DISS. ETH NO. 19001

Zero-Copy Network Communication:

An Applicability Study of iWARP beyond Micro Benchmarks

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

PHILIP WERNER FREY

MSc ETH CS, ETH Zurich

born January 4, 1981

citizen of

Zurich, Switzerland

accepted on the recommendation of

Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Timothy Roscoe, co-examiner
Dr. Bernard Metzler, co-examiner
Prof. Dr. Dejan Kostic, co-examiner

2010

Abstract

Technology trends suggest that the fastest networks will continue to provide link bandwidths close to the memory and I/O subsystem limits. Because of these trends and the inevitable overhead in the TCP/IP stack implementations, an increasing share of a host's capacity is dedicated to pure network I/O and therefore unavailable to application processing. In this thesis, we assess the benefit of applying *Remote Direct Memory Access* (RDMA) as a means to mitigate the shortcomings of TCP/IP-based communication on high-performance interconnects. With RDMA, data is placed directly in the application memory of a remote computer. In bypassing the operating system and eliminating intermediate copying, RDMA promises to reduce the host overhead of large data transfers significantly, thereby making it attractive for implementing distributed applications. In a nutshell, RDMA over Ethernet (also known as *iWARP*) is a TCP/IP offload engine plus direct memory access from the network interface card to the application memory.

In the first part of this thesis, we analyze why offloading the TCP stack to the network adapter alone is not sufficient for significant host overhead reduction—a zero-copy mechanism, like the one offered by iWARP/RDMA is required. We do not discuss how to implement iWARP/RDMA as this has been done by many researchers and the industry. Rather, the objective is to assess its benefits and implications with regard not only to user applications but also to the whole operating system which has not been done yet. To that end, we perform a thorough study and identify the hidden costs of iWARP/RDMA in terms of performance as well as programming- and protocol design complexity. Finally, we provide a set of optimizations without which RDMA loses all its advantages.

The goal of the second part is to show by example how applications can leverage the potential of iWARP/RDMA when adhering to the conditions and optimizations presented in the fist part of the thesis. We demonstrate the performance advantages in different application domains like high-definition media dissemination or distributed databases. Thanks to the host overhead elimination, we achieve unprecedented performance. Finally, we illustrate the inevitable programming- and protocol complexity, due to the new interface and semantics, by enhancing a legacy sockets-based application with the RDMA abstraction and provide an assessment of when such a complex transformation is worthwhile.

Kurzfassung

Die Datenraten moderner Computer Netzwerke bewegen sich oft im oberen Bereich der Arbeitsspeicher- und I/O Subsystem Geschwindigkeiten, wie Technologietrends wiederholt gezeigt haben. Dies führt dazu, dass ein Grossteil der Rechenkapazität für den Datentransfer (innerhalb des Computers) aufgewendet werden muss und somit nicht mehr für andere Anwendungen zur Verfügung steht. In dieser Abhandlung soll untersucht werden, ob und in wie weit es uns *Remote Direct Memory Access* (RDMA) erlaubt, modernste Netzwerke zu nutzen ohne die kommunizierenden Computer mit dem Datentransfer zu belasten. Einfach gesagt, ermöglicht RDMA den direkten Zugriff über ein Netzwerk auf entfernten Arbeitsspeicher. Die Kernideen sind dabei das Umgehen des Betriebssystems sowie das Vermeiden von unnötigen Datenkopien im Netzwerkstack des Computers. Dadurch wird die Kommunikation wesentlich effizienter und interessant für verteilte Anwendungen.

Im Folgenden beschäftigen wir uns vor allem mit *iWARP*, einer Implementierung von RDMA über Ethernet. Im ersten Teil der Abhandlung untersuchen wir, warum es nicht genügt den TCP/IP Stack auf den Netzwerkadapter auszulagern und warum das Vermeiden der Kopieroperationen essentiell ist. Unser Ziel ist es, den Nutzen und die Implikationen von RDMA in Bezug auf die Anwendungen und das Betriebssystem zu verstehen und nicht eine RDMA Implementation zu beschreiben. Zu dem Zweck werten wir eine Reihe von Testergebnissen aus, die Aufschluss geben über die zu erwartende Leistungssteigerung aber auch versteckte Kosten von RDMA und die Komplexität solch neuer Systeme. Um den Leistungsvorteil von RDMA zu maximieren, schlagen wir anschliessend eine Reihe von Optimierungen vor.

Das Ziel des zweiten Teiles ist es, an Hand von konkreten Beispielen zu zeigen, wie unterschiedliche Anwendungen das Potential von RDMA zu ihren Gunsten nutzen können. Wir demonstrieren die Vorteile am Beispiel der Verbreitung von High-Definition Video an viele Zuschauer sowie von grossen verteilten Datenbanken. Wir zeigen, dass durch die Eliminierung unnötiger Arbeit auf den kommunizierenden Computern, die Anwendungen direkt von der performanten Netzwerk Infrastruktur profitieren können. Zu guter Letzt schreiben wir eine existierende, Sockets-basierte Anwendung für RDMA um und erörtern, unter welchen Umständen so eine Transformation sinnvoll ist.

Acknowledgements

I would like to express my gratitude to the people who have supported me during the course of this work.

First, I would like to thank the people at IBM Research-Zurich. Above all, I owe many thanks to Bernard Metzler who has provided me with a lot of invaluable conceptual as well as technical help. Many thanks for all the fruitful and humorous discussions! Moreover, I need to thank Andreas Hasler for all the hours he spent towards the successful completion of the high-definition media dissemination project. Many thanks also go to Charlotte Bolliger for proof-reading my papers and to Fredy Neeser for the introductory assistance to the RDMA subject. Last but not least, I want to express my gratitude to Patrick Droz for providing me with all the necessary resources and management support. Many thanks also go to Felix Marti from Chelsio Corporation for providing me with insights on the RDMA-enabled network adapters.

I am equally grateful for all the support I received from the Systems Group at ETH Zurich. First and foremost, I owe many thanks to my advisor, Gustavo Alonso, who supervised the whole work, kept challenging my endeavors and provided me with a lot of well-appreciated guidance. Furthermore, I am thankful for the challenging questions and helpful advice from my co-advisor Timothy Roscoe. Special thanks also go to Jens Teubner as well as Romulo Goncalves and Martin Kersten for the fruitful collaboration on the Data Roundabout project. Finally, I am deeply grateful to Rene Mueller, Michael Duller, Adrian Schuepbach, Jan Rellermeyer, Akhilesh Singhania, Thomas Heinis and Nico Schottelius for the numerous technical discussions and joyful company.

Contents

1 Introduction

We start this thesis by giving the motivation for the work at hand, followed by a summary of the contributions and a brief overview of the content of the subsequent chapters.

1.1 Motivation - Network Communication Revisited

Following "Moore's Law", computing power per machine doubles every two years on average. However, network technology performance has recently grown at a much faster pace (e.g., Ethernet technology has evolved from 100 Mb/s to 10 Gb/s in a much shorter time frame). Technology trends suggest that the fastest networks will continue to provide link bandwidths close to the memory and I/O subsystem limits on most hosts. Because of this trend and the unavoidable overhead in common TCP/IP stack implementations such as application data copying, context switches and protocol processing, an increasing share of a host's processing power is dedicated to pure network I/O and therefore unavailable to application processing [CJRS89]. Hence, end systems that use TCP sockets are likely to remain the bottleneck in high-speed data communication performance in current as well as in future hardware.

Remote Direct Memory Access (RDMA) is an alternative mechanism to TCP sockets. With RDMA, data is placed directly in the application memory of a remote computer [HCPR]. In bypassing the operating system and eliminating

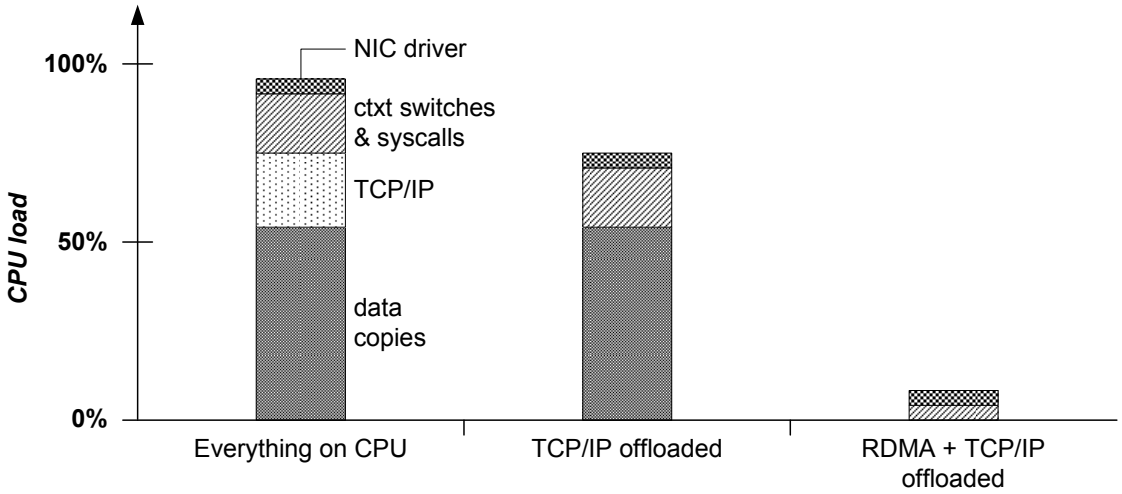


Figure 1.1: CPU load distribution for bulk data transfers across a high-speed network. At the far left, all tasks are performed by the host CPU. In the middle, the TCP/IP stack processing is offloaded to the network adapter. On the far right, the TCP/IP offloading is extended with RDMA capabilities.

intermediate copying across buffers (zero copy), RDMA significantly reduces the CPU cost of large data transfers as well as the end-to-end latency, thereby making it very attractive for implementing distributed applications [LWK+03, NCTP07, KCH+07]. Having the CPU available for computation while receiving and sending data at a very high rate is important for various applications such as a distributed real-time analysis of large scientific experiments or high-definition video streaming to a substantial number of clients. Although the ideas behind RDMA are not new [Dav91, DWB+93, ST93, DP93, CCC97, CGY01], it has only been with the constant increase in network bandwidth that they have become a necessity not only for proprietary high-bandwidth fabrics such as InfiniBand [ibt] but also for Ethernet based TCP/IP [CJRS89]. Early attempts to reduce the CPU load caused by high-bandwidth TCP/IP communication, offloaded the entire TCP/IP stack onto the network interface card (NIC). This proved not to be enough [Mog03]. Today, the approach most favored is a TCP/IP offload engine (TOE) plus direct memory access (DMA) from the NIC to application memory [RB03, BC02]. In a nutshell, this is what RDMA over Ethernet [RMTB05] (also known as *iWARP*) provides.

The advantages of RDMA over plain TCP/IP offloading can be demonstrated with a simple experiment [MNF]. Transferring bulk data as fast as possible over a standard TCP/IP connection reveals a CPU load distribution like the one shown in Figure 1.1. Most CPU cycles are spent on data copying within the local host (leftmost chart). Although offloading the TCP/IP stack onto the NIC shows some

improvement (middle), only the TOE in combination with RDMA reduces the CPU load significantly by eliminating the data copying (rightmost chart). This is the reason why RDMA is being looked at with increasing interest as a key design element of future distributed systems [DW07a, NSL+08, Pak08]. Nevertheless, it is also facing a lot of criticism [Geo06, MSG04].

1.2 Problem Statement

RDMA usage is nowadays limited to high-performance computing and storage applications. Its deployment in other application domains is still due. We have added RDMA support to a number of those applications and were disappointed to find that the performance benefit was not always significant.

We argue, that a profound investigation is needed in order to gain the necessary insight and understanding of the technology at hand. Only this will enable us to rigorously characterize the circumstances under which the benefit of RDMA becomes substantial. In a second step, we want to show the obtainable benefit by means of real-world applications which clearly go beyond micro benchmarks.

In essence, we look for an answer to the following question: "What do we gain with RDMA and which are the ideal circumstances of its deployment?".

1.3 Contributions of this Thesis

In short, we first identify the sweet spots of iWARP/RDMA through a set of experiments and ease the application development by providing a more intuitive consumer interface. Based on that, we then present a number of real-world RDMA applications which we have built on one hand to confirm our findings and on the other hand to demonstrate the advantage of iWARP/RDMA for applications outside the high-performance computing (HPC) domain. In more detail, the contributions are the following:

- Even though, iWARP/RDMA promises a significant performance potential, it is not widely deployed today. One reason for that is the complicated and error prone interface on which the industry has agreed. We hence present a library with a simpler interface to facilitate iWARP application development. Nevertheless, we are able to preserve the performance and flexibility of the original interface. In-depth understanding of the inner workings of RDMA is hence no longer required. The library code has been returned to the industry alliance[1] where it was well received.

[1] http://www.openfabrics.org

- Another reason for the limited deployment of iWARP is the necessary hardware which is still quite expensive (around $800 per adapter) and which has only become available on the mass market in late 2007. In order to bridge the gap between the expensive but powerful iWARP and the ubiquitous but lower-performance TCP, we have implemented an iWARP/RDMA software adapter which extends ordinary Ethernet NICs with iWARP capabilities. This allows for machines, which do not have RDMA hardware installed, to be integrated into iWARP/RDMA networks. Furthermore, as we will see, the expensive hardware is not always necessary. Details on our early version, termed *SoftRDMA*, are published in:

  [NMF10] Fredy Neeser, Bernard Metzler, and Philip W. Frey. SoftRDMA: Implementing iWARP over TCP kernel sockets. *IBM Journal of Research and Development. Special Issue on Network-Optimized Computing*, (2010, in press).

  A more recent version, termed *Softiwarp*, has been presented at:

  [MNF09] Bernard Metzler, Fredy Neeser, and Philip W. Frey. A software iWARP driver for OpenFabrics. In *OpenFabrics Alliance Sonoma Workshop*, 2009.

- A third reason for the low acceptance of iWARP/RDMA is the uncertainty about the performance benefit for a given application. Throughout various experiments, we have been able to identify the sweet spots of the technology and, based on our findings, present a metric which allows for a more accurate assessment of the performance potential at hand. Furthermore, we suggest a number of application optimizations that can be applied when designing the explicit memory management as required by RDMA. The original work is published in:

  [FA09] Philip W. Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, 2009 (Best-Paper Award).

- Based on the findings mentioned above, we have implemented a number of real-world applications. First, we have transformed the distributed C/C++ compiler extension *distcc* from the sockets abstraction to the iWARP/RDMA interface. While the performance advantage is not substantial, the work describes the steps required to replace the legacy socket interface with iWARP and reasons about the various design decisions. Furthermore, a solution is provided for making files, residing on secondary storage, RDMA-accessible. The work is published in:

[FMN10] Philip W. Frey, Bernard Metzler, and Fredy Neeser. Enabling applications for RDMA: Distributed compilation revisited. *Technical report, IBM Research RZ3764*, January 26, 2010.

- The second application, high-definition multimedia data dissemination, matches the sweet spot of iWARP/RDMA. We have built an iWARP-capable video server, which is able to serve the content in real-time to a substantial number of clients at a negligible CPU load. In this work, we provide a one-on-one comparison of our solution against HTTP/TCP as well as RTP/UDP and are able to show the superiority of RDMA. In particular, we highlight the benefits of the one-sided, asynchronous operations offered by iWARP. Furthermore, this application serves as an excellent use case for the aforementioned software iWARP adapter as we cannot require each client to be equipped with RDMA hardware. The original work can be found in:

  [FHMA09] Philip W. Frey, Andreas Hasler, Bernard Metzler, and Gustavo Alonso. Server-efficient high-definition media dissemination. In *Proceedings of the 19th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 49–54, 2009.

- Finally, we have built a novel distributed database architecture which operates on rotating data sets. In particular, our architecture, termed *Data Roundabout*, is able to exploit all computing resources at hand offering good performance and scalability characteristics. Thanks to iWARP/RDMA, we no longer have to avoid network communication at all cost but can leverage the potential provided by the communication hardware. This allows us to spend more CPU cycles on the actual data processing. We show the performance benefit through a set of join algorithms implemented on top of the *Data Roundabout*. Furthermore, we compare the iWARP results with plain TCP. The work has been published in the following two places:

  [FGKT09] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the 5th International Workshop on Data Management on New Hardware*, pages 27–33, 2009.

  [FGKT10] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. A spinning join that does not get dizzy. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, 2010.

1.4 Overview of this Thesis

The thesis consists of two parts:

Part I. The first part starts with background information on TCP and iWARP/RDMA which is required to follow the argumentation presented subsequently. As part of the related work, the history of iWARP is outlined (Chapter 2). In Chapter 3, we look at a real iWARP system in full detail. In particular, we present the requirements for enabling RDMA/iWARP from an operating system (OS) as well as from a user application perspective. A demonstration of the system in practice together with a set of initial benchmarks concludes the chapter. Subsequently, in Chapter 4, we point out the issues with regard to the explicit memory management required by RDMA. Also, we suggest a number of optimizations to make RDMA more efficient with regard to real-world applications. Finally, the critical parameters for unleashing the full performance potential as well as for assessing the potential benefit of iWARP are presented.

Part II. In the second part, we present applications which are more than just micro benchmarks. In Chapter 5, we illustrate the steps necessary to transform a legacy application from the socket interface to iWARP at the example of the distributed C/C++ compiler *distcc*. In particular, we will demonstrate the benefit of the asynchronous API offered by RDMA. In Chapter 6, we build an RDMA-based high-definition media dissemination platform that outperforms HTTP/TCP and RTP/UDP by far not only in terms of serviceable clients but also in terms of the induced server load. Here, we focus particularly on the one-sided semantics of the RDMA operations. In Chapter 7, we move on to the database domain and suggest a novel database architecture based on rotating data sets—the *Data Roundabout*. iWARP/RDMA allows us to ship large amounts of data without incurring any significant cost on the hosts. We thus do not have to avoid network communication and can use the CPU cycles for the data processing which results in a most efficient utilization of the resources at hand. Chapter 8 finally concludes this thesis.

Part I

iWARP/RDMA Communication Principles

2 RDMA Background

The first part of this thesis provides background information required to follow the techniques and argumentation presented subsequently. Before we introduce and discuss the Remote Direct Memory Access (RDMA) communication model, we briefly review the issues encountered when applying the classical socket interface to high-speed[1] interconnects such as 10 Gigabit Ethernet [inta].

2.1 TCP Sockets - The Classical Communication Interface

The Transmission Control Protocol (TCP) [Pos81, Ste94, Tan02] is one of the universally accepted transport layer protocols today. In this section, we revise some of the inner workings of TCP. This is necessary in order to understand why the socket interface is limited with regard to multi-gigabit interconnects. The TCP/IP performance issues on high-speed interconnects have been documented by various researchers [CJRS89, KP93, KP96, FHH+03, BSP04, FBB+05] and we thus restrict ourselves to the most apparent ones.

The TCP protocol specification [Pos81] dates back to 1981 and has not changed significantly since then. However, the infrastructure on which the protocol operates has changed dramatically in the last thirty years. As a consequence, the protocol

[1] High-speed refers to conditions in which the network bandwidth is high relative to the bandwidth of the host memory and CPU.

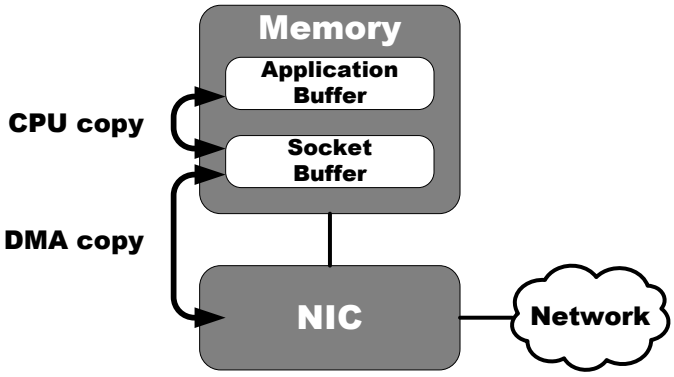


Figure 2.1: Simplified, logical in-host data path for TCP transfers. The data is intermediately stored in the kernel socket buffer before it is either sent out by the network interface controller (TX) or copied into the user space buffer (RX).

in its original form does no longer yield the most efficient resource utilization. Yet, the performance limitations imposed by TCP are not primarily a result of the network protocol itself. Rather they are a consequence of the in-host data path imposed by the socket abstraction.

One major problem is the indirect data placement as shown, in a simplified version, in Figure 2.1. On the *transmit path*, when the `write()` operation is issued, the data is first copied by the CPU from its location in the application buffer (user space) into a temporary socket buffer (kernel space). Thereafter, the TCP/IP stack implementation (running in the kernel) puts the data into an appropriate TCP packet by adding control information such as the ports, sequence numbers, checksum etc. Finally, the packet is brought to the network interface controller by a DMA copy. On the *receive path*, the procedure is the same but in reverse order. First, the data is DMA'ed from the NIC into the socket buffer from where it is then copied by the CPU into the application address space. The last step is triggered by the application issuing a `read()` command.

Already in this simplified illustration, the overhead caused by the intermediate copying through kernel space is apparent. As we will show in a moment, this copying causes an even larger overhead in reality. With network bandwidth still increasing exponentially, shipping data using TCP/IP over recent high-speed interconnects (i.e., 10 Gigabit Ethernet and newer) induces a non-negligible overhead on the end-hosts mainly due to the intermediate copying described above which results in a reduction of the available compute resources. As a consequence, the applications running on the end-hosts are not able to realize the full potential of the underlying network infrastructure.

In the following, we demonstrate the problems of socket-based communication by means of a simple experiment: we transfer bulk data as fast as possible over

a standard TCP/IP connection. Our setup for that consists of two HS21 IBM BladeServers. Each of them is equipped with a quad core Intel Xeon CPU running at 2.33 GHz, 32 KB L1 data cache and 32 KB L1 instruction cache, 4 MB unified L2 cache and 8 GB of main memory.

2.1.1 CPU Overhead

First, we demonstrate the implications, with regard to the CPU, when transferring data at a bandwidth of 1 Gbps and compare the result with 10 Gbps. To that end, we exchange random bulk data over the network using increasing buffer sizes from 1 B to 1 GB. Figure 2.2 reveals the following:

- The network link can be saturated with moderate CPU load for 1 Gbps (Figure 2.2(a)).
- For the 10 Gbps link, this is no longer true (Figure 2.2(b)). The receive-side CPUs are running under full load but the link can still not be fully utilized. There are virtually no CPU cycles left for application processing.
- The receive side (RX) induces more CPU load than the transmit side (TX) in both cases.
- The achieved throughput generally depends on the message size. For the 1 Gbps link this means larger is better. For the 10 Gbps case, the cache sizes have to be taken into account. There, the throughput also depends on whether the data fits into L1 cache (32 KB in our case), L2 cache (4 MB) or none of them.

Our analysis on how the CPU cycles are spent confirm earlier findings [FHH+03]: most of the cycles are spent on data touching operations such as *copying* and *checksum calculation* given that the messages are sufficiently large (Figure 2.2(c)). Further CPU cycles are spent on the TCP/IP stack processing, on system calls and context switches as well as on the NIC driver itself. We are thus bound by the *per-byte cost*. For small messages, on the other hand, we are bound by the *per-packet cost* and the protocol processing dominates. For more details, refer to [KP93] and [FHH+03].

An early, generally accepted rule of thumb states that about 1 Hz of CPU power is required for utilizing 1 bps of the network link. However, the CPU speeds have been increasing faster than the memory bus speeds which causes the CPU to spend more time waiting for the memory bus [CGY01]. As these waiting times are often too short for a context switch, the CPU cycles are essentially lost causing a shift in the above rule of thumb meaning that with recent architectures, 1 bps on

the network requires *even more* than 1 Hz of CPU power. Assuming a 10 Gbps interconnect, at least 10 GHz of CPU power would hence be required to utilize the link.

Furthermore, we see a trend in the evolution of processors away from increasingly faster single cores towards multiple cores per machine, today. Unfortunately, this does not solve the TCP/IP overhead problem because the network stack processing can not be parallelized easily [NBF07]. In addition to that, the CPU is not the only bottleneck as we will illustrate in the next section.

2.1.2 Memory Bus Traffic

A second, less apparent but equally important problem of the socket abstraction is the *high memory bus traffic* caused by the intermediate copying [FHH+03, BSP04, RB03].

While Figure 2.1 has shown a simplified version of the actual data path, Figure 2.3 shows a more complete picture for outbound data.[2] Looking closely at the data path on the transmit side, we find that the following steps are performed:

1. CPU reads data from the application buffer.
2. CPU writes the data to the socket buffer.
3. NIC performs a DMA read from the socket buffer.

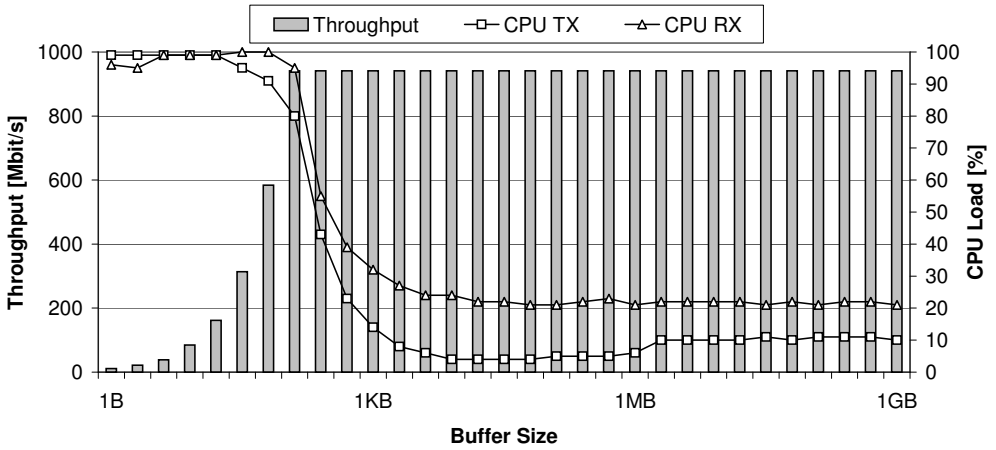
In practice, this means that for every byte we transfer over the network, 2-4 bytes cross the memory bus (depending on whether the data fits into the cache [FHH+03]). In order to perform a data transfer over the network at 10 Gbps we hence require a memory bus that can sustain up to 40 Gbps (or even 80 Gbps if we communicate in full-duplex mode). This does not yet include normal memory bus traffic unrelated to the network transfers. The issue is aggravated when moving to higher bandwidths, for instance, by using multiple NICs per node in parallel [PMB09].

