ⑩ and sends the message to the target node. Back at the source node (Figure 5.5), when the *receive* message arrives, the NI invalidates the target *send* buffer slot by resetting its *valid* field ⑪, indicating its availability to be reused. In practice, a *receive* operation is just syntactic sugar for a special remote write operation, which resets the valid field of a send buffer slot.

²For now, assume a simple round-robin policy. We investigate smarter message dispatch policies in Chapter 5.

5.4.1 Additional Benefits of Native Messaging

We introduced native messaging support as an enabler for NI-controlled dynamic load balancing. However, it offers additional benefits as compared to emulated messaging, making it an appealing feature even for systems where load balancing features are not of interest. In fact, a weakness of emulated messaging is its waste in CPU cycles, which is a precious resource, especially in face of the growing imbalance in the scaling of network and compute capabilities (see Section 2.5). As compared to native messaging, emulated messaging has two sources of CPU inefficiency. The first source of inefficiency is the busy polling of each core on multiple memory locations to check for message reception. A distributed system with 1000 communicating nodes would require 1000 distinct polling locations. Polling on the head of each location implies that 1000 distinct cache blocks are brought into an L1 data cache, completely thrashing it.

The second source of inefficiency, which is specific to soNUMA, is an additional considerable processing overhead associated with *packetizing* the data that is sent in the message, as each of the message's cache blocks requires its own message header. The reason per-cache-line headers are necessary is the fact that the maximum packet payload size in a soNUMA packet is a single cache block, which is typically 64 bytes. A core receiving an incoming message cannot determine whether a valid packet has arrived at the location it is polling on or how many packets the full message comprises, unless (i) each cache block contains metadata indicating its validity as a newly arrived message; and (ii) the first cache block's header contains the total message size. Hence, a message's packetization involves embedding this required information at the message's source, before it is sent to the destination over the network. The cost of this packetization process is similar to the overhead of software mechanisms used to guarantee atomicity of one-sided reads, which involve embedding per-cache-line metadata headers in all transferred data. The overhead of such a software atomicity mechanism was thoroughly analyzed and evaluated in Chapter 4.

Native messaging does not introduce any packetization overheads, as the NI keeps track of a message's packet arrivals and determines when a full message has arrived. It also significantly limits the number of locations each core needs to poll on, as it virtually "collapses" all message

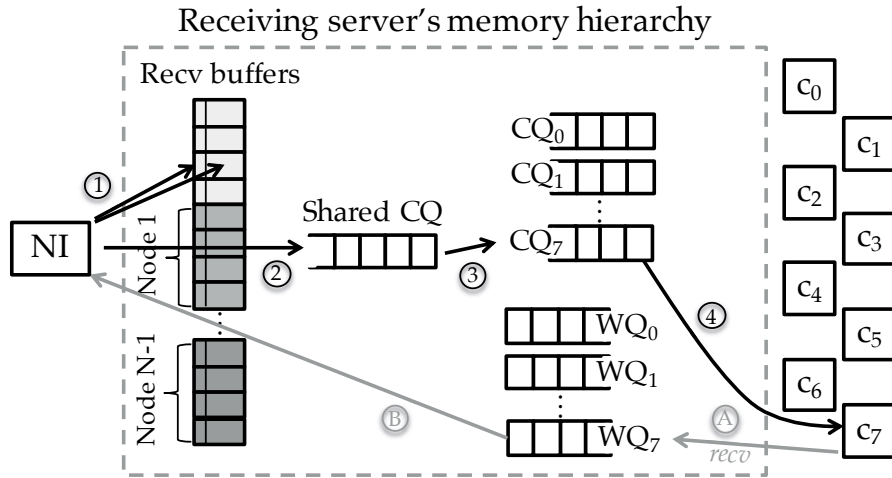


Figure 5.7 – Messaging mechanism extensions for load balancing.

receive locations into a single memory location per core, namely the head of each core's CQ.

5.5 Dynamic Load Balancing Design

With the NI's newly added ability to recognize and manage message arrivals, we can now proceed to introduce NI-integrated dynamic load balancing capability. Load balancing policies implemented by the NIs can be sophisticated and can take various affinities and parameters into account (e.g., certain types of RPCs serviced by specific cores, or data-locality awareness). Implementations can range from simple hardwired logic to microcoded state machines. However, we opt to keep a simple proof-of-concept design, to illustrate the feasibility and effectiveness of load balancing decisions at the NIs.

We design our load balancing functionality as an extension of the messaging mechanism we introduced in Section 5.4. Figure 5.7 depicts the receiving end of a message, extending Figure 5.6 with the required additions for our load balancing design. In addition to the per-core CQs where messages are received by cores, we introduce a new in-memory structure, the *shared CQ*. The *shared CQ* is an intermediate memory-resident queue of waiting requests where the NIs enqueue received requests, before notifying any core of a message's reception by writing into a specific CQ.

5.6. soNUMA Extensions for Dynamic Load Balancing

As in the case of the baseline messaging mechanism, the NI writes each received *send* packet directly into the specified receive slot and increments that receive slot's counter ①. With the addition of the shared CQ, the difference is that when the counter matches the send operation's total packet count, the NI enqueues the pointer to the completed entry in a *shared CQ* instead of immediately picking a core to dispatch the request to ②. At a later point in time, as soon as a core is free and ready to receive a new request to process, the head of the *shared CQ* is copied into the free core's CQ ③, notifying the core of the next received request to process. The benefit of this intermediate buffering is that global FIFO order of message arrival is maintained.

In order to implement step ③, the NI maintains limited state per core: whether the core is currently busy processing a previous *send* request. Reception of a *receive* request from a core (Figure 5.7's step ⑥) implies that the core is done processing the previous *send* request, so the NI can dispatch the first *send* request waiting in the *shared CQ* to the freed core.

5.6 soNUMA Extensions for Dynamic Load Balancing

We now detail the extensions required to soNUMA's protocol and RMC hardware to enable the NI logic to balance load by taking dynamic load dispatch decisions. Load balancing itself is completely transparent to the protocol and mainly affects soNUMA's Remote Request Processing Pipeline (RRPP). Most protocol and RMC pipeline modifications are required to implement native messaging support.

Additional hardware state. Most of the additional state required for messaging (i.e., the *send* and *receive* buffers) is allocated in memory. The only information that requires constant fast access—and hence should be kept in dedicated SRAM—is the send and receive buffer metadata: their location and size. We associate a messaging domain with every registered context. On each node, the maintained state per registered context originally includes a memory address range per node and a QP per local core. We extend the context state with the messaging state, which includes the base virtual addresses for the *send* and *receive* buffers, the maximum message size

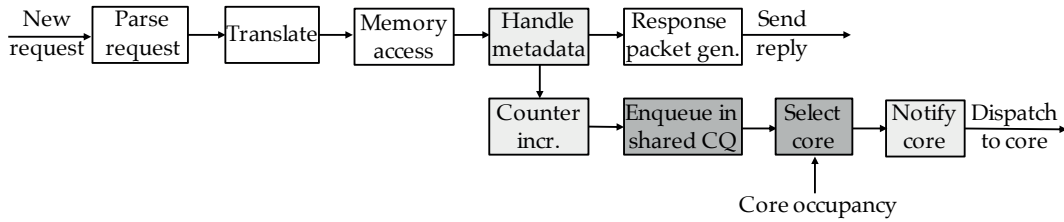


Figure 5.8 – Extensions of soNUMA’s RRPP for load balancing support.

(*max_msg_size*), the total number of nodes participating in the messaging domain (N), and the number of messaging entries per node (S); in total, a modest premium of 20B per context entry.

RMC logic extensions. We extend soNUMA’s RMC pipelines to support the new messaging primitives and load balancing functionality. From an RMC perspective, reception of a new *send* or *receive* request through a WQ is very similar to the reception of a remote write operation. Unlike the baseline soNUMA protocol, the network header of multi-packet messages has to additionally contain the total message size, which is necessary information for the remote end to identify when all of a message’s packets have been received. In Chapter 4, we saw the same requirement for SABRes. The only considerable hardware changes are identified in the RRPP .

Figure 5.8 provides a high-level view of the modified RRPP, with the added stages highlighted in two shades of gray. Light gray indicates added stages required by messaging, and dark gray denotes load-balancing-related stages. The RRPP originally handles independent cache-block-sized requests by (i) parsing the incoming request, (ii) translating the target virtual address, expressed in the request as a combination of *context_id* and *offset* within the context, (iii) accessing the target memory location, and (iv) creating a response packet and sends it back to the requesting node. We add five more stages.

The first added stage, "Handle metadata", is after the "Memory access" stage (i.e., after a *send* request’s payload has been written to memory). In the case of a *receive* request, the RRPP simply resets the *valid* byte field of the target send buffer slot (see Section 5.4, "Receive operation"). In the case of a *send* request, the RRPP performs a fetch-and-increment operation to the corresponding *counter* field of the target receive buffer slot (see Section 5.4, "Send operation").

The following added stage, "Counter incr.", checks if the counter's incremented value matches the message's total length (carried in each packet's network header). If all of the *send* operation's packets have arrived, the next stage is activated, which enqueues a pointer to the completed send buffer slot in the shared CQ. "Select core", monitors the occupancy of the cores and maintains state about their current status to take message dispatch decisions, with the goal of keeping load balanced. The complexity of this stage can widely vary based on the logic and algorithm involved in taking load balancing decisions. In our current design, load balancing decisions are simply based on greedy FIFO message dispatch to the first available core. The state that has to be maintained per core is minimal: a core's status is either occupied or available. Whenever there is an available core and the shared CQ is not empty, the "Select core" dequeues the first entry from the shared CQ and passes it along with the selected core's ID to the final stage. Finally, the "Notify core" stage sends a special message to the Request Completion Pipeline (RCP) and marks that core as busy. The message carries the selected core's ID and a pointer to the receive buffer slot that contains the request ready to be serviced. Upon the message's reception, the RCP enqueues an entry in the target core's CQ, thus notifying the core of the incoming request's location. The core is marked again as available as soon as the Request Generation Pipeline (RGP) receives a subsequent request containing a *receive* operation.

Overall, the additional hardware complexity is modest, thus compatible with architectures featuring ultra-lightweight protocols and on-chip integrated NIs, such as soNUMA. Given the RMC's fast access to its local memory hierarchy, it is possible to virtualize most of the bulky state required for the messaging mechanism's send and receive buffers in the host's memory. Hardware requirements are limited to a small additional fraction of dedicated SRAM capacity, while NI logic extensions are contained and straightforward.

5.7 Chapter Summary

The most challenging distributed applications require rapid inter-node communication with tight tail latencies. In the case of short-lived RPCs with microsecond-scale service times, a key factor

Chapter 5. Integrated Tail-aware Load Balancing

determining RPC tail latency on modern manycore server chips is how they are distributed from the network to the available CPU cores. Modern NICs statically partition and dispatch network load to cores, being oblivious to the cores' dynamic load. Such approach achieves *load distribution*, but not *load balancing*: the resulting load per core can significantly vary, resulting in increased RPC response time tail latencies. In this chapter, we identified that ICONIC architectures such as soNUMA, which feature on-chip NIs, provide a unique opportunity of dynamic NI-to-core load dispatch, alleviating inter-core load imbalance that arises from static load distribution decisions. By allowing the NI to dynamically monitor the load on the CPU cores and dispatch messages to them accordingly, we enable synchronization-free load balancing capable of approaching the quality of an ideal queuing system. As a prerequisite for NI-driven load balancing, we introduced a lightweight native messaging mechanism that can be easily supported with minimal hardware by on-chip integrated NIs. In addition to being an enabler for NI-driven load balancing, the native messaging mechanism is also more CPU-friendly than soNUMA's original messaging mechanism, which was emulated on top of one-sided operations. We implement and demonstrate the effectiveness of both native messaging and dynamic load balancing on an instance of an ICONIC architecture in Chapter 8.

Implementation and Evaluation of an Part II

ICONIC Architecture

6 Manycore Chip Design

In the second part of this thesis, we describe a concrete implementation of an ICONIC architecture based on the Scale-Out NUMA (soNUMA) protocol. We gradually enhance the implemented system with all the features introduced in Part I, and evaluate their performance impact.

soNUMA was introduced in Chapter 3 as a proof-of-concept architecture showcasing that a hardware-software co-design of the entire networking stack—lean user-level network protocol, tight NI integration, and high-performance fabric—can bridge the gap between local and remote memory. To build a real system based on soNUMA, additional effort in adapting the conceptual protocol to modern technology trends is necessary. Specifically, given server technology trends, rack-scale systems will soon feature SoCs with dozens of cores (e.g., Scale-Out Processors [29, 119] or tiled manycores [56]), high-bandwidth memory interfaces supplying well over 100GBps of DRAM bandwidth per socket, and SerDes or photonic chip-to-chip links, allowing for low-latency and high-bandwidth intra-rack communication. A rack-scale system features many such servers, tightly integrated in a supercomputer-like fabric. A key emerging challenge in such systems is a manycore NI architecture that would enable effective integration of on-chip resources with supercomputer-like off-chip communication fabrics to maximize efficiency and minimize cost.

Traditional NIs hang off the chip’s edge, adjacent to the I/O pins. Existing chips with on-chip

NIs (e.g., [107, 109]) follow the same approach, keeping the NI at the edge, albeit on chip. Unfortunately, placing NIs at the chip’s edge in a manycore SoC incurs prohibitively high on-chip coherence and NOC latencies on accesses to the NI’s internal structures. We find that on-chip latency can be particularly high (up to 80% of the end-to-end latency) for fine-grain (e.g., cache block size) accesses to remote in-memory objects. Because of the demand for low remote memory access latency, such edge-based NI placements are not desirable.

Alternatively, there are manycore tiled processors with lean per-tile NIs directly integrated into the core’s pipeline [20]. While per-tile designs optimize for low latency, they primarily target fine-grain (e.g., scalar) communication and are not suitable for in-memory rack-scale computing with coarse-grain objects from hundreds of bytes to tens of kilobytes. Moreover, current per-tile designs are highly intrusive in microarchitecture, which is undesirable for licensed IP blocks (e.g., ARM cores) used across many products. Finally, these designs have primarily targeted single-chip systems rather than distributed in-memory systems, which rely on fast remote memory access for high performance.

This thesis is the first research effort that evaluates the design space of manycore NIs for emerging verb-based network protocol stacks¹, such as RDMA or soNUMA, which are getting increasing traction due to the need for frequent fast remote memory access. Our study reveals that: (i) there is a need for per-tile NI functionality to eliminate unnecessary coherence traffic for fine-grain requestor-side operations; (ii) given high coherence-related NOC latencies, the software overhead to trigger one-sided operations is amortized, obviating the need for direct remote load/store operations in hardware to accelerate them; (iii) bulk transfer operations overwhelm the NOC resources and as such are best implemented at the chip’s edge; and (iv) response-side operations (i.e., remote requests to a SoC’s local memory) do not interact with the cores and are therefore best handled at the chip’s edge.

We use these observations and propose three manycore NI architectures: (1) NI_{edge} , the simplest design, with NIs along a dimension of the NOC at the chip’s edge, optimizing for bandwidth and

¹We use the term *verb-based protocol* to refer to protocols that rely on memory-mapped queue-based communication between the CPU and the NI, such as soNUMA’s Queue Pair model.

low on-chip traffic, (2) $NI_{per-tile}$, the most hardware-intensive design, with an NI at each tile to optimize for lower access latency from a core to NI internals, and (3) NI_{split} , a novel manycore NI architecture with a per-tile frontend requestor pipeline to initiate transfers, and a backend requestor pipeline for data handling plus a response pipeline servicing remote accesses to local memory, both integrated across the chip's edge. The NI_{split} design optimizes for both latency and bandwidth without requiring any modifications to the SoC's cache coherence protocol, the memory consistency model or the core microarchitecture.

Focusing on a rack-scale deployment, we assume a 512-node 3D-torus-connected rack with 64-core mesh-based SoCs, and use cycle-accurate simulation to compare our three proposed manycore NI architectures to a NUMA machine of the same size and show that:

- On-chip coherence and NOC latency dominate end-to-end latency in manycore SoCs for in-memory rack-scale systems, amortizing the software overhead of one-sided operations. As such, intrusive core modifications to add hardware load/store support for remote operations are not merited.
- An NI_{edge} design can efficiently utilize the full bisection bandwidth of the NOC while incurring 16% to 80% end-to-end latency overhead as compared to NUMA.
- An $NI_{per-tile}$ design can achieve end-to-end latency within 3% of NUMA, but can only reach 25% of the bandwidth that NI_{edge} delivers for large (8KB) objects, due to extra on-chip traffic.
- An NI_{split} design combines the advantages of the two base designs and reaches within 3% of NUMA end-to-end latency, while matching NI_{edge} 's bandwidth.

6.1 Key Design Considerations

6.1.1 Application Requirements and Technology Trends

Today's networking technologies struggle to satisfy the heavy demands of datacenter applications that query and process massive amounts of data in real time. User data is growing faster than ever, and providing applications with fast access to it is fundamental. Because datasets commonly exceed the capacity of a single cache-coherent NUMA server by several orders of magnitude, distributing data and computation across multiple servers (a.k.a., scale-out) has become the norm.

Unfortunately, most such applications must address large amounts of data in little time [17], with implications in terms of both latency and bandwidth. Many datacenter applications are hard to partition optimally as they rely on irregular data structures such as graphs, making the poor locality of reference a fact of life. Other applications, such as distributed key-value stores, force clients to go over the network in order to access just a few bytes of user data. Most key-value stores today operate on object sizes between 16 and 512 bytes, which are typical of datacenter applications [16, 156]. Similarly, Facebook's Memcached pools typically have objects close to 500 bytes in size [16]. Accesses to such small objects are bound by the network latency (up to $100\mu s$), which can increase the overall latency as compared to local memory access latency (100ns) by three or more orders of magnitude.

The corresponding bandwidth requirements are equally dramatic for datacenter applications. Lim et al. [104] measure object sizes in file servers, image servers and social networks varying in size from a few to tens of KBs. Many graph processing and MapReduce applications require more coarse-grained accesses and thus are mostly bound by the bisection bandwidth. In such applications, bandwidth requirements grow with the size of the system, as the fraction of data local to a given node is inversely proportional to the number of nodes.

Overall, application requirements highlight the criticality of providing both low-latency and high-bandwidth access to remote memory. Given an optimized network stack such as RDMA or soNUMA for fast remote memory access, the integration of the NI logic, which glues compute

with the network, introduces significant design considerations. We identify two attributes in tomorrow's servers that drive our design choices. First, research results and industry trajectory are pointing in the direction of server processors with dozens of cores per chip [29, 56, 119]. Thus, remote memory architectures will have to cope with realities of a *fat* node with many cores and non-trivial on-chip communication delays. Second, the emerging System-on-Chip (SoC) model for server processors favors features that can be packaged as separate IP blocks and eschews invasive modifications to existing IP (e.g., most licensees of ARM cores do not acquire a costly ARM architectural license that would allow them to modify core internals).

6.1.2 QP-Based Interface for Remote Memory Access

soNUMA relies on RDMA-inspired one-sided memory operations with architectural support in a specialized NI to achieve rack-scale low remote memory access latency. In such RDMA-like protocol implementations, the cores communicate with the NI via in-memory control structures, typically Queue Pairs (QPs), to schedule remote operations and get notified of their completion. The QP-based communication of RDMA introduces a non-trivial scheduling overhead for remote operations, as illustrated in Figure 6.1. Each remote read or write requires the execution of multiple instructions on the core to create an entry in the Work Queue (WQ). The local NI polls on the WQ and upon the creation of a new request, the NI reads the corresponding WQ entry, generates a request and injects it into the network. Upon the response's arrival, the local NI takes protocol-specific actions to complete the request and notifies the application by writing to a Completion Queue (CQ). The QP-based approach is clearly more complex from the programming perspective than a pure load/store model, but is highly flexible and does not require modifying the core.

The underlying technology and mechanisms used to connect the controller to the cores play a significant role for the end-to-end latency and bandwidth. Most existing RDMA-based solutions rely on PCIe to connect the NI, introducing significant interaction overhead. soNUMA leverages cache coherence to make this interaction as cheap as possible, but still introduces non-negligible overhead due to sequences of coherence-related on-chip messages, triggered by each cache block

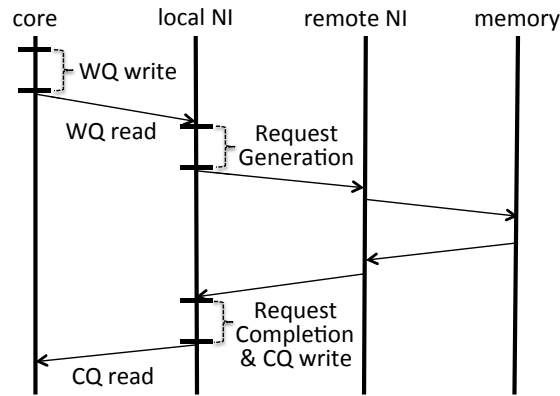


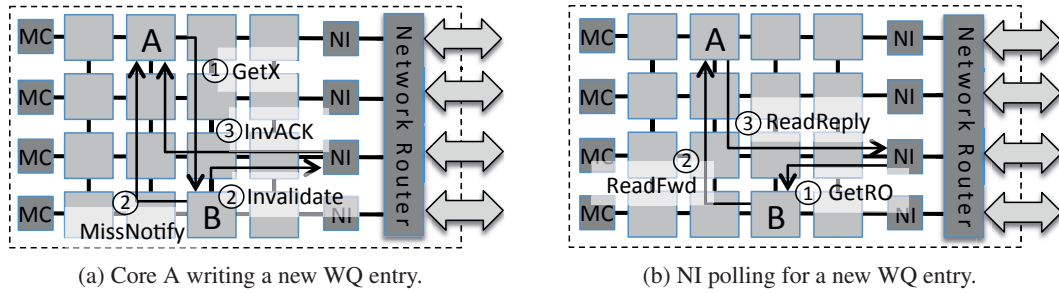
Figure 6.1 – QP-based remote read.

transfer.

QP-based communication has the potential to drive remote access latency down to a small factor of DRAM latency, at rack scale in the near future, and even at datacenter scale in the long run. The QP-based approach is appealing because it does not require modifying the instruction set to support remote reads/writes. By using memory-mapped queues, applications can interact with the NI directly and bypass the OS kernel, which is a major source of overhead. This interaction between CPU cores and the NI is straightforward for CPUs with small core counts; however, its scalability and effectiveness with modern server CPUs, which features tens of cores, is questionable and has not been thoroughly investigated. In this chapter, we identify the challenges of NI placement and core-NI communication in a QP-based remote memory access model for such manycore chips.

6.2 Manycore Network Interfaces

In this section, we investigate the design space for scalable network interfaces for manycore CPUs, their latency and bandwidth characteristics, and the costs and complexities associated with their integration.

Figure 6.2 – Core and NI WQ interactions on an NI_{edge} design.

6.2.1 Conventional Edge-Based NI

Recent high-density chassis- and rack-scale systems are based on server processors with a few cores and an on-chip NI [50, 107, 109]. Although integrated, the whole NI logic is still placed at the chip's edge, close to the pins that connect the chip to the network. We refer to this NI architecture as NI_{edge}.

Emerging scale-out server processors [119] (e.g., Cavium's ThunderX [110, 111]), and tiled manycores (e.g., EZChip's TILE-Mx [56]) already feature from several dozens to up to 100 ARM cores. Because of this trend toward *fat* manycores, NI_{edge} may not be optimal for chips that feature fast remote memory access powered by a QP-based model. In particular, the QP-related traffic between the cores and the NIs must traverse several hops on the NOC, and as the NOC grows with the chip's core count so does the number of hops to reach the chip's edge. Moreover, every cache block transfer triggers the coherence protocol, which typically requires several messages to complete a single transfer. Thus, the QP-related traffic (i.e., WQ/CQ read/write in Figure 6.1) becomes a significant fraction of the end-to-end latency.

Figures 6.2a and 6.2b illustrate the critical path for reads and writes, respectively, in an example manycore chip with NI_{edge}. Without loss of generality we assume a manycore chip with a mesh NOC and a statically block-interleaved shared NUCA LLC with a distributed directory and a 3-hop invalidation-based MESI coherence protocol. As such, a block's home tile location on the chip is only a function of its physical address. The NI also includes a small cache to hold the QP entries, which is integrated into the LLC's coherence domain and is bypassed by all of the NI's

data (non-QP) accesses.