It should now be evident that increasing the CPU speed or adding more CPUs to the system is not enough to be able to sustain the increasing, aggregate throughput—only the elimination of the unnecessary copy operations is [Mar02].

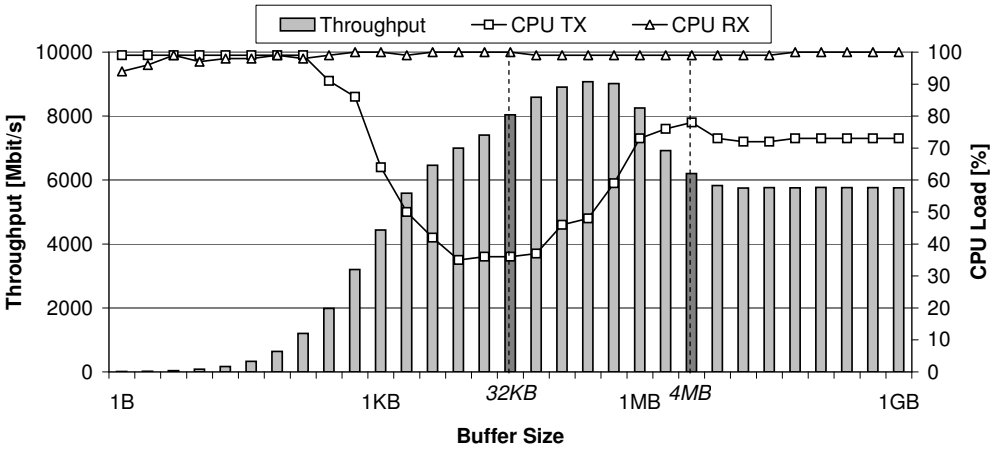
2.1.3 Context Switches

In addition to the CPU load and memory bus traffic caused by the TCP stack processing and data copy operations, the socket-based communication also causes

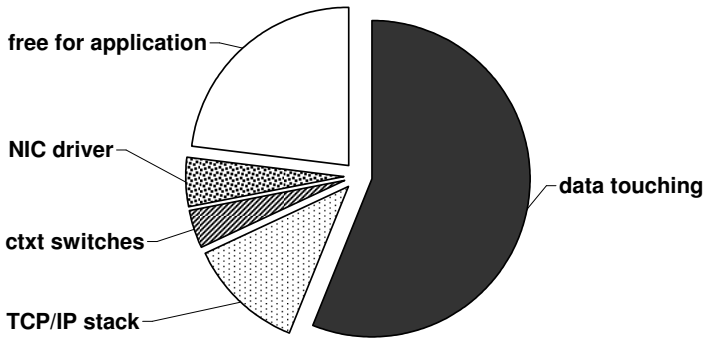
[2] The receive path is essentially the same but in reverse order.



(a) TCP bulk data transfer on 1 Gbps Ethernet.



(b) TCP bulk data transfer on 10 Gbps Ethernet.



(c) CPU load distribution for large messages on 10 Gbps.

Figure 2.2: In-kernel TCP is able to fully saturate the 1 Gbps network link but not the 10 Gbps one due to its expensive in-host data path.

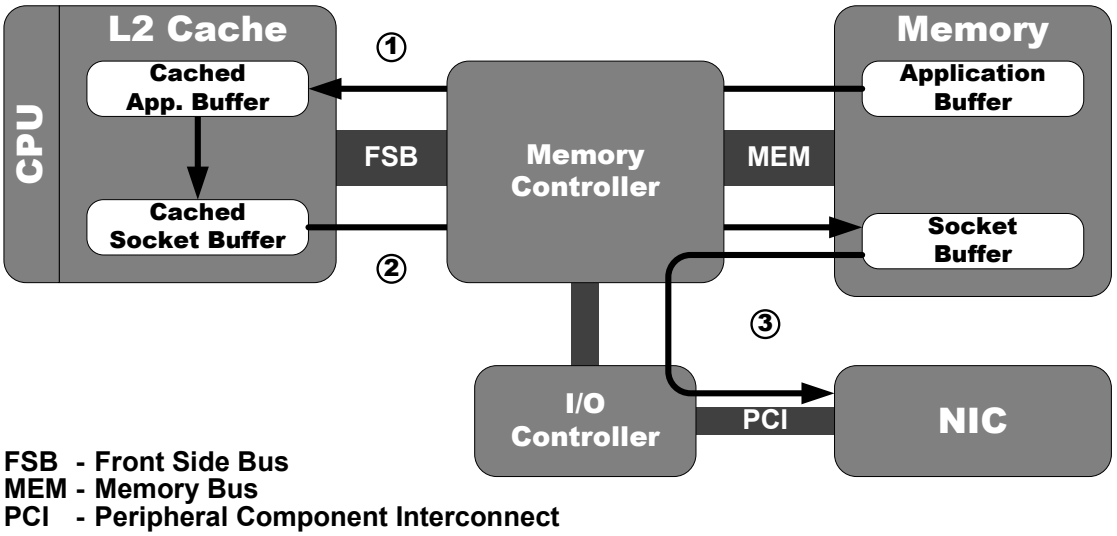


Figure 2.3: Full in-host transmit data path for socket-based data transfers. The same data crosses the memory bus several times.

a large number of context switches when running at the aforementioned, high data rates. Context switches do not only interrupt the currently running task but also cause cache-pollution which can result in a non-negligible slowdown of the whole system [MB91].

The frequent context switches are mainly due to interrupts raised by the NIC to signal updates in the data propagation. This is particularly problematic at the receive side as the NIC raises a lot of interrupts in order to notify the kernel about new inbound data being present. In Chapter 6, we will present an example which shows the negative impact of the high context switch rate in practice.

2.2 Reducing the Communication Overhead

In the previous section, we have identified the major cost factors for TCP data transfers over high-speed Ethernet as being the high CPU load and memory bus traffic. The overhead is caused not only by the intermediate data copies and the protocol processing but also by the numerous context switches and interrupts. In the following, we discuss proposed solutions to reduce this overhead.

2.2.1 Offloading the TCP Stack is not Enough

Traditionally, the TCP/IP protocol stack is implemented as part of the operating system (OS) kernel which implies that it runs on the host CPU(s). As we have

discussed in the preceding section, processing the stack requires a non-negligible share of CPU cycles when handling large data throughput as seen in high-speed networks.

In order to reduce the load on the CPU, researchers have proposed to offload (some of) the TCP stack processing to the NIC. Early implementations have focused on offloading only the compute-intensive parts, such as the checksum calculation. With the ongoing bandwidth increase, eventually the whole stack including all protocol layers below TCP were offloaded to specialized NICs, called *TCP Offload Engines* (TOE).

Even though the TOE approach sounds promising and is able to outperform in-kernel TCP/IP [FBB+05], it has never been widely adopted. The reason for this is two-fold [Mog03]: first, there are still fundamental performance issues. A TOE can reduce the unnecessary intermediate copies (which are responsible for the majority of the CPU cost) but is not able to fully eliminate them. Furthermore, it fails at significantly reducing the context switch rate. Second, its deployment in practice proved to be more complex than anticipated for a number of reasons [Mog03].

2.2.2 The RDMA Idea

The TCP offloading approach illustrated above operates on the socket abstraction and thus cannot avoid the intermediate copying through the kernel socket buffer. *Remote Direct Memory Access* (RDMA), on the other hand, introduces a radically different interface (discussed in detail in Section 3) which allows a system to place the communicated data directly into its final memory location (in user space) without any additional or intermediate data copies through kernel space. This *zero-copy* or *direct data placement* capability provides the most efficient network communication possible. An RDMA-enabled network interface controller (RNIC) provides a hardware accelerated RDMA-ed network stack instance in addition to the conventional network stack in the operating system.

RDMA, thus, enables transferring data from the memory of one host *directly* into the memory of another with minimal involvement of the CPUs in the transfer and thereby essentially extends the well-known local DMA model with network capabilities.

Figure 2.4 illustrates the basic idea at the example of a single data transfer: first, the CPUs program their RNICs with the necessary data placement information which consists of the memory addresses, the length of the data to be exchanged as well as an ID which uniquely identifies the buffer (called *Steering Tag*, or *STag*). Second, the sending RNIC fetches the data (without involvement of the OS kernel) from the local application source buffer using DMA and moves the data across the network to the receiving RNIC which places the inbound data directly into its final location at the destination buffer of the application. Finally,

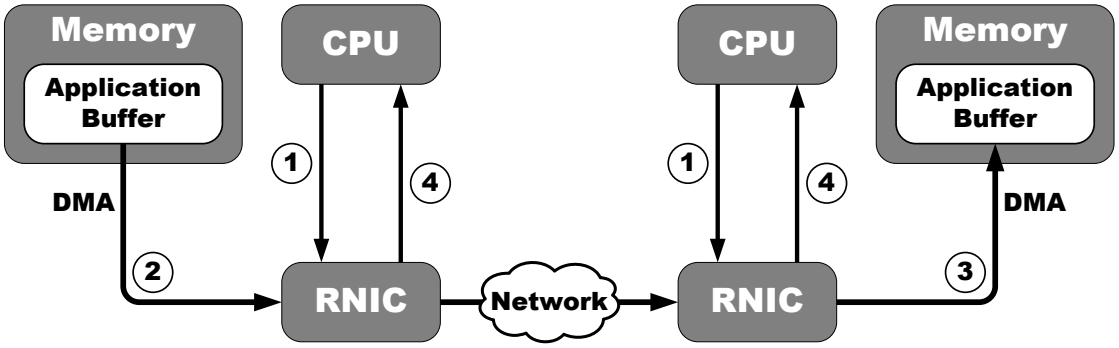


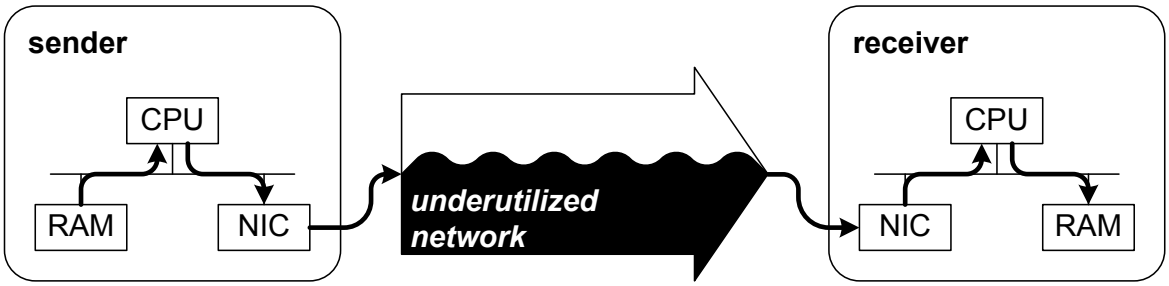
Figure 2.4: A data transfer using RDMA-enabled NICs.

the RNICs notify their CPUs when the data transfer has completed. Note that there is no intermediate buffering of the data in kernel space on either side

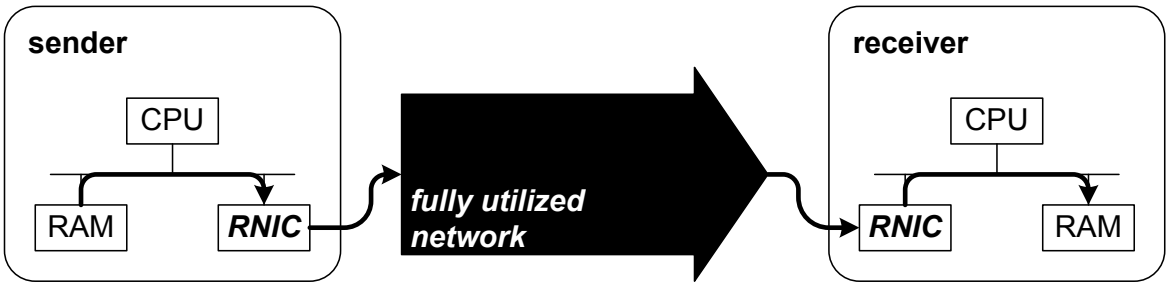
In order to fully bypass the OS and to avoid temporary buffering and associated memory bus transfers (as it is done in sockets), the RDMA programming interface requires applications to explicitly manage their communication buffers. Using RDMA's clear communication buffer ownership rules, the application temporarily passes control over its buffers to the RNIC for direct data placement. This allows for a true zero-copy data transfer while avoiding all in-host copy operations. Only such zero-copy architectures can deliver the entire available network bandwidth up to the application without causing heavy local traffic on the memory bus and CPU as the link speed of the interconnect technologies available today (i.e., 10 Gigabit Ethernet [inta]) have caught up with the memory bandwidth available in modern architectures. While initially based on proprietary network technologies such as InfiniBand, the advent of TCP/IP-based RDMA, called iWARP [RB03, RMTB05] (see Chapter 3), and the standardization of RDMA APIs make RDMA suitable for legacy applications.

With iWARP/RDMA, we hence have a suitable communication mechanism that allows us to fully utilize the high-speed interconnect while inducing only a minimal overhead on the end-hosts (illustrated in Figure 2.5). This is mainly attributed to the avoidance of intermediate copies thanks to the direct data placement and OS bypassing capabilities which not only require almost no CPU cycles during data transfers but also significantly reduce the memory bus traffic. As most systems, also RDMA is by no means perfect. We will carefully examine its practical limitations in Chapter 4.

In the following, we are going to discuss related research projects focusing on host-efficient network communication before we discuss in detail how RDMA can be enabled on the ubiquitous Ethernet using the iWARP protocol stack and illustrate the application programming interface offered by iWARP/RDMA (Chap-



(a) CPU-driven, host-local data copying induces an overhead which prevents the hosts from fully utilizing the available link bandwidth.



(b) Thanks to the direct data placement mechanism provided by RDMA-enabled NICs (RNIC), the network can be fully utilized with minimal overhead on the end-hosts.

Figure 2.5: Avoiding host-local overheads like intermediate copying is key to fully leverage the bandwidth provided by high-speed networks.

ter 3).

2.3 Related Work

This section presents work related to network optimizations in general and RDMA in particular. We start with a selection of interesting, early approaches which aim at reducing network communication overhead in general and the cost of TCP in particular. Among others, they have led to what is known today as the *Virtual Interface Architecture* (VIA), a specification describing user level access to network resources. InfiniBand is an early implementation of these VIA principles and is hence discussed subsequently followed by iWARP which implements the VIA principles on the IP suite. The work related to the individual application use cases presented in Part II of this thesis can be found in their respective chapters.

2.3.1 TCP/IP Optimizations

Various optimizations of TCP/IP based communication were proposed after Clark et al. [CJRS89] observed and documented some fundamental performance limitations. The limitations they found were mainly due to data touching operations such as checksum calculations and data copying for large transfers rather than the protocol processing itself. Kay and Pasquale [KP96] argued in their work, that the cause of the TCP/IP overhead depended on the size of the data to be exchanged. For small messages—typically used for control messages targeting low latency—the non-data touching operations, such as protocol processing, consumed a majority of the total processing time [KP93] while on larger data transfers, most of the overhead was caused by checksum calculations and intermediate copying which hindered a high throughput.

Efficiency in communication can be achieved through various design features [WM87, ST93, Dav91]. The three main options are to

- optimize the processing of the protocol architecture [Bla96],
- optimize the OS support for the data transport [DP93] and
- careful utilization of hardware acceleration and function offloading [DWB+93].

Several suggestions were made to reduce the number of times, application data had to be accessed [DAPP93]. On the transmit side, the copy-on-write scheme was proposed [DWB+93] to reduce data touching: when a program wanted to send data, the system set the state of the memory pages to read-only until the network interface had completed the data transmission. The data was directly read from its original location in memory (without intermediate copying). If, however, the

application wrote to these pages, the memory manager interrupted the program and copied the data into a temporary buffer in order to guarantee correctness of the transfer. This approach required changes to the memory manager of the operating system. Furthermore, the application had to be aware of this mechanism in order to achieve good performance—it should not write the pages while they were being transmitted. Also, this mechanism was limited to the transmit side even though the larger overhead typically incurs on the receiver side.

Chase et al. [CGY01] discussed a number of TCP optimizations which allowed for an overhead reduction on both sides. In terms of end system optimizations, they distinguished between per-packet and per-byte overheads. To reduce the former, they proposed extended frames (larger packets) and interrupt coalescing to amortize the packet handling costs across multiple packets. The latter was tackled with zero-copy networking based on the copy-on-write approach in combination with page remapping. Furthermore, they offloaded the checksum calculation to the network interface controller and pursued an integrated copy/checksum calculation strategy. The shortcomings of their system, especially with respect to the zero-copy aspect were that the maximum transmission unit (MTU) of the underlying network had to be equal to the page size used. Furthermore, the application buffers were required to be page aligned so that the network interface could deposit inbound packet data at page boundary.

In order to reduce the TCP/IP overhead [CGY01, KP96], concepts were developed to avoid redundant copies not only on the operating system [kJC96, ST93], but also on the application level [RAC97, DIFL96] as well as on the network interface [DWB+93, LC95, vEBBV95, PF01]. In terms of the network interface, various systems have been built and were attached to the host on different buses [LC95]. The SUN SAHI, Fore SPA-200, Myrinet LANai as well as the IBM SP2 were attached to the I/O bus of the host and therefore quite far away from the CPU. Others like the Cray T3D, the Meiko CS-2 and the Intel Paragon were directly coupled to the memory bus and thus much closer to the processor. The HP Medusa and Afterburner are examples of controllers attached to the graphics bus. Detailed discussions on the host interface attachment can be found in the work by Smith et al. [ST93] or by B. Davie [Dav90].

The *Afterburner* network card [DWB+93] which supported a single-copy stack at rates up to 1 Gbps was proposed by Dalton et al. The idea behind their single-copy mechanism[3] was to have a dedicated area of memory on the card itself which was shared between the processor and the network interface. The communication between the OS and the card happened through FIFO queues. A big advantage of Afterburner was that it could offer performance improvements for the sender as well as for the receiver without changing the application interface.

[3]We would refer to this mechanism as zero-copy in today's terminology.

A different approach to address the copy overhead issue with respect to high bandwidth I/O was taken by Druschel and Peterson [DP93]. They proposed a facility termed *fast buffers* (or *fbufs*), which combined virtual page remapping with shared virtual memory thereby enabling an efficient cross-domain buffer management. In contrast to the Afterburner, the (shared) data buffers were located in the main memory of the host and not on the NIC. Remapping pages, however, required several physical page table updates which could result in substantial overhead.

Blackwell [Bla96] suggested a technique to improve protocol performance for protocols that used small messages rather than bulk data transfers. He argued that the main overhead incurred with small messages was due to poor locality in protocol processing code which meant that the processor spent more time loading protocol code from memory than moving packet contents. To mitigate that problem, he proposed to reschedule the protocol layer processing based on a (generally applicable) locality driven layer processing scheme (LDLP).

Menon and Zwaenepoel [MZ08] reviewed the TCP performance issues with regard to small messages for recent systems that feature not only uniprocessors but also Symmetric Multiprocessing (SMP) environments and virtualized hosts. According to their findings, reducing the communication overhead for small messages becomes even more important with recent systems. Their work illustrated the effects of new hardware on the per-packet overhead and suggested optimizations for the receive path. The sending side was not considered.

Due to the different pace in the evolution of processors and the host bus system, particularly the memory bus, faster processors will not be able to solve the TCP issues as documented by Markatos [Mar02]. As TCP/IP performance does not scale linearly with processor speeds, communication abstractions which are not based on sockets have to be considered.

2.3.2 User Level Networking

Already in the early 90s, various research groups advocated the so-called *user level networking* principle [Dav91, ST93, vEBBV95] for systems then termed *clusters of workstations*.

The goal of user level networking was two-fold. First, it aimed at offloading the packet processing tasks to dedicated devices in order to save compute resources on the host which were scarce at that time. Second, user level networking promised to lower communication latency considerably compared to socket implementations. Common arguments for implementing network functionality in user space included increased flexibility, easier maintenance, and the possibility to allow optimizations which were specific to the applications and could improve their performance [TNML93, EM95].

The above goals of the proposed user level communication were tackled by

avoiding system calls, allowing the network interface controller to directly access the user application payload in memory and by improving the notification process on reception of network data as well as on completion of data transfers. These techniques are today summarized as *kernel-bypassing*: the OS is only involved in control tasks and taken out of the critical data path (the data follows a so-called *fast path*).

Von Eicken et al. [vEBBV95] built upon the user level networking idea and suggested U-Net, a communication architecture built on off-the-shelf communication hardware that provided processes with a virtual view of the network interface. In their system, every application was given the illusion of having its own, private network interface. They argued, that the entire protocol stack should be placed at user level and that the operating system and hardware should allow protected user level access directly to the network whereby the kernel must be removed from the critical path and the applications were allowed to customize their communication layers as needed. All buffer management and processing tasks were thus moved to user space which enabled data to be sent directly out of the application data structures. U-Net offered support not only for traditional inter-networking protocols but also for novel communication abstractions like Active Messages [vECGS92]. The basic idea behind Active Messages was that each message had a header containing the address of a user space handler to be executed upon message arrival. The contents of the message would then be passed as an argument to the handler thereby exploiting the performance and flexibility of modern interconnects.

In user level networking, systems calls were often avoided by offering hardware support for user level memory-mapped message passing. Systems that had incorporated this technique included the Connection Machine CM-5 supercomputer [LAD+92], the MIT J-Machine multicomputer [NWD93], the MIT Alewife [ABC+98] which was a large-scale multiprocessor that integrated both cache-coherent, distributed shared memory and user level message-passing in a single integrated hardware framework and last but not least the Cray T3E [ABGS97]. All of them either offered device interfaces located in memory or dedicated memory-mapped processors for performing the message passing. Joerg et al. [HJ92] proposed an improvement on the memory mapped network interface design through a tighter, more optimized coupling of network and processor by utilizing processor-mapped registers. The motivation behind it was the following: in order to send a message, the processor had to execute a series of store operations to the memory mapped network interface and to receive a message, a series of load operations was required. These load and store operations could be eliminated with processor-mapped registers.

The Hamlyn interface architecture [BJM+96] proposed by HP labs, described another efficient interface between multicomputer interconnection fabrics and the

host processors. It used sender-based memory management to eliminate receive buffer overruns and software-induced packet loss, provided applications with direct hardware access in the user level networking spirit and offered protection between applications running on a host. The setup consisted of standard HP workstations connected through a Myrinet interconnection fabric.

Other projects used the processor address translation mechanism to enable transmission of physical addresses from user space to the network interface thereby allowing direct memory access (DMA) transfers of the payload data to and from memory [Dav91]. This allowed for larger data transfers while preserving memory protection. In those systems, the network interface typically had a (number of) shared message queue(s) through which data transfer requests were communicated from the application to the NIC. The messages on the queues contained at least the start address of the data as well as the length.

Examples of this are the SHRIMP [BDFL96, DIFL96], FLASH [KOH+98] or Tempest and Typhoon [RLW94] projects. The SHRIMP (Scalable High-Performance Really Inexpensive Multi-Processor) project from Princeton aimed at building high-performance servers from a network of commodity PCs and commodity operating systems. As its name suggests, the primary motivation was that such a system was significantly cheaper than a custom-designed multicomputer. As a novel way of communication, Dubnicki et al. [DIFL96] suggested in the context of the SHRIMP project to map remote memory into the local application address space thereby enabling direct data transfers between the sender's and receiver's virtual address spaces. Before such a data transfer could take place, the receiver had to export a region of its memory which the sender had to import. Support from the hardware side as well as from the OS was required. The data transfers within the hosts were realized through DMA operations. The Stanford FLASH (Flexible Architecture for SHared memory) was a single-address space machine consisting of a large number of processing nodes connected by a low-latency, high-bandwidth interconnection network. FLASH utilized a custom-designed node controller to achieve cache-coherent shared memory and low-overhead user level message passing. Tempest was a novel interface proposed by Reinhardt et al. that exposed low-level communication and memory-system mechanisms allowing application performance improvements. These mechanisms were implemented by the Typhoon hardware platform which incorporated a fully-programmable user level processor in the network interface.

A large part of the efforts for tighter and more efficient integration of the network interface with the processor(s) presented above, eventually resulted in the *Virtual Interface Architecture* (VIA) [CCC97] specification jointly proposed by Compaq, Intel and Microsoft in December 1997.

2.3.3 The Virtual Interface Architecture

Clusters of standard servers were becoming cost-effective alternatives to expensive, large-scale mainframes. In order to achieve a good overall performance, efficient communication between these servers was crucial. Through the Virtual Interface Architecture (VIA) [CCC97], the industry had provided a standard for efficient cluster communication. VIA's main concern was the interface between the network and the application(s) running on top of it.

In the time before VIA, various System Area Networks (SAN) were developed for building high performance clusters. The interfaces to these SANs were proprietary and unique, however. The lack of standardization limited the number of applications that were developed for them. On the other side, system designers started to use Ethernet as basis for building cluster networks due to its standardization, wide availability and relatively low cost. Unfortunately, Ethernet failed to realize the performance potential of the underlying network hardware. Cluster computing therefore faced a trade-off between performance and standardization. VIA aimed at filling this gap while being independent of the underlying network infrastructure, processor architecture and operating system [CR02].

Design challenges of virtual networks as described by the VIA specification were addressed by Mainwaring and Culler [MC99]. The work investigated the effectiveness of network virtualization at scale and demonstrated that it was feasible but challenging because virtual networks required dealing with the interaction between layers of the system and across nodes in the network. In that context, they illustrated design considerations with regard to naming and protection, delivery and error model as well as communication events and threads from an application and OS perspective.

VIA Communication Principles

In the following, we briefly outline the system principles as defined in the Virtual Interface Architecture Specification.

Like many of the research projects mentioned before, VIA aimed at reducing the magnitude of the software overhead incurred during network data exchanges. A fundamental design principle is the bypassing of the OS kernel on the performance critical data path. Furthermore, as the name suggests, VIA provides the applications running on top of it with the illusion of having a dedicated network interface. In terms of the user level spirit, VIA provides the user application with the facility to create and directly manipulate communication endpoints (VI's).

Figure 2.6 shows a high level perspective of the VIA architecture. In contrast to the socket abstraction where all communication functionality is provided by the OS, VIA splits the functions between a *VI consumer* (the users of the VI) and the

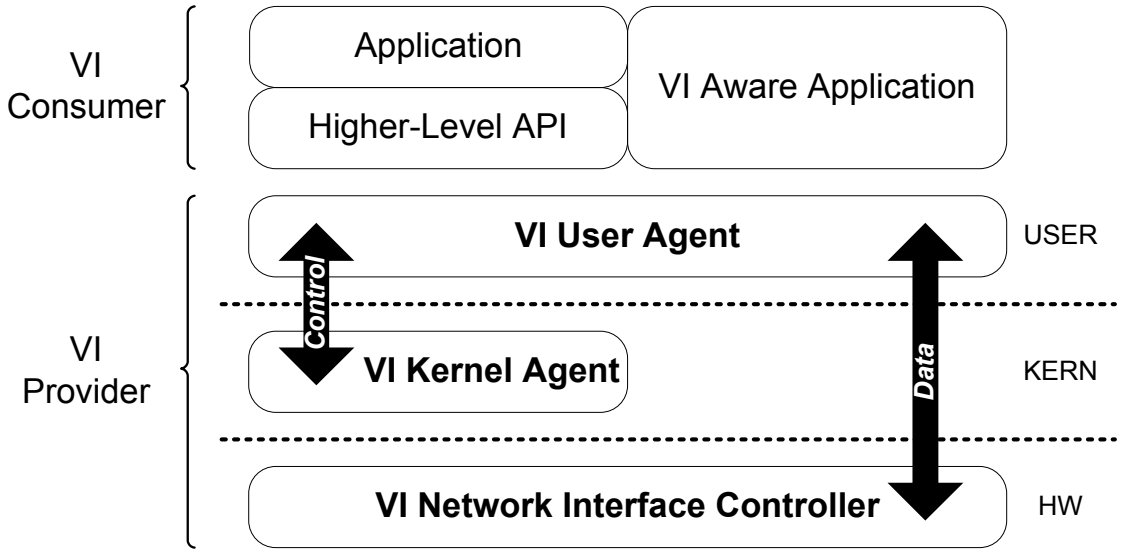


Figure 2.6: VIA model for OS bypassing. The specification distinguishes between the *VI consumer* (application) and the *VI provider* (network subsystem). Data is transfered on a fast path bypassing the OS.

VI provider (the kernel implementation of the Virtual Interface Architecture as well as the hardware). The VI provider is responsible for the protected sharing of the network controller, virtual to physical buffer address translations and the like. Furthermore, it provides a reliable transport service. Within the VI provider, the control path (to allocate or change the state of resources) is separated from the performance critical data path which bypasses the OS kernel completely.

An important design decision in that respect is that in the VIA, the buffer allocation and management is performed by the user rather than by the OS. The user must explicitly register virtual Memory Regions which can thereafter be used as buffers for transmitting and receiving data. As the network interface controller accesses these user buffers through DMA operations (for zero-copy data transfers), the virtual addresses have to be translated to physical ones. Furthermore, the underlying pages must be pinned to prevent the data from being swapped out to secondary storage in case of memory pressure—this is a major differentiator compared with TCP/IP stacks.

VIA is designed with an asynchronous communication model between the application and the network controller (Figure 2.7). A virtual interface consists of a pair of Work Queues (WQ): a Send Queue and a Receive Queue. VI consumers post Work Requests (WR) to these queues to send and receive data. Such a Work Request contains all the information needed by the VI provider to execute the data transfer (i.e., address of the communication buffer, offset, length of the data

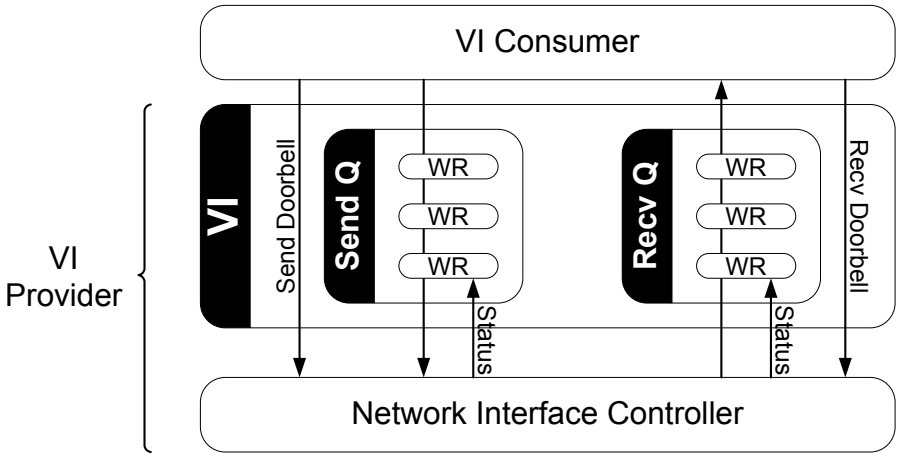


Figure 2.7: The VI consumer posts data transfer operations in terms of Work Requests (WR) onto queues managed by the VI provider. Doorbells are used to notify the NIC about new WRs being present on the respective queues.

transfer as well as a buffer identifier). The VI provider asynchronously reaps WRs from the WQs, processes them and updates their status upon completion. The consumer thereafter removes the completed WRs from the queues. Each queue has an associated Doorbell through which the consumer notifies the provider that new WRs have been posted to the queue.

The WRs can be used to issue one of the following communication operations:

Send/Receive: The *Send* and *Receive* operations are well known operations from message passing systems. Each *Send* operation must have a matching *Receive* operation at the remote end. In VIA terms, these operations are called *two-sided* because the data exchange naturally involves both ends of the communication channel. A *Send* WR specifies where the data should be taken from (on the local machine) and the *Receive* WR on the remote machine specifies where the inbound data is to be placed.

RDMA: On the other hand, there are the *one-sided* Remote Direct Memory Access (RDMA) operations including *RDMA Write* and (optionally) *RDMA Read*. For these RDMA operations, only the application issuing the operation is actively involved in the data transfer. At the remote end, the data is placed by the NIC without involvement of the application logic. An *RDMA Write* WR, for instance, therefore not only specifies where the data should be taken from (locally) but also where it is to be placed (remotely). RDMA operations require a buffer advertisement prior to the data exchange.

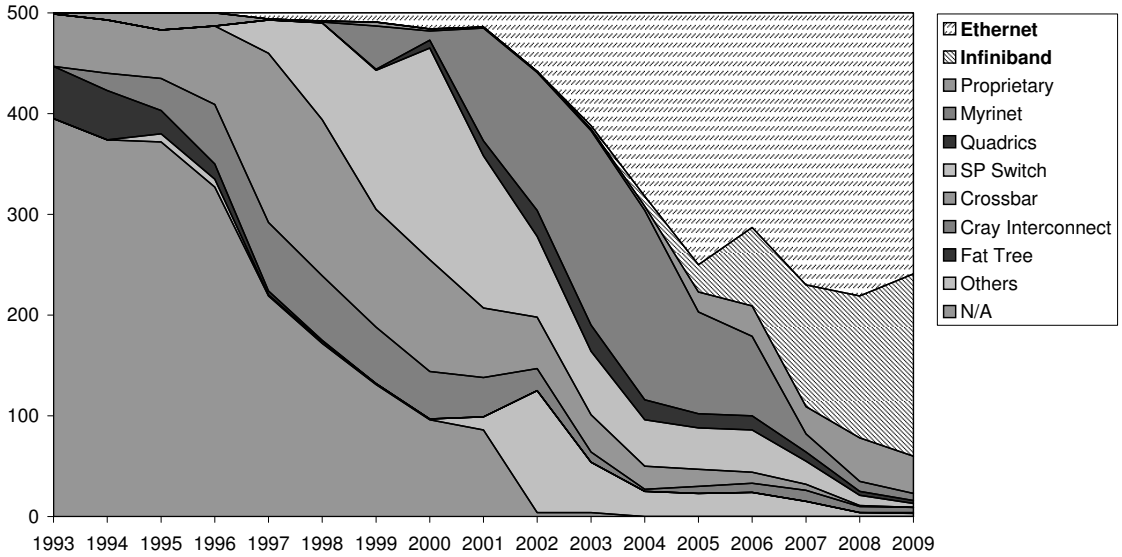


Figure 2.8: Top500 Supercomputer Sites—Interconnect family history. Today, Ethernet has a share of over 50%, followed by InfiniBand with slightly more than 36%.

VIA Implementations

The Virtual Interface Architecture is an abstract specification that does neither define an exact API nor any implementation details with respect to the verbs provider—these are left open for the vendors.

As of today, there are two predominant implementations of the VIA specification: the proprietary InfiniBand and the Ethernet-based iWARP (see Figure 2.8). In the following, we briefly outline some implementation aspects of these two. The rest of this thesis will then focus and be based on iWARP only. More details on iWARP, its API and enablement options are presented in the next chapter.

InfiniBand. InfiniBand (IB) is the result of merging two competing connection standards, Future I/O by Compaq, HP as well as IBM and Next Generation I/O developed by Intel, Microsoft and Sun Microsystems. The InfiniBand Trade Association (IBTA) [ibt] maintains the InfiniBand Architecture (IBA). The first IBA specification was released in 2000. InfiniBand is based on a point-to-point switched I/O fabric and intended for connections within systems (module-to-module) as well as for data center environments (chassis-to-chassis). It provides both, reliable messaging (*Send/Receive*) as well as RDMA operations (*RDMA Write*/*RDMA Read*). Its main application is as a low latency, high bandwidth interconnect in data centers and High Performance Computing (HPC) environments today. Because IB was designed from ground up, it implements state-of-the-art mechanisms which make it the current leader in terms of network bandwidth and communication la-

tency. A side effect of this approach is that the specification seems over-engineered in various aspects—a lot of the features are defined as optional and hardly ever implemented in practice. InfiniBand specifies a set of *verbs* [HCPR] that describe the semantics of the interface without defining a precise API (this is again left to be defined by the vendors).

iWARP. With the recent advent of 10 Gigabit Ethernet [inta], this low-cost, switched-fabric interconnect used primarily for general purpose communication and storage networking is advancing into the HPC space. Ethernet was originally defined in 1975 running at 1 Mbps over copper. During the last 20 years, its bandwidth has steadily increased to 100 Mbps, 1 Gbps and 10 Gbps. 40 Gbps Ethernet is on the horizon and a standard for 100 Gbps is already under development. Even though it cannot quite offer the performance of proprietary interconnects like IB or Myrinet [RA07], its standards-based interface, support for legacy Ethernet fabrics and lower cost provide significant customer benefits. IB and Myrinet on the other hand require new infrastructure to be developed, deployed and managed [Rec03].

The Internet protocol (IP) suite defined by the Internet Engineering Task Force (IETF) is ubiquitous today. The IP suite (commonly known as TCP/IP) specifies management, network and transport protocols. Recently, the IETF has specified a set of companion protocols for *RDMA communication over TCP/IP* [RMTB05]: *MPA* [CER+07], *DDP* [SPRC07] and *RDMAP* [RMC+07]. The topmost protocol, RDMAP, defines the high-level semantics. The DDP protocol then constitutes how the RDMA payload is to be tagged, transferred to and placed at the communication buffers of an application (zero-copy). MPA finally serves as an auxiliary layer that transports the discrete DDP packets over a TCP stream. This so called iWARP[4] stack enables RDMA connectivity over low-cost, Ethernet-based network infrastructures. Think of iWARP as Ethernet extended with the features of InfiniBand.

Based on the economies of scale, multi-gigabit Ethernet [inta] together with iWARP have the potential to become the unifying interconnect in the data center, relegating proprietary technologies such as InfiniBand, Myrinet, and Quadrics[5] to niche markets [RA07]. The prices for previous Ethernet adapters have been dropping fast and it is probably safe to expect the same for the 10 Gigabit models. Major server vendors already include 10 GbE per default. Also the prices per switch port are tumbling. In terms of physical connections, optical standard interfaces, 10GBase-T (i.e., Cat5 and RJ45) as well as CX4 are currently available leaving the choice to the customers. Adaptation to existing infrastructure compo-

[4]The provenance of "iWARP" is controversial. One convincing explanation is to read it as an acronym for "Internet Wide Area RDMA Protocol".

[5]Quadrics fell victim to the global recession in June 2009
http://bits.blogs.nytimes.com/tag/quadrics

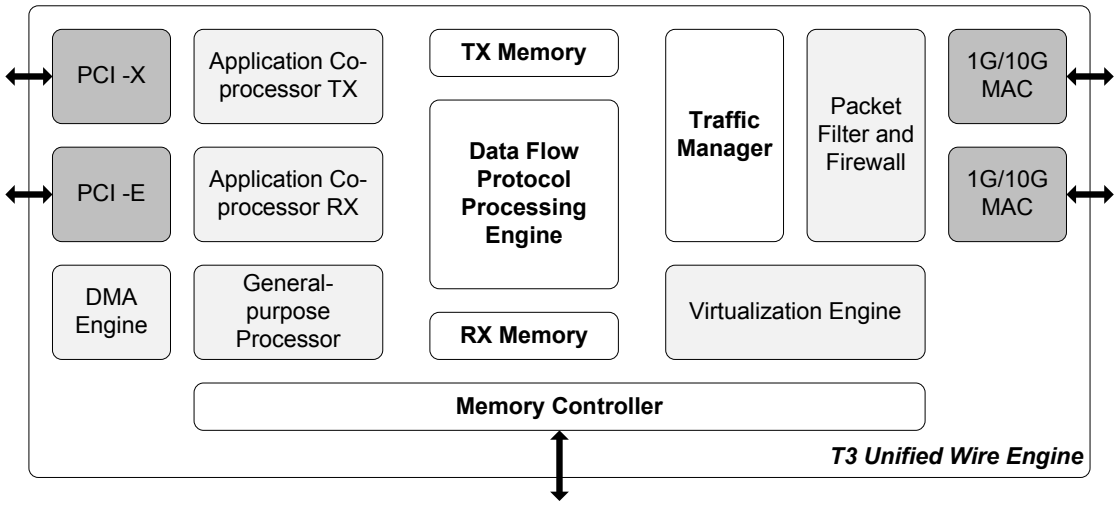


Figure 2.9: T3 Unified Wire Adapter by Chelsio Communications. It is built around a data flow protocol processing engine with high bandwidth external memory interfaces.

nents is thus ensured. The long term vision of iWARP is to be able to provide a fabric convergence (*unified wire*) across data centers. While traditionally, different interconnects were used for LAN/NAS, SAN and HPC, moving everything to Ethernet has a number of advantages: it simplifies data center wiring (no gateways, single switch type, etc.), lowers maintenance and operating costs (single admin, fewer licenses, etc.) and still provides a high enough performance. Another very promising property of 10 Gigabit Ethernet is its extended operating distance of 40 km, which allows a complete rethinking of physical data-center layout and its integration with the Internet. A thorough comparison between InfiniBand and iWARP in the context of server I/O networks and data center consolidation was presented by R. Recio [Rec03].

Some vendors already offer complete 10 Gigabit iWARP hardware solutions (e.g., Chelsio Communications or NetEffect[6]), termed RDMA-enabled NICs or *RNICs*. While these adapters follow the VIA specification (zero-copy based data transfers on RDMA), they can also be used as simple, high bandwidth Ethernet adapters which are fully compatible with legacy Ethernet components. An RNIC offers full offload of the whole protocol stack (RDMAP all the way down to Ethernet) and thus essentially consists of an extended TCP offload engine together with a DMA engine [Mog03]. Figure 2.9 shows the internal block level architecture of the Terminator 3 Unified Wire Engine by Chelsio[7] which was used throughout this

[6]NetEffect was bought by Intel in October 2008.
[7]http://chelsio.com/unifiedwire_eng.html

thesis. We will discuss iWARP in detail in Chapter 3.

The idea of combining the VIA principles with the IP suite was proposed by Buonadonna and Culler [BC02]. Their system, called Queue Pair IP, implemented Queue Pair operations over a subset of TCP, UDP and IPv6 protocols on a programmable NIC. They could show that QP-based communication over IP has a potential similar to InfiniBand but would run on a legacy infrastructure. With that, they provided a proof-of-concept for the iWARP approach.

2.3.4 iWARP/RDMA Applicability

Research, and later also the industry, have largely focused on the enablement of efficient user level networking in general and iWARP/RDMA in particular. However, there are still very few real applications out there that directly leverage the performance potential. As we will illustrate in the following chapters, iWARP/RDMA is best suited for applications that operate on large amounts of data such as real-time high-definition video dissemination (see Chapter 6), digital image retrieval or applications which are accessing and processing large scientific data sets (see Chapter 7).

As a consequence, research has focused a lot on performance evaluations of these new RDMA network interface controllers [DW05,BcFB+07,DWM06,FBB+05]. Liu et al. have also investigated their impact with respect to power consumption [LPA09] and could show that RDMA requires less power than comparable TCP/IP-based communication. A thorough study and comparison of different high-performance networks was presented by Bell et al. [BBC+03].

RDMA Application Obstructions

Even though the performance improvement, overhead reduction and power efficiency have shown to be promising, there are a number of reasons for the limited application of RDMA so far. The most important issue is the lack of a generally accepted API. Today, there are two competing proposals: the Interconnect Transport API (IT-API) [The] proposed by the Interconnect Software Consortium which is part of the Open Group and the OpenFabrics API [ofe] proposed by the OpenFabrics Alliance which stems from Open InfiniBand. Currently, the OpenFabrics API seems more popular because, unlike the IT-API, it supports RDMA over InfiniBand *and* iWARP. Furthermore, it has become part of the vanilla Linux kernel. Also, the OpenFabrics Alliance receives significant support from and is collaborating with important OS vendors (Novel, Red Hat and Microsoft) and leading chip manufacturers.

Another obstacle for a broad acceptance of RDMA is the API itself. It is not only radically different from the socket interface but also much more complex to

program (see Section 3.3). This implies that enabling an application for RDMA is coupled with a significant amount of non-trivial recoding. There is no straightforward mapping from socket-based to RDMA-based communication because of the explicit communication buffer management required by RDMA (cf. Chapter 5).

To mitigate the problem of application transformation, the InfiniBand Trade Association has proposed the *Sockets Direct Protocol* (SDP). It provides a standard wire protocol to support the socket abstraction over RDMA and thus eliminates the need for application redesign. While it was originally limited to InfiniBand, it has become transport agnostic and hence can also be used on top of iWARP. As was shown by Balaji et al. [BNV+04], SDP improves the performance compared to standard TCP sockets but is not able to realize the full potential of InfiniBand. The applications running on SDP are not RDMA-aware and thus utilize their buffers in a way which is not optimal for RDMA—we will discuss this in detail in Chapter 4. In a follow-up paper, Balaji et al. [BBJP06] propose *Asynchronous Zero-Copy SDP*, a mechanism with which allows the approaches for asynchronous sockets to be used with ordinary (synchronous) sockets. This, however, works only with InfiniBand because their design relies on atomic remote operations which are not supported on iWARP.

A third reason for the lack of applications is that InfiniBand requires special, expensive equipment and 10 Gbps iWARP RNICs have not been commercially available until recently.

Explicit Memory Registration

A fundamental difference between user level networking and sockets is the explicit buffer management. For socket-based communication, the kernel provides intermediate buffers to store the communication data. In the case of RDMA, the user application itself is responsible for providing the necessary buffers. As we will see in Chapter 4, the way these buffers are managed has a significant impact on the overall application performance. Furthermore, not every application can be tailored such that it can realize the potential of RDMA due to the overhead of registering Memory Regions for holding the buffers. Based on our findings, we will present some optimizations in terms of buffer management and memory registration (Chapter 4).

The fact that RDMA requires an explicit communication buffer management has been identified as a drawback before. A detailed analysis of the memory registration cost in the Mellanox InfiniBand software stack was presented by Mietke et al. [MRB+06]. Even though it was not iWARP and not based on the OpenFabrics RDMA stack (as in our case), the issues were similar. Unlike in this work, however, the only optimization put forward was to use large pages which resulted in a cost reduction of 15%. Arbitrarily increasing the page size is not suitable in many

contexts and has nonnegligible side effects. Our proposed solution (Chapter 4) is less intrusive and over all more effective. Also Nieplocha et al. [NTSP02] have documented the memory (de-)registration overhead and even suggested to stream data via preallocated registered memory buffers in a pipelined fashion. However, they focus on small data sets only and do not take memory bus limitations into consideration. Furthermore, their pipelined streaming is not generally applicable as it requires dedicated threads and processes on the communicating hosts as well as a custom protocol between them.

Bell and Bonachea [BB03] proposed a novel DMA registration strategy termed Firehose. They aimed at using one-sided RDMA operations in the general case which might necessitate the registration of a significant portion of the total physically available memory. Their approach targeted Global Address Space (GAS) languages based on a globally-shared memory across clusters. While their experiments were based on Myrinet, they also hinted at other similar approaches based on Quadrics. Firehose seems well-suited for the GAS case but lacks general applicability.

The idea to deregister buffer memory lazily was proposed by Tezuka et al. [TOHI98]. Upon receiving a deregistration request from the application, the subsystem would not actually remove the registered segment but keep it in a cache to reduce future buffer registration costs. The memory was unpinned only when the cache exceeded a certain limit. Also Ou et al. [OHH09] propose a sophisticated cache for registered Memory Regions that is slightly more efficient than the Pin-down Cache by Tezuka et al. Most popular RDMA subsystems as well as the Linux kernel do not support MR caching yet. Even though it sounds like a good idea, kernel support is required to keep the cache consistent since the user has access to a variety of operating system calls to alter the memory layout and thereby potentially destroying earlier cached registrations. A detailed description of the issues can be found in the work by Wyckoff and Wu [WW05]. Our proposed optimizations do not require a modified kernel and are therefore more generally applicable.

Woodall et al. [WSBM06] propose a pipelined memory registration approach which is application agnostic. They propose to split large messages into a number of small ones. For each message, an individual Memory Region is registered on-demand before the data transfer. If this is done in a pipelined fashion, the registration process can be hidden behind the actual data transfer. However, the depth of the pipeline is bound by the number of available network adapters because each adapter can only handle one registration at a time—at least two adapters are thus needed to lower the overall registration latency. Furthermore, as we will illustrate in Section 3.4, RDMA performs best if the buffers exchanged are large contradicting the split approach. The performance of this approach (even if many network adapters are available) suffers when the data to be exchanged is small, due to the

constant registration overhead (Chapter 4).

An important detail of the memory registration is that, once registered, each MR has a fixed size. This is particularly critical on the receive side of a connection where the Receive Work Queue elements are consumed from the Receive Queue (RQ) in *FIFO order* by the RDMA adapter—they are not matched to the appropriate size. Thus, a data transmission fails if an inbound *Send* is too large for the next pending Receive Work Queue element. Shipman et al. [SBB+07] have suggested a combination of the Shared Receive Queue (SRQ) mechanism and several parallel connections to mitigate this problem. While their approach allows the sender to choose the size of the remote receive buffer, it requires several simultaneous RDMA connections for a single, logical application channel which severely limits the scalability for upper level applications. Last but not least, with multiple connections in parallel, the message ordering guarantee is lost which might lead to race conditions.

Existing Applications

The iWARP/RDMA application domain is fairly narrow to-date. There are, however, some experimental implementations and even a small number of proposals for protocol extensions to leverage this novel networking facility on a broader scale.

Storage. Despite RDMA being designed for offering I/O overhead reduction in terms of *memory* access, there exist a number of suggestions for applying the RDMA/VIA principles and features to network-attached storage applications operating on to *disks*. In their work, Dalessandro and Wyckoff [DW07b] discussed a number of alternative approaches for shipping data—stored in files—using RDMA. In particular, they pointed out the issue of bringing the data from disk into a user buffer only to transmit it again through the OS kernel. They found that the most efficient way for transmitting data was to apply the kernel sendfile mechanism in a pipelined fashion.

Callaghan et al. [CLRC+03] proposed an RDMA extension for the Network File System (NFS) version 4 [SCR+03]. They suggested a subtle enhancement of NFS with RDMA in a way which did not require changes in the applications running on top of it: RDMA was to be used as the transport for the Remote Procedure Call (RPC) messages which are ubiquitous in NFS. They distinguished between large (data) and small (control) messages. As the direct data placement functionality is only beneficial for large messages, they used conventional send operations for control message exchanges. Noronha et al. [NCTP07] have recently documented shortcomings of the original proposal and suggested some optimizations focusing on the memory management as well as the NFS RPC control message sequence.

Another proposal in the area of Network Attached Storage (NAS), that tried to leverage user level networking features was the Direct Access File System (DAFS)

by Magoutis et al. [MAF+02]. The file system was implemented in user space rather than in kernel for the same arguments brought forward by the VIA community: flexibility, ease of maintenance and customization potential for individual applications. In their paper, Magoutis et al. provided an extensive comparison between DAFS and NFS and were able to show low latency, good bandwidth utilization and low CPU overhead through a number of experiments and benchmarks. In a later study [MAFS03], Magoutis et al. proposed an alternative to their original DAFS which could offer the same performance but was simpler in design (however not as portable). They presented optimizations to RPC-based data transfers by using RDMA and pre-posting of application receive buffers. Furthermore, they suggested Optimistic RDMA (ORDMA) as an alternative to RPC in order to improve throughput for small messages. By extending DAFS with ORDMA, they could again improve the performance of the system. While this optimization showed an improved performance for small RPC I/Os due to the reduced response time, it breaks compliance with the RDMA verbs on which the industry has agreed.

In the space of Storage Area Networks (SAN), providing a block abstraction to clients, iSCSI [SMS+04] has emerged which provides the hosts with the illusion of having disks locally attached where in reality they are distributed across an IP-based SAN. Recently, the IETF has proposed a standard for improving iSCSI performance with iWARP/RDMA in their iSCSI extension for RDMA (iSER) [KCH+07, DDW07]. The motivation was to utilize iWARP as a transport for iSCSI in order to achieve direct, in-order as well as out-of-order placement of SCSI data into pre-allocated buffers while maintaining in-order data delivery. Similar to the NFS over RDMA approach presented before, iSER has also utilized *RDMA Read* and *RDMA Write* operations for large data transfers and send operations for control information.