Figure 6.2a shows the sequence of coherence transactions required for core A to write to a WQ entry. The core sends a message to request an exclusive copy (*GetX*) of the cache block where the head WQ entry resides. The message goes to the requested block's home directory, which happens to be at core B ①. Because the NI polls on the WQ head, the directory subsequently invalidates NI's copy of the cache block, and concurrently sends the requested block to core A, notifying it (*MissNotify*) to wait for an invalidation acknowledgement from the NI ②. The NI then invalidates its copy of the requested block and sends an acknowledgement (*InvACK*) to resume core A ③. Once core A resumes, it also sends an acknowledgement concurrently to the directory to conclude the coherence transaction (arrow omitted in Figure for clarity).

Figure 6.2b shows the sequence of coherence transactions required for the NI to read a new WQ entry. The NI requests a read-only copy (*GetRO*) of the block from the directory ①. Because the block is modified in core A's cache, the directory sends a forward request to core A ②. Core A forwards a read-only copy of the modified block to the NI ③, downgrading its own copy, and resuming the NI. Once the NI resumes, it also sends an acknowledgement to the directory with a copy of the block to keep the data in the LLC up to date and conclude the coherence transaction (arrow omitted in Figure for clarity).

The on-chip coherence overhead in terms of message count is similar in the case of CQ interactions. The only difference is that the roles of the core and the NI are reversed, with the NI writing entries in the CQ, and the core polling on the CQ head.

We quantify the latency cost of these interactions through a case study. Table 6.1 presents a breakdown of the average end-to-end latency for a remote read operation under zero load in a QP-based rack-scale architecture featuring 64-core SoCs with a mesh NOC and NI_{edge} . In this breakdown, we assume communication between two directly connected chips (i.e., one network hop apart). The details of the modeled configuration can be found in Section 6.4. The table also includes the latency breakdown for a base NUMA machine (e.g., Cray T3D [94, 152]), which does not incur any QP-related on-chip communication overheads, as a point of comparison.

| Latency Component | QP-based model | Latency Component | NUMA |
|--------------------------------|----------------|--------------------------------|------|
| A1) WQ write (core) | 104 | B1) Exec. of load instruction | 1 |
| A2) WQ read (NI) | 95 | B2) Transfer req. to chip edge | 23 |
| A3) Intra-rack network (1 hop) | 70 | B3) Intra-rack network (1 hop) | 70 |
| A4) Read data from memory | 208 | B4) Read data from memory | 208 |
| A5) Intra-rack network (1 hop) | 70 | B5) Intra-rack network (1 hop) | 70 |
| A6) CQ write (NI) | 79 | B6) Transfer reply to core | 23 |
| A7) CQ read (core) | 84 | | |
| Total (2GHz cycles) | 710 | Total (2GHz cycles) | 395 |
| Overhead over NUMA | 79.7% | | - |

Table 6.1 – Latency comparison of a QP-based model and a pure load/store interface.

Table 6.1 indicates that the overhead of the core writing a new WQ entry and the NI reading it can measure up to ~ 200 cycles (entries A1 & A2), while the overhead for NUMA to send a request to the chip’s edge is only 24 cycles (B1 & B2). The network and memory access at the remote node incur the same latency in both systems. Finally, the QP-based model requires ~ 160 cycles to complete the transfer via a CQ entry that is written by the NI and read by the core (A6 & A7), while for NUMA the response is sent directly to the issuing core (B6).

The overall overhead of the QP-based model over a NUMA machine is almost 80%. The QP-based interactions that require multiple NOC transfers dominate the end-to-end latency. Moreover, in this example, the software overhead of creating a WQ entry for a RISC core is roughly a dozen arithmetic instructions plus two stores to the same cache block. Similarly, reading the CQ involves four instructions including a load. Therefore, the software overhead of reading/writing the QP structures is negligible compared to the overall on-chip latency. These results suggest that supporting a load/store hardware interface for remote accesses as in NUMA machines is an overkill because its impact would be negligible on the end-to-end latency for manycore chips.

NI_{edge} becomes competitive, however, with an increase in transfer size. The QP-based model allows for the core to issue a request for multi-cache-block objects through a single WQ entry. Such a request is subsequently parsed by the NI and “unrolled” directly in hardware, completing multiple block transfers before having to interact with the core again, thus amortizing the QP-related overheads over a larger data transfer. The net result is that NI_{edge} exhibits robust bandwidth characteristics with the QP-based remote access model. In contrast, a NUMA machine primarily

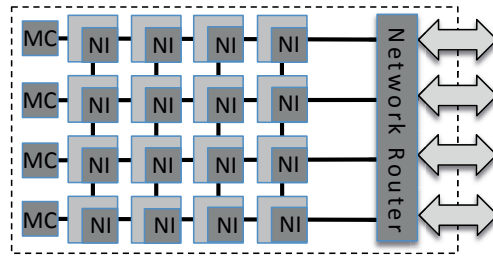


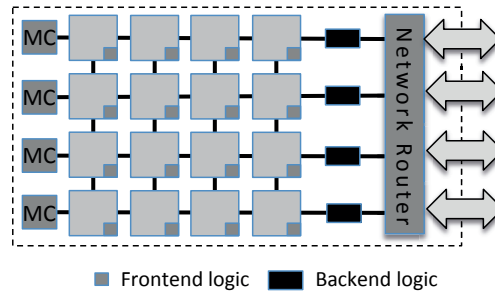
Figure 6.3 – $NI_{per-tile}$ design.

supports a single cache block transfer and, without specialized NI support, it would suffer in performance from prohibitive on-chip traffic.

6.2.2 Per-Tile NI

An alternative NI design is to collocate the NI logic with each core ($NI_{per-tile}$, Figure 6.3). Such a design would mitigate QP-related traffic, thereby using the NOC only for the direct transfer of network packets between each tile and the network router at the chip's edge. To eliminate the coherence traffic between the core and NI, while precluding changes to the core or the LLC coherence controllers, the NI cache must be placed close to the core with care. We discuss the details of the NI cache design in Section 6.2.4.

Unfortunately, while $NI_{per-tile}$ minimizes the initiation latency for small transfers, it suffers from unnecessary NOC traversals for large transfers. To access a large object in remote memory, the NI issues a separate pair of request and response messages for each cache block. Thus, a large request is transformed into a stream of cache-block-sized requests, which congests the NOC on its way to the chip's edge. Similarly, the responses congest the NOC because every single response message must be routed to the NI, which the request originated from, before its payload is sent to its corresponding home LLC tile. Therefore, in contrast to NI_{edge} , $NI_{per-tile}$ optimizes for latency, while suffering from lower bandwidth. We compare and contrast the latency and bandwidth characteristics of these two NI designs in Section 6.5.

Figure 6.4 – NI_{split} design.

6.2.3 Split NI

To overcome the limitations of the NI_{edge} and $NI_{per-tile}$ designs, we propose a novel design that optimizes for both low latency and high bandwidth. Our design is based on the fundamental observation that an NI implements two distinct functionalities that are separable: (i) a *frontend* component including the NI cache, which interacts with the application to initiate a remote memory access operation, and (ii) a *backend* component, which accesses data. We therefore split each NI into these two components. We replicate the NI’s frontend at each tile, so that each frontend is collocated with the core it is servicing to minimize the QP coherence overhead. The backend is replicated across the chip’s edge, close to the network router. The split NI design (NI_{split} , Figure 6.4) achieves the best of both NI_{edge} and $NI_{per-tile}$ worlds. It provides low QP interaction latency without generating unnecessary NOC traffic and optimizes for both fine-grained and bulk transfers.

6.2.4 NI Cache

In the NI_{edge} design, each NI is attached to an edge tile, extending the mesh as shown in Figure 6.2. Each NI includes a small cache that holds the QP entries and acts like a core’s L1 data cache participating in the LLC’s coherence activity. The NI cache has a unique on-chip tile ID, which is tracked by the coherence protocol much like a core’s L1 cache.

In contrast, $NI_{per-tile}$ and NI_{split} collocate their NI cache with a core at each tile to mitigate coherence traffic induced by QP interactions. Unfortunately, a naive collocation of the NI

cache at each core does not eliminate the traffic because all QP interactions require consulting with the home directory in the LLC for the corresponding cache blocks. To guarantee that the traffic remains local, the system must migrate the home directories for the QP entries to their corresponding NI tile requiring additional architectural and OS support (e.g., as in R-NUCA [73]). Alternatively, sharing the L1 data cache between the core and NI would eliminate all traffic but is highly intrusive because the L1 data cache is on the critical path of the pipeline. Moreover, commodity ARM cores are licensed as an entire IP block and making changes to the core would be prohibitively expensive.

Instead, the cache for these two NI designs is attached directly to the back side of L1, at the boundary of the core's IP block. Unlike the NI_{edge} cache, this cache directly snoops all traffic from the L1's back side (e.g., as in a write-back or victim buffer). The NI cache and the core's L1 at each tile collectively appear as a single logical entity to the LLC's coherence domain while physically decoupled. Such an integration obviates the need to modify the on-chip coherence protocol and guarantees preserving the base memory consistency model (e.g., Reunion vocal/mute cache pair [158], FLASH/Typhoon block buffers [74, 96, 147]).

NI Cache Coherence Transition Diagram

Without loss of generality, we assume a non-inclusive MESI-based invalidation protocol with an inexact directory (i.e., non-notifying protocol). Exact directories are also implementable but would require a more sophisticated finite-state machine to guarantee that a single shared copy is tracked by the directory between the NI cache and the L1 data cache.

Much like typical L1 back-side buffers, the $NI_{per-tile}$ (or NI_{split}) cache snoops all traffic from both L1 and the directory, looking for addresses matching the registered QPs. The cache can provide a block upon a miss in L1 as long as the request message conforms with the cache state. Otherwise, the request is forwarded to the directory. Similarly, if the directory requests a block that is currently shared by the cache, the cache acts on the message, forwards it to L1, waits for a response from L1, and responds back to the directory.

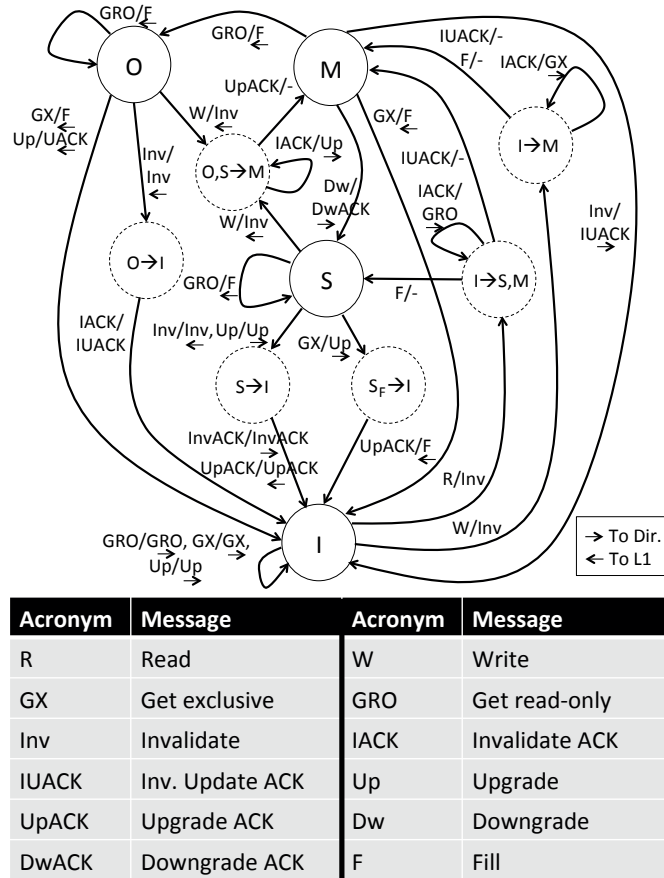


Figure 6.5 – NI cache coherence state diagram.

Figure 6.5 shows the full coherence state diagram for the NI cache. The NI cache’s coherence controller is designed to remain invisible to both the core’s L1 cache and the directory, so that the chip’s coherence mechanism can remain unmodified. To achieve that, it absorbs certain coherence messages that are related to previous coherence messages generated by itself, acting as a proxy of the L1 cache or the directory. The coherence state diagrams of the L1 cache and the directory remain unaffected.

A frequent case that occurs under normal system operation is the $NI_{per-tile}$ (or NI_{split}) cache holding a block in the modified state because of a CQ write, and the core polling on that block, requesting a read-only copy. Under a MESI-based protocol, the cache cannot respond with a dirty block to a read-only request, so it would have to write it back to the LLC first. To optimize for this common case, we introduce an owned state (O on the diagram), only visible to the NI cache

controller. This way, the cache can directly forward a clean version of the requested block to L1 ($M \rightarrow O$ transition on the diagram), while keeping track of its modified state, so that the block eventually gets written back to the LLC upon its eviction.

6.3 A Case Study with Scale-Out NUMA

We use soNUMA to illustrate and instantiate the design points of the previous section, namely NI_{edge} , $NI_{per-tile}$, and NI_{split} .

6.3.1 soNUMA NI Scaling and Placement

Chapter 3 provided a detailed functional overview for each of the three soNUMA pipelines. We now revisit each of the three pipelines (Request Generation Pipeline – RGP, Request Completion Pipeline – RCP, Remote Request Processing Pipeline – RRPP) and illustrate how they can be scaled and mapped to the different NI designs presented in Section 6.2.

The RRPP pipeline is the only pipeline that does not interact with the cores. Therefore, in all the designs we consider, it lies at the chip’s edge nearest to the network router. In order to fully utilize the NOC bandwidth, multiple independent RRPPs are spread out along the edge (e.g., one per edge tile in a tiled CMP as shown in Figure 6.2).

In an NI_{edge} design, the RGP/RCP scales like the RRPP – one pair per edge tile along a chip edge. In an $NI_{per-tile}$ design, a full RGP/RCP pair is replicated per tile and collocated with each core to minimize QP traffic. As described in Section 6.2.2, an essential requirement to reap the benefits of such a collocation is the proper integration of the NI cache with the chip’s default coherence protocol and the core’s L1 cache.

Both NI_{edge} and $NI_{per-tile}$ are suboptimal: the former latency-wise and the latter bandwidth-wise. We next discuss how to overcome the limitations of these designs in soNUMA with the NI_{split} design, which physically splits the RGP and RCP into a frontend and a backend.

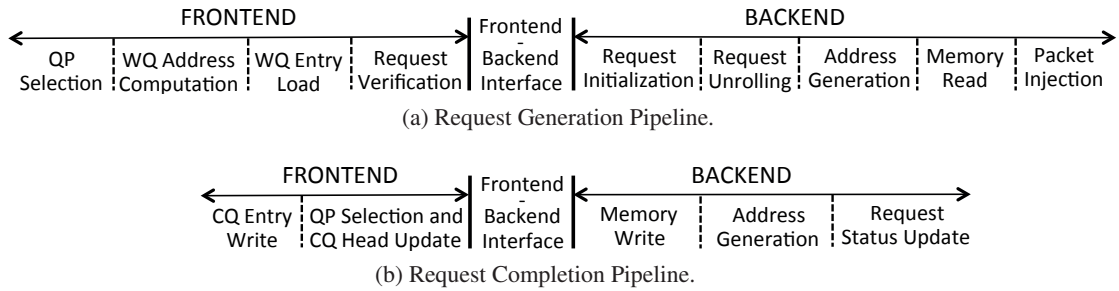


Figure 6.6 – Logical separation of soNUMA’s RGP and RCP into a frontend and a backend.

RGP Frontend/Backend Separation

The frontend/backend split comes naturally in the RGP by separating the stages that interact with the WQs (frontend) from those that act on WQ requests by generating network packets (backend).

Figure 6.6a details the functionality of RGP in stages. The RGP frontend selects a WQ among the registered QPs, computes the address of the target WQ, loads the WQ head, and checks if a new entry is present. The RGP backend initializes the NI’s internal structures to track in-flight requests, unrolls large requests into cache-block-sized transactions, computes and translates the address of the target data and reads it from memory (for writes), and finally injects a request packet in the network.

The *Frontend-Backend Interface* is the boundary between the frontend and the backend. In the NI_{edge} and $NI_{per-tile}$ designs, it is simply a pipeline latch. For the NI_{split} design, the *Frontend-Backend Interface* is an additional stage that generates and sends a NOC packet containing a valid WQ entry from the RGP frontend to its corresponding backend.

RCP Frontend/Backend Separation

The RCP frontend/backend split follows a similar separation of concerns as that of RGP. The RCP backend receives network packets and accesses local application memory to store the remote data. Once all the response packets of a given request have been received, the frontend notifies the application of the request’s completion by writing to the CQ.

Figure 6.6b shows the RCP frontend and backend. The backend is responsible for updating the status of in-flight requests, computing the target virtual address at the local node, and storing the remote data at the translated address. The RCP frontend updates the CQ head pointer in RCP's internal bookkeeping structures and writes a new CQ entry at the CQ's head.

Similar to the RGP, the *Frontend-Backend Interface* is a latch in NI_{edge} and $NI_{per-tile}$ designs. For NI_{split} , it is an additional stage that packetizes and pushes a new CQ entry into the NOC from the RCP's backend to its corresponding frontend.

In the NI_{split} soNUMA design, we integrate the RGP and RCP frontends in each tile (Figure 6.4), thus minimizing the overhead of transferring the QP entries between the NI and the core. The interaction between the core and the frontend logic is handled by the mechanism described in Section 6.2.4. The RGP and RCP backends are replicated across the chip's edge nearest to the network router. Scaling the backend across the edge allows utilization of the NOC's full bisection bandwidth by locally generated requests.

6.3.2 Other Design Issues

Mapping of Frontends to Backends. There is no inherent limitation in the binding of a pipeline frontend to a backend. In this work, we consider a simple mapping, whereby all the frontends of a NOC row map to that row's backend, minimizing frontend-to-backend distance.

Mapping of Incoming Traffic to RRPPs. Distribution of incoming requests to the chip's RRPPs is address-interleaved to minimize the distance to the request's destination tile. This functionality can be trivially supported in the network router by inspecting a few bits of each request's offset field in its soNUMA header under the following assumptions: (i) the directory and LLC are statically address-interleaved across the chip's tiles, and (ii) the address bits that define a block's home location in the tiled LLC are part of the physical address (i.e., these bits fall within the page offset), so this location can be determined prior to translation. Such traffic distribution minimizes on-chip traffic and latency, as it guarantees a minimal number of on-chip

hops for each request to reach its home location in the LLC.

On-Chip Routing Implications. In our evaluated chip designs, we place NIs (RRPPs and RGP/RCP backends) on one side of the chip and memory controllers (MCs) on the opposite side. We found that on-chip routing is critical to effective bandwidth utilization, and conventional dimension-order routing, such as XY or even O1Turn [153], can severely throttle the peak data transfer bandwidth between the chip’s NI and MC edges. Such behavior occurs because most packets originating at a remote node (i.e., remote requests as well as responses to this node’s requests) end up as DRAM accesses, since the requested or delivered data is typically not found in on-chip caches. Under XY routing, all memory requests are first routed to the edge columns, where the MCs reside, and then turn to reach their target MC. The NOC column interfacing to the MCs turns into a bottleneck, reducing the overall bandwidth. If YX routing is used instead, a similar problem arises with responses originating at the MC tiles.

Recent work has proposed Class-based Deterministic Routing (CDR) [6] as a way of overcoming the MC column congestion bottleneck. CDR leverages both XY and YX routing, with the choice determined by the packet’s message class (e.g., memory requests use YX routing while responses XY).

In the soNUMA design, the MC-oriented policy employed by CDR is insufficient, as edge-placed NIs (such as RGP/RCP backends in the case of NI_{split}) can also cause peripheral congestion. To avoid the edge column with the NIs becoming a hotspot, we modify CDR by defining a new packet routing class for directory-sourced traffic; all messages of this class are routed YX, while the rest follow an XY route. This policy results in better utilization of the NOC’s internal links and reduced pressure on the NOC’s edge links, as directory-sourced traffic never turns at the chip’s edges.

| | |
|--------------|---|
| Cores | ARM Cortex-A15-like; 64-bit, 2GHz, OoO, 3-wide dispatch/retirement, 60-entry ROB |
| L1 Caches | split I/D, 32KB 2-way, 64-byte blocks, 2 ports, 32 MSHRs, 3-cycle latency (tag+data) |
| LLC | Shared block-interleaved NUCA, 16MB total 16-way, 1 bank/tile, 6-cycle latency Mesh: 1 tile/core, NOC-Out: 8 tiles in total |
| Coherence | Directory-based Non-Inclusive MESI |
| Memory | 50ns latency |
| Interconnect | 16B links. 2D mesh: 3 cycles/hop NOC-Out: Flattened Butterfly: 2 tiles/cycle Tree Networks: 1 cycle/hop |
| NI | 3 independent pipelines (RGP, RCP, RRPP) one RRPP per row (8 in total) |
| Network | Fixed 35ns latency per hop [165] |

Table 6.2 – System parameters for simulation on Flexus.

6.4 Methodology

Simulation. We use Flexus [174], a full-system cycle-accurate simulator, to evaluate our 64-core chip designs. The parameters used are summarized in Table 6.2. The NIs for all NI designs are modeled in full microarchitectural detail.

We focus our study on a single node, with remote ends emulated by a traffic generator that matches the outgoing request rate of the node that is simulated by generating incoming request traffic at the same rate. Incoming requests are address-interleaved among RRPPs as described in Section 6.3.2.

We assume a fixed chip-to-chip network latency of 35ns per hop [165] and monitor the average servicing latency of local RRPPs that are simulated in detail. This RRPP latency is added to the network latency (which is a function of hop count), thus providing the roundtrip latency of a request once it leaves the local node.

Interface Placement. We evaluate three different placements of the RGP and RCP NIs: NI_{edge} , $NI_{per-tile}$, and NI_{split} . For all three placements, RRPP NIs are placed across a chip’s edge, next to the network router. Such placement provides the ensemble of these NIs access to the full chip

bisection bandwidth for servicing of incoming requests. Memory controllers are placed on the opposite side of the chip.

Memory and Network Bandwidth Assumptions. The focus in this study is the investigation of the implications of NI design on manycore chips. As such, we intentionally assume high-bandwidth off-chip interfaces to both memory and the intra-rack network that do not bottleneck our studied workloads. Technology-wise, high-bandwidth memory interfaces are emerging in the form of on-package DRAM [12] and high-speed SerDes. For instance, Micron’s Hybrid Memory Cube provides 160GBps ($15\times$ more than a conventional DDR3 channel) with a quad of narrow SerDes-based links [85]. On the networking side, the recently finalized IEEE 802.3bj standard codifies a 100Gbps backplane Ethernet running over a quad-25Gbps interface, with Broadcom already announcing fully compliant 4×25 Gbps PHYs. Beyond that, chip-to-chip photonics is nearing commercialization [83], with 100Gbps signaling rates demonstrated and 1Tbps anticipated [15].

Network-on-chip. Since we do not throttle the network or memory bandwidth, the NOC becomes the main bandwidth limiter. We use a mesh as the baseline NOC topology and apply CDR to route on-chip traffic, as described in Section 6.3.2. We also validate the applicability of our observations on latency-optimized NOCs through a separate case study with NOC-Out [118], the state-of-the-art NOC for scale-out server chips.

Microbenchmarks. The goals of this work are to understand the implications of NI design choices on the latency and bandwidth of remote memory accesses in rack-scale systems. Toward that goal, our evaluation relies on microbenchmarks as a way of isolating software and hardware effects in tightly-integrated messaging architectures such as soNUMA while also facilitating a direct comparison to a hardware-only scheme (i.e., a NUMA architecture with a load/store interface to remote memory).

We use a remote read microbenchmark to measure the latency and bandwidth behavior of the

evaluated NI designs. For latency, we study the unloaded case: a single core issuing synchronous remote read operations. Bandwidth studies are based on a remote read microbenchmark, in which remote reads are issued asynchronously: as long as there is space left in the WQ, the application keeps enqueueing new remote read requests, while occasionally polling its CQ for completions. If the 128-entry WQ is full, the application spins on the CQ until an earlier request's completion frees a WQ entry.