A radically different suggestion for SAN over RDMA was brought forward by Narasimhamurthy et al. [NGSH05]. They proposed the Quanta Data Storage (QDS), a novel architecture for SANs, as an alternative to iSCSI and argued that the iSCSI and iSER stacks had grown too large with the consequence that some of the compute intensive functionality (e.g., checksum calculation) was duplicated and the header overhead had become significant. Therefore they collapsed the whole stack into their Effective Cross Layer (ECL) which incorporated ideas from iSCSI, iWARP and TCP but was optimized for data storage. Their main principle was to deal with the data in fixed blocks (quanta) rather than treat them as a byte stream which seemed more natural for storage class applications. In contrast to the other solutions, they implemented their ECL completely in software. An interesting design decision was to create an asynchronous protocol in which most of the compute intensive tasks were performed on the client in order to take load

off the server. In Chapter 5, we have pursued a similar approach and designed an asynchronous, RDMA-based protocol for distributed source code compilation.

High Performance Computing. Another prominent use case for RDMA is the Message Passing Interface (MPI) ubiquitously used by large-scale scientific applications running on clusters [VM03, SBM+05, TG03, BSL07, LWK+03]. We will not go into details of MPI over RDMA here, though.

Java. Today, a substantial fraction of the applications are written in Java. Therefore, also the Java community has shown interest in efficient, high performance communication promised by the user level networking principles. The need for Java InfiniBand support in particular and low overhead, high performance communication in general was expressed by Zhang et al. [ZHH+07]. Java can only compete with C/C++ in the HPC market if it offers support for efficient I/O.

However, Java is not a good fit for the VIA paradigms. First of all, Java is not designed for explicit user level buffer management. There is, for instance, the garbage collector which frees buffers which are no longer in use. With the user level networking principle, however, a buffer might not be in use by the application but the ownership might have been transferred to the underlying NIC—garbage collector modifications would be required to handle that case. Furthermore, due to the indirect buffer layout used in Java VMs (JVM), the data is (in most cases) not directly accessible by the NIC—a fundamental assumption of the VIA principles. So the fundamental challenge in combining Java and RDMA is how and where to manage the buffers. The two apparent options (that do not require changes to the JVM) are:

- Keep the buffers on the Java heap. This strategy allows seamless and efficient access for Java applications but due to the indirect nature of the internal Java buffer structure, they are not directly accessible by the NIC. The Java Native Interface (JNI) can be used to make the buffers accessible by the NIC. However, moving the data between the Java heap and the native code requires a copy for all but the primitive data types.

- Manage the buffers in the native code. In this scenario, the buffers are easily accessible by the NIC but Java applications must access them through JNI calls. We would face the same copy issue as before and are not transparent to the applications.

Baker et al. [BCS06] reasoned about the buffer management in the context of High Performance Computing in Java. While they did not discuss user level networking, their work suggested that an intermediate buffering layer for an efficient HPC messaging system in Java could be realized through direct buffers provided by the Java New I/O (Java NIO). Also, they highlighted the memory management

issues with regard to the Java garbage collector (constant creation and destruction of buffers is expensive). Their findings can directly be applied for efficient I/O in the communication context as well. The problem with the direct buffers, however, is that they are outside the scope of the Java garbage collector.

An early proposal for a VIA-aware RPC mechanism, called J-RPC, was presented by Chang et al. [CvE98]. In their work, they illustrated (un-)marshalling problems and pointer issues which stem from the indirect buffer management scheme of Java. Furthermore, they came to the conclusion that true zero-copy communication was only possible for arrays of primitive types unless the JVM was modified.

Despite all this, it is tempting to use the Java Native Interface to implement an API for user level networking. JNI allows a Java application to be extended with functionality which cannot entirely be implemented in Java. Unfortunately, the JNI requires a data copy for all but the primitive types which limits its zero-copy suitability as illustrated by Welsh and Culler [WC00]. The authors presented a detailed overhead comparison between the JNI path and comparable C code and showed the limitations of JNI. Therefore, they suggested Jaguar [WC00] as a remedy. Jaguar was a mechanism which offered efficient access to system resources for high performance I/O (not limited to communication) while preserving the protection of the Java environment. In particular, Welsh and Culler demonstrated a way to offer efficient user level networking for Java applications by translating Java byte code to inlined machine code sequences at compile time. Even though their experiments looked promising, they were limited to as little as a few 100 Mbps.

At the same time, Javia [CvE00] was presented. Javia was designed as a Java interface customized for the VI architecture (not a general I/O interface like Jaguar). Chang and von Eicken addressed the buffer management issues and suggested two approaches. The fist one used JNI and thus required copying. The buffers were managed in the native code. The second approach introduced a special buffer class in Java in combination with an extension of the garbage collector. While this approach required a custom JVM, it was able to avoid the intermediate copy step between the Java heap and the native interface.

Recently, Huang et al. have proposed the Java Direct InfiniBand (Jdib) [HZH+07]. Their solution aimed at exploiting the RDMA capabilities of InfiniBand by offering the IB verbs API to Java applications. In their work, they confirmed the difficulty of passing data between the Java heap and JNI. Nevertheless, they utilized JNI and even achieved decent performance. However, it was not quite clear how they have implemented the mapping between the VIA principles (direct data access) and JVM (indirect buffer scheme). Also, it was not clear what kinds of Java objects they have used for their experiments. This is vital because, as stated above, transferring primitive types can be done by reference in JNI; all the rest requires

copying.

Summary and Outlook.

So even though IB and iWARP promise a significant overhead reduction, improved latency and higher throughput due to the VIA principles, they require applications to be re-written in order to realize the full potential. In some cases, this is not possible or the performance benefit is not large enough to justify the effort. In the upcoming chapter, we will illustrate the iWARP host integration as well as the proposed API with its implications in detail. A thorough assessment of when the application of RDMA-based communication is beneficial is then provided in Chapter 4.

3

iWARP: RDMA over Ethernet

While we have hinted at RDMA over Ethernet (also known as iWARP) in the previous chapter (Section 2.3.3), we will now discuss what is necessary to ultimately enable an application for iWARP/RDMA. The approach taken to do this is bottom-up: we start by looking at the wire protocol and move up to the application level.

First, we will illustrate the iWARP protocol stack as proposed by the IETF. In that context, we will discuss our extension to the Wireshark [wir] network protocol analyzer which allows the inspection and dissection of iWARP traffic. Thereafter, we will look at the host system integration of the iWARP/RDMA subsystem. In particular, we will describe our software-based iWARP solution which enables iWARP communication on hosts without RDMA hardware. Next, we discuss interfaces offered to the RDMA consumer (the user application). As we will see, the general interface is rather cumbersome to program and error prone. Therefore, we suggest a simplified API which allows the fast development of iWARP-based applications while conserving the performance and flexibility of the original interface. Also, we will experiment with a radically different interface to RDMA: the well-known UNIX file abstraction. Last, we will see iWARP in action: we first show a simple but complete iWARP application based on our lightweight library to give future programmers a head start and then present a suite of micro benchmarks to visualize the performance of RDMA over Ethernet.

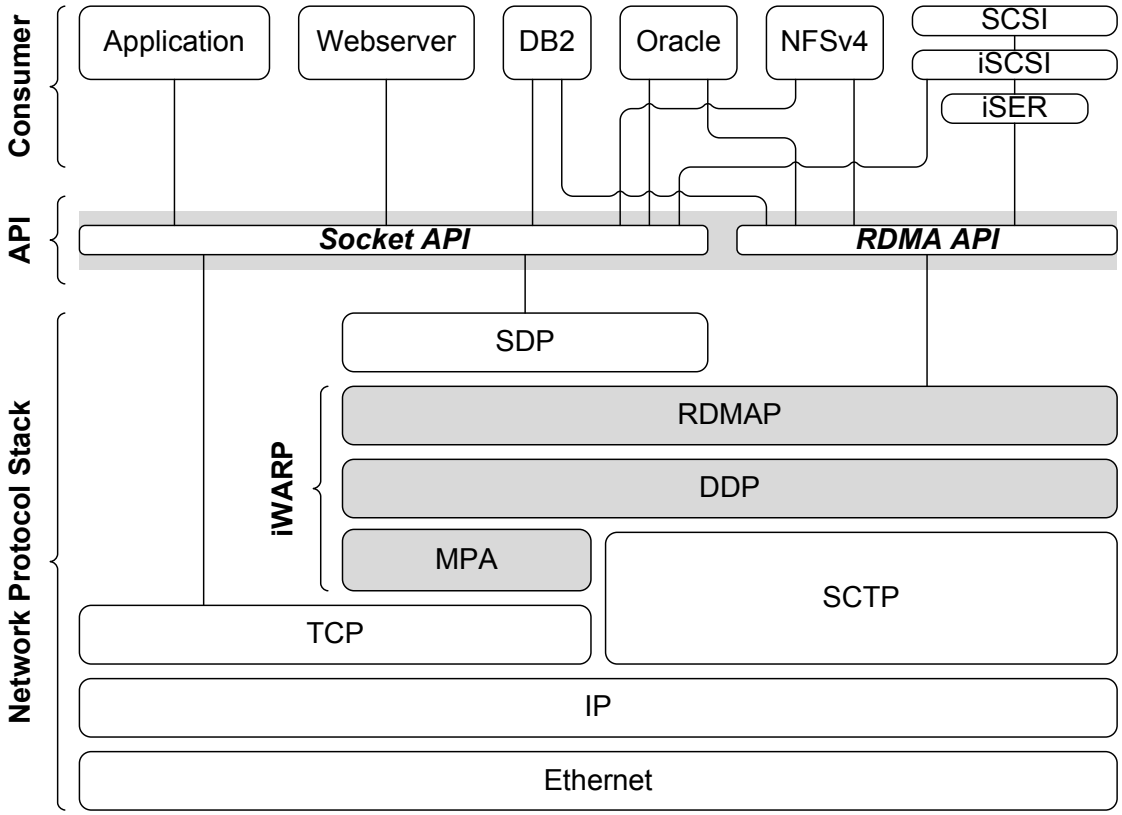


Figure 3.1: iWARP protocol stack, API and some applications.

3.1 The Protocol Stack

In order to enable RDMA over Ethernet, the IETF has standardized three companion protocols over TCP/IP [RMTB05]. These are, from bottom up, the *Marker PDU Aligned Framing for TCP* (MPA) [CER+07], the *Direct Data Placement over Reliable Transports* (DDP) [SPRC07] and finally the *Remote Direct Memory Access Protocol* (RDMAP) [RMC+07]. Figure 3.1 depicts the whole stack. In the following, we focus only on the *Network Protocol Stack* part of the figure. The API will be discussed in Section 3.3 and applications are presented in Part II of this thesis.

The three iWARP building blocks depicted in the figure provide the following functionality:

MPA Marker PDU Aligned Framing acts as a data adaptation layer between TCP (stream-based) and DDP (packet-based). It keeps the reliable, in-order delivery of TCP while adding the preservation of higher-level protocol record boundaries. In other words, it preserves the header alignment for DDP. To

that end, MPA splits the outbound data into *Framed Protocol Data Units* (FPDUs) and (optionally) inserts *Markers* into the byte stream at fixed intervals to recover from header misalignment on reception. Each marker contains the offset to the previous header. Being able to locate the header of the upper layer protocol (DDP) is useful to hardware network adapters that use DDP to directly place inbound data into the user buffer without requiring that the packets arrive in order. Additionally, MPA appends an (optional) Cyclic Redundancy Check (CRC32c) to each FPDU to prevent data corruption [SP00]. Last but not least, MPA is used to signal the transition from normal TCP stream mode to full RDMA operation of the connection. For detailed information see *RFC 5044*.

DDP The Direct Data Placement layer provides information for directly placing inbound data (which potentially arrives out of order) in the appropriate user level application buffer. This is the core protocol for enabling zero-copy data transfers over Ethernet. As we have discussed earlier, avoiding the intermediate copies (through the kernel) is highly desirable as it leads not only to a reduced memory bus traffic but also to a lower CPU load and fewer context switches. DDP operates on messages and can either run over MPA plus TCP or the Stream Control Transmission Protocol (SCTP) [SXM$^+$00]. MPA was mainly introduced because TCP is much more wide spread than SCTP which makes RDMA-style communication over TCP more desirable. DDP distinguishes between the *tagged* and *untagged* buffer model. The tagged model allows the local peer to advertise a named buffer to the remote peer. Such a buffer is assigned a unique tag, called *Steering Tag* or *STag* which enables the remote peer to specify where the data is to be placed at the local peer. In the untagged model, on the other hand, the local buffer is kept anonymous and the local peer specifies where inbound data is to be placed. Reliable, in-order delivery is provided for both, the tagged and untagged buffer model. The additional information provided by DDP allows the user level buffer to be used as reassembly buffer even in the case where the MPA or SCTP messages arrive out of order. More information on the exact delivery semantics, the packet format and the like can be found in *RFC 5041*.

RDMAP The Remote Direct Memory Access Protocol, finally, provides the data transfer semantics over DDP that enable a kernel bypass implementation. RDMAP distinguishes between *one-sided* (*RDMA Write* and *RDMA Read*) and *two-sided* (*Send/Receive*) operations. With one-sided operations, only the application layer of the peer issuing the operation is involved. At the other peer, the data transfer is handled entirely by the RDMA hardware (i.e.,

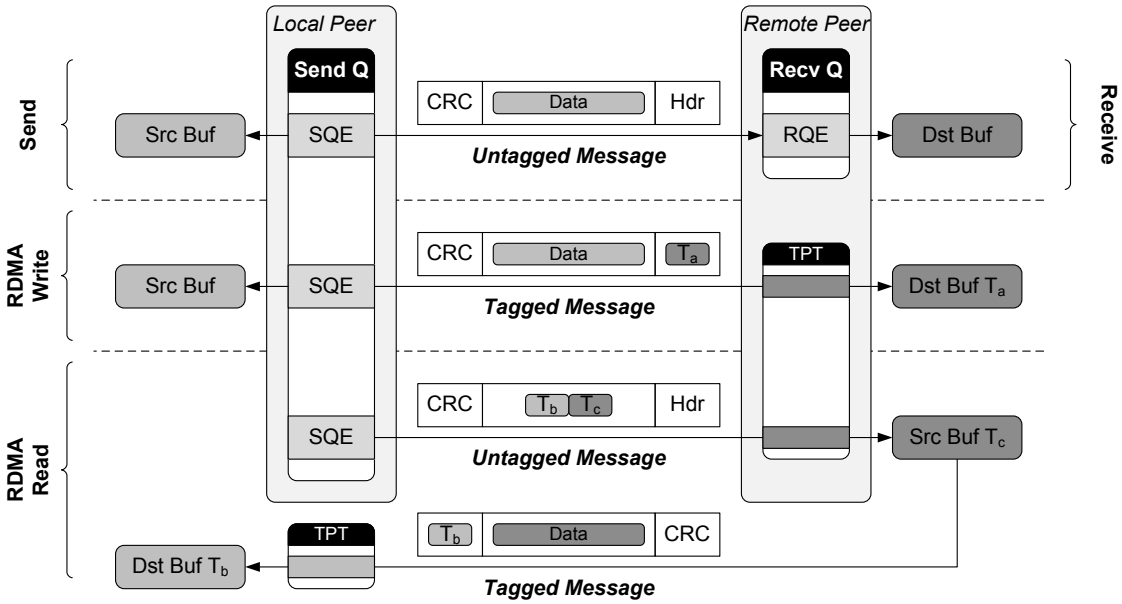


Figure 3.2: RDMA operations. The *untagged* model is used for the *Send* and *RDMA Read Request* messages whereas the *tagged* model is used for *RDMA Write* and *RDMA Read Response* messages.

the RNIC). With two-sided operations, on the other hand, the application layers of both peers are involved. The one-side operations require a prior buffer advertisement and thus utilize the tagged buffer model from DDP whereas the two-sided operations follow the untagged buffer model. RDMAP is described in full detail in *RFC 5040*.

Figure 3.2 illustrates the use of the tagged and untagged buffer model. The topmost data transfer represents the two-sided *Send/Receive* operation. The Send Queue Element (SQE) on the Send Queue (SQ) contains the location of the source buffer. At the remote peer, the destination buffer is identified by the corresponding Receive Queue Element (RQE) of the Receive Queue (RQ). Note, that the message on the wire does not contain any source or destination buffer information—tags—and is thus called *untagged*. For a formal introduction of the queue-based interaction, see Section 3.3.

The data transfer in middle represents a one-sided *RDMA Write* operation. Here, the SQE not only contains the location of the source buffer (like in the case of a *Send* operation) but also the sink buffer tag (T_a). As the local peer is able to steer the data directly into a given (pre-advertised) named buffer at the remote end, this operation follows the *tagged* buffer model. On the remote peer, the incoming data is verified against the Translation and Protection Table (TPT)

but there is no corresponding RQE.

The last operation depicted is the *RDMA Read*. In contrast to the other two operations, the *RDMA Read* consists of a ping-pong message exchange on the wire. The local peer starts by sending a *RDMA Read Request* carrying the source (remote) and destination (local) tags T_c and T_b. The remote peer then replies with a *RDMA Read Response* message containing the actual data as well as the destination buffer information supplied with the *RDMA Read Request* (T_b). The *RDMA Read Response* is essentially an inverse *RDMA Write*. As with all named buffers, the data exchange is verified against the respective TPTs.

3.1.1 Protocol Analyzer Extension

In order to visualize the iWARP traffic on the wire, we have extended the well-known Wireshark network protocol analyzer [wir]. Without our extension, Wireshark displays all iWARP/RDMA data simply as payload of the TCP protocol. Having the iWARP protocol header fields in a human readable format aided not only in the task of verifying our software iWARP implementation (see Section 3.2.2) but also helps in tracing iWARP applications. Furthermore, it is helpful in understanding the inner workings of the iWARP protocol suite from a wire perspective. We have chosen Wireshark because it is open source software and has an active developer community.

Figure 3.3 displays a sample packet dissection done by our iWARP aware Wireshark. The packet displayed is an MPA connection request frame—we are thus in the process of establishing an iWARP connection. As we can see, the connection initiator does not want to use markers (Marker flag: False) but will attach a valid CRC to each FPDU (CRC flag: True). Without our extension, Wireshark would stop dissecting the data at TCP level and display the MPA request frame as a series of octets which would then have to be analyzed by hand.

Support for new protocols is added to Wireshark by means of dissectors. We have thus added dissectors for for full MPA, DDP and RDMAP inspection. Our implementation features reassembly of individual MPA FPDUs into full DDP/RDMAP messages. Furthermore, we are able to verify the attached CRC, provide filtering capabilities for packets of interest and detect protocol discrepancies. Dissecting iWARP traffic is a non-trivial task for a number of reasons: first, the protocol headers are interleaved (see below). Second, MPA optionally inserts markers within the byte stream. These have to be tracked because they can end up not only in the payload but also within the header of a message. Third, individual FPDUs must be reassembled to reconstruct meaningful, complete DDP/RDMAP messages. Our code has become part of the official Wireshark release (since version 1.2.0).

Figure 3.4 shows the complete iWARP headers for *Send* (3.4(a)), *RDMA Write* (3.4(b)), *RDMA Read Request* (3.4(c)) and *RDMA Read Response* (3.4(d))

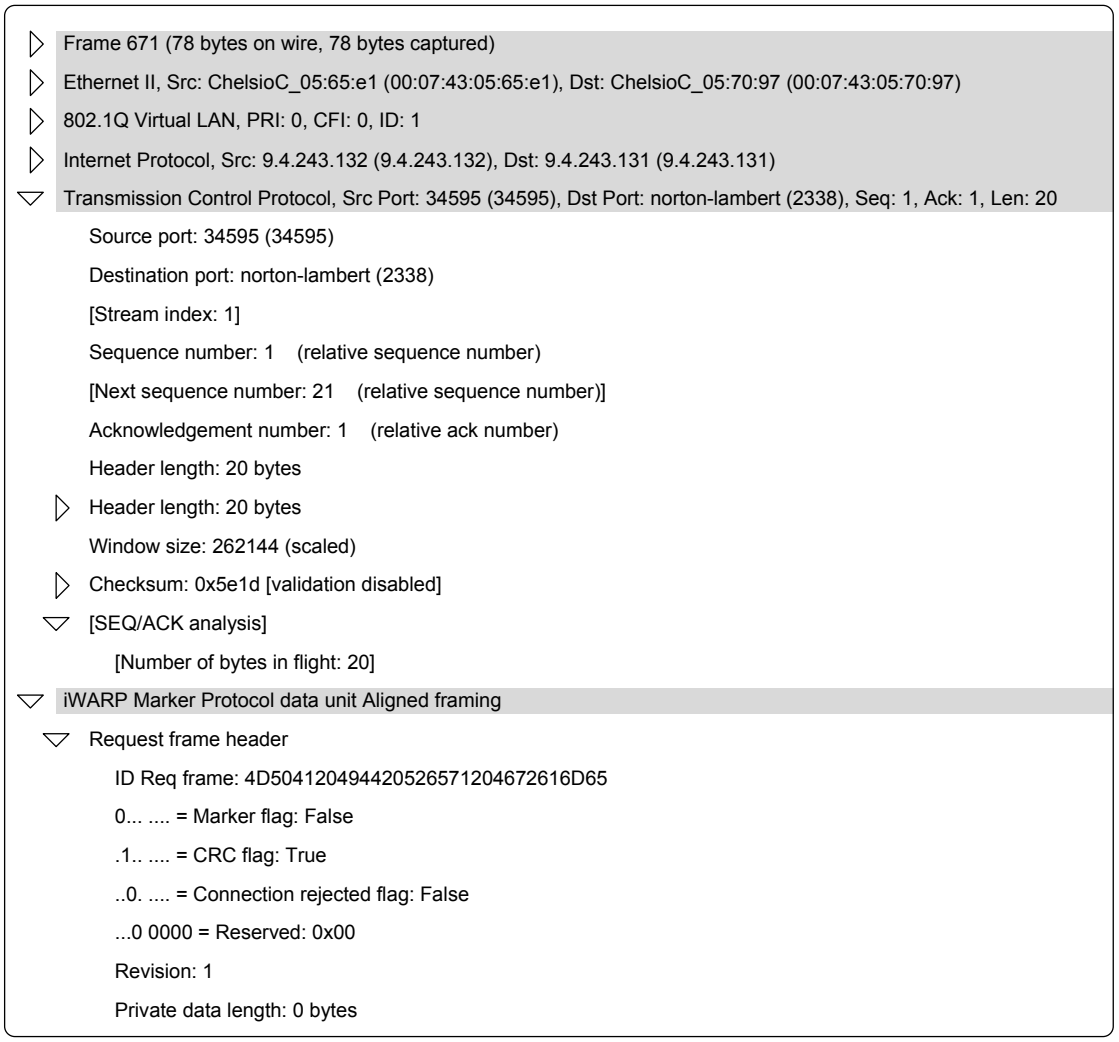
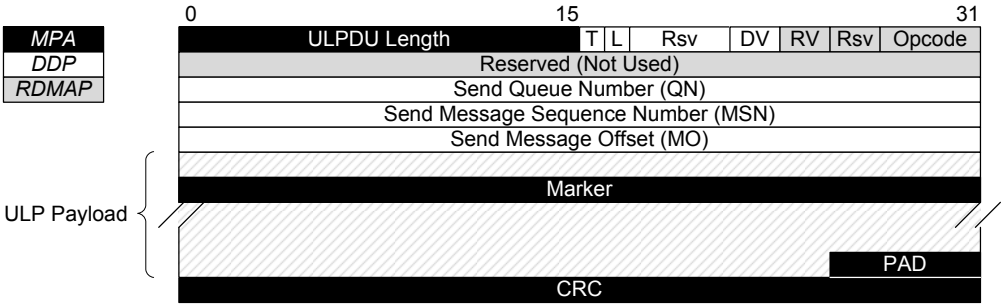
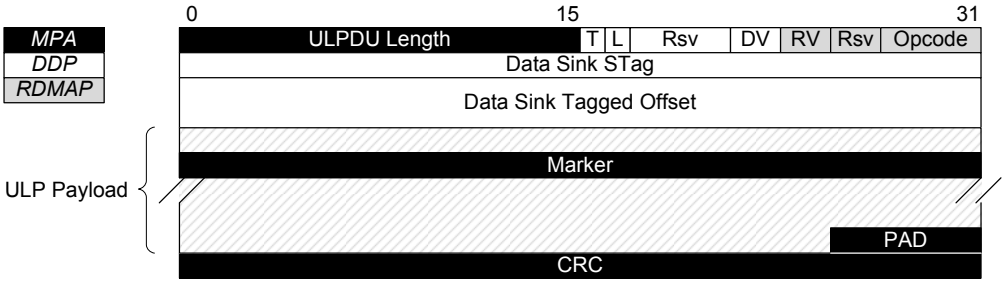


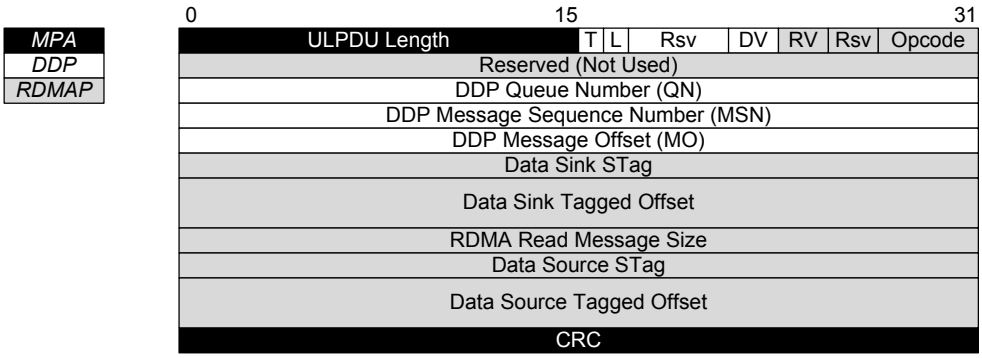
Figure 3.3: Wireshark protocol analyzer. iWARP traffic is dissected into a human readable format rather than just being displayed as and array of octets.



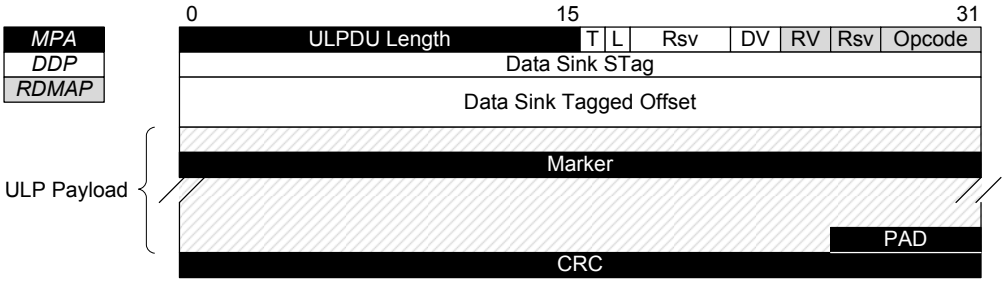
(a) Send FPDU.