We vary the size of the remote reads from 64B to 16KB. Both the soNUMA context, the memory region accessed by remote requests, and local buffers, where requested remote data are written to, are sized to exceed the aggregate on-chip cache capacity, forcing all accesses to hit DRAM. We monitor the metrics of interest (latency, bandwidth) in 500K-cycle windows and run the simulation until the metric's value stabilizes (i.e., when the delta between consecutive monitoring windows is less than 1%).

6.5 Evaluation

6.5.1 Latency Characterization

We first provide a tomography of the end-to-end latency for a single block transfer and show where time goes for each of the three evaluated NI designs. We then show the latency sensitivity of a read request to the size of the transfer.

Single-Block Transfer Latency Breakdown

Table 6.3 shows the latency breakdown for a single-block remote read request. The first three design points show the performance for a messaging-based design, differing in the placement of the NIs that interact with the cores. The last column of the table is a projection of the performance of an ideal NUMA machine, which can access remote memory through its load/store interface without any of the overheads associated with messaging. We optimistically assume that issuing a load/store instruction only requires a single cycle. The cost of traversing the NOC from the

| Latency Component | NI_edge | Latency Component | NI_per-tile | Latency Component | NI_split | Latency Component | NUMA projection |
|-----------------------------------|--------------|-------------------------------|--------------|---------------------------------|--------------|-----------------------------------|-----------------|
| WQ write software overhead | 104 | WQ write software overhead | 13 | WQ write software overhead | 13 | Remote read issuing (single load) | 1 |
| WQ read and RGP processing | 95 | WQ entry transfer | 5 | WQ entry transfer | 5 | | |
| | | RGP Processing | 7 | RGP frontend processing | 4 | | |
| | | Transfer request to chip edge | 23 | Transfer request to RGP backend | 23 | Transfer request to chip edge | 23 |
| Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 |
| RRPP servicing | 208 | RRPP servicing | 208 | RRPP servicing | 208 | RRPP servicing | 208 |
| Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 |
| RCP processing and CQ entry write | 79 | Transfer reply to RCP | 23 | RCP backend processing | 4 | Transfer reply to requesting core | 23 |
| | | RCP processing | 11 | Transfer reply to RCP frontend | 23 | | |
| | | CQ entry transfer | 5 | RCP frontend processing | 8 | | |
| | | CQ read software overhead | 10 | CQ entry transfer | 5 | | |
| CQ read software overhead | 84 | CQ read software overhead | 10 | CQ read software overhead | 10 | | |
| Total (2GHz cycles) | 710 | Total (2GHz cycles) | 445 | Total (2GHz cycles) | 447 | Total (2GHz cycles) | 395 |
| Overhead over NUMA | 79.7% | Overhead over NUMA | 12.7% | Overhead over NUMA | 13.2% | - | |

Table 6.3 – Zero-load latency breakdown of a single-block remote read.

core to the edge, network latency, and reading the data at the remote end are the same as for the messaging interface.

A critical observation is that the actual software overhead to issue and complete a remote read operation is mainly attributed to microarchitectural aspects rather than the number of instructions that need to be executed. While NI_{edge} suggests that the software overhead is as high as 188 cycles to issue and complete a request (the sum of *WQ write* and *CQ read* software overheads in Table 6.3), the other two designs show that the actual instruction execution overhead is just 23 cycles. The remaining 165 cycles are the result of bouncing a QP block between the core's and the NI's caches via the normal cache coherence mechanisms.

Although modern coherence mechanisms are considered to be extremely efficient for on-chip block transfers, these results indicate that high-performance NI designs should not rely on the assumption that coherence-powered transfers are free from a latency perspective. Coherence protocols intrinsically introduce points of indirection, which can turn a single transfer into a long-latency sequence of several multi-hop chip traversals. These subtle interactions must be taken into consideration when architecting a high-performance NI.

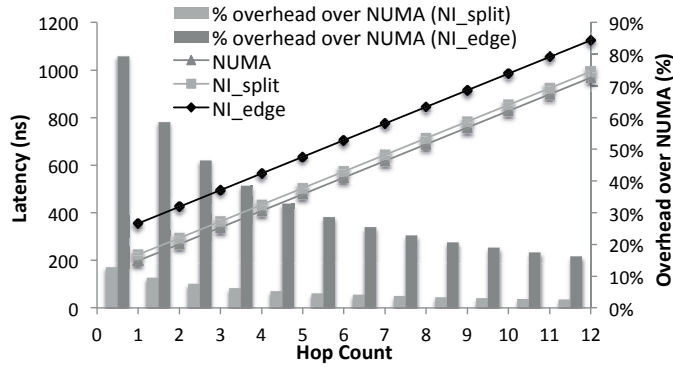


Figure 6.7 – Projection of the end-to-end latency of a cache-block remote read operation for multiple intra-rack network hops. Bars map to the right y-axis, lines to the left y-axis.

Scaling to More Network Hops

To put the latency numbers in perspective, Figure 6.7 projects the end-to-end latency for reading a single cache block in large-scale distributed memory system, accounting for multiple network hops. The projection is based on Table 6.3’s breakdown, accounting for 70 cycles (35ns) of network latency per hop, per direction. We assume a 512-node deployment in a 3D torus topology; the average and maximum hop counts between two nodes in such deployment are 6 and 12 respectively. This model is more representative of future large rack-scale systems. A similar approach could be applied at datacenter scale, assuming future optic datacenter-scale networks, with a couple of key differences: (i) the network topology would be indirect (typically a fat tree), meaning that messages would go through a number of intermediate switches (with their associated added latency), but also the average and maximum number of hops would be lower; and (ii) the latency per hop would be higher, as a result of longer distances.

Figure 6.7 shows that the additional on-chip transfers related to QP interactions that occur in the case of NI_{edge} account for a significant fraction of the end-to-end latency, inducing a 28.6% overhead over NUMA for six network hops. In comparison, NI_{split} significantly reduces the time spent on QP interactions, bringing the end-to-end latency within 4.7% of NUMA. Even in the worst case of traversing the entire diameter of the modeled 3D torus, the difference in the end-to-end latency overhead between NI_{edge} and NI_{split} is still significant: 16.2% vs. 2.6%

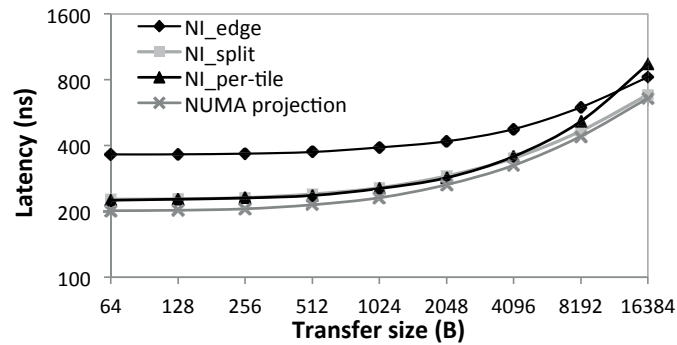


Figure 6.8 – End-to-end latency for synchronous remote reads.

over NUMA. These results indicate that a high-performance NI design must consider the node’s microarchitectural features, but highly invasive microarchitectural modifications (such as those associated with a load/store interface to remote memory) are not warranted.

Latency of Larger Requests

Figure 6.8 shows the end-to-end latency of a synchronous remote read operation in an unloaded system assuming a single network hop per direction. We project the latency of an ideal NUMA machine by subtracting the latencies associated with QP interactions in the NI_{split} design as shown in Table 6.3.

We observe that as the transfer size increases, the relative latency difference between NI_{edge} , NI_{split} , and NUMA shrinks because the cost of launching remote requests through QP interactions is amortized over many cache blocks. However, that is not the case for $NI_{per-tile}$, which observes the highest latency among all evaluated designs for the largest transfer sizes. This behavior is caused by unrolls of large transfers into cache-block-sized transactions which, in the $NI_{per-tile}$ design, take place at the source tile. Because each network request packet is encapsulated inside a NOC packet, it requires two flits to transfer from the source tile to the network router at the chip’s edge. Meanwhile, unrolls happen at a rate of one request per cycle, resulting in queuing at the source tile. While a wider NOC would alleviate the bandwidth pressure caused by unrolling at the source tile, the cost-effective solution is to provide hardware support for offloading bulk transfers to the chip’s edge, as is done in the NI_{edge} and NI_{split} designs.

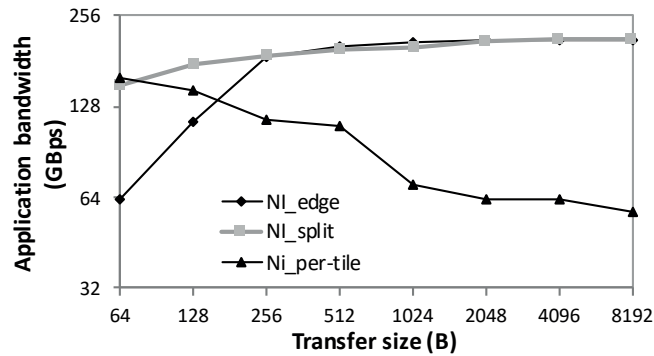
6.5.2 Bandwidth Characterization

For bandwidth measurements, all 64 cores are issuing asynchronous requests of varying sizes. Figure 6.9a shows the aggregate application bandwidth for each of the three NI designs. The bandwidth is measured as the rate of data packets written into local buffers by RCPs for locally initiated requests, and the rate of data packets sent out by RRPPs in response to remote requests serviced at the local node. Because of the way remote traffic is generated, these two rates are always balanced, and the reported aggregate bandwidth is the sum of the two.

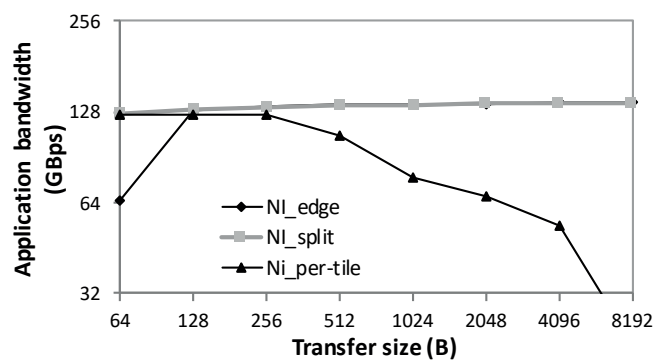
Both NI_{edge} and NI_{split} reach a peak bandwidth of 214GBps, or 107GBps per direction. It is unlikely that the bandwidth can be pushed any further using the same NOC; NOC traffic counters report an aggregate bandwidth of 594GBps, with the bulk of it crossing the bisection whose bidirectional bandwidth is 512GBps. The aggregate consumed bandwidth is $2.7\times$ higher than the application bandwidth demand; the difference is attributed to a plethora of NOC packets that are not carrying application data. These other packets include coherence messages and evicted LLC blocks requiring a write-back to memory.

As Figure 6.9a shows, $NI_{per-tile}$ and NI_{split} reach higher bandwidth than NI_{edge} for small transfer sizes. NI_{edge} suffers from ping-ponging of the WQ and CQ entries between the cores and the NIs, particularly when a cache block containing WQ entries gets polled and transferred to the NI before the application completely fills it with requests; a similar effect occurs with CQ cache blocks that are invalidated by the core while new completions are processed at the NI. With larger transfer sizes, QPs are accessed less frequently, thus diminishing their effect on performance.

Whereas NI_{edge} is inefficient for small transfers, the performance of the $NI_{per-tile}$ design degrades at large transfer granularities. The reason is that the NIs in this design unroll the requests inside the NOC, resulting in a flood of packets streaming from the tiles to the edges. By the time the backpressure reaches the source tiles, the network is completely congested. A similar problem occurs with responses: once they arrive at the network router, they are first sent back to the source NI, regardless of the final on-chip destination of the payload, thus introducing an unnecessary



(a) With CDR routing.



(b) With X-Y routing.

Figure 6.9 – Application bandwidth for asynchronous remote reads.

point of indirection, which further increases on-chip traffic. The congestion problems could be mitigated through smarter (e.g., source-based) flow-control, but the aggregate bandwidth would still be inferior to the other two designs because of the extra on-chip traffic due to the per-tile NI placement.

Clearly, efficient handling of large unrolls requires having a block handling engine at the edge, which receives a single command, does the data transfers, and finally notifies the requester upon completion. This observation is not limited to messaging, but equally applies to load/store NUMA systems as well.

Finally, Figure 6.9b shows the same results when X-Y on-chip routing is used instead of our modified CDR policy baseline (see Section 6.3.2). The bandwidth curves the three NI designs are qualitatively similar for both CDR and X-Y routing, but the peak bandwidth of the bandwidth-

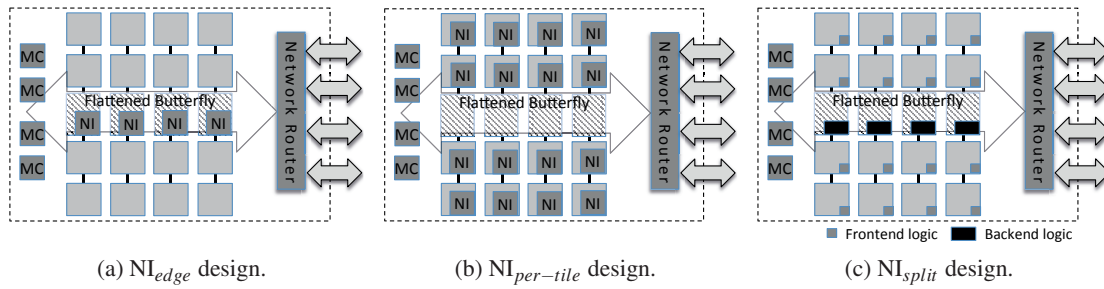


Figure 6.10 – NI design space for NOC-Out-based manycore CMPs. Striped rectangles represent LLC tiles.

optimized designs (NI_{edge} and NI_{split}) is 35% lower with X-Y as compared to CDR (138GBps versus 214GBps). The reason for that bandwidth degradation is that X-Y routing results in early congestion of the NOC links on the two edge columns of the chip, where the MCs and the NIs reside.

6.5.3 Effect of Latency-Optimized Topology

In this section, we show that trends and conclusions derived from the mesh-based study are equally valid for latency-optimized NOCs. To that end, we evaluate the various NI design options using NOC-Out [118], a state-of-the-art latency-optimized NOC for scale-out server chips. In the NOC-Out layout, LLC tiles form a row in the middle of the chip and are richly interconnected via a flattened butterfly. Cores lie on both sides of the LLC row, and the cores of each column are chained via a simple reduction/dispersion network that connects them to their column’s corresponding LLC tile.

Figure 6.10 illustrates the three NI design options in the context of NOC-Out. The LLC tiles are spread across the middle of the chip and are interconnected via the flattened butterfly to each other, the MCs, and the network router. In all three designs, the RRPPs (not shown) are placed across the chip’s LLC tiles rather than the chip’s edge, as the rich connectivity of these tiles provides access to the full bisection bandwidth. For the same reason, the RGP and RCPs in the case of NI_{edge} are collocated with the RRPPs. While NI_{middle} would be a more accurate term for this placement, we continue using NI_{edge} for consistency. $NI_{per-tile}$ features full RGP and

| Latency Component | NI _{edge} | Latency Component | NI _{per-tile} | Latency Component | NI _{split} | Latency Component | NUMA projection | | |
|-----------------------------------|--------------------|-------------------------------|------------------------|---------------------------------|---------------------|-----------------------------------|-----------------|----------------------------|-----|
| WQ write software overhead | 62 | WQ write software overhead | 13 | WQ write software overhead | 13 | Remote read issuing (single load) | 1 | | |
| WQ read and RGP processing | 46 | WQ entry transfer | 5 | WQ entry transfer | 5 | | | | |
| | | RGP Processing | 7 | RGP frontend processing | 4 | | | | |
| RGP to network router | 7 | Transfer request to chip edge | 16 | Transfer request to RGP backend | 9 | Transfer request to chip edge | 16 | | |
| Intra-rack network (1 hop) | 70 | | | Intra-rack network (1 hop) | 70 | | | Intra-rack network (1 hop) | 70 |
| RRP servicing | 132 | | | RRP servicing | 132 | | | RRP servicing | 132 |
| Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | Intra-rack network (1 hop) | 70 | | |
| Network router to RCP | 7 | Transfer reply to RCP | 16 | Network router to RCP backend | 7 | Transfer reply to requesting core | 16 | | |
| RCP processing and CQ entry write | 53 | | | RCP backend processing | 4 | | | | |
| | | | | Transfer reply to RCP frontend | 9 | | | | |
| | | | | RCP frontend processing | 8 | | | | |
| CQ read software overhead | 53 | CQ entry transfer | 5 | CQ entry transfer | 5 | | | | |
| | | CQ read software overhead | 10 | CQ read software overhead | 10 | | | | |
| Total (2GHz cycles) | 500 | | 355 | | 357 | | 305 | | |
| Overhead over HW NUMA | 63.9% | | 16.4% | | 17.0% | | - | | |

Table 6.4 – Zero-load latency breakdown of a single-block remote read (NOC-Out).

RCP pipelines at each core, while NI_{split} has an RGP/RCP frontend per core and an RGP/RCP backend per LLC tile.

Single-block Transfer Latency Breakdown with NOC-Out

Table 6.3 in Section 6.5.1 showed the end-to-end latency breakdown of a synchronous remote read for all NI designs on a mesh-based chip. We repeat that latency tomography for a NOC-Out on-chip interconnect and show the results in Table 6.4. The latency-optimized NOC-Out noticeably speeds up on-chip interactions both at the local and the remote end. The end-to-end latency’s reduction for all designs falls in the range of 20–30%, compared to their mesh-based counterparts. Improvements at the source side originate from accelerated QP interactions and faster transfers of requests and responses between the NIs located in the inner chip and the network router at the chip’s edge. The flattened butterfly interconnecting the network router, LLC tiles, and memory controllers reduces the time spent at the remote end by 37% (132 cycles) as compared to mesh-based designs (208 cycles).

Despite the fact that significantly less time is spent on on-chip interactions as compared to the mesh, all of the previously made observations still hold. While the latency gap between NI_{edge} versus $NI_{per-tile}$ and NI_{split} is narrowed, the end-to-end latency improvement of the two latency-optimized NI designs over NI_{edge} still measures up to $\sim 30\%$. This result indicates that

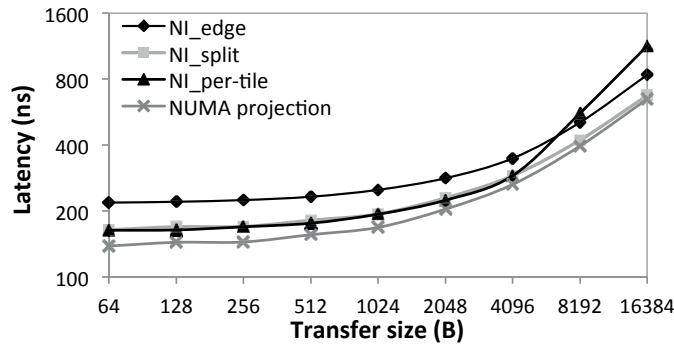


Figure 6.11 – Latency for synchronous remote reads on NOC-Out.

on-chip QP interactions still account for a considerable fraction of the end-to-end latency, even on latency-optimized NOC topologies.

Latency and Bandwidth Measurements with NOC-Out

Figure 6.11 shows the end-to-end latency of synchronous remote read operations of various sizes for all three NI designs. For small transfers, NOC-Out delivers up to 30% lower latency than mesh (Figure 6.8). Examining the sources of improvement, we find that latency is reduced both at the source and remote nodes. Improvements at the source node originate from accelerated QP interactions and faster transfers of requests and responses between the NIs and the network router. At the remote node, the flattened butterfly speeds up the access latency to the LLC and MCs by 37% compared to mesh-based designs.

Comparing the latency gap between NI_{edge} and the other two designs, we observe that it is narrowed compared to the mesh topology, yet the latency of NI_{edge} is still up to 30% greater than that of NI_{split} and $NI_{per-tile}$. This result indicates that on-chip QP interactions still account for a considerable fraction of the end-to-end latency even in latency-optimized NOC topologies.

Bandwidth results for NOC-Out appear in Figure 6.12. The general trends are identical to those observed with a mesh (Figure 6.9). However, the peak bandwidth achieved with NOC-Out is significantly lower than that in mesh-integrated NIs. The reason for the low throughput is the

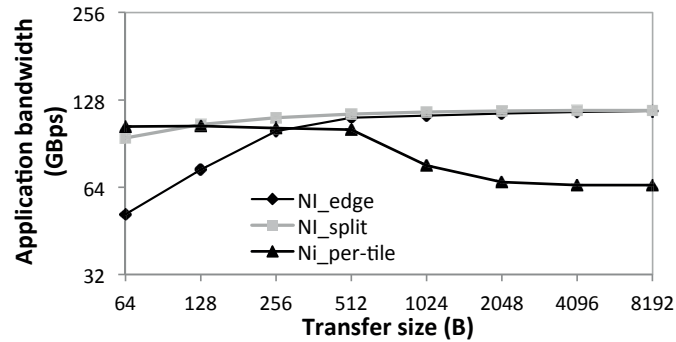


Figure 6.12 – Application bandwidth for asynchronous remote reads on NOC-Out.

highly contended LLC in the NOC-Out organization, which has significantly fewer tiles and banks than its mesh-based counterpart.

6.6 Chapter Summary

The emergence of manycore server chips in the context of integrated rack-scale fabrics, where low latency and high bandwidth between nodes is crucial, introduces new challenges for on-chip NI integration. Because of inherently high on-chip latencies on manycore chips, initiation and termination of remote operations that take place at the NIs can become a first-order performance determinant for remote memory access, especially for emerging QP-based models. We investigated three different integrated NI designs for manycores. The classic NI_{edge} integration approach, where the NIs are placed across a chip’s edge, can utilize the full bisection bandwidth of the NOC, but suffers from significant latency overheads due to costly on-chip core-NI interactions. The $NI_{per-tile}$ design integrates the NI logic next to each core, rather than the chip’s edge, and delivers end-to-end latency for fine-grained remote memory accesses that is within 3% of hardware NUMA’s latency. However, the $NI_{per-tile}$ design generates excess traffic that reduces the bandwidth for bulk data transfers significantly. To achieve the best of both worlds, we propose a novel manycore NI design, NI_{split} , that delivers the latency of $NI_{per-tile}$ and the bandwidth of NI_{edge} .

7 LightSABRe in Action

Chapter 4 introduced SABRe, a new one-sided operation with the rich semantics of an atomic object read, and LightSABRe, a hardware implementation of SABRe coupled with soNUMA's NI. In this chapter we evaluate the performance impact of LightSABRe on distributed systems. We consider a system featuring manycore servers implementing soNUMA and the NI_{split} architecture, as introduced in Chapter 6. We briefly address the implications arising from applying the LightSABRe mechanism on a manycore NI architecture, then proceed to describe our methodology and performance evaluation.

7.1 Manycore NI Architecture Implications on LightSABRe

Figure 7.1 shows a chip implementing a manycore NI_{split} architecture. Each of the chip's RRPPs features a LightSABRe, which handles incoming SABRes. Multiple RRPPs give rise to the implication of balancing requests across them, as already discussed in Section 6.3.2 in the context of basic one-sided operations. However, the load distribution concern is different for SABRes, as multiple packets of the same request are related to each other; in contrast, for other one-sided operations, every incoming cache-block-sized request is handled independently. Since SABRes can be arbitrarily long and can differ in size, distributing a single SABRe across multiple RRPPs would be desirable for optimal load balancing. Supporting such distribution

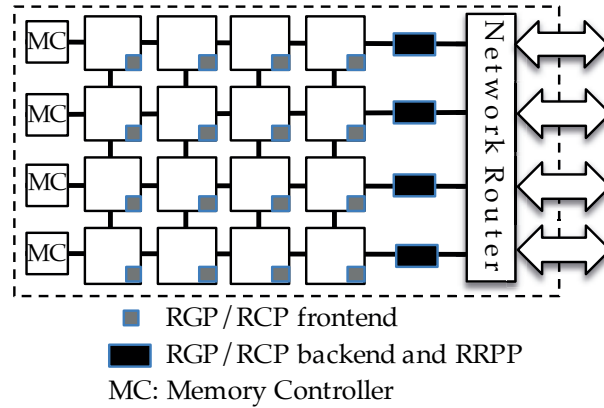


Figure 7.1 – Multicore chip layout based on NI_{split} architecture.

would require breaking a SABRe operation into multiple sub-operations that would *all together* have to be atomic, introducing additional hardware design complexity to implement distributed logic to enable that. Furthermore, given that each transfer originates from a single RGP, all reply packets have to be routed back to that pipeline’s matching RCP, which will ultimately become the transfer’s bandwidth bottleneck. Therefore, the additional complexity required for inter-SABRe distribution seems unwarranted and our implementation of LightSABRe for soNUMA maps each SABRe to a single RRPP.

7.2 Methodology

System organization. We evaluate LightSABRe by modeling two directly connected 16-core chips that implement soNUMA with LightSABRe-enhanced RMCs. Figure 7.1 shows the layout of the modeled chips, which implement the NI_{split} design introduced in Chapter 6. This means that RGPs and RCPs are split into frontends and backends; frontends are replicated per core and handle the memory-mapped queue-based interaction with the cores, while backends are replicated across the chip’s edge, for efficient data handling. RRPPs, which are the pipelines implementing the LightSABRe mechanism, are monolithic and replicated across the chip’s edge.

Core-to-RMC-backend and SABRe-to-RRPP mapping. Our design maps each SABRe to a single RRPP rather than across multiple of them. This design choice has a significant

| | |
|--------------|--|
| Cores | ARM Cortex-A57-like; 64-bit, 2GHz, OoO 3-wide dispatch/retirement, 128-entry ROB, TSO |
| L1 Caches | 32KB 2-way L1d, 48KB 3-way L1i, 64-byte blocks 2 ports, 32 MSHRs, 3-cycle latency (tag+data) |
| LLC | Shared block-interleaved NUCA, 2MB total 16-way, 1 bank/tile, 6-cycle latency |
| Coherence | Directory-based Non-Inclusive MESI |
| Memory | 50ns latency, 4×25.6GBps (DDR4) |
| Interconnect | 2D mesh, 16B links, 3 cycles/hop |
| RMC | 3 independent pipelines (RGP, RCP, RRPP) @ 1GHz one RGP/RCP frontend per core (Figure 7.1) four RGP/RCP backends & RRPPs across edge |
| LightSABRe | 16 32-entry stream buffers per RRPP |
| Network | Fixed 35ns latency per hop [165], 100GBps |

Table 7.1 – Flexus simulation parameters for LightSABRe on soNUMA.

advantage in terms of simplicity, but may introduce load-balancing concerns. Assuming uniform behavior across cores regarding remote memory accesses, we map each row of cores to the row’s corresponding RGP backend, and all SABRes from a given RCP backend to the matching remote RRPP. In our setup, this static mapping does not result in any load imbalance issues. A more flexible, possibly dynamic load-balancing policy might be worth investigating in the future.

Simulation. We use the Flexus [174] full-system cycle-accurate simulator to evaluate the LightSABRe-enhanced soNUMA system. Table 7.1 summarizes the used parameters.

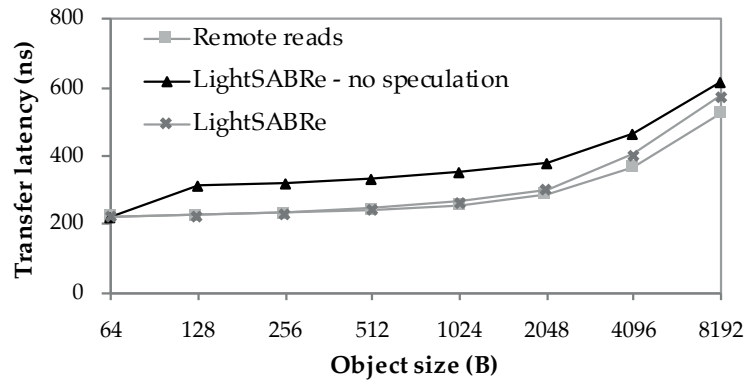
Applications. We use a simple microbenchmark to study the performance of LightSABRe in isolation. Our microbenchmark launches a number of writer threads that update objects in their local memory, or reader threads that access objects in remote memory using one-sided soNUMA operations (remote reads or SABRes) in a tight loop.

We also use FaRM [53] to evaluate the effect of LightSABRe on a full software stack. FaRM is a transactional system for distributed memory with an underlying key-value data store, that uses RDMA for fast remote memory access. In particular, FaRM uses one-sided reads to access remote objects over RDMA, while writes are always sent to the data owner over an RPC. FaRM implements atomic remote object reads via optimistic concurrency control by encoding per-

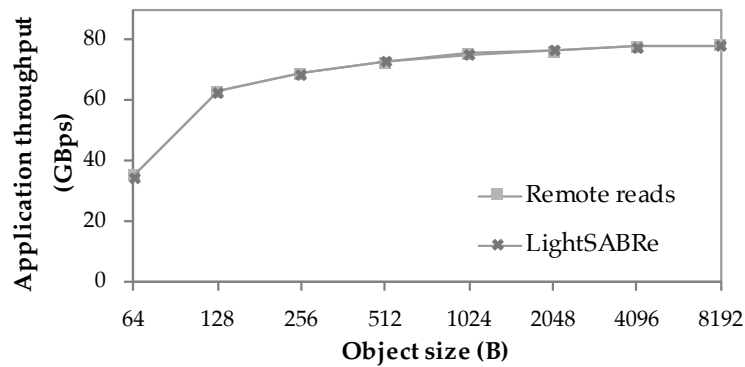
cache-line versions in the objects. The framework detects atomicity violations for (local or remote) reads, should they overlap with a concurrent write to the same object, and retries the read operation. FaRM provides a fast path for lock-free single-object remote read operations, which are strictly serializable with FaRM's general distributed transactions, without invoking the distributed transactional commit protocol. As discussed extensively in Section 4.1.2, the per-cache-line versions mechanism imposes CPU overheads related to the extraction of useful data from the data store by stripping off embedded metadata. An additional consequence of embedded metadata is the requirement for intermediate system-managed buffering before the clean data can be exposed to the application, thus giving up on the zero-copy benefit of one-sided read operations.

We performed the following major modifications to FaRM: (i) we ported the FaRM core from a standard RDMA interface to soNUMA; and (ii) because of Flexus constraints, we ported FaRM from Windows/x86 to Solaris/UltraSPARC III. We also replaced a number of system calls in FaRM, such as timer-related calls, with their most efficient counterparts on Solaris.

We evaluate two implementations of atomic lock-free reads with different object layouts in the FaRM data store. In the baseline implementation, we use soNUMA's remote read primitives combined with the original FaRM data object store (per-cache-line versions layout) and post-transfer atomicity checks in software. In the SABRe implementation, we remove these per-cache-line versions from the objects' layout and use LightSABRe to enforce atomicity. The SABRe implementation also removes the intermediate buffering for the data transferred from remote memory; instead, the one-sided operation can directly write the—already clean—data into the application buffer (zero-copy).



(a) End-to-end latency.



(b) Bandwidth.

Figure 7.2 – Microbenchmark with one-sided operations.

7.3 Evaluation

7.3.1 Latency and Throughput Characterization

We first use a single-threaded microbenchmark that issues synchronous operations, remote reads and SABRes, to assess their latency. To illustrate the benefit of the LightSABRe mechanism over a basic hardware mechanism for SABRes that serializes the version check before data access, we evaluate the performance of both mechanisms (*LightSABRe* versus *LightSABRe - no speculation*). Remote data is memory resident and the local buffer at the source is LLC resident. Figure 7.2a shows the soNUMA transfer latency (from issuing to completion) as a function of the transfer size. For single-block transfers, *remote reads* and both types of *LightSABRe* achieve the same latency, as expected. For larger transfers, the latency of SABRes using the

Chapter 7. LightSABRe in Action

LightSABRe - no speculation mechanism is significantly higher than *remote reads*, because of the read-version-then-data serialization. *LightSABRe* successfully remove this overhead, matching the latency of *remote reads*. The latency difference between *remote reads* and *LightSABRe* in the case of large transfers (2KB) is attributed to the load distribution to RRPPs: while remote reads are balanced on a per-block basis across RRPPs, each SABRe is assigned to a single RRPP.

Figure 7.2b shows the peak throughput of 16 threads issuing asynchronous remote operations (remote reads and SABRes). The remote reads and LightSABRe have identical throughput curves, illustrating that (i) peak theoretical bandwidth (20GBps per RRPP) is reached with both operation types, and (ii) introducing state at the RRPPs does not hurt throughput. The throughput curve of *LightSABRe - no spec.* is also identical, and therefore omitted.

7.3.2 Conflict Sensitivity

We extend the synchronous microbenchmark used in Section 7.3.1 to evaluate LightSABRe's end-to-end effect in the presence of atomicity violations. We use the per-cache-line versions technique to provide atomicity in software, using remote reads. After every transfer, the microbenchmark unpacks the transferred data into an application buffer, checking for atomicity violation in the process. With LightSABRe, such an atomicity check mechanism is not required. In both cases, the end result is the same: a remote operation completes when the clean data is read by the core.

We employ 16 reader threads on one chip and vary the number of writers from 0 to 16 on the other, for a throughput sensitivity analysis as the conflict probability grows. To achieve a perceivable change in conflict probability, we limit the number of objects to 100, making all accesses LLC resident. Readers access all remote objects uniformly at random, while each writer repeatedly writes a predefined subset of the objects (Concurrent Reads Exclusive Writes model [103]). Upon a conflict detection, readers immediately retry reading the same object.

Figure 7.3 compares the microbenchmark's throughput for remote atomic reads of 128B, 1KB, and 8KB objects, when using the software per-cache-line versions mechanism versus LightSABRe. In all cases, we observe a performance degradation as the number of writers, and, consequently,

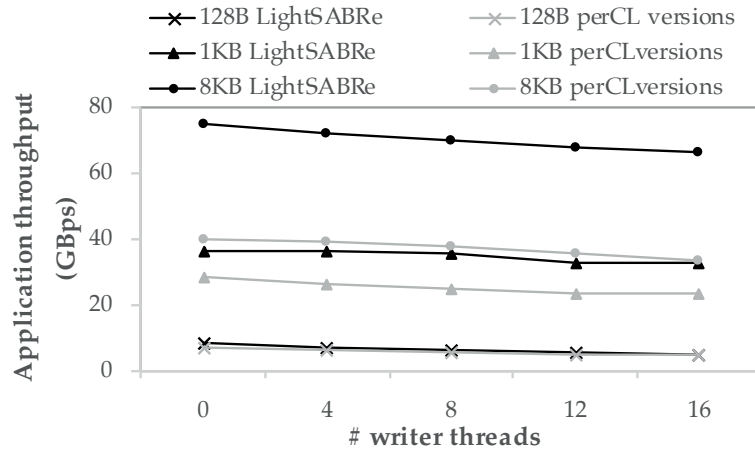
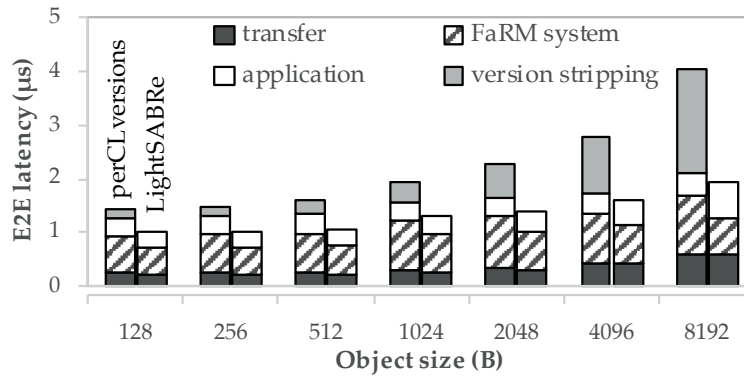


Figure 7.3 – Application throughput with increasing conflict rate.

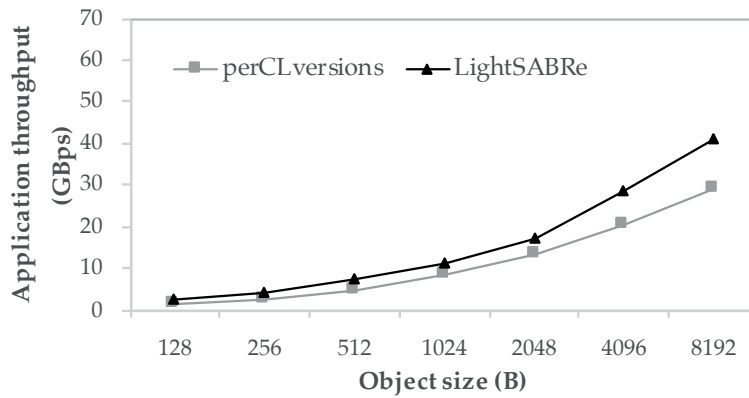
conflict probability, increases. The throughput difference between the software and hardware atomicity enforcement method is a direct result of the reduced end-to-end latency delivered by LightSABRe. We observe an opposite trend for small and large objects. For 128B objects, the application throughput gap between LightSABRe and the software mechanism shrinks from 15% to 3% as the conflict probability increases. In contrast, for 1KB and 8KB objects, the throughput gap grows from 30% to 41%, and from 87% to 97%, respectively. The reason for these differences is two-fold. First, the benefit from removing the software atomicity check is proportional to the object size. Second, atomicity success or failure of completed SABRes is directly exposed to the application through the transfer’s Completion Queue entry. This action is object-size agnostic. In contrast, the cost of software atomicity detection grows with the object size. Therefore, the larger the object size and the conflict probability, the greater the benefit for LightSABRe.

7.3.3 FaRM Key-Value Store

We conclude the evaluation by testing LightSABRe on a read-only key-value store application running on top of FaRM [53]. The first node allocates a number of FaRM objects in its memory, which a single reader thread running on the second node accesses continuously by issuing key-value lookups over synchronous one-sided operations: remote reads versus SABRes. All remote



(a) End-to-end latency breakdown.



(b) Application throughput.

Figure 7.4 – FaRM KV store: baseline versus LightSABRe.

memory accesses miss in the remote LLC and go to main memory.

Figure 7.4a shows the latency breakdown for different object sizes and each of the two evaluated FaRM versions (baseline versus SABRes). LightSABRe considerably reduces the end-to-end latency for atomic remote object reads for all object sizes. We identify two main sources of benefit. The direct benefit comes from the fact that the use of SABRes completely removes the software overhead of version stripping and atomicity checking. The second, implicit, benefit is that SABRes shrink the total instruction footprint, thus reducing frontend stalls, which are critical to performance and a major concern in modern server workloads [62]. As pointed out in the methodology, SABRes not only deprecate the code for software atomicity checks, but also the FaRM code that deals with intermediate buffering, as SABRes allow soNUMA to directly write into the application buffer (zero-copy). We found the application’s instruction working

set to be in the 40–50KB range, which results in L1i conflict misses, even though we deploy a next-line instruction prefetcher. The use of SABRes reduces the instruction working set by $\sim 7\%$, relaxing core frontend pressure. Using SABRes only increases the application’s latency component (Figure 7.4a), because the accessed object is located in the LLC, as opposed to the baseline where the software atomicity check implicitly brings the clean object in the L1d.

The application has two distinct phases: a low ILP/MLP phase with an IPC of 0.8 to 1, and a high-MLP phase, when the transferred remote data is read by the core. In the case of small objects, the largest fraction of the performance benefit provided by SABRes comes from the first phase. The combination of reduced instruction footprint (no version stripping or intermediate buffering code) and a slightly reduced instruction miss ratio results in a 35% overall latency improvement for 128B remote object accesses.

In contrast, the greatest benefit of SABRes for large objects comes from the high-MLP phase, increasing the performance benefit to 52% for 8KB objects. We do not model a data prefetcher, which would be capable of shrinking the gap between SABRes and the baseline for large objects. However, we significantly optimized the version stripping kernel by hand-tuning assembly code to maximize the MLP, at 1KB data chunks; thus, our results for object sizes up to 1KB are guaranteed to get maximum MLP, which a data prefetcher would not improve. Assuming a *perfect* data prefetcher that identifies the access to an object and directly brings all of it in the L1d, so that only the LLC access latency of accessing the first 1KB is exposed, the performance benefit of using SABRes would shrink from 52% to 30-35% for 8KB objects.

The latency benefit of using LightSABRe also results in throughput improvement. We now use 15 FaRM reader threads that access remote objects using synchronous remote operations (reads or SABRes). Figure 7.4b shows that LightSABRe delivers a throughput improvement of 30-60% depending on the object size, as compared to the baseline.

Finally, we evaluate the performance of local reads for the two FaRM object store implementations. While LightSABRe is not involved in local accesses, it’s an enabler for keeping the object store unmodified (i.e., no embedded per-cache-line versions), which implicitly results in faster local

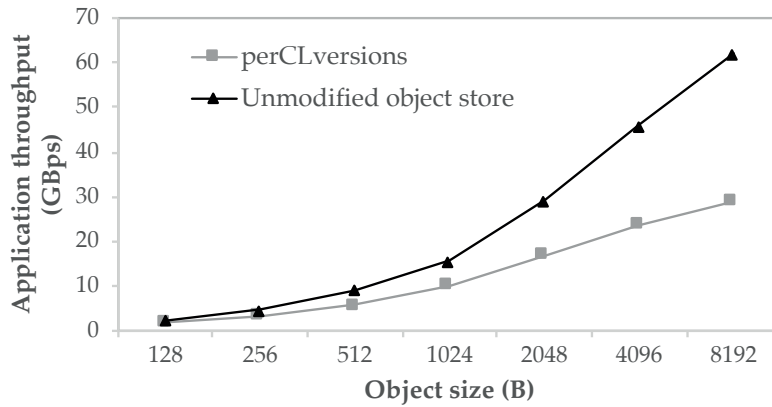


Figure 7.5 – FaRM local reads throughput comparison.

reads. Figure 7.5 shows the application throughput achieved for a read-only key-value lookup kernel on FaRM, with 15 FaRM reader threads issuing read requests to local memory only. We observe a throughput increase of 20% for 128B objects, which grows to 53% for 1KB objects, and a striking $2.1\times$ for 8KB objects. Thus, using SABRes also results in a substantial acceleration of local reads, which are performance-critical even in distributed memory environments, especially in the case of locality-aware applications.

7.4 Chapter Summary

Our LightSABRe evaluation demonstrates the benefit of introducing new one-sided operations with stronger semantics to implement operations that are ubiquitously performed by distributed systems. The introduction the SABRe atomic remote object read operation results in considerable software simplification and significant performance improvements. We report remote object read throughput improvements of up to 97% for a microbenchmark and up to 60% for a key-value lookup application running on top of the full software stack of a modern distributed object store.

8 Tail-Aware Balancing of μ s-Scale RPCs

Chapter 5 introduced the opportunity to implement synchronization-free dynamic load balancing of incoming network messages to cores of a manycore server CPU within the NI logic. Doing so has the potential of significantly improving the RPC throughput of a manycore server under tight tail latency constraints, especially for the most challenging short-lived RPCs with a service time of just a few microseconds. In this chapter, we implement and evaluate our NI-integrated load balancing mechanism on a server implementing soNUMA the NI_{split} architecture, as introduced in Chapter 6. We also implement and evaluate the lightweight native messaging mechanism we introduced in Chapter 5, which is a prerequisite for our load balancing implementation. We first address the design and implementation implications arising when scaling the number of NIs from a single to multiple, and then proceed with our methodology and performance evaluation.

8.1 Manycore NI Design Implications

Scaling the native messaging mechanism to multiple NIs introduces the same implications that surfaced in the case of SABRes. We follow the same approach: As multiple packets of the same message are related with each other (the NI increments the message's corresponding receive slot counter), we opt to steer all packets of the same packet to the same RRPP to reduce inter-NI coherence traffic.

From the load balancing perspective, the implications of multiple NIs are more challenging. As multiple NIs handle message receptions, there is a need to dispatch work from multiple sources to multiple destinations (cores), which inherently implies a multi-queue system. From our theoretical queuing system analysis in Section 5.1, the performance of multi-queue systems is inferior to single-queue systems. However, there is a spectrum of resource pooling versus achieved performance, that—from the queuing system organization perspective—introduces a range of options. Motivated by Section 5.1’s analysis, we consider implementations of two queuing systems, each of which strikes a different tradeoff point in the complexity/performance design space. For our evaluated 16-core chip, illustrated in Figure 8.1, we consider an implementation of the theoretically ideal single-queue system (1×16) and an implementation of a 4×4 queuing system, which is modestly inferior in terms of performance to 1×16 , but represents a simpler design. We first describe the 4×4 implementation and then the extensions required to upgrade it to a 1×16 queuing system.

8.1.1 4×4 Queuing System

A manycore NI_{split} architecture enables seamless scalability of networking capabilities with the number of cores, in terms of latency and bandwidth. However, the distributed nature of the NI logic introduces a challenge: The otherwise independent NI backends (each of which features an RRPP taking message dispatch decisions) need to coordinate to balance incoming load across cores. Driven by the observation that even a multi-queue system with a modest number of serving units per queue can approach the performance of a single-queue system, we constrain the number of cores every NI backend can dispatch load to.

Figure 8.1 demonstrates such an example for a 16-core tiled chip with a mesh on-chip interconnect: an NI backend limits its dispatch decisions to the cores residing on the same mesh row. Such a decision results in a load balancing configuration that corresponds to a 4×4 queuing system. As compared to a 1×16 system, 4×4 compromises load balance and, ultimately, tail latency, in favor of reduced complexity. Each NI backend has its own *shared CQ* and its own set of cores to dispatch incoming messages to. Therefore, no inter-NI coordination is required neither when a

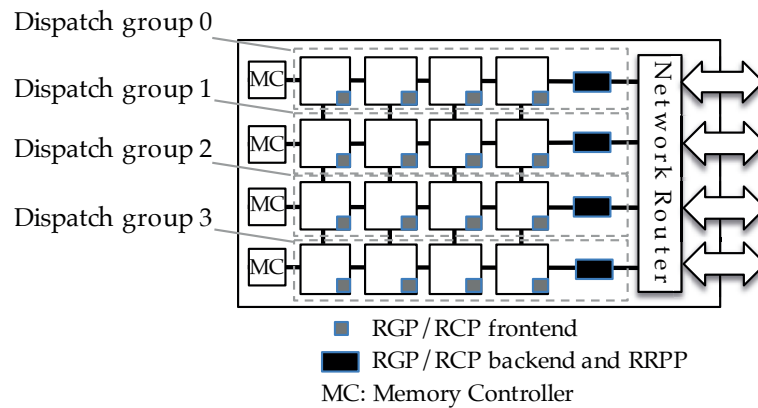


Figure 8.1 – Message dispatch groups on a multicore chip.

new message arrival is enqueued in the shared CQ, nor when a core becomes available, waiting for a new message to be dispatched to it by an NI. Furthermore, each core only notifies a single NI about its occupancy (one-to-one instead of one-to-many communication). By assigning a disjoint set of cores to every NI backend, this 4×4 implementation eschews the need for inter-NI coordination and maintains the original manycore NI design’s scalability, where all NI backends operate independently.

8.1.2 1×16 Queuing System

A 4×4 system organization strikes a balance between complexity and performance by eschewing inter-NI communication. However, given the load dispatch mechanism’s specialized nature and the predictable inter-component communication patterns it results in, the additional requirements to implement a single-queue system may prove to be modest and hence justifiable. One option would be to approach the problem as a small-scale distributed system, with the goal of optimizing the inter-NI synchronization necessary when an incoming message needs to be enqueued into a single shared CQ, and when it’s time for a message to be dispatched from the shared CQ to a core. The solution would not be necessarily limited to purely algorithmic approaches, but could also involve hardware support, such as dedicated links interconnecting the handful of NI backends to guarantee fixed-time communication between them (e.g., a similar concept has been used to enable snoopy coherence over a mesh on-chip interconnect [45]).