(b) RDMA Write FPDU.



(c) RDMA Read Request FPDU.



(d) RDMA Read Response FPDU.

Figure 3.4: iWARP protocol FPDUs. MPA, DDP and RDMAP protocol headers are transported as TCP payload. DDP and RDMAP headers are interleaved.

which are recognized and dissected by our extension. The black pieces belong to MPA, the white ones to DDP and the light gray ones to RDMAP. Each of the four messages represents an FPDU. Each FPDU encapsulates a DDP segment with RDMAP information. One or several of these segments form a DDP/RDMAP message. The respective fields of each FPDU are described in the following.

The MPA information contains the Upper Layer Protocol Data Unit (ULPDU) Length in number of bytes (not including the MPA information itself), potential markers, padding at the end of the ULP payload and the CRC.[1]

The first octet of the DDP header is referred to as the control field. It states whether the message follows the tagged or untagged buffer model (T) and whether this is the last segment of a DDP message (L). The rest of the DDP message depends on the buffer model. For an untagged message (i.e., either *Send* or *RDMA Read Request*), the queue number of the data sink's untagged buffer queue (constant across all DDP messages for a given connection), the message sequence number (increased by one for each new message) as well as the message offset of the current segment (relative to the beginning of the whole DDP message) are specified. In other words, the queue number specifies the connection (like a port in TCP), the sequence number enables order preservation between individual DDP messages and the message offset enables order preservation among individual segments within a DDP message. This allows data to be placed directly in the application buffer even if it arrives out of order (for an RNIC). The tagged DDP messages (i.e., *RDMA Write* and *RDMA Read Response*) are a bit simpler and shorter. They contain the STag of the data sink (destination buffer) as well as the offset within that buffer. The offset can be specified either relative to the beginning of the buffer or as an absolute virtual address. As we will see later in this chapter, the performance and overhead are nearly equivalent for the tagged and untagged buffer model. The differences are mainly relevant in terms of the semantic for upper layer application protocols.

Also RDMAP starts with a single octet of control information which primarily contains the operation code (Opcode) indicating the data transfer type (*Send*, *RDMA Write*, etc.). For the *Send*, *RDMA Write* and *RDMA Read Response*[2], the RDMAP does not add further information. Only the application payload is inserted. For the *RDMA Read Request*, however, the source and destination information as well as the length of the data transfer are specified. Note that there is no payload attached to that message.

A simplified view of these operations was provided in Figure 3.2. Further protocol details can be found in the respective RFC documents (see beginning of Section 3.1).

[1] The CRC is filled with zeros if it is disabled.

[2] The *RDMA Read Response* is essentially an inverse *RDMA Write*.

3.1.2 Security Considerations

The iWARP stack does not provide any security features to prevent spoofing, tampering and information disclosure attacks. As it is based on TCP/IP, however, standard security services such as IPsec (network) or SSL/TLS (transport) can be applied without modification.

In terms of remote memory access, the RDMA enabled NIC (RNIC) is responsible to verify a number of things before fetching or placing data. First of all, the STag (buffer ID) must be valid which means that the buffer must exist and have been registered with the RNIC. Second, the address specified by the operation must be within the address range of the buffer referenced by the STag—the address cannot be beyond the end or before the start address of the buffer. Furthermore, the end of the data transfer operation must not exceed the available, registered buffer space. If these criteria are not met, a so-called base-and-bounds exception is thrown and the data transfer fails. Also, each iWARP object lives within a *Protection Domain* (PD) for resource isolation. The RDMA enabled NIC must assure that data transfers execute only within their respective PD. By this mechanism, it is possible to isolate buffers of different processes from each other: it is not possible to access a buffer of a foreign PD. We will look into more details of the iWARP object management later in the context of the RDMA API.

3.2 Host System Integration

In the previous section, we have seen the wire protocol of iWARP. Now it is time to look into the integration of iWARP/RDMA into the host system. As we have outlined in Section 2.3, there are different ways of doing this. In the following, we will focus on the method chosen by the OpenFabrics Alliance within their OpenFabrics Enterprise Distribution (OFED) on which all our work is based. As mentioned earlier, OFED is the most widespread RDMA software stack thanks to the strong participation from various important players of the industry[3], its dual support for InfiniBand (IB) as well as iWARP and its availability for Linux and Windows.

3.2.1 The OpenFabrics Software Stack

The OpenFabrics software stack (OFED)[4] provides the necessary software support by means of user level libraries and kernel extensions to bridge RDMA enabled hardware (i.e., IB and iWARP adapters) with user applications. While it was

[3]Intel, AMD, Cisco, IBM, Chelsio, Sun Microsystems, Oracle, Microsoft and others.
[4]http://www.openfabrics.org/downloads/OFED/

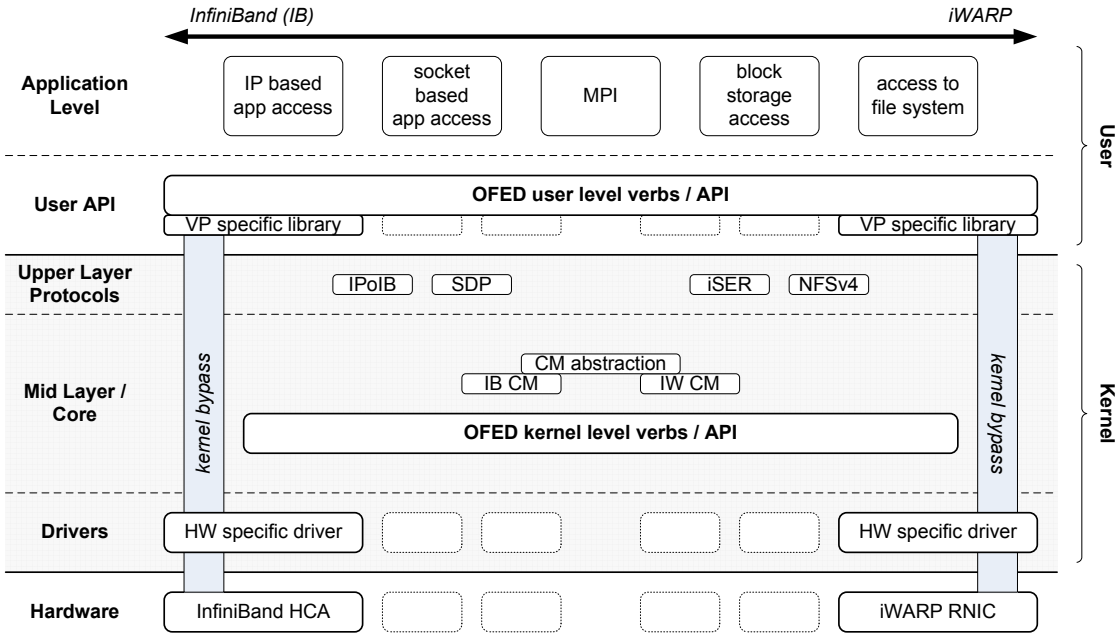


Figure 3.5: The OpenFabrics software stack (OFED). The illustration is reduced to the aspects which are relevant for this thesis.

originally designed exclusively for InfiniBand, support for RDMA over Ethernet was added with the standardization of the iWARP stack by the IETF and the emergence of RNICs. The benefit of this duality is that OFED can be used to run applications (unmodified) on IB as well as iWARP fabrics—the API and thus also the user applications have become transport agnostic. In particular, legacy InfiniBand applications from the HPC corner can now also run on Ethernet. However, the stack has grown quite large and complex. To make matters worse, there is no real documentation because too many parties work on the stack in parallel. As we will discuss in the following section, also the API has demanded compromises when iWARP support was added—particularly the connection management is rather laborious. For these reasons and the ones mentioned in Section 2.3.4, developers are still reluctant to move TCP/IP-based applications to RDMA despite the potential for significant performance improvement.

Figure 3.5 depicts selected aspects of the OFED stack which are relevant for this discussion. The left side of the illustration shows aspects more specific to IB while the right side focuses on iWARP. At the very top, we have the user space with the application level and the user libraries (to be discussed in Section 3.3.2). In the center, the kernel space is shown including some upper layer protocols mentioned earlier and the mid layer hosting the core functionality of the stack. This includes not only the RDMA object management but also the connection management

(CM) for IB and iWARP. Thanks to the CM abstraction, user applications are agnostic to the connection establishment and teardown details of the respective fabrics. The hardware specific device drivers finally conclude the kernel code. At the bottom, RDMA hardware such as an InfiniBand Host Channel Adapter (HCA) or an RDMA enabled NIC (RNIC) is shown. As described in the virtual interface architecture, a *fast path* bypassing the operating system kernel is provided for an efficient data propagation between the network adapter and the application layer.

The stack further distinguishes between general access functionality and vendor specific libraries and drivers. In the user API of Figure 3.5, we find a general purpose OFED user level verbs library which implements the API provided to applications according to the RDMA verbs specification [HCPR]. Below that, there are verbs provider (VP) specific libraries (e.g., for the Chelsio T3 adapter) that map the general functions to device specific ones. Like this, each vendor can implement its private fast path to the hardware, for instance. The same functional division is found in kernel space where the OFED core contains the general access functionality (e.g., for a kernel verbs consumer) which then invokes the individual device drivers.

Further details on the OpenFabrics stack can be found on their website[5] or in the tour through the OFED stack provided by J.George [Geo].

3.2.2 Softiwarp: iWARP Communication without an RNIC

The OFED stack is open source software and therefore extensible. We have made use of that and developed *Softiwarp*—an iWARP verbs provider implemented entirely in software.[6] The benefit of Softiwarp is that any ordinary NIC can now be used for iWARP communication. From an application point of view, it is simply another verbs provider. Within the OFED stack, Softiwarp is realized as an additional hardware specific driver, targeting legacy Ethernet NICs that are not RDMA-capable by themselves.

The implementation of Softiwarp follows the standard design of the OFED stack[7]: it consists of a user library attached to the general OFED verbs library and a device driver (kernel module) which plugs into the OFED core (see Figure 3.6). Even though our module uses unmodified kernel sockets for the data exchange, Softiwarp allows for the same communication semantics as an RNIC (e.g., asynchronous interface, one-sided operations, etc.) because it maps the whole RDMA object hierarchy in software which is otherwise implemented in hardware. In contrast to the hardware solutions, however, Softiwarp does not provide a fast path

[5] http://www.openfabrics.org

[6] In the course of my thesis, I have been co-developing this software RDMA solution under the lead of B.Metzler (see http://www.zurich.ibm.com/sys/software).

[7] Currently, there is only a Linux version of Softiwarp.

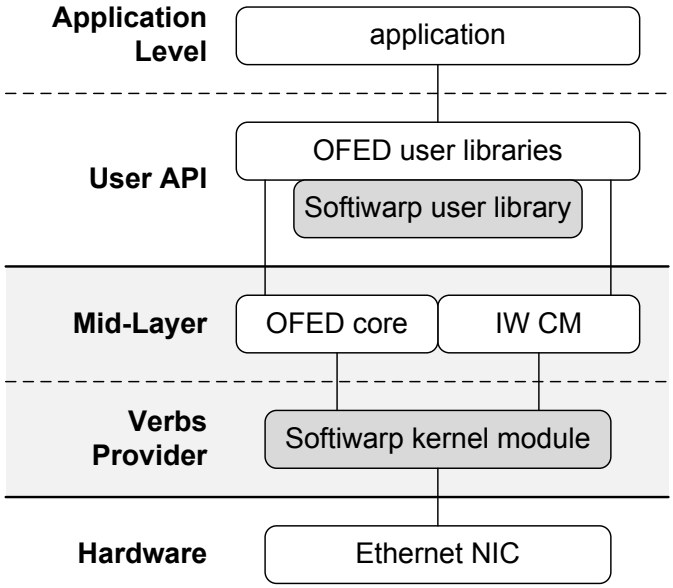


Figure 3.6: OFED software stack extended with Softiwarp.

between the user library and the device.

For obvious reasons, we do not achieve the same performance and overhead reduction as a true RNIC (cf. Section 3.4). However, hardware support is not always necessary and often too expensive. Softiwarp therefore makes RDMA attractive for a whole range of applications for which RDMA would otherwise not be an option. Since it is wire-compatible with an RNIC, Softiwarp allows for mixed setups consisting of hardware- and software enabled RDMA. Typical examples for this are client/server scenarios (e.g., video streaming, Chapter 6) where a single server must be able to sustain a high aggregate throughput while the clients are numerous and only require a small fraction of the total bandwidth for which hard-

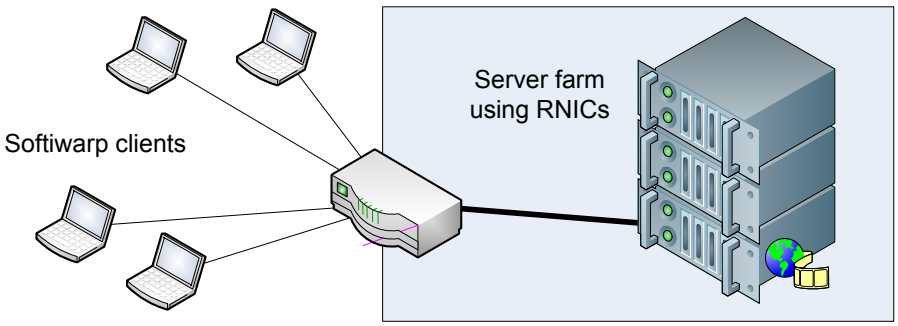
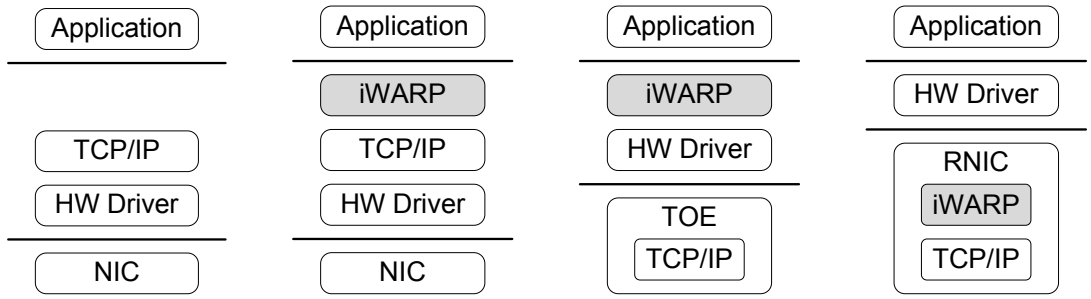


Figure 3.7: Asymmetrical client/server setup. The servers are equipped with RNICs for performance while the clients run Softiwarp for cost reasons.



(a) Classical (b) Softiwarp (c) Softiwarp/TOE (d) RNIC

Figure 3.8: Different offloading options. From the conventional TCP/IP stack (a) over iWARP in software without (b) and with (c) TCP/IP offloading to full hardware offload (d).

ware acceleration and zero-copy are not required (Figure 3.7). In such a setup, the server can be equipped with a real RNIC (for performance) while the clients run Softiwarp (for cost reasons). It is important to understand that *both sides* of the connection must have iWARP/RDMA capabilities or else the server cannot perform zero-copy RDMA data transfers and has to fall back to ordinary TCP communication inducing the well-known overhead. Another interesting use case for Softiwarp is to have it as a fallback option within a server in case the real RNIC fails.

In contrast to the function offloading principle followed by RNICs, we on-load the iWARP processing onto the general purpose CPU(s), see Figure 3.8. With today's multicore systems, dedicating a CPU core to iWARP processing can be a cost-effective alternative [RMI+04]. Furthermore, techniques such as the Intel I/O Acceleration Technology [ioa] allow network data to be moved more efficiently through recent CPUs. Additionally, a TCP offload engine (TOE) could be used to take the TCP/IP stack processing load off the CPU. Still, Softiwarp is unlikely to replace RNICs in the near future due to the remaining memory bus limitation. An experimental evaluation of the performance offered by Softiwarp based on micro benchmarks vis a vis real RNICs is presented in the upcoming Section 3.4.

Softiwarp, was first presented by B.Metzler et al. [MNF09] at the 2009 International Sonoma Workshop where the transmit and receive path within the kernel module were discussed. The feedback was positive and we thus have returned the code to OpenFabrics for Softiwarp to become part of the standard Linux kernel. Before we had the OFED-based Softiwarp, the kernel module (then called *Soft-RDMA*) was standalone and featured a different API, the IT-API [The]. Details on SoftRDMA, which eventually led to Softiwarp, were published by Neeser et al. [NMF10].

Related Projects

The Ohio Supercomputer Center has presented an alternative software implementation of the iWARP protocol stack [DDW05]. They provided a kernel-space and a user-space version together with a set of wrapper functions that followed the OpenFabrics verbs. The implementation itself was not designed within the OFED framework, however. The Ohio stack suggested its own, non-standardized API which made applications that were built on it less portable and fabric-aware. Concerning the performance, CRC calculation in software resulted in a latency increase by a factor of two. The maximum throughput on 1 Gigabit Ethernet of about 920 Mb/s could only be reached with packets of size 16 KB or larger and with CRC disabled. Our previous implementation, SoftRDMA, on the other hand achieved the same throughput already with 2 KB packets (or 4 KB if CRC was enabled). The CPU load induced by the Ohio stack was also considerably higher than that of SoftRDMA, especially at the receiver side. For results on the current implementation, see Section 3.4.

Balaji et al. [BJVP05] have addressed the lack of backwards compatibility of iWARP communication by introducing yet another software stack to emulate iWARP. Their stack was enhanced with a so-called *extended socket interface* which could provide the necessary backwards compatibility for socket based applications by overloading the standard socket implementation. Furthermore, they claimed to have exposed the richer feature set of iWARP (i.e., asynchronous interface, zero-copy and one-sided operations) to be used by the applications with minimal modifications. However, they did not provide any details on what that meant in practice and how large these minimal modifications really were. Furthermore, they did not address the important discrepancy between the implicit and explicit buffer management from an application perspective. Performance wise, their iWARP stack with the extended sockets lagged significantly behind plain TCP in terms of latency and throughput due to the slow inter process channel (IPC) between the threads which they used for enabling asynchronous processing. Also locking of the shared queues seemed to be a problem. As in the case of the Ohio stack [DDW05], their kernel implementation outperforms the user space implementation.

A thorough comparison between host-based, host-offloaded (RNIC) and partially offloaded iWARP processing was performed by Balaji et al. [BcFB+07]. In particular, they focused on iWARP specific aspects like support for out-of-order data placement, MPA marker handling and CRC calculation. They showed that CRC calculation was one of the most expensive tasks in the iWARP stack. Furthermore, they pointed out the importance of having a true scatter/gather DMA engine on the NIC for efficient marker handling and out-of-order data placement. While they were able to outperform their RNIC—which did not have a true scatter/gather DMA engine—by offloading only parts of the functionality, this is no

longer likely to be possible with today's hardware.

An even more radical approach was proposed by Binkert et al. [BSR06]. In order to meet the needs of future high performance TCP/IP networking, they suggested the integration of the NIC with the CPU for backwards compatibility. Their novel NIC design strove at being simpler while providing a performance equivalent to a conventional DMA-based NIC. Instead of creating the usual overhead and complexity of DMA descriptor management, they exposed a raw FIFO interface to the device driver. With that, they were able to avoid the copy overhead on the receive side and could decouple packet header inspection from payload copying.

3.3 Consumer Interfaces

After having presented the common option for integrating RDMA into today's computer systems, we will now discuss how this iWARP/RDMA subsystem is accessed from a user application perspective.[8] To that end, we start with a short overview of the RDMA verbs [HCPR] on which the industry has agreed. As the verbs only provide the semantics but no concrete interface, we will thereafter continue our journey with the API exported by the OpenFabrics stack. Finally, we will present two alternative interfaces with the potential for simplifying application development.

3.3.1 RDMA Verbs

The *RDMA Protocol Verbs Specification* [HCPR] describes the interface semantics that build the basis for the interaction between applications and the RDMA subsystem. Every RDMA enabled NIC (RNIC)[9] has to follow them for compliance. However, the exact details of the implementation are left open to the RNIC vendors. In this section, we will highlight the key aspects of the verbs which are relevant for the subsequent discussion. For that, we start by looking at the defined RDMA programming abstractions.

RDMA Object Overview

Before RDMA operations can be executed, we not only need to establish the connection but also have to create a number of programming objects. These objects live within the RNIC. The verbs consumer merely holds references to them. Figure 3.9 depicts the main objects described by the verbs in a resource creation

[8] Even though it is possible to access the RDMA subsystem also as a kernel consumer, we restrict the discussion to the user level.

[9] This also includes software implementations like Softiwarp.

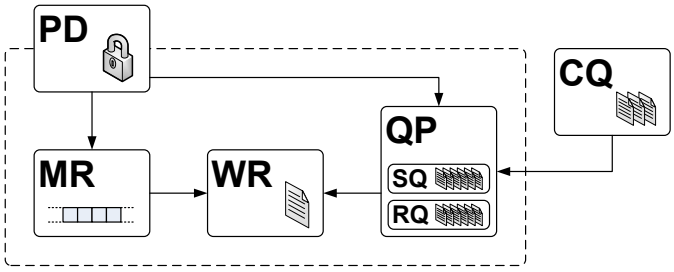


Figure 3.9: Resource creation dependency diagram for the main RDMA verbs objects: *Protection Domain* (PD), *Completion Queue* (CQ), *Queue Pair* (QP) with *Send Queue* (SQ) and *Receive Queue* (RQ), *Memory Region* (MR) and *Work Request* (WR).

dependency diagram. The complete diagram can be found in the verbs specification [HCPR] on page 26.

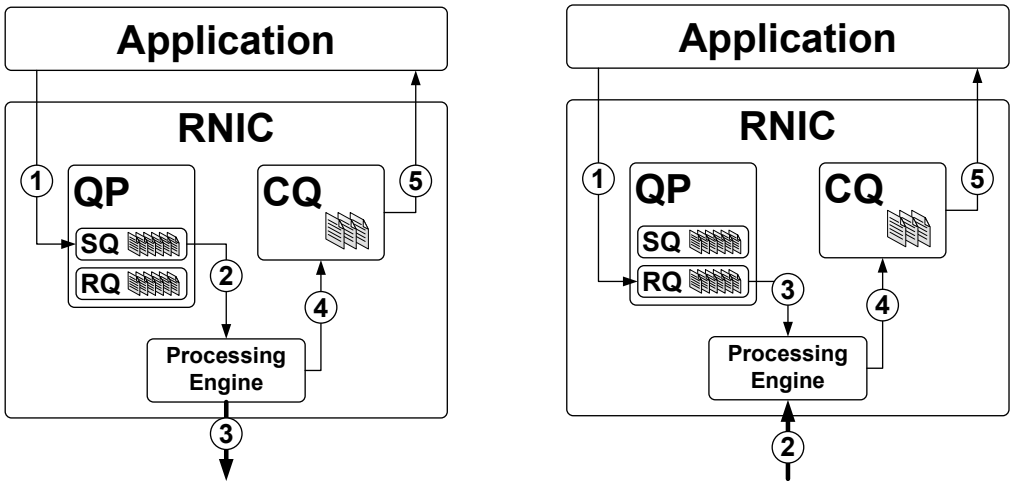
PD The root object is the *Protection Domain* (PD), which provides a security mechanism for mutual resource isolation between different processes. Upon creation of the PD, an ID is returned which is used to refer to the PD and create further objects within it.

CQ Then comes the *Completion Queue* (CQ) which acts as a container for *Work Completions* (WC). If requested, WCs are generated by the verbs provider (RNIC) whenever an RDMA operation has terminated in order to signal completion and indicate the status of the operation (error or success) to the application level.

QP With the PD and CQ in place, a *Queue Pair* (QP) can be created which is used by the application to post requests for RDMA operations to the verbs provider. It consists of two queues (hence the name): the *Send Queue* (SQ) and the *Receive Queue* (RQ). The SQ is used for send type messages such as *Send*, *RDMA Write* and *RDMA Read* as well as for local operations. The RQ, on the other hand, holds Receive Work Requests which are consumed on inbound *Send* messages. Think of a QP as a connection endpoint abstraction similar to TCP sockets.[10] Each QP lives within a PD and has an associated CQ for completion reporting.

MR Within the PD, user communication buffers, termed *Memory Regions* (MRs), can be created. A Memory Region is essentially a user buffer which is registered with the verbs provider. After such a registration, the MR is identified

[10] Like a socket, the QP also has different states depending on the lower-layer connection status.



(a) Post Send (b) Post Receive

Figure 3.10: Communication queues for sending and receiving data.

within the RDMA subsystem by its unique *Steering Tag* (STag). Furthermore, it has a (user virtual) starting address and a size. Like the QP, each MR lives within a PD for protection against access from processes running in different PDs.[11]

WR In order to execute an RDMA operation, we finally need to create a *Work Request* (WR). WRs are used to describe the operation to be executed (i.e., operation type, address(es), STag(s), length, etc.). Send WRs are appended to the Send Queue and Receive WRs to the Receive Queue (see upcoming section on Work Request Processing).

Now that the programming abstractions are in place, we can look closer into the consumer/provider interaction.

Queue-based Consumer/Provider Interface

Following the Virtual Interface Architecture, the RDMA verbs propose a queue-based communication between the verbs consumer (application) and the verbs provider (RNIC) which allows for an *asynchronous interaction* enabling the overlap of communication and computation. The value of this feature will be discussed throughout the examples presented in Part II of this thesis. Figure 3.10 depicts the communication scheme for sending (3.10(a)) and receiving (3.10(b)) data.

[11]It is possible use the same PD for different QPs to allow sharing of MRs.

As mentioned previously, the application posts *Work Requests* (WR) onto the Work Queues of its Queue Pair (QP) to submit operations to the RNIC (step 1 in Figure 3.10). For sending data, the application creates a *Send Work Request* (Send WR) and posts it onto the *Send Queue* (SQ). The processing engine of the RNIC then asynchronously reaps the Work Request from the SQ (in FIFO order) and performs the send data transfer (steps 2 and 3 in Figure 3.10(a)). A Send Work Request can trigger one of the following operations: *Send*, *RDMA Write* or *RDMA Read*.[12]

Figure 3.10(b) illustrates the data receive path. *Receive Work Requests* (Receive WRs) are posted to the *Receive Queue* (RQ) of the QP on the RNIC. They contain local placement information for inbound data, transported within *Send* messages. The RNIC consumes exactly one Receive WR for each inbound *Send* (in FIFO order).