A second option that turns out to lead to a surprisingly simple and effective solution, is to centralize the last step of message reception and dispatch, instead of handling it in a distributed fashion. One of the NI backends—henceforth referred to as *NI dispatcher*—is statically assigned to handle the last stage of message dispatch to all the available cores. The number of shared CQs is reduced to one, and it's only accessible by the NI dispatcher. Network packet and data handling still benefits from the parallelism offered by the manycore NI architecture, as all NI backends still independently handle incoming network packets and access memory directly. However, once an NI backend writes all packets comprising a message in their corresponding receive buffer slots, it creates a special message completion packet and forwards it to the NI dispatcher over the chip's default on-chip interconnect. Once the NI dispatcher receives the message completion packet, it enqueues the information in its shared CQ, from which point on the dispatch mechanism is the same as described above for the 4×4 queuing system. Because all the incoming messages are collected in a single queue, the NI dispatcher can dispatch load to *all* 16 cores, in contrast to the 4×4 implementation, which limited load dispatch within four separate dispatch groups. The ability of the NI dispatcher to dispatch load to all 16 cores is the key characteristic that enables the transition from a 4×4 to a 1×16 queuing system.

Having a single NI dispatcher introduces a point of centralization that eschews synchronization, but raises scalability concerns. However, for modern server processor core counts and a simple dispatch policy, the required dispatch throughput should be easily sustainable by a single centralized hardware unit, while the additional latency overhead of indirection (i.e., from any NI backend to the NI dispatcher) is negligible. From the throughput perspective, even assuming an RPC service time as low as 500ns associated with a message's reception, that translates to a requirement of a message dispatch every ~ 31 ns or ~ 8 ns for a 16-core and a 64-core chip, respectively. Both dispatch frequencies are modest enough for a single hardware dispatch component to handle. Latency-wise, the indirection from any NI backend to the NI dispatcher would cost just a couple of on-chip interconnect hops, adding just a few nanoseconds to the end-to-end message delivery latency. In conclusion, the centralized dispatch approach seems to be sufficiently flexible for reasonable system sizes. In case of exotic system deployments where the above assumptions do

not hold, alternative dispatch options with limited flexibility, such as the 4×4 design introduced in Section 8.1.1, remain relevant solutions.

8.2 Methodology

In this section, we detail our methodology for evaluating our design’s effectiveness in balancing load transparently in hardware. We additionally evaluate the performance of our lightweight native messaging mechanism, and compare its performance and memory footprint to that of the emulated messaging mechanism offered by the baseline soNUMA protocol.

8.2.1 Load Balancing

System organization. We model a single 16-core chip with emulated remote ends, implementing soNUMA with a manycore NI_{split} design, as illustrated in Figure 8.1. We emulate a 200-node cluster, with remote nodes emulated by a traffic generator generating synthetic *send* requests following Poisson arrival rate of configurable lambda from randomly selected nodes of the emulated cluster. The traffic generator also generates synthetic replies to the modeled node’s outgoing requests. We model the 16-core chip using Flexus [174] cycle-accurate simulation, with the same configuration parameters as the ones detailed in Section 7.2.

Microbenchmark. We use a multithreaded microbenchmark that emulates different service time distributions, where each thread executes the following actions in a loop: (i) spins on its CQ, waiting for a new *send* request; (ii) when a new request arrives, emulates the execution of an RPC by spending processing time X , where X is provided by a service time distribution generator detailed below; (iii) when the artificial processing time has passed, generates a random response in a 512B buffer and enqueues a *send* request as a response to the incoming *send*; and (iv) issues a *receive* request corresponding to the processed *send* request, marking the end of the incoming request’s processing.

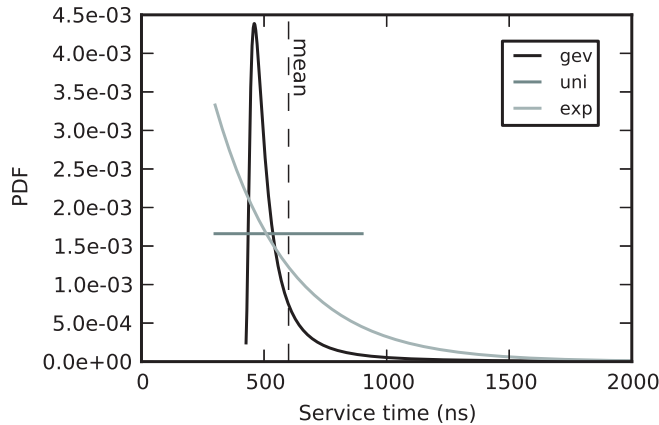


Figure 8.2 – Modeled service time distributions.

Emulated RPC processing time. We develop an RPC processing time generator that generates values that follow a selected distribution. We experiment with four different distributions: fixed, uniform, exponential, and generalized extreme value (GEV). Fixed represents the ideal case, where all requests always take the same processing time. Uniform represents a more challenging case, assuming that there is a continuous range of service times that are equally probable. Exponential and GEV represent more realistic expectations for service times, as they exhibit infrequent long tails, which can be caused by hard to predict events like TLB misses, interrupts, or page faults. GEV in particular resembles the service time distribution observed in web search engines [72]. The overall service time for an emulated RPC (i.e., the time a CPU core is occupied with servicing an incoming RPC request) is the sum of the processing time generated by the RPC processing time generator, plus the time required to write the response to the RPC in a local buffer and execute a *send* and a *receive* operation.

We focus on emulating the execution profile of latency-sensitive communication-intensive software layers with fine-grained requests, such as data stores. For instance, HERD [89] is a key-value store designed to service every lookup request with only two memory lookups, resulting in an average service time of ~ 300 ns. We use 300ns as the base service time and add an extra 300ns *on average*, following one of the aforementioned four service time distributions. Figure 8.2 illustrates the PDFs of the resulting RPC processing time.

Compared load balancing implementations. We first compare the performance of three hardware-based load balancing implementations, which correspond to three different queuing systems: 1×16 , 4×4 , and 16×1 . 1×16 , 4×4 correspond to the implementations described in Section 8.1. 16×1 is represented by a system with statically partitioned dataplanes, where every incoming message is statically assigned to a core at the time of its arrival, without any rebalancing possibility. Next, we compare the best-performing hardware load distribution implementation, 1×16 , to a software-based counterpart. In our software implementation, NIs enqueue incoming *send* requests into a single CQ from which all 16 threads pull requests in FIFO order. To minimize the thread synchronization overhead we implement an MCS queue-based lock [125] to poll for new requests.

We evaluate all configurations in terms of 99th percentile latency as a function of throughput. We measure each request’s latency as the time from the reception of a *send* message until the thread that ends up servicing the request posts a *receive* operation.

8.2.2 Messaging

System organization. We model two directly connected 16-core servers with the same configuration parameters for Flexus [174] cycle-accurate simulation as the ones detailed in Section 7.2.

Microbenchmark. We build a Netpipe [159] microbenchmark to evaluate the performance of the two messaging mechanisms, emulated and native, on soNUMA. The emulated messaging mechanism emulates *send/receive* operations over one-sided operations, as presented in Section 3.5.4. We use a ping-pong loop between two communicating servers to determine the end-to-end one-way latency of different message sizes. We use a single thread on each server; the source and destination buffers are LLC resident.

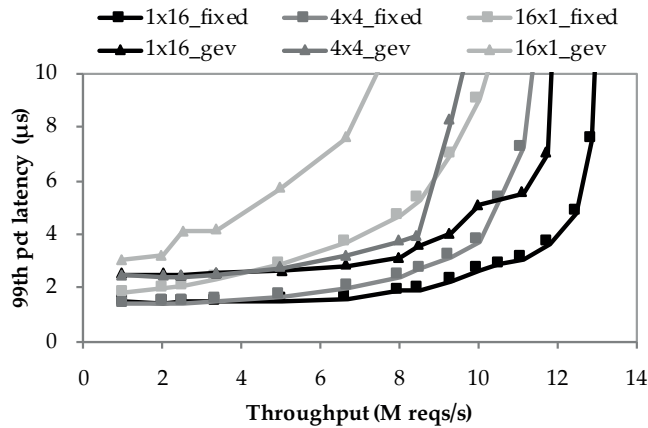


Figure 8.3 – Load balancing with three different queuing system implementations in hardware.

8.3 Evaluation

8.3.1 Load Balancing: Hardware Queuing Systems

Figure 8.3 compares the performance of our three evaluated hardware-based load balancing implementations. We set the Service Level Objective (SLO) in terms of acceptable tail latency at $10\mu\text{s}$, which corresponds to $10\times$ the average request service time we measured on the unloaded system. In favor of clarity, Figure 8.3 only shows two of the four evaluated RPC processing time distributions: fixed and GEV. These two distributions represent the ones that are the least and most affected by the underlying queuing configuration (see Section 5.1).

As predicted by Section 5.1’s theoretical queuing results, 1×16 consistently delivers the best performance, thanks to its superior flexibility in dynamically balancing load across all 16 available cores. For the same reason, 4×4 outperforms 16×1 , the only queuing configuration that offers no dynamic load balancing flexibility at all. For the fixed distribution, 1×16 delivers 16% and 29% higher throughput under SLO, as compared to 4×4 and 16×1 respectively. For GEV, this throughput improvement grows to 26% and 76%, respectively.

The flexibility to balance load from a single queue to multiple cores not only results in higher peak throughput under SLO, but also lower tail latency before reaching saturation load. 1×16 and 4×4 deliver roughly the same tail latency up to 4×4 ’s earlier saturation point, but the tail

latency difference of these two systems compared to 16×1 is significant, even for low load. For example, at a load of 5 million requests/s, which corresponds to only 50% of the 16×1 system's peak throughput with a fixed service time distribution, 16×1 's tail latency is almost $2 \times$ higher than 1×16 's.

In conclusion, integrated load balancing support in NI hardware can significantly improve system throughput under tight tail latency goals. Implementations that enable the full flexibility of dispatching incoming requests to all available cores (i.e., 1×16) deliver the best performance. However, even the performance of implementations with limited balancing flexibility, such as the evaluated 4×4 configuration, is competitive. As the implementation complexity of a true single-queue system incurs some additional design complexity, such limited-flexibility alternatives introduce viable options for system designers willing to sacrifice some performance in favor of simplicity.

8.3.2 Comparison to Queuing Model

The performance results in Section 8.3.1 qualitatively meet the expectations set by the queuing analysis presented in Section 5.1. We now quantitatively compare the obtained results to the ones expected from the purely theoretical models, to determine the performance gap between our implementations and the theoretical best.

To match the queuing system implementations to representative queuing models, we devise the following methodology. We measure the average service time S of our implementation; a part D of this service time is synthetically generated to follow the value distribution of choice (fixed, uniform, exponential, GEV) and the rest, $S-D$, is spent on the rest of the microbenchmark's code (e.g., event loop, execution of a *send* operation as an RPC response and a *receive* operation to free the RPC slot—see Section 8.2.1). We conservatively assume that this $S-D$ part of the service time follows a fixed distribution. Using discrete-event simulation, we model and evaluate the performance of theoretical queuing systems with a service time S , where:

Chapter 8. Tail-Aware Balancing of μ S-Scale RPCs

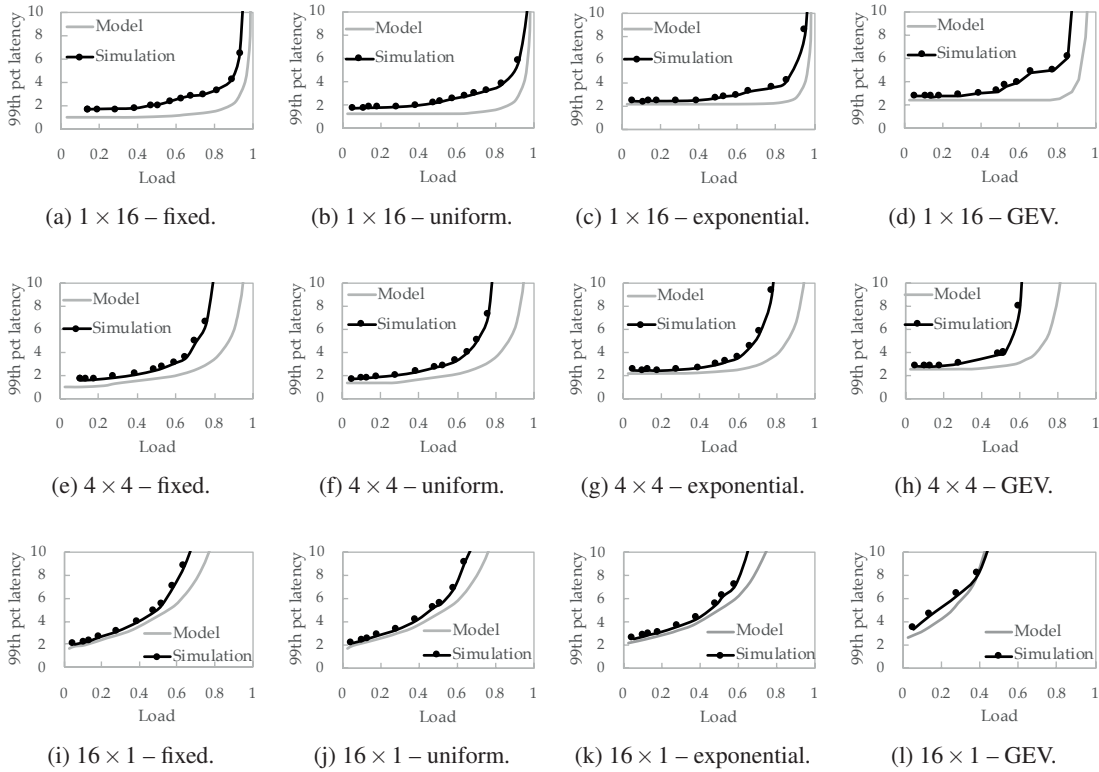


Figure 8.4 – Performance of three hardware load balancing implementations (1×16 , 4×4 , 16×1) as compared to a theoretical queuing model, for four service time distributions: fixed, uniform, exponential, GEV. Tail latency shown as a multiple of the average service time S .

- $\frac{D}{S}$ of the service time follows a certain distribution (fixed, uniform, exponential, GEV).
- $\frac{S-D}{S}$ of the service time is fixed.

Figure 8.4 compares our implementation to the theoretical queuing model. The graphs show the 99th percentile latency as a function of system load for three different queuing configurations (1×16 , 4×4 , 16×1) and four different distributions for the D part of the service time. We set the SLO in terms of 99th percentile response latency to $10 \times$ the average service time S . For the 1×16 and 16×1 configurations (Figures 8.4a–8.4d and 8.4i–8.4l, respectively), our implementations are as close as 3% to the model, and within 15% in all cases. For the 4×4 configuration (Figures 8.4e–8.4h), the performance gap between the implementation and the model is larger: 16% for the fixed, uniform, and exponential distributions, and 26%—the largest difference across the board—in the case of GEV. We attribute the gap between the implementations and the model to

contention that emerges under high load in the implemented systems, which is not captured by the model. Furthermore, assuming a fixed service time distribution for the $S-D$ part of the service time is an optimistic simplifying assumption: modeling variability for this latency component would have a detrimental effect on the model's achieved performance, thus shrinking its gap from the implemented systems.

1×16 , being the best-performing configuration, is the system of key interest. As compared to the other two configurations, 1×16 demonstrates the smallest performance gap with the model, ranging from 3% to a maximum of 16%. The takeaway is that our implementation leaves no significant room for improvement; the design decisions of centralizing dispatch and maintaining zero-depth request queues at the cores do not introduce performance concerns.

8.3.3 Hardware Versus Software Load Balancing

Figure 8.5 compares the performance of our hardware-based load balancing implementation to a software implementation, both of which implement the same theoretically optimal queuing system (i.e., 1×16). The key difference between the hardware and software implementation is the management of load dispatch from the shared CQ to a core. In the case of the software implementation, a synchronization mechanism, in this case an MCS lock, is necessary for cores to atomically pull incoming requests from the queue. In contrast, our hardware load balancing mechanism does not incur any synchronization costs, as the NI itself dispatches requests to available cores.

The software implementation is directly competitive to the hardware implementation for low system load, but because of the single lock all cores contend on, it reaches saturation significantly faster. As a result, our hardware implementation delivers 2.3–2.7 \times higher throughput under SLO, depending on the request processing time distribution. A comparison between Figures 8.3 and 8.5 reveals that the 1×16 software implementation is not only inferior to the 1×16 hardware implementation, but all other evaluated hardware implementations as well. The fact that even the 16×1 hardware implementation is superior to the software 1×16 implementation indicates

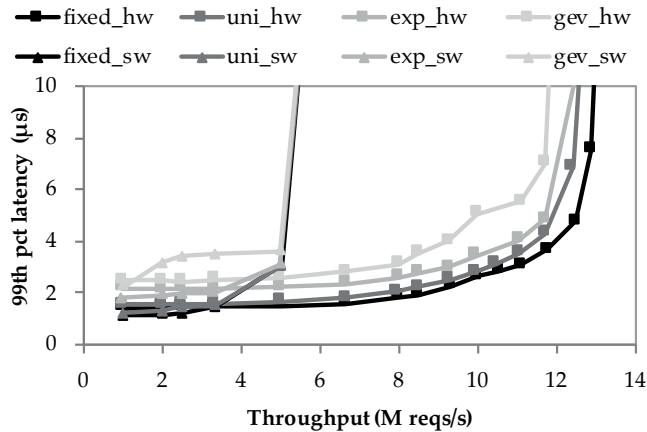


Figure 8.5 – Load balancing performance of a 1×16 queuing system: Hardware vs. software implementation.

that the benefit from providing load dispatch flexibility does not offset the synchronization cost associated with it. That is a direct consequence of the very short-lived nature of the RPCs we focus on.

8.3.4 Messaging Performance

Figure 8.6 shows the one-way messaging latency for the two evaluated messaging mechanisms: native and emulated over one-sided operations. As emulated messaging offers two variations that represent different tradeoff points (push versus pull), we evaluate both to experimentally determine the optimal boundary between the two mechanisms by setting the push to pull switching threshold to 0 and ∞ in two separate runs.

For small transfers, the push model outperforms the pull model, as the cost of an additional network roundtrip outweighs the packetization overhead. The opposite is true for large transfers. We find 1024 bytes to be the smallest transfer size for which the pull model outperforms the push model. A third curve (*Emulated, thr=1KB*) shows the performance of emulated messaging with the threshold set to 1KB, representing the best performance of emulated messaging by combining the two models at their best performing transfer size ranges.

The latency of native messaging is on par with emulated messaging for sub-cache-block transfers

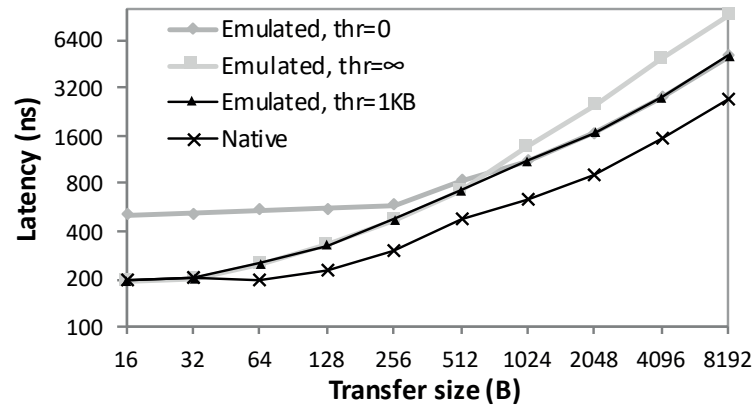


Figure 8.6 – Messaging latency.

(~ 200 ns). Cache-block-sized transfers are 20% faster with native messaging, because 64B transfers require two network packets for the emulated messaging mechanism with the push model (in addition to the packetization overhead). The latency difference between native and emulated messaging grows to up to 45% for KB-sized transfers. The sources of this difference are two. The primary latency overhead for emulated messaging is an additional *memcpy* of the data to be moved to a *rendezvous* location, where it has to remain until the receiver reads it using a remote read operation. Another latency overhead is the additional network roundtrip that the pull model's *rendezvous* technique introduces, which, in this particular experimental setup of two directly connected nodes, is of secondary importance.

8.3.5 Messaging Memory Requirements

Our native messaging mechanism heavily relies on in-memory data structures to keep hardware additions modest. Here, we analytically estimate the memory requirements for these structures, and compare them to those of emulated messaging.

Table 8.1 shows the number of send/receive buffer slots required per target node for different message sizes, with the goal of sustaining a messaging bandwidth of 100Gbps per node pair. The number of slots required is computed via straightforward application of Little's Law, using the target bandwidth and the measured roundtrip latency of messages of specific size as inputs.

| Message size (B) | Latency (ns) | Required slots |
|------------------|--------------|----------------|
| 64B | 397 | 78 |
| 256B | 500 | 25 |
| 1024B | 827 | 11 |
| 4096B | 1716 | 6 |

Table 8.1 – Estimation of buffering slots required for a peak target messaging throughput of 100Gbps, as a function of message size.

After computing the number of required slots per participating node, we use it to compute the messaging mechanism’s overall memory requirements.

Figure 8.7 shows the total memory footprint of our messaging mechanism as a function of the number of participating nodes and the maximum supported message size. Note the logarithmic scale on the y axis. The total memory footprint is broken down into three components: (i) the send buffers; (ii) the receive buffers; and (iii) the counters coupled with the receive buffers to keep track of incoming packets per message. Of those three components, only (ii) is required for the emulated messaging mechanism. The memory footprint for each component is trivially computed as follows:

- *Send buffers* = $32B \times num_slots \times num_nodes$
- *Receive buffers* = $max_msg_size \times num_slots \times num_nodes$
- *Receive counters* = $64B \times num_slots \times num_nodes$

The first observation from Figure 8.7 is that even in the case a of maximum message size of 4KB and a cluster size of 1024 nodes, the total memory required per node for the messaging buffers is less than 25MB. The memory requirement is acceptable even for more challenging use cases. For instance, assuming an application that requires native messaging support at peak throughput for both large (e.g., 4KB) and small (64B) messages, many receive slots of a large size should be allocated (78 slots of 4KB in this case). Even in this case, the aggregate per-node memory

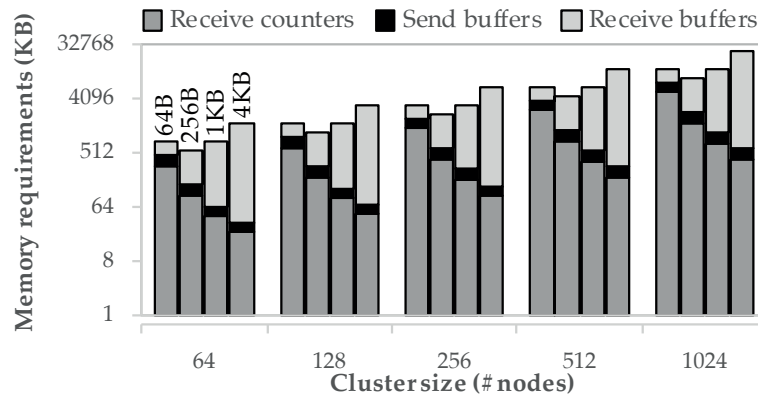


Figure 8.7 – Total memory footprint of messaging mechanism for different cluster and message sizes (64B, 256B, 1KB, 4KB).

footprint for a 1024-node deployment measures up to 319MB. An alternative approach would be the deployment of more than one messaging contexts to better accommodate different message sizes. Such an approach is reminiscent of the buddy memory allocation technique, used in various aspects of memory management. In the previous 1024-node example, allocating two message contexts instead of a single one—one for 64B messages and one for 4KB messages—reduces the aggregate memory footprint from 319MB to 37MB.

The second observation from Figure 8.7 is that the additional memory footprint overhead of native messaging as compared to emulated messaging is modest. In the case of small messages, even though the relative overhead is significant ($2.5\times$ memory footprint increase for 64B messages and $1.4\times$ for 256B), the absolute values are negligible as compared to the available memory capacity in modern servers. For 1KB and 4KB messages, the relative memory footprint overhead of native versus emulated messaging drops to 9% and 2% respectively.

8.4 Chapter Summary

In this chapter, we evaluated a proof-of-concept dynamic load balancing mechanism integrated in the NI logic. We proposed two implementations that represent different appealing points in the design plane of implementation complexity versus load balancing flexibility. For the most challenging RPCs with μs -scale execution times, both implementations significantly outperform

Chapter 8. Tail-Aware Balancing of μ S-Scale RPCs

pre-existing approaches to balancing incoming network load. Our study is limited to a single synthetic application with emulated service time distributions, but our findings promising, motivating further research efforts on NI-integrated load balancing policies.

Related and Future Work **Part III**

9 Related Work

In this chapter, we discuss prior work related to the topics this thesis touched upon. Section 9.1 discusses work related to the soNUMA architecture and programming model, and network interface integration; Sections 9.2 and 9.3 discuss topics related to SABRes and on-chip load balancing of incoming network load, respectively.

9.1 soNUMA and NI Integration

9.1.1 Partitioned Global Address Space

soNUMA exposes the abstraction of a partitioned global virtual address space, but offers a different interface to access local and remote memory. The programmer has to be aware of data location and distinguish between local and remote. The notion of such an address space is not new; the parallel programming model of PGAS (Parallel Global Address Space) has been around for more than a decade and emerged as a good fit for distributed shared memory architectures. PGAS relies on compiler and language support to provide the abstraction of a shared address space on top of non-coherent, distributed memory [36]. Languages such as Unified Parallel C [36] and Titanium [176] require the programmer to reason about data partitioning and be aware of data structure non-uniformity. However, the compiler frees the programmer from the burden of ensuring the coherence of the global address space by automatically converting accesses to

remote portions into one-sided remote memory operations that correspond to soNUMA's own primitives. PGAS also provides explicit asynchronous remote data operations [23], which also easily map onto soNUMA's asynchronous library primitives. The efficiency of soNUMA remote memory access primitives would allow PGAS implementations to operate faster.

9.1.2 Software Distributed Shared Memory

Unlike the memory hierarchies exposed by PGAS and soNUMA, software distributed shared memory (DSM) systems provide global coherence. Systems such as IVY [99], Munin [27] and Threadmarks [10] expose a global *coherent* virtual address space and rely on OS mechanisms to “fault in” pages from remote memory on access and propagate changes back, typically using relaxed memory models for performance reasons. Alternatively, software DSM can be implemented within a hypervisor to create a cache-coherent global guest-physical address space [33], or entirely in user-space via binary translation [150]. While both software DSM and soNUMA operate at the virtual memory level, the former typically operates at the page level, while the latter targets fine-grained accesses. Shasta [150] and Blizzard [151] offer fine-grain DSM through code instrumentation and hardware assistance, respectively, but in both cases with non-negligible software overheads. soNUMA's design for fine-grained access allows accessing a single cache block from remote memory within a small factor ($\sim 3\times$) over a local memory access.

9.1.3 Cache-Coherent NUMA

In the 90s, ccNUMA designs emerged as a promising approach to scale shared-memory multiprocessor performance by interconnecting thin symmetric multiprocessor server nodes with a low-latency and high-bandwidth network. These machines provided a globally coherent distributed memory abstraction to applications and the operating system. Examples include academic prototypes such as Alewife [7], Dash [98], FLASH [74, 96], Fugu [121] and Typhoon [147], and commercial products such as SGI Origin [97] and Sun Wildfire [60, 70]. While most targeted

coherence at cache block granularity, machines with programmable controllers also enabled support for bulk transfers [59, 74] broken down into a stream of multiple cache blocks. Today’s multi-socket servers are cache-coherent NUMA machines with a few multicore sockets that use either Intel’s QPI or AMD’s HTX technology.

soNUMA shares the non-uniform aspect of memory with these designs and leverages the lower layers of the ccNUMA protocols (routing and link), but does not enforce global cache coherence. soNUMA uses a stateless protocol, whereas ccNUMA requires some global state such as directories to ensure coherence, which limits its scalability. The ccNUMA designs provide a global physical address space, allowing conventional single-image operating systems to run on top. The single-image view, however, makes the system less resilient to faults [32]. In contrast, soNUMA exposes the abstraction of global virtual address spaces on top of multiple operating system instances, one per coherence domain, thus maintaining the scalability trait of scale-out architectures. soNUMA leverages the local on-chip coherence within each small coherence domain to accelerate the exchange of data and metadata between the cores and its NI that executes the communication protocol, the RMC. We find that moving to *fat* manycore chips has major implications on NI placement for both fine-grained and bulk transfers, which were previously not explored.

9.1.4 User-Level Messaging

User-level messaging eliminates the overheads of kernel transitions by exposing communication directly to applications. Hybrid ccNUMA designs such as FLASH [96], Alewife [7], FUGU [121], and Typhoon [59] provide architectural support for user-level messaging in conjunction with cache-coherent memory. FLASH and Typhoon feature a programmable processor at the NI, which executes software handlers in response to a message’s reception, implementing a form of Active Messages [169]. Alewife features memory-mapped network registers, where messages are queued waiting for the CPU to execute a software handler, which either directly loads the message contents or sets up a DMA transfer from the network registers to the host’s memory. The messaging implementation in FUGU is based on Alewife’s mechanism. In contrast, soNUMA’s

original protocol design allows for an efficient implementation of message passing entirely in software using one-sided remote memory operations. Even in the native messaging mechanism we introduced in Chapter 5, the NI is not involved in any handler execution. In contrast, all message destinations are fixed in advance (setup of a *messaging domain* as introduced in Section 5.4) and registered with the NI. The NI directly writes messages into these destinations allocated in the destination host's memory, and creates a message arrival notification for the CPU once all of a message's packets have arrived.

Our messaging mechanism bears semblance to SHRIMP's user-level messaging [21]. SHRIMP's NI maps physical memory ranges of two nodes to each other, and sending a message from node A to node B boils down to the NI copying data from node A's memory region to the corresponding mapped memory location on node B. This roughly corresponds to our messaging mechanism's memory allocation policy, where each set of send buffer slots on the source node has a one-to-one correspondence to receive buffer slots on the destination node.

9.1.5 Remote Memory Access

Hardware support for direct remote memory access has been commercialized in the past in Cray supercomputers [94, 152]. Cray T3D/T3E implemented *put* and *get* instructions that applications could use to directly access a global memory pool. RDMA technology is the modern incarnation of remote memory access and is available in commodity clusters equipped with host channel adapters such as Mellanox ConnectX-series [123] that connect into InfiniBand or Converged Ethernet switched fabrics [79]. To reduce complexity and enable SoC integration, soNUMA only provides a minimal subset of RDMA operations; in particular, it does not support reliable connections, as they require keeping per-connection state in the adapter.

Unlike the Cray machines, RDMA and soNUMA do not rely on a load/store interface to access remote memory, but deploy explicit remote memory access commands written into memory-mapped queues (QPs). For PCIe-attached RDMA adapters, transferring information through the QPs between the CPU and adapter card is costly mainly because of the DMA operations over the

PCIe interface. In contrast, for properly integrated controllers as in the case of soNUMA, the overhead of QP-based communication is a negligible fraction of the end-to-end latency. Therefore, extending commercial CPUs with a special load/store interface for direct remote memory access is an unnecessary hardware cost and complexity.

SHRIMP [21] uses a specialized NI, which creates a mapping between physical memory regions of different machines and automatically keeps their contents synchronized by performing direct remote writes. Cashmere [162] leverages DEC's Memory Channel [66], a remote-write network, to implement a software DSM. Unlike SHRIMP and Cashmere, soNUMA also allows direct reads from remote memory.

9.1.6 Coherent NI Integration

One advantage of soNUMA over prior proposals on fast remote memory access is the tight integration of the NI into its local CPU's coherence domain. The advantage of such an approach was previously demonstrated in Coherent Network Interfaces [129], which leverage the coherence mechanism to achieve low-latency communication of the NI with the processors, using cacheable work queues. That work, however, did not consider NI integration with large manycore chips where on-chip data transitions represent a significant fraction of the end-to-end latency.

More recent work showcases the advantage of integration, but in the context of kernel-level TCP/IP optimizations, such as a zero-copy receive [20, 78, 102]. Our RMC is fully integrated into the local cache coherence hierarchy and does not depend on local DMA operations. Furthermore, these efforts on NI optimization were focused on traditional networking and were thus inevitably engaged with expensive network protocol processing. In contrast, we focus on specialized NIs for RDMA-like communication, in which execution of remote operations only requires low-cost user-level interactions with memory-mapped queues and minimal protocol processing. As the base assumptions about the protocols are different, so are the main bottlenecks and key design considerations for each type of NI.

While the soNUMA protocol is similar in style to RDMA, it represents a much simplified version

of it, making the integration of the RMC into the local cache-coherence domain practical. Such integration provides substantial latency benefits, not only because the coherence mechanisms enable the fastest bouncing of QP entries between the cores and the RMC, but also because all control data structures, such as the QPs and page tables, can be kept in large on-chip caches shared with the CPU. Coherent integration also allows soNUMA to provide *global atomicity* by implementing atomic operations within a node’s cache hierarchy; global atomicity cannot be delivered by modern RDMA controllers (i.e., atomicity of concurrent writes by a CPU and its RDMA NIC to the same cache block is not guaranteed). For example, that limitation has led the designers of DrTM [171] to perform *all* writes (local and remote) to data objects in local memory through the local RDMA NIC. Coherent integration can also be leveraged to introduce new operations with stronger semantics and richer functionality. An example of such an extension is the SABRe operation introduced in Chapter 4 of this thesis. Our NI hardware support for SABRes, LightSABRe, leverages the on-chip coherence mechanism to guarantee atomic reads of data objects straddling multiple cache blocks, extending atomicity of remote memory access beyond the limit—for both RDMA and baseline soNUMA—of a single cache line.

9.2 Hardware Support for Atomic Remote Object Reads

Chapter 4 introduced SABRe, a novel one-sided operation with the semantics of an atomic remote object read. The implemented NI hardware extension that supports SABRes, LightSABRe, relies on the protocol controller’s (e.g., soNUMA’s RMC) coherent integration and on the software contract of a standardized object layout in memory to simplify hardware requirements.

9.2.1 Hardware-Software Contract

The hardware simplicity of LightSABRe stems from the insight that objects in data stores are structured, and this software-provided guarantee can be harnessed. A similar observation has been made and leveraged before in the context of HTM: object-aware HTM relies on the organization of data as software objects to tackle the capacity limitations of traditional HTM [95].

9.2.2 Atomic Chunk Operations

A large body of work has been done in providing atomic access to memory chunks in shared-memory architectures [22, 30, 31, 49, 71, 145, 146, 173]. While these mechanisms can be used in a distributed memory environment to provide SABRes, they deliver broader functionality than simple atomic range reads at the cost of increased hardware complexity and intrusive hardware modifications. In contrast, LightSABRe only requires simple and contained extensions to the integrated network protocol controller, without any further chip modifications (e.g., caches, cache and coherence controllers); thus, integration into commercial chips with conventional block-based coherence protocols is more practical.

9.2.3 Memory Subsystem Support

Tagged memory has been extensively investigated in the context of security and data integrity [37, 41, 163, 178]. Variations of such architectures can also be found on real machines, such as the Soviet Elbrus processors in the 70s [106], the J-machine in the 90s [160], and Oracle's more recent M7 chip [8]. The hardware tags embedded in memory can be leveraged as a mechanism for concurrency control, e.g., as a hardware implementation of per-cache-line versions. The *destination-side* protocol controller could use these versions to identify atomicity violations while servicing a SABRe. While functionally similar to its software counterpart, such a hardware mechanism would be significantly more efficient, with the added benefit of leaving the data store's layout unmodified.

HICAMP [35] effectively provides snapshot isolation for all software objects through hardware multiversioning, thus preventing read-write conflicts. Integration of protocol controllers for one-sided operations with HICAMP is an interesting case where SABRes are provided by default, without any special hardware extensions.

9.2.4 SABRe: One-Sided Operation or RPC?

In the broader sense of RPCs, extending the network protocol controller with one-sided operations with stronger semantics (such as SABRes) and the addition of destination-side accelerators is semantically as much of an RPC mechanism as it is a one-sided operation. LightSABRe can be perceived as a simple fixed-functionality hardware RPC unit that reaps all the benefits of one-sided operations, and addresses the shortcomings of software RPCs at the price of limited flexibility: minimized latency for atomic object reads from remote memory and massive MLP, which is, in general, unattainable with RPCs, as their concurrency is fundamentally limited by the number of available cores.

9.2.5 Destination-Side Concurrency Control

In Section 4.2.1, we discussed different approaches to concurrency control in distributed systems, reaching the conclusion that *destination-side* concurrency control mechanisms are superior to *source-side*. Microsoft FaRM's original design [53] implements a *source-side* concurrency control to achieve atomic remote object reads. Every cache line of an object is enhanced with a version. Once a remote object is read using a one-sided read operation, the CPU strips off these per-cache line versions and compares them against each other to verify object read atomicity. This operation incurs a considerable overhead in terms of latency and wasted CPU cycles, which motivated the introduction of the new SABRe operation and its corresponding NI extensions to provide hardware support for atomic remote object reads.

Soon after we proposed SABRe, Microsoft introduced a similar operation by leveraging its Catapult architecture [28]. In the Catapult architecture, an FPGA is attached in front of every server's NIC as a "bump-in-the-wire", which can be used as a programmable accelerator for inter-server communication, network flow transformations, and application code. Microsoft used Catapult's FPGA to accelerate FaRM's atomic object read mechanism by exposing an interface for a new "atomic object read operation" (i.e., a *SABRe*) [51]. While the operation's semantics are essentially those of a SABRe, their implementation is significantly different. The FPGA strips

off the read object's per-cache-line versions and verifies object read atomicity at the destination server. Switching to this destination-side concurrency control mechanism alleviates several of the original source-side atomicity check mechanism: it removes the CPU overhead of version stripping at the request's source, and reduces bandwidth usage (no versions or inconsistently read objects are put on the wire). However, the object store remains modified, as objects are still stored with embedded per-cache-line versions, negatively affecting the performance of all local read and write operations. In contrast, LightSABRe does not require any modifications of the object store.

9.3 Load Balancing

9.3.1 Load Distribution and Imbalance

The emergence of manycore CPUs and growing networking capabilities have necessitated mechanisms to efficiently spread network load (both network-layer processing for TCP/IP and application-layer processing) to all available cores. Most modern NICs provide such support in the form of Receive Side Scaling (RSS) [126] or Flow Director [82], which split the incoming network load into multiple queues, each of which can be privately assigned to a core. Systems like IX [19] and MICA [103] leverage these mechanisms to significantly boost their throughput under tail latency constraints. However, the disadvantage of RSS/Flow Director is that they blindly spread load across multiple receive queues based on specific network packet header fields, being oblivious to load imbalances that may arise at the CPU level, which can significantly hurt tail latency. MICA has load-imbalance-aware optimizations (CREW mode), which however are handled at the software level and requires an application restart to take effect.

ZygOS [144] is a recent effort to address the shortcomings of statically partitioned dataplanes like IX, which suffer from increased tail latencies when load imbalance across dataplanes arises. ZygOS introduces an intermediate shuffling layer for network messages to enable CPU threads to perform *work stealing* whenever load imbalance across their incoming request queues arises. In effect, ZygOS moves from the *shared-nothing* architecture of partitioned dataplanes to a *shared-something* architecture to strike a tradeoff: pay the price of infrequent synchronization

to enable load rebalancing. For short-lived tasks, with service times on the order of a few tens of microseconds, ZygOS improves throughput under tight tail latency goals by up to 25% as compared to the state-of-the-art IX dataplane. However, because of the added synchronization overhead added by the shuffling layer, there is still significant room for improvement, which is inversely proportional to the target application's service time. For service times of $25\mu\text{s}$ and $10\mu\text{s}$, ZygOS achieves 88% and 75%, respectively, of an ideal queuing system's throughput under tail latency constraints. Its efficiency is expected to drop further for service times of only a couple of microseconds, which are common for simple—yet ubiquitous—software layers such as distributed object stores (e.g., Memcached). Our proposed dynamic load balancing design, integrated in the NI logic, takes advantage of low-latency core-NI interactions enabled by on-chip NI integration to monitor on-chip load in real time and dynamically distribute load to cores, offering strong resilience to load imbalance without any added synchronization overhead. The key conceptual difference from ZygOS is that our approach does not attempt to approach the performance of a theoretically better queuing system through a secondary mechanism that rebalances load, but rather directly implements a queuing system superior to partitioned dataplanes, without any associated synchronization overheads.

9.3.2 Load Balancing Policies

The merits of controlling message dispatch to cores at the NI to eschew software synchronization and balance load for throughput gains have been identified in the past, mainly in the context of parallel protocol handler execution for DSMs [61, 143]. Programming abstractions such as the Parallel Dispatch Queue [61] can be deployed as smarter, programmable load balancing implementations at our NI design's penultimate pipeline stage, which takes the message dispatch decisions.

A large body of work has been investigating load balancing in the context of web services running on datacenters. In such a setting, the problem of distributing incoming load from one or more dispatching units to serving units takes the form of distributing load from a number of frontend servers to multiple backend servers. The high-level problem is the same as distributing load

from one or more NIs to multiple CPU cores on chip; however, the challenges and scale are radically different, leading to dissimilar solutions. The key difference between datacenter-scale and on-chip load balancing is the communication latency between the dispatcher (frontend servers for datacenter, NI for on-chip) and the serving units (backend servers for datacenter, CPU cores for on-chip). In the former case, communication incurs high latency and should be used sparingly, hence, from a performance perspective, it is not possible for the dispatcher to always wait for a serving unit to become free to dispatch a new request.

Examples of load distribution algorithms in datacenter-scale systems are Join-Shortest-Queue (JSQ) [69], Power-of- d (SQ(d)) [25], and Join-Idle-Queue (JIQ) [120]. JSQ is a simple greedy dispatch policy, based on which the frontend server sends every new incoming request to the backend server with the smallest queue of waiting requests. Despite its simplicity, JSQ has been shown to outperform algorithms with higher complexity. However, it does not scale well with a large number of dispatching frontends, as maintaining globally consistent state of every backend's load is expensive. SQ(d) is a more scalable approach that better suits datacenter-scale deployments. At each request arrival, the frontend samples d backends, obtains the number of queued requests at each of them, and dispatches the new request to the backend with the least number of queued requests among the d sampled. While more scalable, the main drawback of SQ(d) is that frontend-backend communication is on the critical path of a request's assignment, significantly contributing to the request's end-to-end response time. Finally, JIQ strikes a balance between JSQ and SQ(d) by decoupling discovery of lightly loaded backend servers from the assignment of incoming requests to backend servers. As soon as a backend server becomes idle, it informs a subset of the frontends of its availability, an operation that occurs off the critical path. The JIQ algorithm is designed to strike a balance between overloading and underutilizing each of the backend servers as they advertise idle work slots, by properly balancing the advertisement of available backends to frontends.

Because on-chip core-NI communication latencies are 1–2 orders of magnitude lower than the service time of the fastest RPCs, the challenges of (i) overlapping communication latency with useful computation, (ii) without resulting in unevenly distributed load across serving units, and

(iii) without excessive communication between the dispatchers and serving units, all of which are significant considerations for large-scale distributed systems, are of minor importance in the context of on-chip load distribution across cores. It is therefore possible to achieve load balancing behavior approaching that of the ideal single-queue system by deferring request dispatch from the dispatcher (NI) to the serving units (cores) until a core becomes free, as demonstrated by our results.

9.3.3 Programmable NIs

Adding programmable compute capabilities to NIs to offload high-level functionality closer to the network is an old idea that has recently seen rekindled interest. For example, FLASH [96] and Typhoon [147] in the 90s featured fully programmable processors at the network interface, enabling custom handler execution upon the reception of a network message. The renewed interest in programmable NIs, marketed as "SmartNICs", is partially motivated by recent technological trends leading to the stagnation of general-purpose logic. The goal of such NIs is the acceleration of networking or even high-level application functionality, to reduce CPU load.

The programmable logic offered by modern NIs comes in the form of either general-purpose cores or FPGA logic. Examples of the former are Mellanox's BlueField, Cavium's LiquidIO, and Netronome's Agilio-CX. Examples of the latter include Mellanox's InnovaFlex, the NetFPGA project [179], and Microsoft's widely successful Catapult project [28]. Another relevant recent line of work that does not strictly fall into any of the above two categories is FlexNIC [92, 93]. FlexNIC draws inspiration from SDN switches, deploying a reconfigurable match table (RMT) processing model, which processes packets through a sequence of match plus action stages. The RMT model has limitations, but its simplicity enables line-rate processing.

The programmable logic offered by these SmartNICs could be leveraged to implement smarter load distribution/balancing decisions than RSS or Flow Director. For example, assuming a key-value store application, rules could be installed in the NI's logic to steer requests to cores based on the content they wish to access. However, the flexibility of existing SmartNICs is

limited compared to the fully integrated NIs we focused on in this thesis, as the CPU logic and the NI logic are segregated by the high-latency PCIe interface. The NI-integrated dynamic load balancing mechanism we proposed relies on nanosecond-scale interaction between the NI and CPU logic, which is only attainable through tight NI integration and CPU-NI co-design.

10 Future Research Directions

10.1 Hardware Heterogeneity in Datacenters

Technological scaling trends contrast starkly with the exponential growth in demand for computational resources. With Dennard scaling and Moore's Law grinding to a halt, the continuous performance improvements, conventionally delivered by general-purpose logic, have stagnated. At the same time, new die-stacked memory devices deliver significant bandwidth improvements, while the InfiniBand Trade Association's roadmap predicts a quadrupling of network bandwidth in the foreseeable future [81]. To illustrate, coupling a modern top-of-the-line manycore server with an InfiniBand network adapte results in a processing budget of fewer than a thousand CPU cycles per packet. By solely relying on general-purpose logic, this imbalance will only become more pronounced.

We need major innovation in hardware/software system design to keep improving datacenter performance and efficiency. The former is a prerequisite for the continuously better and richer services we have become accustomed to. The latter is crucial for environmental reasons: modern datacenters already consume three percent of global electricity and are responsible for two percent of total greenhouse gas emissions. These implications herald an era where logic specialization and heterogeneity are essential features in the design of next-generation datacenter systems.

Chapter 10. Future Research Directions

It is time to depart from the long-established, “one-size-fits-all” CPU-centric view of computing and decentralize functionality, by distributing it to heterogeneous specialized components. As the end of silicon scaling signifies the end of free performance improvements, pushing the envelope of computing requires tailoring hardware to types of computation executed in the most apposite location. Such a major transition requires a deep rethink of the whole system stack, including algorithms, software and hardware architectures. While layering is a fundamental principle we rely upon to build complex systems, most layers and interfaces in modern systems have been established decades ago. With rapidly changing technologies, some of these abstractions are becoming opaque and detrimental to performance. It is, therefore, important to take cross-layer approaches (i.e., view the system stack holistically rather than focus on a single layer) and revisit long-standing assumptions, especially under the upcoming reality of larger datacenters, comprising a broad diversity of heterogeneous compute units.

The transition from general-purpose computing to the era of heterogeneity is an exciting topic to work on, particularly in the context of datacenters—the largest computing systems ever built. Such a transition requires rethinking long-established layers, interfaces, and abstractions to best accommodate new technologies and computing paradigms, and can affect the whole system stack up to the application layer. It is time for system designers to take a step back from the prevalent CPU-centric view of computing systems, and consider network-centric and memory-centric approaches. Examples of such alternative approaches are the present thesis, for the former, and systems designed with the goal of providing efficient near-memory processing capabilities, for the latter (e.g., [55]).

Most datacenter services today are solely handled by CPUs; in some cases, they are specially restructured to also make use of specific accelerators like FPGAs, GPUs or TPUs [88], with the CPU still playing the role of the central coordinator. In a future with servers comprising a multitude of heterogeneous units, it will be critical to remove the CPU from the critical path and allow all units to tap into the network and directly initiate and receive network messages. Enabling heterogeneous units to directly expose remotely invocable services offers (i) performance scalability; (ii) latency improvements, especially critical for the most communication-intensive software

layers relying on fine-grained remote memory access; and (iii) improved tail latency robustness, as a large fraction of the major sources of long tail effects are attributed to unpredictable software events (e.g., interrupts, context switches, garbage collection, etc.). The latter largely depends on the software layers deployed on each of the heterogeneous units, which is a major open research question.

Extreme heterogeneity comes with management complexity that hurts usability; it is crucial to ease programmability of such systems through interface unification, an endeavor that engenders many open questions. What interface should accelerators expose to the network—is it a specific device, a service, or capability to manipulate specific data? What operating system and hardware modifications are required to completely remove the CPU from the critical path but preserve all the important operating system guarantees (e.g., isolation and protection)? These questions introduce an exciting, broad and challenging research direction that requires cross-layer thinking, and successfully addressing them will play a key role in the success of future network-centric systems harnessing diverse hardware heterogeneity.