Whenever the verbs provider has finished processing either a Send WR or a Receive WR, it reports the completion status by appending a corresponding *Work Completion* (WC) to the *Completion Queue* (CQ) attached to the QP (step 4). The consumer then polls the CQ to retrieve the completion (step 5). Only now, the application is guaranteed that the data transfer has completed. The CQ can be used in various ways: it is possible to assign a separate CQ for holding Work Completions from the Send Queue and from the Receive Queue which facilitates completion handling because it is implicitly known whether it belongs to a Send- or Receive Work Request. On the other hand, it is also possible to use just one CQ for the whole QP (or even for several QPs) in order to reduce the number of objects and build a centralized completion handling mechanism.

Memory Management

The interaction between the application and the networking subsystem is not the only difference between RDMA and TCP sockets; also the way in which the communication buffers are managed is radically different. In the sockets interface, the buffers are located in the kernel, hidden from the application. With RDMA, however, the application has to *manage all communication buffers manually* in user space. In practice, this means that an application first has to create buffers of appropriate size by using OS provided memory allocation mechanisms like `mmap` or `malloc`. Thereafter, in order to enable the memory to be accessed by the RNIC, (parts of) this buffer must be registered with the verbs provider through the RDMA subsystem.

From now on, the buffer is referred to as a *Memory Region* (MR). Each MR is assigned a *Steering Tag* (STag) for identification as well as local and remote access

[12]Even though data is "received" with an *RDMA Read* operation, it is scheduled as a send operation.

rights. Furthermore, its virtual starting address, called *Tagged Offset* (TO), and its size are recorded. After MR registration, the TO, size and STag cannot be altered anymore. To alter these parameters, the concept of the *Memory Window* (MW) was introduced in the verbs (see Section 7.10 of the verbs specification [HCPR]).

A thorough analysis and description of the MR registration process is presented in the upcoming Chapter 4.

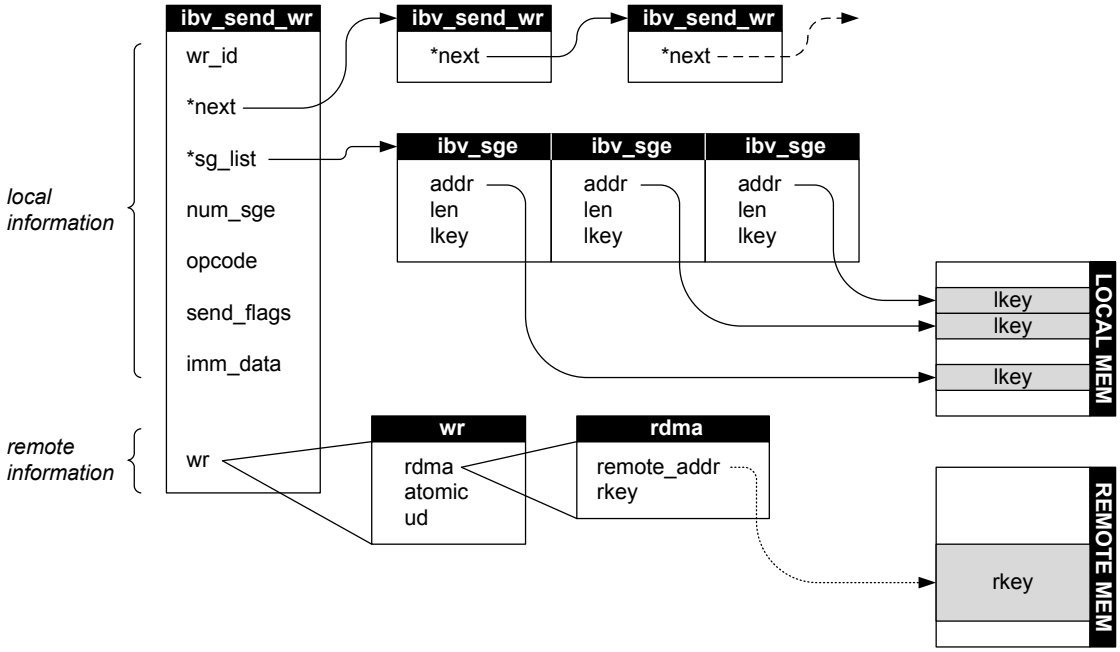
Work Request Processing

The fundamental unit for the application to communicate with the RNIC is the *Work Request* (WR). WRs are created by the application and posted to the Send Queue or Receive Queue depending on the type of the WR.

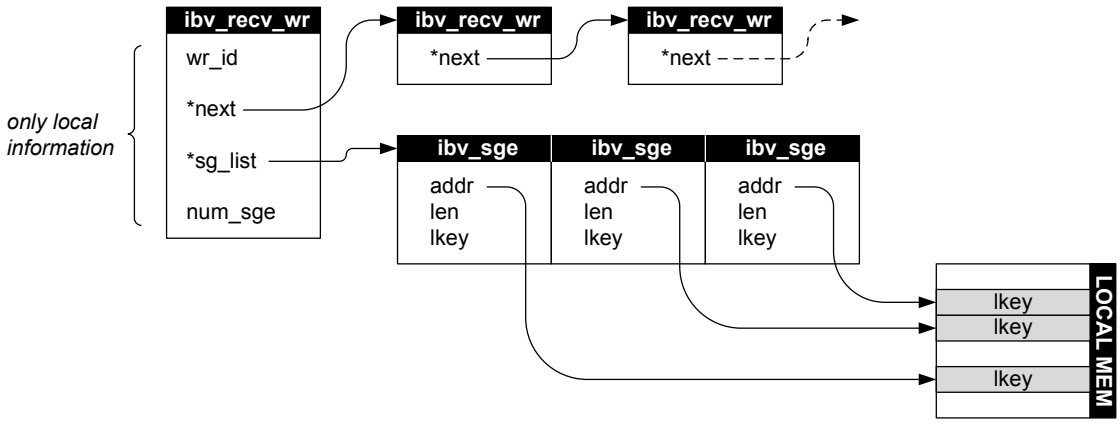
Figure 3.11 displays the format of WRs implemented by the OpenFabrics stack. Figure 3.11(a) illustrates the Send WR while Figure 3.11(b) shows the Receive WR structure. Both WR types carry a user specified WR ID (wr_id) which is reflected in the Work Completion for identification. The *local buffer* is always represented as a scatter/gather list (sg_list) consisting of a number of scatter/gather elements (num_sge). Each such element carries the address, length and STag[13] of the Memory Region it refers to. Furthermore, WRs can be linked (next pointer) which enables multiple WRs to be posted with a single call.

In contrast to the Receive WR, the Send WR contains some additional information. First, there is the operation code (opcode) indicating the type of send operation to be executed (i.e., *Send*, *RDMA Write*, *RDMA Read*). Then, there are some send flags that can be specified. The most important one for iWARP is the *signaled* flag. WRs without this flag set are dropped silently when the data transfer has finished—no Work Completion is created. This flag is particularly useful if the application is not (immediately) interested in the completion of the operation. Since WRs are processed in order, it might be sufficient to request a Work Completion only every n operations in order to reduce the processing overhead. We will see in Section 3.4, that these non-signaled WRs significantly lower the CPU load due to the reduced interaction between the application and the RDMA subsystem. As introduced in Section 3.1, DDP supports *tagged* and *untagged* send messages. For the tagged ones (i.e., *RDMA Write* and *RDMA Read*), the remote buffer is referenced in the WR through the remote address (remote_addr) and remote STag (rkey). Here, the remote address can either be an absolute user virtual address or an offset towards the beginning of the MR referenced by the remote STag. For an untagged message, this information is omitted.

[13] OpenFabrics refers to STags by the InfiniBand naming as *lkey* (local) and *rkey* (remote).



(a) Send Work Request



(b) Receive Work Request

Figure 3.11: Send- and Receive Work Request layout within OFED.

Verbs Interface

Before looking at the API suggested by OpenFabrics, we briefly summarize the operations described semantically in the verbs. First, a verbs provider (RNIC) can be opened and closed. By these two operations, we (un-)assign our RDMA programming objects to a specific network interface. For the Protection Domain, (de-)allocation semantics are described. The Queue Pair as well as the Completion Queue can be created, queried, modified and finally destroyed. In terms of the memory management, Memory Regions and Memory Windows can be (de-)registered. Finally, the verbs describe the operations on an existing Queue Pair and Completion Queue. For the first one, posting to the Send Queue as well as to the Receive Queue are supported. The latter queue can be polled for completions. Alternatively, a completion notification can be requested to learn about newly available Work Completions.

3.3.2 OFED API

We now move on from the semantic description of the RDMA interface to the concrete API used most widely in practice today: the OpenFabrics API provided to interact with the OpenFabrics Enterprise Distribution (OFED). While the API follows the verbs specification, there are some design decisions left open to the verbs provider and addressed by OFED which are worth mentioning in this context. In the following, we illustrate the steps necessary to setup and close an iWARP communication channel and to finally perform actual RDMA data transfers.

iWARP Channel Setup

Setting up an iWARP/RDMA channel is quite cumbersome with the OFED API. In a first set of steps, the network interface (RNIC) is selected. Thereafter, the peer initiating the connection (termed *Initiator*) creates the necessary RDMA objects and sends a connection request to the *Responder*. Upon receipt of such a request, the *Responder* creates his RDMA objects and accepts the connection.

Figure 3.12 lists these steps in detail. Both peers start by creating an *RDMA Event Channel* (1) which is used to receive connection events. Next, an *RDMA Connection Management ID* is created on the Event Channel (2). The IDs are then bound (3) to a particular verbs provider (RNIC) by supplying the respective IP addresses (listen address on the *Responder* and source/destination addresses on the *Initiator*). From now on, the Event Channel can be used to listen for connection events from the selected RNIC.

In the following, the *Responder* and *Initiator* proceed differently: the *Initiator* uses the Event Channel to wait for the events indicating resolution of the addresses

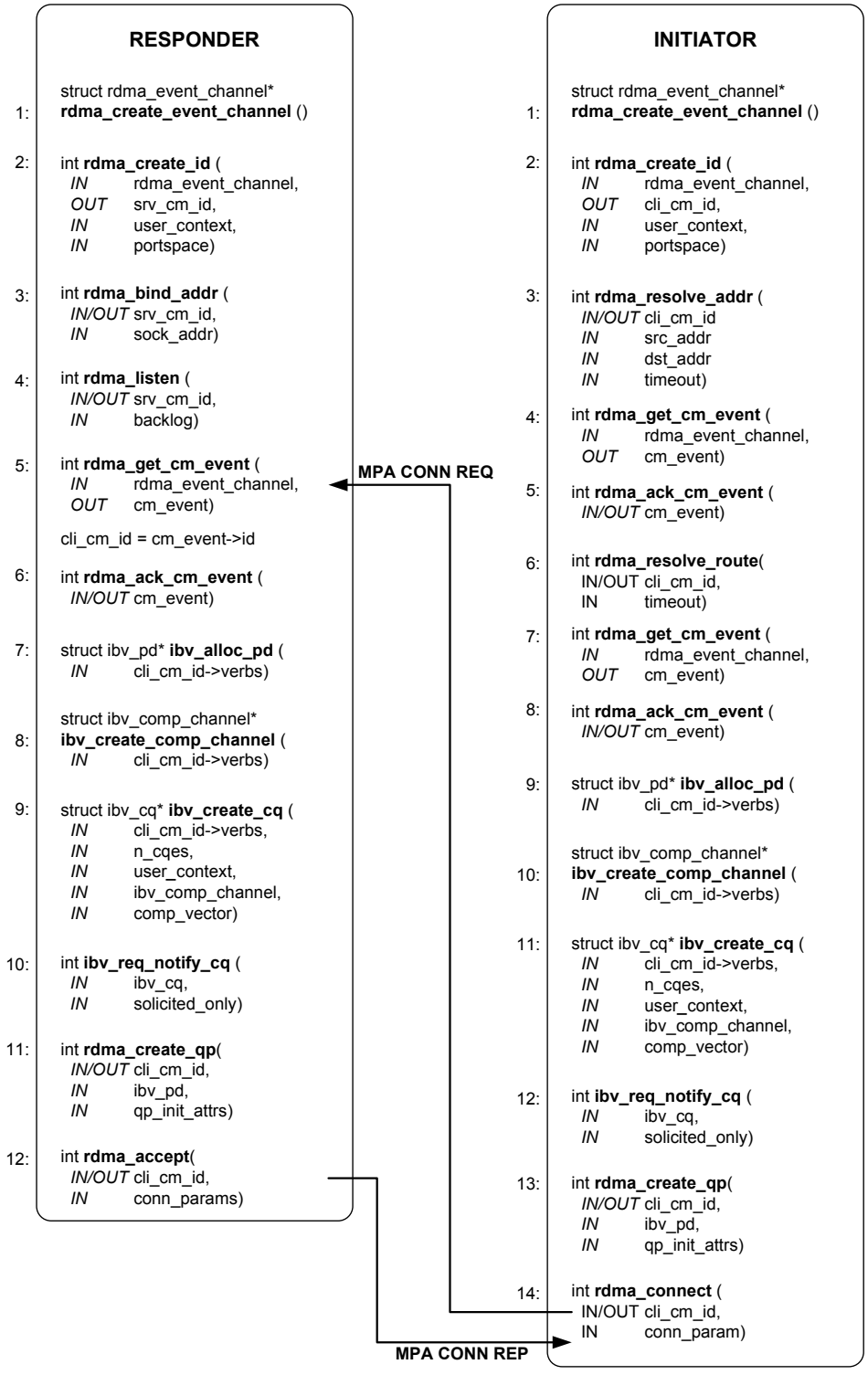


Figure 3.12: Setting up an iWARP/RDMA connection using the OFED API.

and the route to the *Responder* (steps 3–8).[14] These resolution steps are based on the host routing table to find a suitable, local RNIC which is connected to the same network as the *Responder*. After that search has succeeded, the RDMA programming objects are created: first, a *Protection Domain* is allocated (9).[15] Since the *Completion Queue* offers both, polling and notification on completion events, we need another Event Channel (the *Completion Event Channel*) (10). Now, the *Completion Queue* can be created (11) based on the verbs provider and the *Completion Event Channel*. In step (12), a completion notification is requested for the *Completion Queue*—in this example, the *Initiator* makes use of the notification mechanism rather than polling the queue. The last remaining object to be created, is the *Queue Pair* (13). Finally, the MPA connection request is sent (14).

The *Responder*, on the other hand, proceeds as follows: initially, it uses the *RDMA Event Channel* to wait for the connection request from the *Initiator* (steps 4–6). Upon arrival of such a request, it proceeds with the creation of the RDMA programming objects. An important difference is, that the *Responder* uses the *Initiator*-provided *RDMA Connection Management ID* rather than its own. Think of the *Responder* ID as the server socket and the *Initiator* ID as the client socket. The ID of the *Responder* is reserved for accepting further incoming connection requests. After all the objects have been created, the *Responder* accepts the connection request (12) which triggers an *RDMA Connection Established* event on the *RDMA Event Channel* and completes the connection setup.

As we will argue in Section 3.3.3, the connection setup is similar in most cases and can easily be wrapped to simplify application programming.

RDMA Communication and Buffer Management

Once the iWARP/RDMA communication channel is established, we can start issuing RDMA data transfers. To do that, we first need to create some buffers and register them as *Memory Regions* (MRs). In OFED, this is achieved by calling ibv_reg_mr(*pd, *addr, length, access_rights). Each MR lives inside a *Protection Domain* (pd), starts at a given address (addr) and has a certain length (length). The last argument of the registration call (access_rights) is used to specify the local and remote access rights (e.g., remote read-only, local/remote read-write, etc.). The start of the MR as well as its length do not have to match that of the user buffer—it can also lie within it (see Figure 3.13). However, only the part which has been registered can be accessed by the RNIC.

[14] All events have to be acknowledged for resource cleanup purposes after they have been consumed.

[15] The verbs pointer of the *RDMA Connection Management ID* is used to refer to the selected verbs provider.

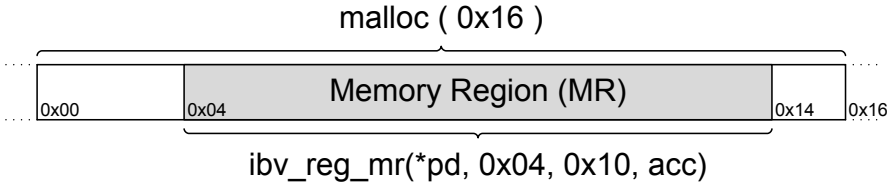


Figure 3.13: Registering a Memory Region of length 16 (0x10) with an offset of 4 (0x04) within a user buffer of length 22 (0x16).

In order to initiate a *Send*, *RDMA Write* or *RDMA Read* operation, a respective *Work Request* has to be posted onto the *Send Queue* of the *Queue Pair* by means of ibv_post_send(*qp, *send_wr, **error_wr). The whole zero-terminated WR list (send_wr) is processed and erroneous WRs are reported through the error_wr pointer. For filling the *Receive Queue* with *Receive Work Requests*, the ibv_post_recv(*qp, *recv_wr, **error_wr) is used analogously.

If a *Completion Event Channel* is in place, we can wait on it for *Work Completions*. To that end, ibv_get_cq_event(*channel, **cq, **context) is used which blocks until there is a new WC on any CQ. The CQ, on which the WC is pending, is returned through the cq pointer. In order to get the WC, we need to poll that CQ by means of ibv_poll_cq(*cp, num_entries, *wc) which returns num_entries WCs or empties the CQ in case there are fewer WCs pending. Through this mechanism, we can find out when an operation, scheduled by earlier posting of a respective WR, has completed—we are guaranteed that the data has been placed in its entirety upon reception of the respective Work Completion.

Channel Teardown

In contrast to the channel establishment, closing an iWARP connection is simple. Both sides, the *Initiator* as well as the *Responder*, can initiate connection teardown by calling rdma_disconnect(cm_id). The *RDMA Connection Management ID* (cm_id) is used to identify the connection to be closed. This causes a *RDMA Disconnect Event* to be generated on the *RDMA Event Channel* on both sides. As soon as this event is received, not only the iWARP- but also the underlying TCP connection are closed. Cleaning up the RDMA programming objects is done in the reverse order as they were created.

3.3.3 iWARP Library

The API suggested by OFED follows the RDMA verbs closely and is suitable not only for iWARP but also for InfiniBand. However, it has become rather cumbersome to program against (especially with regard to the connection management

as demonstrated in the previous section). In order to facilitate the development of iWARP/RDMA applications, we present a novel iWARP library consisting of a set of wrappers on top of the original OFED API. The goal of this effort is three-fold: first, the overhead incurred by the library must be negligible if at all. Second, the original functionality provided by the OFED API must be preserved. Third, the API exported by the new library should allow even non-experts in the field of OFED/RDMA to write such programs. In that respect, we do not primarily focus on reducing the number of lines of code but on eliminating sources of error by hiding the complexity.

In the following, we highlight the key features of the library from a coding perspective. To allow a one-on-one comparison, we follow the same structure as in the discussion of the OFED API (previous Section). A complete example application based on our library and API is shown in the upcoming Section 3.4.

iWARP Channel Setup

In order to ease the transformation from sockets to the RDMA verbs for developers, we have designed the connection management similar to what people are used from sockets. Furthermore, contrary to the OFED API, we keep all the RDMA programming objects (including the *RDMA Connection Management IDs* (CM ID)) in a single structure termed *iWARP Context*. We use this context as an abstraction for the iWARP/RDMA channel since all the programming objects are eventually bound to a single connection on an RNIC anyway. Figure 3.14 illustrates the complete state diagram for our *iWARP Context*. It might look large at first but some of the states are transitioned through automatically (shown in gray) and are only displayed for completeness. Upon creation, the context is INVALID. After having selected the verbs provider, its state changes to VALID and finally to CONNECTED after successful connection establishment.

The two programming steps of the *Initiator* (right-hand side of Figure 3.14) are the following: the connection establishment process is started by allocating the *iWARP Context* using iw_ctx_alloc() for which the remote IP address and port number are specified. In the background, the library creates all the necessary RDMA objects and transitions the CM ID through the address- and route resolution—note that the library handles and dispatches all connection events for us. Upon return of this call, we are ready for sending the connection request by means of iw_connect(). This call blocks until the *Responder* has either accepted or rejected our request. When the call returns successfully, the iWARP channel is ready for data exchanges. These two steps incorporate steps 1–14 of the original OFED verbs.

On the *Responder* side, we prepare for incoming connection requests by calling iw_open() where we specify the listen IP address/port pair and with that select

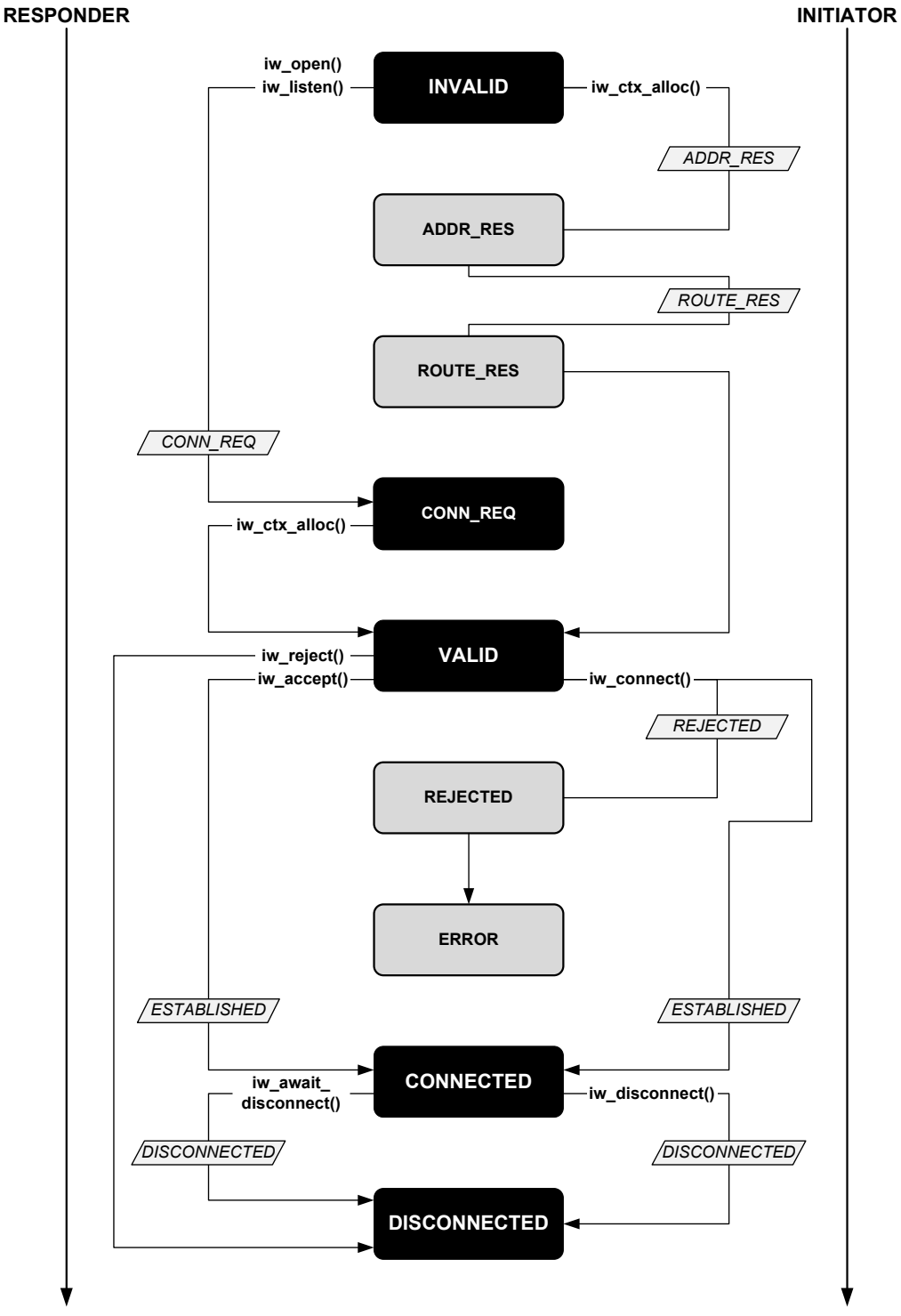


Figure 3.14: Setting up an iWARP/RDMA connection using our simplified iWARP library.

the local verbs provider. We then wait (blocking) for inbound connection requests using iw_listen(). Upon receipt of a connection request, the iWARP Context is moved to the CONN_REQ state. From there, we have to finalize the RDMA object creation by calling iw_ctx_alloc() (corresponds to steps 7–11 of Figure 3.12) before we either accept the request (iw_accept()) or reject it (iw_reject()). After returning from iw_accept(), the iWARP channel is ready.

RDMA Communication and Buffer Management

When using the OFED verbs for communication, first of all, a user buffer has to be created and registered with the RNIC. Thereafter, a list of scatter/gather structures (SGL) is needed—even for a single scatter/gather element—which is referred to by the Work Request. Finally, we have to set all the members of the Work Request structure to describe the desired RDMA data transfer (cf. Figure 3.11). Although this is generally not an issue with respect to the final application performance, it is extremely error prone—a lot of code is necessary to perform (even a simple) RDMA data transfer. The original verbs provide no easy-to-use function to transfer a (part of a) Memory Region to the remote peer. Things get even more complicated when performing the buffer advertisements through RDMA operations as well.[16] Our library fills this gap by providing not only a *simplified buffer management interface* but also an *easy-to-use, in-band buffer advertisement mechanism* as well as *intuitive data transfer and completion handling primitives*.

Buffer Management. For facilitating the communication buffer management, we offer the options of either creating a new buffer (including the allocation) or registering an existing one. We further distinguish between local Memory Regions (lmr) and remote ones (rmr). A new MR is created by calling

- iw_lmr_create(size, access_rights, lmr_out, iw_ctx)

while an existing one can be registered with

- iw_lmr_register(buf_addr, size, access_rights, lmr_out, iw_ctx).

In either case, the developer does not have to worry about the Protection Domain as it is automatically taken from the iWARP Context (iw_ctx). We also provide a function to handle MR deregistration and optional freeing of the underlying memory.