From the hardware organization perspective, the novel scalable on-chip NI design introduced in Chapter 6 represents an appealing starting point. NI backends can seamlessly handle data movement between the homogeneous network and the server’s memory hierarchy, while each heterogeneous compute unit can feature its own NI frontend, responsible for the direct initiation and completion of network transfers and tailored to the compute unit’s unique characteristics.

10.2 Dynamic Load Balancing Extensions

In Chapter 5 we introduced a proof-of-concept design of a mechanism for dynamic load balancing, integrated in the NI logic. While the current simple implementation demonstrates the opportunity for dynamic load balancing, there is significant room for improvement in terms of utility, flexibility, and system integration.

10.2.1 Advanced Load Balancing

In its current form, our dynamic load dispatch mechanism only considers homogeneous compute units (i.e., same CPU cores) and a single application running on all compute units, while it distributes incoming network messages to all units, solely based on their current load. To enable deployment of such a load dispatch mechanism on a real system, these limitations should be addressed.

First, it is essential to support multi-tenancy. Multiple applications will be concurrently executing on the same server, each of which will expose its own set of RPCs that can be invoked from the network. The NI should be able to distinguish among them, and match incoming RPCs to computation units that are in a state capable of servicing every given RPC (e.g., in the context of conventional multicore CPUs, distinguish the core currently running a thread that belongs to the right process). Achieving that would require OS involvement: exposing the NI's load dispatch logic to the OS and making it part of the thread scheduling process (e.g., notify the NI of a thread's migration).

Second, it is worth considering smarter load distribution policies than just the currently implemented "first-available" policy. For example, an alternative load distribution policy could be data-locality-aware. Such a policy would be a great fit for software stacks like the MICA key-value store [103], which offers a shared-nothing data access mode (EREW—Exclusive Reads/Exclusive Writes) where each fraction of the dataset is only accessible by a single CPU core.

Finally, in the face of emerging intra-server heterogeneity (see Section 10.1), it is important to extend the load dispatch mechanism to extend beyond the notion of service logic running on homogeneous CPU cores and cover different types of units with different computational and memory access characteristics. Heterogeneity broadens the range of interesting load dispatch policies and exacerbates the challenge of the load dispatch mechanism's system integration (i.e., (operating system and high-level architecture aspects)).

10.2.2 Proactive Versus Reactive Load Balancing

Our proposed NI-driven load balancing mechanism relies on rapid on-chip core-NI communication to optimize for cross-core load balance proactively. Because accurately predicting the service time of an incoming RPC at arrival time is a difficult open problem [72, 117], our approach to tackling load imbalance is the deference of a request's processing assignment until a core becomes free. An alternative approach to such proactive load balancing would be a reactive policy that assigns incoming requests to cores eagerly, but offers a rebalancing mechanism that is triggered whenever load imbalance across cores is detected. As mentioned in Section 9.3.1, one such approach of reactive load balancing is work stealing, implemented in the similar context of balancing RPCs to cores by the ZygOS system [144]. Each approach, proactive and reactive, has its own benefits and drawbacks.

A reactive load balancing mechanism, such as work stealing, offers high flexibility, as it can be dynamically modified and controlled in software. However, it introduces performance overheads in the form of "task migrations", which can be considerable for RPCs with very short service times. Task reassignments also result in priority reordering (out-of-order processing) of incoming requests, which negatively impacts tail latency. For example, such an implementation of work stealing in ZygOS results in 25% performance drop as compared to perfect load balance for RPCs with a 10 μ s service time, a performance gap expected to be larger for shorter RPCs. Our implementation of proactive load balancing introduces latency overhead related to propagating core occupancy information to the NI and dispatching a new request from the NI to the core. However, the physical core-NI proximity, which is a direct consequence of on-chip NI integration, renders this overhead negligible, as our comparison to the theoretical queuing model demonstrates in Section 8.3.2. Because our implementation's proactive dispatch decisions are based in hardware, the flexibility in terms of dispatch policies are limited as compared to software-controlled reactive rebalancing techniques, such as work stealing.

The efficiency of each approach, proactive versus reactive, is largely determined by the workload. For RPCs that execute for several 10s of μ s and demonstrate low service time variability, the need

for load rebalancing will be infrequent enough to make the mechanism's overhead negligible. In such cases, software-based load rebalancing implementations are good enough. For more challenging workloads, with service times of a couple of μs and high service time variability, the performance superiority of proactive load balancing will be pronounced. A detailed quantitative analysis of these key tradeoffs of the two load balancing approaches, as well as their sensitivity to workloads characteristics, are worth investigating in the future.

10.2.3 Scaling to CPUs with Hundreds of Cores

Our evaluation of NI-driven dynamic load balancing was based on a modestly sized CMP of 16 cores, because of practical considerations related to the scalability of cycle-accurate simulation. Our results indicate that centralizing RPC dispatch decisions to a single NI does not introduce considerable overheads for the evaluated 16-core chip, and we expect this to apply to most modern server-grade CPUs. As discussed in Section 8.1.2, even in an extreme case scenario of a CPU servicing RPCs with service time as low as 500ns, the requirements for message dispatch from a single NI to the cores is one every $\sim 31\text{ns}$ or $\sim 8\text{ns}$ for a 16-core and a 64-core chip, respectively. Both dispatch frequencies are modest enough for a single hardware dispatch component to handle.

Centralizing the load dispatch decision is, however, not a universal solution. In a potential future CPU with 100s of cores, the overhead of centralizing dispatch decisions can become considerable, necessitating a more scalable approach. One possible approach could be the partitioning of resources, by statically limiting the number of cores each NI can dispatch load to, similar in concept to the 4×4 queuing configuration presented in Section 8.1.1. While in our evaluated 16-core system the performance difference between 4×4 and a single-queue system implementation is considerable, grouping a larger number of cores per dispatch group (e.g., a 4×64 configuration for a 256-core chip) may approach the performance of a single-queue configuration. We demonstrated this effect in a different context in [136]: our study demonstrates that pooling the memory of multiple servers together can absorb the inherent inter-object popularity skew of distributed object stores. The larger the server group, the more

capable it is of successfully re-distributing the load within the group, absorbing the negative impact of load imbalance across individual servers. The same observation is applicable to our context of balancing load across cores of a single manycore chip: with large enough dispatch groups, the effects of load imbalance can be significantly mitigated.


A second approach to balancing load on a CPU with 100s of cores would be the design of a hierarchical mechanism with several stages, with each stage reducing the number of candidate cores. Such an approach eschews the need for centralizing all dispatch decisions at the cost of slightly increased dispatch decision latency. Finally, beyond a certain system scale, a third plausible approach would be to revisit load balancing policies investigated in the context of datacenter-scale web services, such as the ones discussed in Section 9.3.2.

10.2.4 Load Balancing Opportunities with PCIe-Attached NIs

Tight NI integration with the CPU enables nanosecond-scale interaction of NI and compute logic, enabling unprecedented flexibility in terms of dynamic load balancing decisions. While integrated NIs represent ideal candidates for such a mechanism, dynamic load balancing could also be applicable, in a constrained form, to discrete PCIe-attached NIs. As several modern NIs offer programmable logic in a bump-on-the-wire architecture (e.g., Mellanox BlueField and Innova Flex), which can be leveraged to implement adaptive load dispatch policies. An interesting research question is to what extent such added flexibility can improve performance over static load distribution mechanisms like RSS and Flow Direction. Because of the microsecond-scale communication delays between the CPU cores and the NIC, introduced by the PCIe interface, the opportunity will be limited as compared to integrated NIs, but will have broader applicability. For the same reason, the design constraints and dispatch policies will be significantly different from the ones investigated in Chapter 5 (e.g., going after single-request queue depth per core would dramatically hurt throughput).

11 Conclusion

The conventional boundaries of network and compute are blurring. Increasing performance demands in communication-intensive datacenter environments require network and compute to start fusing, bringing the NI—the bridge between the two—to the epicenter of such integration efforts. The NI in future servers should be part of the CPU as much as part of the network: effectively leveraging the increasing network bandwidth resources will require the NI and CPU to be co-designed and tightly integrated. Tight integration enables offloading application-level functionality from the CPU to the NI, offering opportunities for vertical optimization of modern rich application software stacks. This thesis demonstrated the importance of proper NI integration in server chips and its co-design with CPU resources, the potential of introducing new network operations with richer semantics, and the new opportunities that arise from extending the network endpoints with richer functionality.

- 
- [1] “Apache Cassandra,” <http://cassandra.apache.org/>. (Cited on page 52)
- [2] “Memcached,” <http://memcached.org/>. (Cited on pages 52 and 74)
- [3] “Project Voldemort,” <http://www.project-voldemort.com/>. (Cited on page 52)
- [4] “Redis,” <http://redis.io/>. (Cited on page 52)
- [5] “First International Workshop on Rack-scale Computing (WRSC),” <https://www.microsoft.com/en-us/research/event/first-international-workshop-on-rack-scale-computing-wrsc-2014/>, 2014. (Cited on page 12)
- [6] D. Abts, N. D. E. Jerger, J. Kim, D. Gibson, and M. H. Lipasti, “Achieving predictable performance through better memory controller placement in many-core CMPs.” in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 451–461. (Cited on page 111)
- [7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. A. Kranz, J. Kubiawicz, B.-H. Lim, K. Mackenzie, and D. Yeung, “The MIT Alewife Machine: Architecture and Performance.” in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, 1995, pp. 2–13. (Cited on pages 154 and 155)
- [8] K. Aingaran, S. Jairath, G. K. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki, “M7: Oracle’s Next-Generation Sparc Processor.” *IEEE Micro*, vol. 35, no. 2, pp. 36–45, 2015. (Cited on page 159)

Bibliography

- [9] Amazon, “Amazon ElastiCache,” <http://aws.amazon.com/elasticache/>. (Cited on page 52)
- [10] C. Amza, A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, “ThreadMarks: Shared Memory Computing on Networks of Workstations.” *IEEE Computer*, vol. 29, no. 2, pp. 18–28, 1996. (Cited on page 154)
- [11] Anandtech, “ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 Exynos Review.” [Online]. Available: <http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/6> (Cited on page 47)
- [12] —, “Haswell: Up to 128MB On-Package Cache.” [Online]. Available: <http://www.anandtech.com/show/6277/haswell-up-to-128mb-onpackage-cache-ulv-gpu-performance-estimates> (Cited on page 113)
- [13] —, “Living on the Edge: Intel Launches Xeon D-2100 Series SoCs.” [Online]. Available: <https://www.anandtech.com/show/12409/intel-launches-xeon-d-2100-series-socs-edge> (Cited on page 11)
- [14] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, “FAWN: a fast array of wimpy nodes.” *Commun. ACM*, vol. 54, no. 7, pp. 101–109, 2011. (Cited on page 52)
- [15] K. Asanović, “A Hardware Building Block for 2020 Warehouse-Scale Computers,” USENIX FAST Keynote, 2014. (Cited on pages 55 and 113)
- [16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store.” in *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64. (Cited on pages 52, 85, and 98)
- [17] L. A. Barroso, “Three Things to Save the Datacenter,” ISSCC Keynote, 2014. [Online]. Available: http://www.theregister.co.uk/Print/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future/. (Cited on page 98)

- [18] L. A. Barroso, M. Marty, D. Patterson, and P. Ranganathan, “Attack of the killer microseconds.” *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017. (Cited on pages 3 and 74)
- [19] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, “The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane.” *ACM Trans. Comput. Syst.*, vol. 34, no. 4, pp. 11:1–11:39, 2017. (Cited on pages 80 and 161)
- [20] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, “Integrated network interfaces for high-bandwidth TCP/IP.” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006, pp. 315–324. (Cited on pages 24, 96, and 157)
- [21] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, “Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer.” in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 142–153. (Cited on pages 156 and 157)
- [22] C. Blundell, M. M. K. Martin, and T. F. Wenisch, “InvisiFence: performance-transparent memory ordering in conventional multiprocessors.” in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 233–244. (Cited on pages 61 and 159)
- [23] D. Bonachea, “Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, Version 2.0,” 2007. (Cited on page 154)
- [24] Boston Limited, “Boston Limited Unveil Their Revolutionary Boston Viridis,” 2011. [Online]. Available: <http://www.boston.co.uk/press/2011/11/boston-limited-unveil-their-revolutionary-boston-viridis.aspx/>. (Cited on page 13)
- [25] M. Bramson, Y. Lu, and B. Prabhakar, “Randomized load balancing with general service time distributions.” in *Proceedings of the 2010 ACM SIGMETRICS International Confer-*

Bibliography

- ence on Measurement and Modeling of Computer Systems*, 2010, pp. 275–286. (Cited on page 163)
- [26] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, “TAO: Facebook’s Distributed Data Store for the Social Graph.” in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013, pp. 49–60. (Cited on page 52)
- [27] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, “Implementation and Performance of Munin.” in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991, pp. 152–164. (Cited on page 154)
- [28] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, “A cloud-scale acceleration architecture.” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13. (Cited on pages 14, 16, 160, and 164)
- [29] Cavium Networks, “Cavium Announces Availability of ThunderX™: Industry’s First 48 Core Family of ARMv8 Workload Optimized Processors for Next Generation Data Center & Cloud Infrastructure,” 2014. [Online]. Available: <http://www.cavium.com/newsevents-Cavium-Announces-Availability-of-ThunderX.html>. (Cited on pages 95 and 99)
- [30] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, “BulkSC: bulk enforcement of sequential consistency.” in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 278–289. (Cited on page 159)
- [31] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, “A Scalable, Non-blocking Approach to Transactional Memory.” in *Proceedings of the 13th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2007, pp. 97–108. (Cited on page 159)

- [32] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, “Hive: Fault Containment for Shared-Memory Multiprocessors.” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995, pp. 12–25. (Cited on page 155)
- [33] M. Chapman and G. Heiser, “vNUMA: A Virtual Shared-Memory Multiprocessor.” in *Proceedings of the 2009 USENIX Annual Technical Conference (ATC)*, 2009. (Cited on page 154)
- [34] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, “Fast and general distributed transactions using RDMA and HTM.” in *Proceedings of the 2016 EuroSys Conference*, 2016, pp. 26:1–26:17. (Cited on page 17)
- [35] D. R. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, “HI-CAMP: architectural support for efficient concurrency-safe shared structured data access.” in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, 2012, pp. 287–300. (Cited on page 159)
- [36] C. Coarfa, Y. Dotsenko, J. M. Mellor-Crummey, F. Cantonnet, T. A. El-Ghazawi, A. Mohanti, Y. Yao, and D. G. Chavarría-Miranda, “An evaluation of global address space languages: co-array fortran and unified parallel C.” in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005, pp. 36–47. (Cited on page 153)
- [37] J. R. Crandall and F. T. Chong, “Minos: Control Data Attack Prevention Orthogonal to Memory Model.” in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004, pp. 221–232. (Cited on page 159)
- [38] D. E. Culler, A. C. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick, “Parallel programming in Split-C.” in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC)*, 1993, pp. 262–273. (Cited on page 34)

Bibliography

- [39] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, “Manycore network interfaces for in-memory rack-scale computing.” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 567–579. (Cited on page 8)
- [40] A. Daglis, D. Ustiugov, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, “SABRes: Atomic object reads for in-memory rack-scale computing.” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13. (Cited on page 8)
- [41] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: a flexible information flow architecture for software security.” in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 482–493. (Cited on page 159)
- [42] Data Center Knowledge, “Google Data Center FAQ,” 2017. [Online]. Available: <http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq/> (Cited on page 1)
- [43] T. David, R. Guerraoui, and V. Trigonakis, “Everything you always wanted to know about synchronization but were afraid to ask.” in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 33–48. (Cited on page 42)
- [44] M. Davis and D. Borland, “System and Method for High-Performance, Low-Power Data Center Interconnect Fabric,” WO Patent 2,011,053,488, 2011. (Cited on page 11)
- [45] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, “SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering.” in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 25–36. (Cited on page 137)
- [46] J. Dean, “Building Software Systems At Google and Lessons Learned [video],” 2011. [Online]. Available: <http://www.youtube.com/watch?v=modXC5IWTJI> (Cited on page 10)

- [47] J. Dean and L. A. Barroso, “The tail at scale.” *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013. (Cited on page 74)
- [48] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store.” in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 205–220. (Cited on pages 10, 52, and 74)
- [49] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, “DMP: deterministic shared memory multiprocessing.” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIV)*, 2009, pp. 85–96. (Cited on page 159)
- [50] A. Dhodapkar, G. Lauterbach, S. Li, D. Mallick, J. Bauman, S. Kanthadai, T. Kuzuhara, G. S. M. Xu, and C. Zhang, “SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips,” in *HOTCHIPS-XXIII*, 2011. (Cited on pages 13, 55, and 101)
- [51] A. Dragojevic, “The Configurable Cloud: Accelerating Hyperscale Datacenter Services with FPGAs,” *Keynote at the Workshop on Multi-Core and Rack-Scale Systems (MaRS)*, 2017. (Cited on page 160)
- [52] A. Dragojevic, D. Narayanan, and M. Castro, “RDMA Reads: To Use or Not to Use?” *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 3–14, 2017. (Cited on page 17)
- [53] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, “FaRM: Fast Remote Memory.” in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414. (Cited on pages 12, 16, 50, 52, 53, 54, 56, 58, 59, 63, 127, 131, and 160)
- [54] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, “No compromises: distributed transactions with consistency, availability, and performance.” in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 54–70. (Cited on pages 12 and 16)

Bibliography

- [55] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. N. Pnevmatikatos, “The Mondrian Data Engine.” in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 639–651. (Cited on page 168)
- [56] EZchip Semiconductor Ltd., “EZchip Introduces TILE-Mx100 World’s Highest Core-Count ARM Processor Optimized for High-Performance Networking Applications,” Press Release, 2015. [Online]. Available: <http://www.tilera.com/News/PressRelease/?ezchip=97>. (Cited on pages 11, 95, 99, and 101)
- [57] Facebook, “Building express backbone: Facebook’s new long-haul network,” 2017. [Online]. Available: <https://code.facebook.com/posts/1782709872057497> (Cited on page 10)
- [58] Facebook, “Facebook company info,” 2018. [Online]. Available: <https://newsroom.fb.com/company-info/>. (Cited on page 1)
- [59] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood, “Application-specific protocols for user-level shared memory.” in *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC)*, 1994, pp. 380–389. (Cited on page 155)
- [60] B. Falsafi and D. A. Wood, “Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA.” in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 229–240. (Cited on page 154)
- [61] ———, “Parallel Dispatch Queue: A Queue-Based Programming Abstraction to Parallelize Fine-Grain Communication Protocols.” in *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 1999, pp. 182–192. (Cited on page 162)
- [62] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: a study of emerging scale-out workloads on modern hardware.” in *Proceedings of the 17th International*

-
- Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, 2012, pp. 37–48. (Cited on pages 11 and 132)
- [63] M. Flajslik and M. Rosenblum, “Network Interface Design for Low Latency Request-Response Protocols.” in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013, pp. 333–346. (Cited on pages 21 and 23)
- [64] M. Franklin and G. S. Sohi, “The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism.” in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 58–67. (Cited on page 62)
- [65] —, “ARB: A Hardware Mechanism for Dynamic Reordering of Memory References,” *IEEE Trans. Comput.*, 1996. (Cited on page 62)
- [66] R. Gillett, “Memory Channel: An Optimized Cluster Interconnect,” *IEEE Micro*, vol. 16(2), pp. 12–18, 1996. (Cited on page 157)
- [67] C. Gniady, B. Falsafi, and T. N. Vijaykumar, “Is SC + ILP=RC?” in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 162–171. (Cited on page 61)
- [68] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, “RDMA over Commodity Ethernet at Scale.” in *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016, pp. 202–215. (Cited on page 32)
- [69] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt, “Analysis of join-the-shortest-queue routing for web server farms.” *Perform. Eval.*, vol. 64, no. 9-12, pp. 1062–1081, 2007. (Cited on page 163)
- [70] E. Hagersten and M. Koster, “WildFire: A Scalable Path for SMPs.” in *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 1999, pp. 172–181. (Cited on page 154)

Bibliography

- [71] L. Hammond, B. D. Carlstrom, V. Wong, M. K. Chen, C. Kozyrakis, and K. Olukotun, “Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software.” *IEEE Micro*, vol. 24, no. 6, pp. 92–103, 2004. (Cited on page 159)
- [72] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, “Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services.” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, 2015, pp. 161–175. (Cited on pages 81, 140, and 171)
- [73] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: near-optimal block placement and replication in distributed caches.” in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 184–195. (Cited on page 106)
- [74] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, “Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor.” in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 38–50. (Cited on pages 106, 154, and 155)
- [75] Hewlett-Packard Enterprise, “HP Moonshot System Family Guide,” 2015. [Online]. Available: <http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-6076ENW&cc=us&lc=en/>. (Cited on page 55)
- [76] —, “The Machine: A new kind of computer,” 2015. [Online]. Available: <http://www.labs.hpe.com/research/themachine/>. (Cited on pages 13 and 55)
- [77] High Scalability, “Latency is Everywhere and it Costs You Sales - How to Crush it,” 2009. [Online]. Available: <http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it> (Cited on page 1)

- [78] R. Huggahalli, R. R. Iyer, and S. Tetrick, “Direct Cache Access for High Bandwidth Network I/O.” in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 50–59. (Cited on page 157)
- [79] *IEEE 802.1Qbb: Priority-Based Flow Control*, IEEE, 2011. (Cited on pages 12, 21, and 156)
- [80] InfiniBand Trade Association, *InfiniBand Architecture Specification: Release 1.0*, 2000. (Cited on pages 12 and 21)
- [81] —, “InfiniBand Roadmap,” 2018. [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=technology_overview (Cited on pages 2, 15, and 167)
- [82] Intel Corp., “Introduction to Intel® Ethernet Flow Director and Memcached Performance,” <http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf>. (Cited on pages 75 and 161)
- [83] —, “Moving Data with Silicon and Light,” 2013. [Online]. Available: <http://www.intel.com/content/www/us/en/research/intel-labs-silicon-photonics-research.html> (Cited on page 113)
- [84] Internet live stats, “Google search statistics,” 2018. [Online]. Available: <http://www.internetlivestats.com/google-search-statistics/>. (Cited on page 1)
- [85] J. Jeddloh and B. Keeth, “Hybrid Memory Cube New DRAM Architecture Increases Density and Performance,” in *2012 International Symposium on VLSI Technology (VLSIT)*, 2012. (Cited on page 113)
- [86] J. Johnson, *GUI Bloopers 2.0: Common User Interface Design Don'Ts and Dos*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. (Cited on page 1)
- [87] N. P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.” in *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, 1990, pp. 364–373. (Cited on page 63)

Bibliography

- [88] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-Datcenter Performance Analysis of a Tensor Processing Unit.” in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12. (Cited on page 168)
- [89] A. Kalia, M. Kaminsky, and D. G. Andersen, “Using RDMA efficiently for key-value services.” in *Proceedings of the ACM SIGCOMM 2014 Conference*, 2014, pp. 295–306. (Cited on pages 12, 16, 38, 52, 83, and 140)
- [90] —, “FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs.” in *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, 2016, pp. 185–201. (Cited on page 32)
- [91] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. M. Brooks, “Profiling a warehouse-scale computer.” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169. (Cited on pages 3, 11, and 74)
- [92] A. Kaufmann, S. Peter, T. E. Anderson, and A. Krishnamurthy, “FlexNIC: Rethinking Network DMA.” in *Proceedings of The 15th Workshop on Hot Topics in Operating Systems (HotOS-XV)*, 2015. (Cited on page 164)

- [93] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy, “High Performance Packet Processing with FlexNIC.” in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*, 2016, pp. 67–81. (Cited on page 164)
- [94] R. Kessler and J. Schwarzmeier, “Cray T3D: A New Dimension for Cray Research,” in *Comcon Spring '93, Digest of Papers*, 1993. (Cited on pages 102 and 156)
- [95] B. Khan, M. Horsnell, I. Rogers, M. Luján, A. Dinn, and I. Watson, “An Object-Aware Hardware Transactional Memory System.” in *Proceedings of the 2008 IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2008, pp. 93–102. (Cited on page 158)
- [96] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy, “The Stanford FLASH Multiprocessor.” in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 302–313. (Cited on pages 106, 154, 155, and 164)
- [97] J. Laudon and D. Lenoski, “The SGI Origin: A ccNUMA Highly Scalable Server.” in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 241–251. (Cited on page 154)
- [98] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam, “The Stanford Dash Multiprocessor.” *IEEE Computer*, vol. 25, no. 3, pp. 63–79, 1992. (Cited on page 154)
- [99] K. Li and P. Hudak, “Memory Coherence in Shared Virtual Memory Systems.” *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989. (Cited on page 154)
- [100] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore

Bibliography

- architectures.” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480. (Cited on page 45)
- [101] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, “Architecting to achieve a billion requests per second throughput on a single key-value store server platform.” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 476–488. (Cited on page 52)
- [102] G. Liao, X. Zhu, and L. N. Bhuyan, “A new server I/O architecture for high speed networks.” in *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2011, pp. 255–265. (Cited on page 157)
- [103] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, “MICA: A Holistic Approach to Fast In-Memory Key-Value Storage.” in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 429–444. (Cited on pages 52, 59, 130, 161, and 170)
- [104] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, “Thin servers with smart pipes: designing SoC accelerators for memcached.” in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 36–47. (Cited on pages 52 and 98)
- [105] LinkedIn, “How LinkedIn Uses Memcached,” <http://www.oracle.com/technetwork/server-storage/ts-4696-159286.pdf>. (Cited on page 52)
- [106] Linley Group, “The Russians Are Coming,” *Microprocessor Report*, February 1999. (Cited on page 159)
- [107] —, “X-Gene 2 Aims Above Microservers,” *Microprocessor Report*, September 2014. (Cited on pages 11, 55, 95, and 101)
- [108] —, “Cortex-A57 Is Most Efficient CPU,” *Microprocessor Report*, February 2015. (Cited on page 47)

- [109] —, “Oracle Shrink Sparc M7,” *Microprocessor Report*, September 2015. (Cited on pages 11, 55, 95, and 101)
- [110] —, “Cavium Beefs Up ThunderX2 CPU,” *Microprocessor Report*, June 2016. (Cited on pages 11 and 101)
- [111] —, “Cavium Thunders Into Servers,” *Microprocessor Report*, February 2016. (Cited on pages 11 and 101)
- [112] —, “Phytium Samples 64-Core ARMv8,” *Microprocessor Report*, September 2016. (Cited on page 11)
- [113] —, “Centriq Aces Scale-Out Performance,” *Microprocessor Report*, November 2017. (Cited on page 11)
- [114] —, “Epyc Relaunches AMD Into Servers,” *Microprocessor Report*, June 2017. (Cited on page 11)
- [115] —, “X-Gene 3 Up and Running,” *Microprocessor Report*, March 2017. (Cited on page 11)
- [116] —, “Xeon Scalable Reshapes Server Line,” *Microprocessor Report*, July 2017. (Cited on page 11)
- [117] J. R. Lorch and A. J. Smith, “Improving dynamic voltage scaling algorithms with PACE.” in *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2001, pp. 50–61. (Cited on page 171)
- [118] P. Lotfi-Kamran, B. Grot, and B. Falsafi, “NOC-Out: Microarchitecting a Scale-Out Processor.” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 177–187. (Cited on pages 113 and 120)
- [119] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi, “Scale-out processors.” in *Proceedings of*

Bibliography

- the 39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 500–511. (Cited on pages 11, 95, 99, and 101)
- [120] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. G. Greenberg, “Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services.” *Perform. Eval.*, vol. 68, no. 11, pp. 1056–1071, 2011. (Cited on page 163)
- [121] K. Mackenzie, J. Kubiawicz, A. Agarwal, and F. Kaashoek, “Fugu: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor,” in *Proceedings of the 1994 Workshop on Shared Memory Multiprocessors*, 1994. (Cited on pages 154 and 155)
- [122] Y. Mao, E. Kohler, and R. T. Morris, “Cache craftiness for fast multicore key-value storage.” in *Proceedings of the 2012 EuroSys Conference*, 2012, pp. 183–196. (Cited on pages 59 and 65)
- [123] Mellanox Technologies, “ConnectX InfiniBand Adapter Cards,” 2011. [Online]. Available: http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX_VPI.pdf (Cited on page 156)
- [124] —, “RDMA Aware Networks Programming User Manual, Rev 1.7,” https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015. (Cited on pages 2, 11, and 22)
- [125] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.” *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991. (Cited on page 141)
- [126] Microsoft Corp., “Receive Side Scaling,” <http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx>. (Cited on pages 75 and 161)
- [127] C. Mitchell, Y. Geng, and J. Li, “Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store.” in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013, pp. 103–114. (Cited on pages 12, 16, 50, 52, 53, 54, and 58)

- [128] S. S. Mukherjee, P. J. Bannon, S. Lang, A. Spink, and D. Webb, “The Alpha 21364 Network Architecture.” *IEEE Micro*, vol. 22, no. 1, pp. 26–35, 2002. (Cited on page 25)
- [129] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, “Coherent Network Interfaces for Fine-Grain Communication.” in *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996, pp. 247–258. (Cited on page 157)
- [130] A. Muzahid, S. Qi, and J. Torrellas, “Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically.” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 363–375. (Cited on page 61)
- [131] J. Nielsen, *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993. (Cited on page 1)
- [132] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, “Scaling Memcache at Facebook.” in *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398. (Cited on page 52)
- [133] S. Novakovic, “Rack-Scale Memory Pooling for Datacenters,” *EPFL PhD Thesis*, 2017. (Cited on pages 3 and 33)
- [134] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, “Scale-out NUMA.” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 3–18. (Cited on pages 8, 36, and 55)
- [135] ———, “An Analysis of Load Imbalance in Scale-out Data Serving.” in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2016, pp. 367–368. (Cited on page 75)

Bibliography

- [136] ———, “The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems.” in *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, 2016, pp. 182–195. (Cited on pages 3, 13, 75, and 172)
- [137] S. Novakovic, A. Daglis, B. Grot, E. Bugnion, and B. Falsafi, “Scale-out Non-uniform Memory Access,” *US Patent 9,734,063*, filed February 27, 2015, issued August 15, 2017. (Cited on page 8)
- [138] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum, “Fast crash recovery in RAMCloud.” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 29–41. (Cited on page 12)
- [139] Oracle, “Oracle Exadata Database Machine,” 2016. [Online]. Available: <http://www.oracle.com/technetwork/database/exadata/overview/index.html/>. (Cited on pages 13 and 55)
- [140] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang, “The RAMCloud Storage System.” *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, 2015. (Cited on page 52)
- [141] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” 1999. (Cited on page 34)
- [142] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe, “Arrakis: The Operating System Is the Control Plane.” *ACM Trans. Comput. Syst.*, vol. 33, no. 4, pp. 11:1–11:30, 2016. (Cited on page 80)
- [143] I. Pragaspathy and B. Falsafi, “Address Partitioning in DSM Clusters with Parallel Coherence Controllers.” in *Proceedings of the 9th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2000, pp. 47–56. (Cited on page 162)
- [144] G. Prekas, M. Kogias, and E. Bugnion, “ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks.” in *Proceedings of the 26th ACM Symposium on*

- Operating Systems Principles (SOSP)*, 2017, pp. 325–341. (Cited on pages 74, 75, 161, and 171)
- [145] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, “Scalable and reliable communication for hardware transactional memory.” in *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2008, pp. 144–154. (Cited on page 159)
- [146] X. Qian, J. Torrellas, B. Sahelices, and D. Qian, “BulkCommit: scalable and fast commit of atomic blocks in a lazy multiprocessor environment.” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 371–382. (Cited on page 159)
- [147] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Tempest and Typhoon: User-Level Shared Memory.” in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 325–336. (Cited on pages 106, 154, and 164)
- [148] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the Social Network’s (Datacenter) Network.” in *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015, pp. 123–137. (Cited on page 85)
- [149] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, “It’s Time for Low Latency.” in *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011. (Cited on pages 2, 10, 21, and 74)
- [150] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, “Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996, pp. 174–185. (Cited on page 154)
- [151] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Fine-grain Access Control for Distributed Shared Memory.” in *Proceedings of the 6th*

Bibliography

- International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 297–306. (Cited on page 154)
- [152] S. L. Scott and G. M. Thorson, “The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus,” in *Hot Interconnects*, 1996. (Cited on pages 102 and 156)
- [153] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, “Near-Optimal Worst-Case Throughput Routing for Two-Dimensional Mesh Networks.” in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 432–443. (Cited on page 111)
- [154] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, “Scalable Hardware Memory Disambiguation for High ILP Processors.” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003, pp. 399–410. (Cited on page 62)
- [155] S. Shelach, “Mellanox wins \$200m Google, Microsoft deals,” <http://www.globes.co.il/en/article-1000857043>, 2013. (Cited on page 12)
- [156] W. Shi, E. Collins, and V. Karamcheti, “Modeling object characteristics of dynamic Web content.” *J. Parallel Distrib. Comput.*, vol. 63, no. 10, pp. 963–980, 2003. (Cited on page 98)
- [157] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network.” in *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015, pp. 183–197. (Cited on page 10)
- [158] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, “Reunion: Complexity-Effective Multicore Redundancy.” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 223–234. (Cited on page 106)

- [159] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, “Netpipe: A Network Protocol Independent Performance Evaluator,” in *IASTED International Conference on Intelligent Information Management and Systems*, 1996. (Cited on pages 23 and 141)
- [160] E. Spertus, S. C. Goldstein, K. E. Schauer, T. von Eicken, D. E. Culler, and W. J. Dally, “Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5.” in *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, 1993, pp. 302–313. (Cited on page 159)
- [161] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch, “Deconstructing the Tail at Scale Effect Across Network Protocols.” *CoRR*, vol. abs/1701.03100, 2017. (Cited on page 16)
- [162] R. Stets, S. Dwarkadas, N. Hardavellas, G. C. Hunt, L. I. Kontothanassis, S. Parthasarathy, and M. L. Scott, “Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network.” in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997, pp. 170–183. (Cited on page 157)
- [163] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking.” in *Proceedings of the 11st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004, pp. 85–96. (Cited on page 159)
- [164] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, “RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits.” in *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006, pp. 32–39. (Cited on page 85)
- [165] B. Towles, J. P. Grossman, B. Greskamp, and D. E. Shaw, “Unifying on-chip and inter-node switching within the Anton 2 network.” in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 1–12. (Cited on pages 46, 112, and 127)

Bibliography

- [166] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, “Speedy transactions in multicore in-memory databases.” in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 18–32. (Cited on pages 59 and 74)
- [167] Twitter, “Memcached SPOF Mystery,” <https://blog.twitter.com/2010/memcached-spoof-mystery>, 2010. (Cited on page 52)
- [168] J. Ullal, “Keynote Talk (Arista Networks),” in *Proceedings of the 18th Annual IEEE Symposium on High Performance Interconnects (HOTI)*, 2010, pp. xv–xvi. (Cited on page 9)
- [169] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation.” in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 256–266. (Cited on page 155)
- [170] Z. Wang, H. Qian, J. Li, and H. Chen, “Using restricted transactional memory to build a scalable in-memory database.” in *Proceedings of the 2014 EuroSys Conference*, 2014, pp. 26:1–26:15. (Cited on page 59)
- [171] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, “Fast in-memory transaction processing using RDMA and HTM.” in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 87–104. (Cited on pages 12, 17, 50, 52, 53, 58, and 158)
- [172] M. Weiss, “Amazon: 3 Million Packages a Day, Alibaba: 12 Million Packages a Day,” *Early Moves: The Future Retail Blog*, March 2017. [Online]. Available: <https://earlymoves.com/2017/03/21/amazon-3-million-packages-a-day-alibaba-12-million-packages-a-day/>. (Cited on page 1)
- [173] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Mechanisms for store-wait-free multiprocessors.” in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 266–277. (Cited on page 159)

- [174] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, “SimFlex: Statistical Sampling of Computer System Simulation.” *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006. (Cited on pages 112, 127, 139, and 141)
- [175] WinterCorp, “Big Data: What Does It Really Cost?” 2013. [Online]. Available: <http://www.teradata.com/Resources/White-Papers/WinterCorp-Special-Report--Big-Data-What-does-it-really-cost/> (Cited on page 9)
- [176] K. A. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. N. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. L. Welcome, and T. Wen, “Productivity and performance using partitioned global address space languages.” in *International Workshop on Parallel Symbolic Computation (PASCO)*, 2007, pp. 24–32. (Cited on page 153)
- [177] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, “The End of a Myth: Distributed Transactions Can Scale.” in *Proceedings of the VLDB Endowment*, 2017. (Cited on page 17)
- [178] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, “Hardware Enforcement of Application Security Policies Using Tagged Memory.” in *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 225–240. (Cited on page 159)
- [179] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, “NetFPGA SUME: Toward 100 Gbps as Research Commodity.” *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014. (Cited on page 164)

Alexandros DAGLIS

Computer Science PhD Candidate
École Polytechnique Fédérale de Lausanne (EPFL)

Computer & Communication Sciences +41 21 69 31385
École Polytechnique Fédérale de Lausanne <http://parsa.epfl.ch/~daglis/>
INJ 239, Station 14, CH-1015 Lausanne alexandros.daglis@epfl.ch

RESEARCH INTERESTS

I am broadly interested in the ongoing transition towards extreme hardware heterogeneity, as it opens a broad range of architectural innovation opportunities. With CPU performance improvements stagnating, it is time for a paradigm shift from CPU-centric to network- and memory-centric computing. Such transition calls for a thorough rethink of well-established practices across layers: hardware architectures, software, and algorithms. I therefore strive to take holistic, cross-layer approaches in my research. My work so far has been focused on rack-scale computing and network-compute integration, targeting the most challenging communication-intensive services. These services are abundant in modern datacenter environments, where service quality guarantees impose tight microsecond-scale tail latency constraints. Blurring the boundaries between network and compute, and pushing functionality with richer semantics to the network paves the way for tail-tolerant, low-latency services.

EDUCATION

- 2012–2018 PhD candidate in COMPUTER SCIENCE
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
Thesis: “Network-compute co-design for distributed in-memory computing”
Advisors: Prof. Babak FALSAFI & Prof. Edouard BUGNION
- 2007–2012 Diploma in ELECTRICAL & COMPUTER ENGINEERING (5-year degree)
National Technical University of Athens (NTUA), Athens, Greece
Thesis: “A study of a dynamic placement policy in a NUCA cache”
Advisor: Prof. Nectarios KOZIRIS
GPA: 9.22/10 (Summa cum laude – top 1%)

AWARDS & HONORS

- 2013–2014 Microsoft Research Fellowship
- 2012–2013 EPFL Computer Science Fellowship
- 2012–2013 Nominated for a *Fulbright scholarship* for graduate studies in the US
- 2010 Honorary distinction by the State Scholarships Foundations of Greece (IKY) for ranking 5/430 in class (undergraduate)

EMPLOYMENT

| | |
|-------------------|---|
| SEP. '12–AUG. '18 | <p>Research Assistant at EPFL, Lausanne, Switzerland</p> <p>Research on Datacenter Technologies and In-Memory Rack-scale Computing</p> <ul style="list-style-type: none">• Lead architect of the Scale-Out NUMA project, introducing a communication protocol and architecture for fast distributed in-memory processing. Designed Scale-Out NUMA's Remote Memory Controller, an on-chip network protocol controller integrated in the processor's coherence domain.• Proposed and evaluated Split NI, the first scalable network interface design for manycore chips.• Proposed and evaluated SABRes, a lightweight hardware extension that provides the network interface with the capability of direct <i>atomic</i> access to data objects in remote memory, without the remote CPU's involvement.• Contributed to the design of RackOut, a system architecture that leverages fast one-sided remote memory accesses (RDMA) for memory pooling and load balancing in skewed data serving workloads.• Contributed to the design of the Mondrian Data Engine, an algorithm-hardware co-design for high-performance data analytics on emerging architectures with near-memory processing capabilities. |
| JAN.–MAY '15 | <p>Systems Research Intern at HP LABS, Palo Alto, USA</p> <p>Software and hardware research for HP's <i>The Machine</i></p> <ul style="list-style-type: none">• Analyzed the microarchitectural behavior and memory access patterns of analytics workloads on Spark.• Investigated opportunities for hybrid hardware-software and partial coherence schemes for <i>shared something</i> architectures. |

PEER-REVIEWED PUBLICATIONS

- [1] **The Mondrian Data Engine.** Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, Dionisios Pnevmatikatos. In Proceedings of the *44th International Symposium on Computer Architecture (ISCA)*, Toronto, ON, Canada, 2017.
- [2] **The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems.** Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, Boris Grot. In Proceedings of the *ACM Symposium in Cloud Computing (SoCC)*, Santa Clara, CA, USA, 2016.
- [3] **SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing.** Alexandros Daglis, Dmitrii Ustiugov, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, Boris Grot. In Proceedings of the *49th International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, 2016.
- [4] **An Analysis of Load Imbalance in Scale-out Data Serving.** Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, Boris Grot. *ACM SIGMETRICS* (Short paper), Antibes Juan-Les-Pins, France, 2016.

- [5] **Manycore Network Interfaces for In-Memory Rack-Scale Computing.** Alexandros Daglis, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, Boris Grot. In Proceedings of the *42nd International Symposium on Computer Architecture (ISCA)*, Portland, OR, USA, 2015.
- [6] **Scale-Out NUMA.** Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, Boris Grot. In Proceedings of the *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, Salt Lake City, UT, USA, 2014.

OTHER PUBLICATIONS

- [1] **Enabling Storage-Class Memory as a DRAM Replacement for Datacenter Services.** Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, Dionisios Pnevmatikatos. *arXiv: 1801.06726v1*. Submitted: 20 Jan. 2018.

INVITED TALKS

Network-Centric Computing for Online Services

- SCS seminar at Georgia Tech February 2018
- CS research seminar at UCLA February 2018
- CS research seminar at Rutgers University March 2018
- EE departmental seminar at Princeton University March 2018
- CIS seminar at the University of Pennsylvania March 2018
- CS research seminar at Imperial College London April 2018

SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing

- 49th International Symposium on Microarchitecture (MICRO), Taipei, Taiwan (conference presentation) Oct. 2016

Chip Design for In-Memory Rack-Scale Computing

- EcoCloud Annual Event, Lausanne, Switzerland May 2016

Manycore Network Interfaces for In-Memory Rack-Scale Computing

- Research day of the Computer Systems Lab of the National Technical University of Athens (NTUA), Athens, Greece Jan. 2016
- 42nd International Symposium in Computer Architecture (ISCA), Portland, USA (conference presentation) Jun. 2015

Hybrid Coherence for "The Machine"

- HP Labs, Palo Alto, USA May 2015

STUDENT SUPERVISION

- Zilu Tian (1st year PhD at EPFL) Sep. 2017–present
 - Currently supervising Zilu in a project targeting memory access efficiency via dynamic

in-memory data transformation through specialized hardware.

- Dmitrii Ustiugov (3rd year PhD at EPFL) Jan.–Aug. 2017
- Supervised Dmitrii on memory hierarchy design leveraging emerging Storage-Class Memory technology as a cheaper and denser DRAM replacement. This work identified the key parameters that enable designs capable of delivering significant cost benefits without sacrificing performance and is currently under review.
- Ivo Mihailovic (Master's thesis at EPFL) Mar.–Apr. 2017
- Supervised Ivo on his Master's thesis, in which he studied the effectiveness of RDMA-friendly data structures and the tradeoff between data overfetch and additional network roundtrips.
- Marina Shimchenko (Summer intern at EPFL) Jun.–Sep. 2016
- Supervised Marina on extending the QEMU emulator to support incremental checkpoints, a critical missing feature, as it dramatically reduces storage requirements for long chains of checkpoints needed for cycle-accurate simulation.
- Zaid Qureshi (2nd year Master's at EPFL) Mar.–Jun. 2016
- Supervised Zaid on a semester project investigating caching policies for data accessed from remote memory over RDMA operations, driven by the insight that remote memory access patterns drastically differ from those of local accesses.
- Hussein Kassir (Master's thesis at EPFL) Feb.–Jun. 2016
- Supervised Hussein (with Mario Drumond) on his master thesis, which involved prototyping the Scale-Out NUMA architecture on HARP, Intel's hybrid CPU-FPGA platform. Hussein is now a PhD student at EPFL.
- MS Course Research Project Mentoring Fall 2015 & Fall 2017
- As the head TA of the Advanced Multiprocessor Architecture course at EPFL, supervised twenty course-tailored research projects in total over two semesters. Sample topics include: shared core frontend for call graph prefetching, graph analytics on GPU, the effect of multithreading on the performance density of server processors, and analysis of modern RPC software stacks.

TEACHING ASSISTANTSHIPS

| | |
|-------------|--|
| Fall 2017 | Advanced Multiprocessor Architecture (MS) – Head TA |
| Fall 2015 | |
| | Prepared the course material (slides, exercises, exams), graded, guided students on individual research projects, and gave guest lectures on distributed memory systems and advanced topics on memory consistency. |
| Fall 2017 | Introduction to Multiprocessor Architecture (BS) – Guest Lecturer |
| Fall 2016 | |
| | Gave guest lectures on cache coherence and memory consistency. |
| Spring 2017 | Topics on Datacenter Design (PhD) |
| | Constructed curriculum, led in-class discussions, graded weekly paper reviews. Course based on paper readings and discussions. |
| Fall 2016 | Computer Architecture I (BS) |
| Fall 2014 | |
| Fall 2013 | |
| | Prepared the course's material (slides, exercises, labs, exams), graded, and taught lab sessions. |

| | |
|----------------------------|---|
| Spring 2016 | Virtual Reality (MS) Prepared a lab exercise based on Google Cardboard and assisted in lab sessions. |
| Spring 2014 Spring 2013 | Real-time Embedded Systems (MS) Assisted in lab sessions. |

PROFESSIONAL SERVICE

CLOUDSUITE BENCHMARK SUITE:

Member of the CloudSuite team since 2012. CloudSuite is one of the first benchmark suites representative of modern datacenter services, and has been used to drive the design of modern datacenter-oriented CPUs, such as Cavium ThunderX. I have been in the core team responsible for the 2nd and 3rd releases of CloudSuite, which upgraded the software packages and added new workloads. In the 3rd release we also included pre-packaged Docker images of all workloads to ease deployment. I was responsible for the Graph Analytics workload in the 2nd release, and the Media Streaming workload in the 3rd release.

FLEXUS SIMULATOR:

Member of the Flexus maintenance team since 2012. Flexus is the only full-system cycle-accurate simulator that can be used to practically model full-blown modern server software stacks, thanks to its integrated SMARTS statistical sampling. The project's latest focus has been on (i) porting Flexus from Simics to the QEMU open-source and widely used emulator, making Flexus widely accessible to the community; and (ii) moving from the Sparc to the ARM ISA, which is emerging in the server space.

TUTORIALS:

- APR. 2016 *Server Benchmarking with CloudSuite 3.0* at EuroSys'16, London, UK.
With Mario Drumond, Javier Picorel, and Dmitrii Ustiugov.
- JAN. 2016 *Server Benchmarking with CloudSuite 3.0* at the 11th HiPEAC conference, Prague, Czech Republic.
With Mario Drumond and Javier Picorel.
- JUN. 2013 *CloudSuite on Flexus* at the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel.
With Djordje Jevdjic and Cansu Kaynak.

FUNDING

In April 2016, drafted the technical part of a *Swiss National Science Foundation* research proposal with Javier Picorel, entitled "Memory-Centric Server Architecture for Datacenters" (Principal PI: Prof. Babak Falsafi). The proposal was ranked at the top of six quality levels and was awarded a full 3-year funding for 2 PhD students.

PATENT

Scale-Out Non-Uniform Memory Access | US Patent 9,734,063, 2017

- With Stanko Novakovic, Boris Grot, Edouard Bugnion, and Babak Falsafi.
- Prototyped on Intel's hybrid CPU-FPGA HARP platform.

- Licensed by a major IT vendor in 2016.

LANGUAGES

GREEK, ENGLISH, GERMAN (limited working proficiency), FRENCH (elementary proficiency).

EXTRACURRICULAR

- Violin Diploma (highest degree), 2012 | Erateio Conservatory, Athens, Greece.
- Principal first violin in the orchestra of Erateio Conservatory, 2008–2012.
- Gold medal in the 5th *World Choir Games* (Graz, Austria), as member of the *Rosarte* choir.