Buffer Advertisement. In order to facilitate in-band buffer advertisements, we provide the following three functions using the iWARP Context (iw_ctx):

[16] Performing the buffer advertisement on a separate TCP channel is simpler but the message ordering guarantee is lost which is dangerous with respect to race conditions.

- iw_post_send_adv(lmr, iw_ctx) to advertise a local MR (lmr),
- iw_post_recv_adv(iw_ctx) to prepare for an inbound advertisement and
- iw_wait_recv_adv(rmr_out, iw_ctx) to blocking wait for the advertisement.

Data Transfer Primitives. While the OFED API always needs a SGL for referring to a local buffer (MR), our library adds support for direct addressing if only one scatter/gather element is used. To further simplify the code, we provide individual functions for each RDMA operation—the iWARP Context is used to select the connection.

- post_send_lmr(lmr_src, offset_src, length, send_flags, iw_ctx)
- post_write_lmr(lmr_src, offset_src, rmr_dst, offset_dst, length, send_flags, iw_ctx)
- post_read_lmr(lmr_dst, offset_dst, rmr_src, offset_src, length, send_flags, iw_ctx)

The above three functions, offer an intuitive interface to the programmer which allows the specification of addresses and offsets for the source and destination buffers rather than having to deal with scatter/gather structures. In addition to these functions, we provide SGL-based versions in case the data is spread across several buffers. Also the *Receive* operation is supported in these two variants (direct addressing and SGL).

Work Completion Handling. Finally, we simplify waiting for the completion of signaled operations. Where in the OFED API, the developer has to write code for installing and (re-)arming an Event Channel which eventually returns an event containing a Completion Queue to be polled, we provide the following function for hiding this complexity:

- await_completions(cq_type, num_wcs, *wc_list, iw_ctx)

The CQ type specifies whether we wait for an event on the Receive Queue or on the Send Queue—we use separate CQs for the Receive- and Send Work Completions. With a single call to be above function, we can reap several Work Completions (num_wcs) which are returned through a pre-allocated list (wc_list). Again, the connection in question is selected by the iWARP Context.

Channel Teardown

Closing an iWARP connection can happen actively with iw_disconnect(iw_ctx) or passively by calling iw_await_disconnect(iw_ctx). The connection to be closed is defined once more by the iWARP Context.

Summary

By hiding the complexity of the OFED API within the library and by combining all the state within the iWARP Context, we not only simplify and accelerate iWARP/RDMA application development but also make the application code less error prone. The connection establishment is reduced from 12 down to 2 calls. Furthermore, the asynchronous event handling is encapsulated and the results are offered through a simple interface. Also, the memory management and Work Request posting tasks have become easier and more natural. Hence, in-depth knowledge of the OFED peculiarities is no longer required.

Since our library is essentially a set of wrappers for the OFED API, it does not limit the original functionality: it is always possible to write code in a form mixed between the OFED API and the iWARP library, in case special features of the original API or individual RDMA programming objects are needed. The performance goal is also met: as we will show by experiment (later in this chapter), there is no significant performance difference between our library and the original OFED verbs.

The library has proven useful through various projects carried out as part of this thesis. Furthermore, it has been fed back to the OpenFabrics community which has shown great interest and does want to include it in future releases of OFED.

3.3.4 The File Abstraction - An Alternative Interface

In addition to the iWARP library, we have conducted a case study and feasibility assessment of an even more radical approach for hiding the complexity. Following the UNIX principle where everything is a file, we propose to use (a subset of) the POSIX file interface for performing data transfers over RDMA. To that end, we have developed a primitive architecture for transparent mappings between standard file operations (i.e., *fopen*, *fclose*, *fwrite*, *fread*) and RDMA operations (i.e., *RDMA Write* and *RDMA Read*). As it is merely a proof of concept rather than a final system, the mapping is straightforward as follows: *fopen* establishes a new connection to the remote host (unless the target is local) and creates the necessary buffer for holding the file content. *fwrite* and *fread* are mapped to *RDMA Write* and *RDMA Read*, respectively. *fclose*, finally, terminates the connection established with *fopen* and frees the buffer again (if it holds the last remaining reference to it).

We have realized that mapping by pre-loading[17] the functions provided by the GNU C Library (Glibc) with our own. In order to separate RDMA data transfers

[17] see http://www.kernel.org/doc/man-pages/online/pages/man8/ld-linux.so.8.html

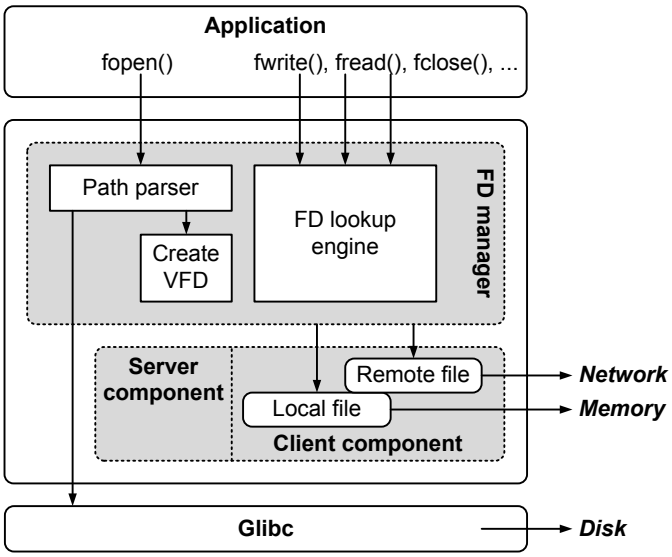


Figure 3.15: Remote file abstraction infrastructure.

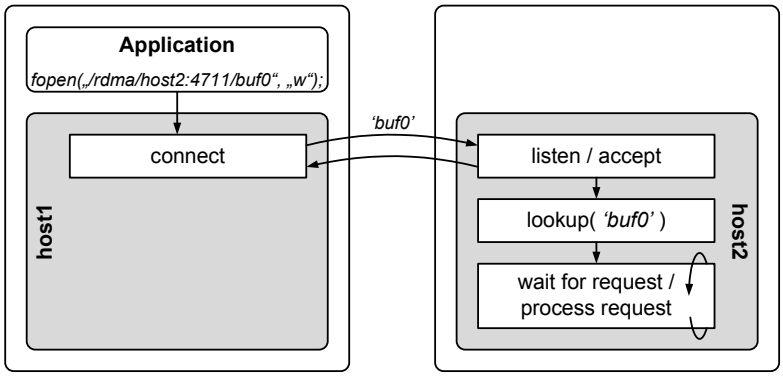
from ordinary file accesses, we use a special path for the RDMA "files":

```
/rdma/<host>:<port>/<buf_name>[:<size_hint>]
```

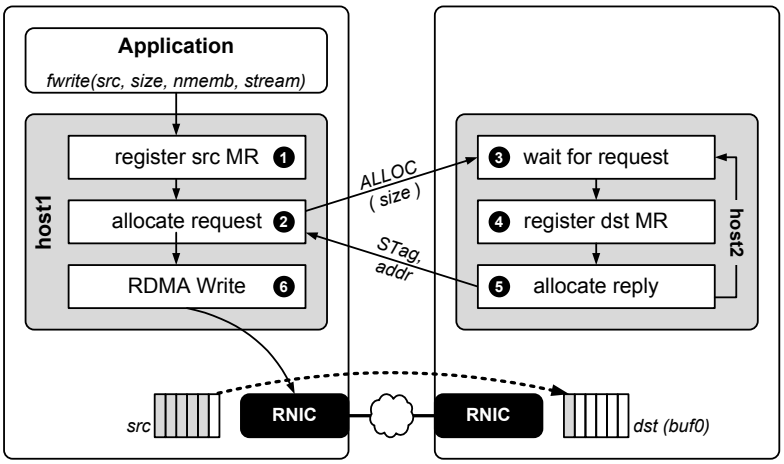
First, a designated directory prefix (`/rdma/`) is provided, followed by the address and port of the host on which the RDMA target buffer should reside (can also be the local host). The buffer itself, as any ordinary file, carries a human-readable name and is internally represented by a file descriptor (FD). As a last, optional argument, the expected size of the buffer can be specified.

Buffers can either reside on the local or on a remote host. Furthermore, buffers can be shared between several nodes and feature access restrictions (e.g., read-only). Sharing a buffer can, for instance, be used to build a video server from which the clients fetch data by reading from the shared buffer residing on the server. Thanks to the performance offered by RDMA, the server load is kept minimal (see also Chapter 6). Yet, there are no changes necessary with regard to the client media player as it can simply access the remote buffer as if it was an ordinary file. Another use case would be the staged processing of data where a number of nodes are connected sequentially to build a pipeline or even 1-to-n communication for map-reduce like processing.

Figure 3.15 depicts the implementation of our architecture which consists of three entities, plugged in between the application and the Glibc: the *file descriptor (FD) manager*, the *server-* and the *client component*. The FD manager is responsible for the buffer management by using OS provided file descriptors. The client component handles requests from the application level whereas the server



(a) Opening a file.



(b) Writing a non-existing file.

Figure 3.16: Examples of remote file operations.

component reacts to requests from the network, such as buffer allocation requests. The path parser within the FD manager processes all *fopen* calls and differentiates between ordinary file operations and RDMA transfers. All the other calls (targeting RDMA buffers) are handled by the FD lookup engine and are forwarded to the client component. If the access is targeting a remote buffer, it is propagated across the network, otherwise it affects local memory.

Figure 3.16 depicts two example runs of the architecture. In Figure (a), a remote buffer, called buf0, is opened with write access on host2 and port 4711. The path parser sees the /rdma/ prefix, creates a virtual file descriptor for that buffer (by opening `/dev/zero` locally) and registers it with the FD manager. Finally, it initiates a connection to the remote peer. The server component of the remote peer accepts the connection request and performs a lookup of the named buffer buf0. We

allocate new buffers lazily which means that they are only created at first access. Figure (b) depicts the initial write operation after having opened the "RDMA file". The FD lookup engine checks whether the local source buffer is already registered as an RDMA MR and if the remote buffer has already been allocated. Since we look at an initial write to a new buffer, none of them have been performed yet. Thus, in the second step, the client component of the local host sends an allocation request (indicating the size) to the server component of the remote peer. The remote peer processes the request by creating and registering a new RDMA MR and sends back the buffer advertisement. Now, the client component can post the *RDMA Write* Work Request to the RNIC for transmitting the data. Reading a remote buffer works analogously. This rather expensive initial buffer setup (allocation and buffer advertisement) can be amortized by repeated data transmissions.

A similar approach to ours was suggested by Goglin and Prylli [GP03] with their *Optimized Remote Filesystem Access* (ORFA). While their approach was similar with respect to the pre-loading idea, they based their implementation on top of a file system while we operate directly on buffers. Our approach has a lower overhead because the buffer access is more direct. On the other hand, having a file system provides a greater flexibility and richer feature set.

In the same spirit, Flouris and Markatos [FM99] argued on the leverage of using remote memory as a network RAM disk. They suggested an architecture, which created a virtual block storage device from non-used memory spread across interconnected workstations. Even though their system was based on TCP and not on RDMA, they could show that remote memory access was faster than local disk access. This was used, for instance, as remote swap space. Replacing TCP with iWARP/RDMA is likely to further improve the performance of their system.

Summary

The architecture, proposed in this section, is able to completely hide the (RDMA) network communication behind the well-known file interface of UNIX similar to network file systems like NFS. However, our architecture is much more lightweight and more direct in terms of buffer access—we do not initiate data transfers using RPC. Thanks to the buffer sharing capabilities, multiple nodes can be easily combined for creating processing pipelines or map-reduce setups.[18] Even though, any application that is operating on files can use our infrastructure for remote data exchanges without modification, the abstraction does not offer the same level of flexibility as the previously presented iWARP library or the original OFED verbs. In the following, we will therefore only focus on the original verbs as well as the iWARP library.

[18] Mutual exclusion and synchronization mechanisms are outside the scope of this work.

3.4 iWARP in Action

We will now go back and present the iWARP library in action: first, we show a complete "Hello World"-like sample application and then continue with a suite of micro benchmarks for assessing the potential of the technology.

3.4.1 The "Hello iWARP" Application

Figure 3.17 shows a simple but complete iWARP application both, from an *Initiator* and from a *Responder* perspective.[19] The application is split into three phases: in the first phase (lines 1–10), the nodes establish an iWARP connection. In the second phase (lines 11–18), the *Responder* performs a buffer advertisement (line 12) upon which the *Initiator* issues a *RDMA Write* into the advertised buffer (line 14) followed by an immediate *RDMA Read* from that buffer (line 15). In essence, the *Initiator* reads back what he has just written in the preceding operation which allows him to locally verify the correctness of the data transfers. Since the *RDMA Write* and *RDMA Read* operations are one-sided, the *Responder* does not know when the *Initiator* has finished the data transfers. Therefore, the *Initiator* signals protocol termination to the *Responder* explicitly through a final *Send* operation (line 17). In the last phase (lines 19–25), the buffers are deallocated and the connection is closed.

While this protocol looks straight forward, there are a few subtle but important details worth discussing here. Due to the asynchronous- and one-sided nature of the interaction, designing an application protocol for RDMA-based communication is highly error prone—particularly facing race conditions.

Send/Receive Synchronization

As mentioned earlier, each inbound *Send* message consumes *exactly one* Receive Work Request from the Receive Queue. This implies two things: first, there must be a Receive WR pending on the RQ when the *Send* message arrives. Second, the Receive WR must reference a destination buffer which is large enough for the entire data of the *Send* message. If the RQ is empty when an inbound *Send* message arrives or if the referenced Memory Region is too small, the data transfer fails and the iWARP connection is terminated—there is no second chance.

In order to prevent race conditions between inbound *Sends* and pending Receive WRs, the protocol must be well synchronized. In our example, such a synchronization can be seen on lines 11–13 at the *Responder* side: we post the Receive WR targeting the control buffer (line 11) *before* we send out the buffer advertisement

[19] The error handling has been omitted.

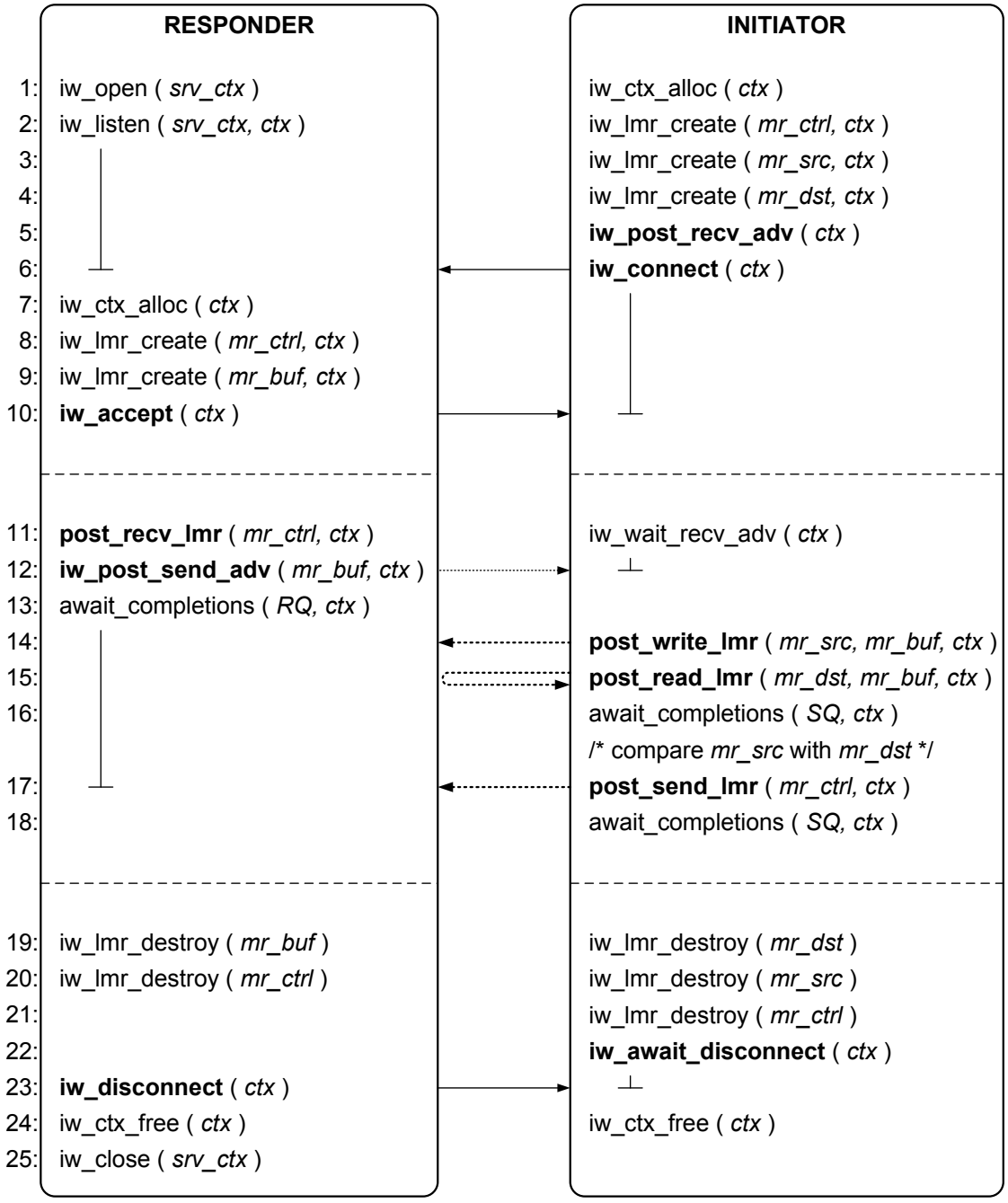


Figure 3.17: "Hello World" on iWARP/RDMA.

for which the *Initiator* is waiting (line 12). This guarantees us that the *Initiator* can not issue the *Send* operation (line 17) before we have a Receive WR in place.

An alternative approach (for more complex protocols) would be to have a large Receive Queue and making sure it is refilled as soon as the number of available WRs drops below a certain threshold. Yet, that method provides no guarantee against race conditions (e.g., on bursty data transmissions) and can be wasteful in terms of RNIC resources, especially if a large number of QPs are handled like this in parallel.

Waiting for Work Completions

Another important protocol design question is when to wait for Work Completions. While it is generally desirable to wait as infrequently as possible, there is the occasional situation where the peers must be synchronized before they can proceed. As can be seen in our protocol, we also wait for the completions of certain operations. In the following, we elaborate on the reasons.

The *Responder* posts two Work Requests: a *Receive* (line 11) and a *Send* (line 12). While it does not have to wait for the completion of the buffer advertisement, it *must wait* for the WC of the Receive WR in order to know when the *Initiator* has completed the one-sided data transfers. As soon as it receives that WC, it can safely terminate the connection.

Also the *Initiator* has to wait for some completions: first, it cannot start issuing the one-sided operations prior to receiving the buffer advertisement (line 11) and thus has to wait there. On the other hand, even though the *Initiator* reads the remote buffer right after having written it, it does not have to wait for the *RDMA Write* completion because the RDMA verbs dictate in-order delivery of the data and processing of the requests on a single iWARP/RDMA channel. This means that the RDMA Read Response is not processed by the *Responder's* RNIC before the *RDMA Write* data placement has finished which guarantees that the data we read back is valid and corresponds to what we have written earlier. However, prior to performing the buffer comparison, we have to wait for the *RDMA Read* completion—the state of the mr_dst buffer is undefined before the respective Work Completion has been generated. We also have to wait again for the final *Send* completion (line 18) or else we might prematurely deallocate the mr_ctrl Memory Region while it is still in use by the RNIC.

RDMA Protocol Design Findings

Send/Receive synchronization difficulties, like the ones mentioned, complicate RDMA protocol design compared to TCP. Furthermore, also the one-sided *RDMA Write* and *RDMA Read* impose certain difficulties. This explains why many application

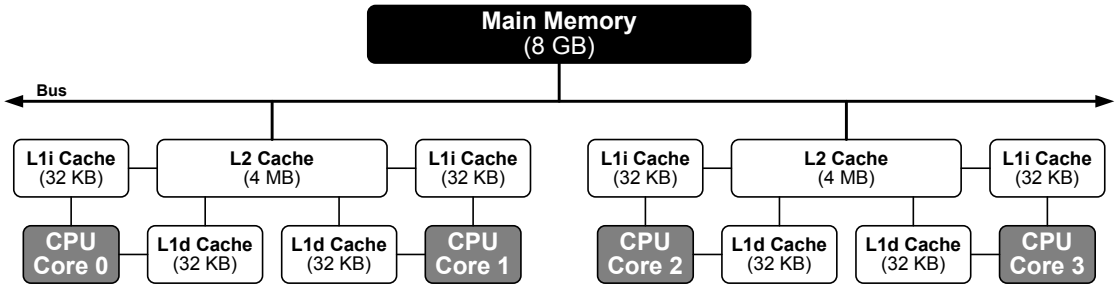


Figure 3.18: Intel Xeon cache layout.

developers are reluctant to move from the sockets interface to the RDMA verbs despite of the performance potential. The even bigger problem of choosing an appropriate buffer size will be addressed in Chapter 4. As we will also discuss in that chapter, the necessary synchronization overhead can (in some situations) even remove all the performance benefits and render RDMA useless for certain applications.

The protocol design difficulties are not only caused by the asynchronous nature of the queues and the one-sided communication pattern but also by the RNICs which are a black box from the programmer's perspective because they do not interact with the operating system for data placement at all. The only way to trace the data exchange, for debugging purposes for instance, is by inspecting the packet flow on the wire of a core network component (e.g., a switch or a router) through port mirroring—running network protocol analyzers like Wireshark on the communicating machines does not surface anything.

3.4.2 Micro Benchmarks

Now that we understand how iWARP works, how it is integrated into the host system and how it can be used by applications, we will look into the achievable performance by means of initial micro benchmarks. The benchmark results provide upper bounds for the actual performance as they were running on dedicated test systems—there was no other load on the machines during the tests. Real-world examples using larger applications which involve also computation and other tasks are presented and discussed in the second part of this thesis. Yet, in order to estimate the potential, the micro benchmarks serve as useful indicators.

Test Environment

Our testbed consists of an IBM BladeCenter containing HS21 BladeServers. Each of them is equipped with a quad core Intel Xeon CPU running at 2.33 GHz, 32 KB

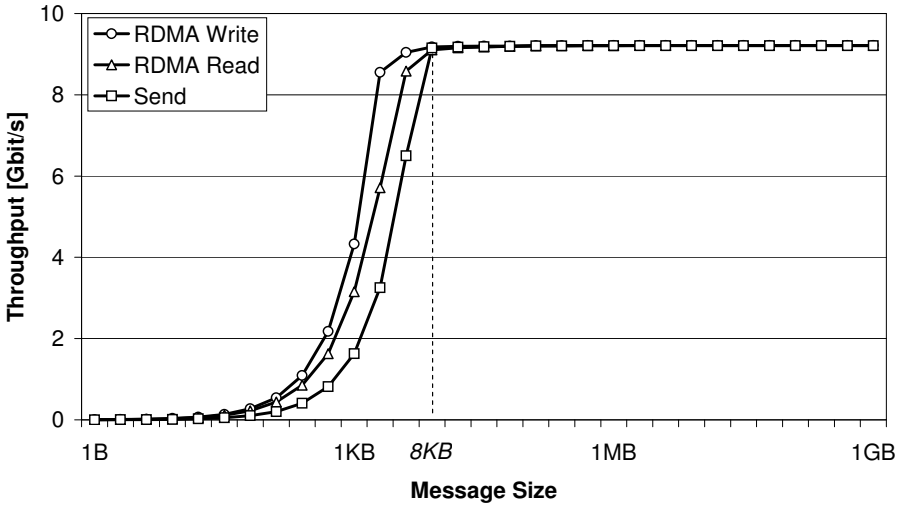


Figure 3.19: Direct comparison of all RDMA operations. While *RDMA Read* and *RDMA Write* perform slightly better for small buffers, the difference vanishes for large buffers.

L1 data cache and 32 KB L1 instruction cache, 4 MB unified L2 cache and 8 GB of main memory (see Figure 3.18).

RDMA hardware support is provided by Chelsio T3 RNICs (S320EM-BCH)[20] which offer full TCP/IP offloading (TOE) and iWARP RDMA support. The RNICs are interconnected through a Nortel 10 Gb Ethernet Switch Module.[21] The BladeServers are running Fedora Core 9 with a 2.6.24 vanilla kernel. The OpenFabrics Enterprise Distribution (OFED v1.3.1) software stack [ofe] serves as OS interface to the RDMA subsystem. Unless stated otherwise, the tests are all based on our iWARP library and not on the plain OFED verbs. A comparison between the verbs and the library is provided as well.

RDMA Operation Comparison

We run three sets of benchmarks: one for each operation type supported by iWARP/RDMA (*Send/Receive*, *RDMA Write*, *RDMA Read*). In each set, we perform a unidirectional bulk data transfer between two nodes. The data is transferred in messages of sizes between 1 Byte and 1 GB. Figure 3.19 shows the throughput measured on the application level for the three operations.

In accordance with Bell et al. [BBC+03], we find that the underlying 10 Gigabit link can only be fully utilized when transferring the data in large enough

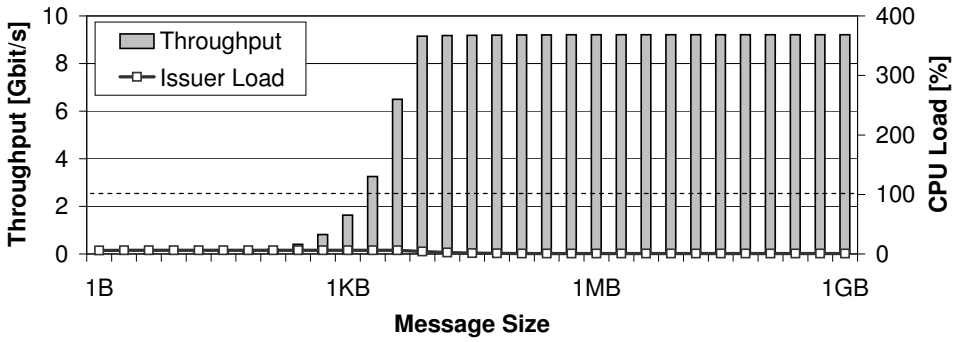
[20] http://www.chelsio.com/assetlibrary/products/S320EM-BCH_Product_Brief.pdf
[21] http://www.bladenetwork.net/BNT-Virtual-Fabric-10G-Switch-Module.html

messages.[22] The reason for that is on one hand the reduced interaction between the application and the RDMA subsystem because shipping the data in larger chunks requires fewer Work Requests to be posted to the Send Queue. On the other hand, the cost for processing small messages is dominated by the per-packet rather than the per-byte cost which renders small data exchanges inefficient. As we will discuss in the subsequent chapter, RDMA is only beneficial for large data transfers—for small ones, TCP is often preferable.

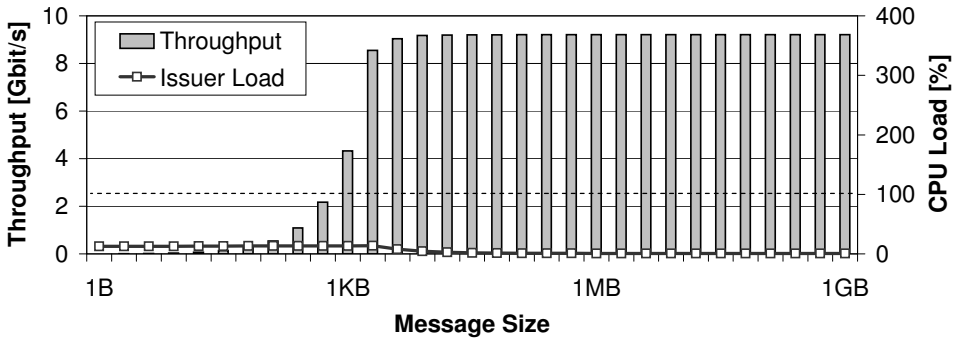
Figure 3.19 further shows that the performance of all operations is roughly equal—except for the band between 128 B and 4 KB where the *RDMA Write* performs best, followed by the *RDMA Read* and finally the *Send* operation. The important implication from this result is that the operation to be used for an application protocol can be chosen based on the desired semantic only (one-sided or two-sided; push-based or pull-based).

Figure 3.20 shows the throughput for each individual operation and includes the CPU load induced on the node issuing the Work Requests. The CPU load measurements were conducted using OProfile [opr]. We have chosen the scale to range from 0 to 400 % to reflet the 4 cores available in the systems. Figure 3.20(a) depicts the result of the two-sided *Send/Receive* communication. It can be seen, that the CPU load is low for small messages (≤ 7 %) and negligible for large messages. The CPU load for small messages is a result of the frequent Work Request posts by the application. The load on the receiving node is roughly equal to that of the sender because the processing overhead is essentially the same. The *RDMA Write* benchmark result is not only in terms of throughput but also in terms of CPU load almost equal to the *Send/Receive* benchmark (see Figure 3.20(b)). However, the CPU load on the receiver is negligible in all cases because the entire data placement is performed in hardware. Surprisingly and in contrast to the other two operations, the *RDMA Read* fully occupies one of our cores at the issuing side (the responder is idle). The reason for this observation is a shortcoming within the current version of the Chelsio T3 chip used in our setup: there is no support for unsignaled *RDMA Read* Work Requests. To minimize the interaction between the application and the RDMA subsystem, we let the RNIC generate a Work Completion only every n operations (with n being as large as possible; maximal 16384 in our case). In practice, this means that we set the *signaled* flag of the Work Request to 0 for the first $n - 1$ operations and to 1 for the final one. By waiting for this last Work Completion, we are (by definition of the verbs) guaranteed that all the previous operations have completed as well. However, this is not possible with *RDMA Read* yet—there, every operation generates a Work Completion which results in an extensive interaction between the application and the RDMA subsystem. Furthermore, appending Work Completions to the Completion Queue within

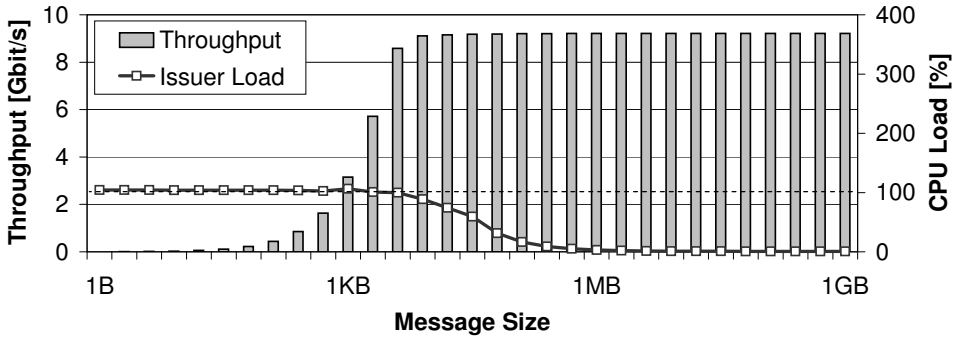
[22] In our setup, the minimal message size required to saturate the link is about 8 KB.



(a) *Send/Receive*



(b) *RDMA Write*



(c) *RDMA Read*

Figure 3.20: iWARP/RDMA micro benchmark. Transferring bulk data between two peers using all the available RDMA operations. RDMA performs best when using large messages ($\geq$ 8 KB).

the subsystems seems to be a comparably expensive task: waiting for n Work Completions and reaping them all in one go did not bring down the CPU load significantly. According to the verbs specification, *RDMA Read* operations must be able to complete unsignaled—Chelsio has announced support for unsignaled *RDMA Read* operations in their new T4 chip. Using only signaled Work Requests results in a comparably high CPU load also for *Send/Receive* and *RDMA Write*. The important message here is that being able to pipeline RDMA data transfer operations and only wait for a completion after a certain (perferably large) number of operations is vital for achieving a low host processing load on the node issuing the operations.

In summary, the benchmarks have revealed the following:

1. In order to fully utilize the available bandwidth at negligible host processing overhead, the data must be exchanged in chunks of a (system-dependent) minimal size ($\geq$ 8 KB in our setup).

2. All three data transfer operations offered by iWARP (*Send/Receive*, *RDMA Write*, *RDMA Read*) perform very similar in terms of CPU load and achievable application throughput (except for a small band where the *RDMA Write* and *RDMA Read* perform better; between 128 B and 4 KB in our setup). Hence, the choice of operation to be used can be made entirely on the most appropriate application semantic.

3. Work Requests should be posted unsignaled whenever possible because the production and consumption of Work Completions is work intensive and eliminates precious CPU cycles which could be spent on application processing otherwise.

iWARP Library Overhead

While the above experiments were all based on our iWARP library presented in Section 3.3.3, we need to compare the results with the plain OFED verbs.

As can be seen in Figure 3.21, there is no significant difference in either the CPU load or the achieved throughput between the plain verbs and our library. The chart only shows the result for the *RDMA Write* benchmark but we have found the same also for *Send/Receive* and *RDMA Read*. In particular, the high CPU load due to the missing support for unsignaled *RDMA Read* operations is not a consequence of the library. In the following, we hence restrict our experiments and applications to be based on the library due to the easier and faster development.

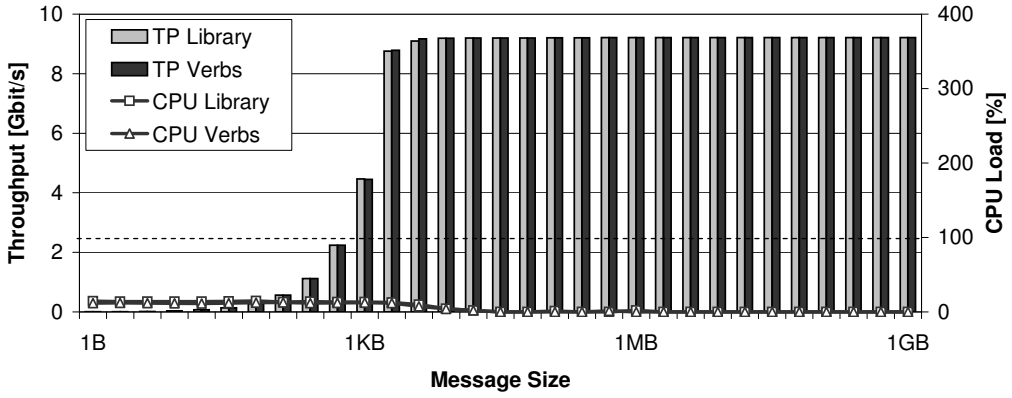


Figure 3.21: Direct comparison of the *RDMA Write* benchmark running on top of the iWARP library and on plain OFED verbs. There is no significant difference—neither in CPU load nor in achieved throughput.

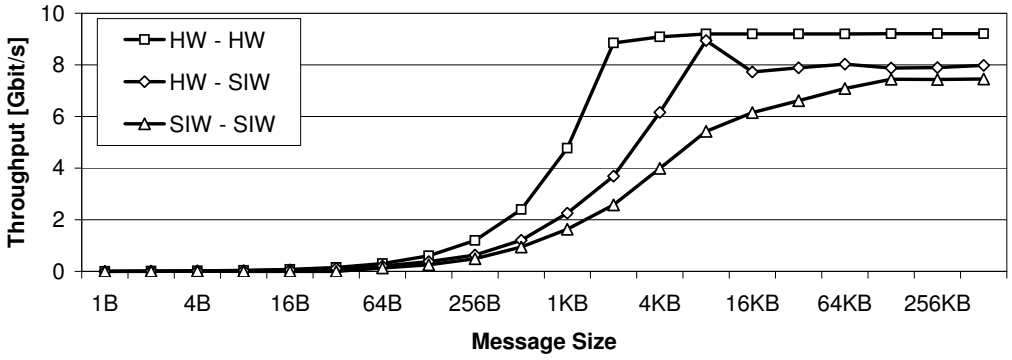


Figure 3.22: Throughput of Softiwarp (SIW) compared to the Chelsio RNIC (HW).

Softiwarp

Figure 3.22 shows the throughput measured with a preliminary version of Softiwarp on the same setup as before.[23] The benchmark was run in two different configurations. The first configuration (SIW - SIW) uses Softiwarp on the sender as well as on the receiver. In the second configuration, we have replaced Softiwarp by an RNIC on the sender (HW - SIW)—this reflects a mixed setup like the one used for the HD media dissemination system presented in Chapter 6. The third configuration (HW - HW) is added as reference.

It is obvious that Softiwarp does not achieve the same performance as a pure RNIC-based configuration. Yet, it provides a throughput which is high enough for 1-to-n configurations where the numerous clients only require a (potentially small) fraction of the total bandwidth. The mixed configuration performs slightly better than the one using Softiwarp on both sides. The SIW - SIW configuration does not provide a performance advantage but might be useful for preliminary iWARP application development and testing on machines which are not (yet) equipped with an RNIC.

There are two more details worth mentioning: first, the minimum message size required to reach the peak throughput is larger with Softiwarp than with RNICs—this must be considered when designing communication protocols for mixed setups. Second, the CPU load of the 'passive' host in one-sided Softiwarp operations is roughly equal to the one issuing the operations because there is no RNIC to handle the data transfer (load is not shown in the chart).

3.5 Summary

In this chapter, we have illustrated the enablement of RDMA over Ethernet with a bottom-up approach. After having described the iWARP protocol stack standardized by the IETF, we have motivated and presented our Wireshark network analyzer extension allowing the visualization of the iWARP traffic on the wire. We have then outlined the host system integration of iWARP/RDMA at the example of the OpenFabrics stack and introduced *Softiwarp*, our kernel module to enable RDMA on hosts which only feature a plain Ethernet NIC. In the context of the interface offered to the verbs consumer, we have argued about the issues of the OFED verbs and proposed a simplified iWARP library which allows an easier and less frustrating start into the world of RDMA application development. Furthermore, we have studied the file abstraction as an alternative interface to completely hide the RDMA communication. Finally, we have demonstrated iWARP in action by means of a "Hello World"-like example application as well as through a set of

[23] We restrict the discussion to *RDMA Write* because the other operations perform similarly.

micro benchmarks that give an initial idea of the performance potential and some limitations of the technology at hand.

3.6 Outlook

With iWARP, RDMA does no longer depend on special purpose (mostly proprietary) infrastructures like InfiniBand but can be used on the ubiquitous IP suite. iWARP, together with Softiwarp, enables on one hand RDMA for new application domains (e.g., HD Media Dissemination, Chapter 6) and on the other hand allows HPC applications to run on Ethernet fabrics. In the following, we will explore the parameter space in which iWARP/RDMA is most beneficial with another set of micro benchmarks before we assess the performance potential on real world applications in Part II of this thesis.

4

The Hidden Cost of iWARP/RDMA

So far, we have motivated iWARP/RDMA, discussed its inner workings and explained how it is used by the application programmer. It is now time to present an in-depth analysis of iWARP/RDMA in practice and identify its hidden costs and pitfalls. Subsequently, we propose optimizations which preserve the benefits of RDMA even under "difficult" circumstances. Finally, taking our findings and optimizations into account, we specify a set of critical parameters which need to be considered in order to assess the benefit of iWARP/RDMA for given application domains.

4.1 Introduction

In Chapter 2 we have motivated and presented RDMA: a mechanism whereby data is moved directly between the application memory of the local and remote computer. In bypassing the operating system, RDMA significantly reduces the CPU cost of large data transfers and eliminates intermediate copying across buffers, thereby making it attractive for implementing distributed applications. With the advent of hardware implementations of RDMA over Ethernet (iWARP; Chapter 3), its advantages have become even more obvious.

When we have started to write benchmarks and applications involving RDMA communication, we realized that there are environments where RDMA cannot provide the expected performance advantage over TCP. In this chapter we therefore analyze the applicability of RDMA in practice and identify hidden costs in the

setup of its interactions that, if not handled carefully, remove any performance advantage, especially in hardware implementations. From an application point of view, the major difference to TCP/IP based communication is that the buffer management has to be done explicitly by the application. Without the proper optimizations, RDMA loses all its advantages. We discuss the problem in detail, analyze what applications can profit from RDMA, present a number of optimization strategies, and show through extensive performance experiments[1] that these optimizations make a substantial difference in the overall performance of RDMA based applications.

4.1.1 Problem Statement

An important differences, which is often underestimated [DW07a], between RDMA and operating system driven TCP/IP communication, is that the application has to *explicitly manage the memory segment(s)* that will be used as communication buffer(s) at runtime. The application has to preregister certain parts of its memory with the RDMA subsystem as source and/or destination buffers before the data transfers. During registration, the memory pages get pinned by the OS making sure they stay resident and cannot be swapped out to disk. The pinned pages are then registered with the RNIC so that it can access them using DMA operations which eliminates the need for OS callbacks and intermediate buffering during the transfers (cf. Chapter 2).

MR registration happens through the resource management path which requires kernel activity and therefore induces a delay as well as a non-negligible CPU load. Even though the expensive data copy operations are avoided with RDMA, creating too much traffic on the resource management path can render RDMA useless. This is particularly true for the explicit buffer management with applications that are not able to reuse their buffers. In this chapter we show that the management of these user space communication buffers is crucial to implementing efficient RDMA communication.

We address the open question to what extent the hidden memory management costs affect the performance of RDMA. Our experiments show that, even for data transfers of moderate size, these hidden costs can completely eliminate the performance advantage of RDMA. We further present the critical parameters such as the increased connection setup time or the more expensive and complex RDMA object management that need to be considered when assessing the value of RDMA for any application.

[1] All the experiments presented in this chapter are based on the original OFED API and library—not on our simplified iWARP library presented before.

4.1.2 Contributions

In this chapter we describe the hidden costs of RDMA and show in detail how to design applications such that they take full advantage of RDMA. We also use our results to characterize which applications are likely to benefit from RDMA.

The contributions of this chapter are three-fold.

- First, we provide extensive performance experiments that show the hidden costs of RDMA and compare them with the potential advantages.
- Second, we describe cost-effective memory management strategies for RDMA and demonstrate their feasibility and performance with experiments on the Chelsio RNIC over 10 Gigabit Ethernet.
- Third, we present a list of the critical parameters based on which the added value of RDMA can be assessed.

4.1.3 Chapter Overview

The chapter is structured as follows: Section 4.2 briefly revises the RDMA mechanisms which are relevant in this context. Section 4.3 continues with the promised cost analysis for establishing an RDMA data path which includes the connection setup and buffer registration. Optimization strategies based on this analysis are presented in Section 4.4 before we summarize and present a raster for assessing the potential of iWARP/RDMA for given applications in Section 4.5.

4.2 RDMA Background

iWARP/RDMA was discussed in the previous chapter and is described in full detail in the RDMA Verbs Specification [HCPR]. In here, we focus only on the aspects that are relevant for our purposes in this chapter.

4.2.1 Asynchronous Communication Interface

In contrast to the classical TCP/IP semantics, all RDMA operations are executed asynchronously. They are described by the application in terms of *Work Requests* (WR) which are posted to the *Work Queues* for asynchronous processing by the RNIC. Since posting a WR is nonblocking and since the actual data transfer described in the WR is handled by the RNIC without CPU involvement, the application can overlap communication with computation. RDMA might be of limited use for an application that cannot profit from this.

4.2.2 RDMA Data Transfer Operations

The data transfer operations offered by RDMA are the *two-sided Send/Receive* as well as the *one-sided RDMA Read* and *RDMA Write*.

For the two-sided operations, the sending application specifies the buffer from which the data to be sent must be taken by posting a Send WR and the receiving application at the other side decides where to place the inbound data by posting a Receive WR in advance. In the case of one-sided operations, on the other hand, the *RDMA Write* copies data from a local MR into a remote one whereas the *RDMA Read* does the opposite without involvement of the remote host. Asynchronous notification about completion of the local Work Request can be demanded for all operations.

The following list presents peculiarities of the above RDMA operations which limit their applicability in certain cases:

- they are executable only on explicitly preregistered buffers
- the receiver of an inbound *Send* message needs to know the size of the inbound data in order to have an appropriate target buffer ready
- one-sided operations require knowledge of the remote buffer which necessitates a prior advertisement
- reregistration of a buffer requires a readvertisement inducing protocol delay
- the remote side of a one-sided operation cannot implicitly be notified of the completion of an operation

4.2.3 Explicit Buffer Management

Applications based on TCP/IP assume implicit communication buffers provided by the OS. The flexibility of that approach comes with the major drawback of requiring intermediate buffer copies, which induce a significant CPU and memory bus overhead as discussed in Section 2.1. The RDMA model on the other hand requires the application developer to allocate his communication buffers (or Memory Regions, MRs) explicitly and to register them with the RNIC for hardware accelerated direct data placement using DMA. Once registered, an MR has a fixed size which can only be changed by deregistering it and thereafter registering the buffer as a new MR. Since the registered MRs block the underlying memory for other applications, they should be deregistered when they are no longer needed. As we will see later in this chapter, MR (de-)registration induces a significant overhead. Applications that cannot reuse their communication buffer(s) therefore lose a significant performance advantage.

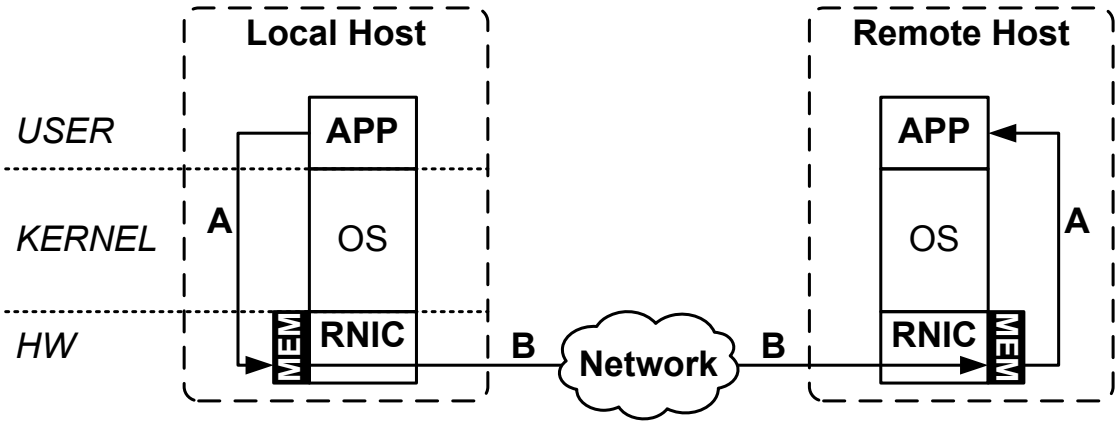


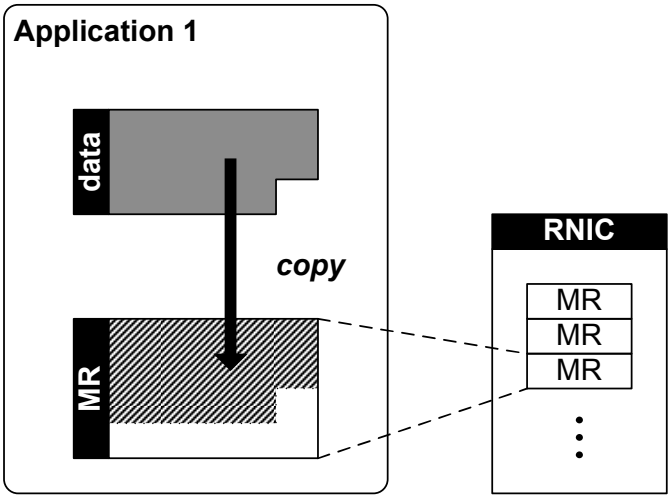
Figure 4.1: Data path for a network transfer bypassing the OS. The data travels directly between the network and the application buffer (A) as well as across the fabric between nodes (B).

4.3 iWARP/RDMA Cost Analysis

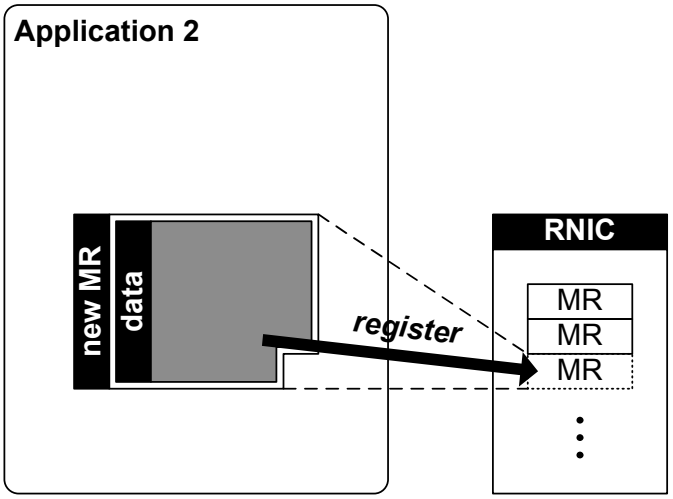
When looking at a network data transfer from a high-level perspective, the data path can be divided into two parts: First, the data is moved locally from the application buffer onto the wire (part A in Figure 4.1). In a second step, the data is transferred across the network (part B) to the remote host where it is moved into the destination buffer of the application.

The performance advantages of RDMA appear only after the whole path is established (A–B–A of Figure 4.1) and does not need to be changed anymore. In the case of iWARP/RDMA, establishing this data path involves setting up an iWARP connection between the nodes followed by the creation of the respective communication buffers (MRs) on each side. Both of these two steps are initialized through the resource management path across the kernel which is what makes them expensive.

Hence, for an application to profit from RDMA, it has to be able to reuse its buffers during operation. However, such reuse is only possible if the application can be designed to output all its data directly to that fixed user virtual memory address interval where the MR is situated. Furthermore the data set must always be of the same size or else either memory is wasted (new data set is smaller) or the transfer fails because the MR is not large enough (new data set is larger). If this is not possible, the application must either copy the data locally into an existing MR (Figure 4.2(a)) or register a new MR on the data (Figure 4.2(b)). Our experiments in this section indicate that only a combined approach is able to keep the overhead low. We use the same setup as described in Section 3.4.



(a) Copy the application output data into an existing MR.



(b) Register a new MR with the RDMA subsystem that fits the application output data.

Figure 4.2: Preparing application output data for RDMA transfer. In the case where the output cannot be directly steered into an existing Memory Region (MR), it either has to be copied into an existing MR or a new one has to be registered.

	1 B	1 KB	1 MB	1 GB
obj create	2 ms	2 ms	2 ms	2 ms
buf create	0.05 ms	0.07 ms	0.52 ms	565 ms
connect	201 ms	201 ms	201 ms	771 ms
send data	0.03 ms	0.04 ms	0.87 ms	869 ms
disconnect	10 μs	10 μs	10 μs	10 μs
buf destroy	0.03 ms	0.04 ms	0.23 ms	110 ms
obj destroy	0.06 ms	0.06 ms	0.06 ms	0.06 ms
total	203.17 ms	203.21 ms	204.68 ms	2317.06 ms

Table 4.1: Breakdown of the processing steps required for transferring n bytes of payload over iWARP/RDMA including the connection establishment and teardown.

4.3.1 RDMA Setup

Before remote DMA access is even possible, an elaborate connection setup following the creation of a number of RDMA objects is necessary (see Chapter 3). In order to assess the overhead compared to a simple TCP handshake, we have measured the time-to-first-byte—the time it takes for a client to connect to a server and send one byte of payload. On our link (round-trip time (RTT) of 25 μs) we have found a time-to-first-byte of 203 ms for RDMA and a mere 0.1 ms for TCP/IP.[2] A factor of two thousand. In the following, we analyze the cost of the individual steps required to exchange data over iWARP/RDMA in order to understand where this difference comes from.

For that, we have extended the time-to-first-byte experiment and transferred different amounts of data between 1 Byte and 1 Gigabyte (we call this experiment time-to-*nth*-byte). Table 4.1 shows the breakdown of the individual steps of the time-to-nth-byte experiment. The individual steps reflect the following tasks on the application level of the initiator side:

obj create. Creation of the following iWARP/RDMA objects: Connection Management ID, Event- and Completion Channels, Protection Domain, Completion Queue and Queue Pair.

buf create. Allocation of the data buffer (malloc) followed by its registration with the RNIC as well as creating the Send Work Request. This step will be discussed in detail in the following sections.

connect. Setting the connection parameters, issuing an MPA connection request and waiting for the connection established notification.

[2]Reading the clock induces an overhead of 0.14 μs which is negligible unless stated otherwise.

send data. Posting the Send Work Request onto the Send Queue of the RNIC and waiting for its Work Completion.[3]

disconnect. Issuing an iWARP/RDMA disconnect down call to the RDMA subsystem in the kernel.

buf destroy and *obj destroy*. Deregistering and freeing the data buffer and destroying the RDMA objects created at the beginning.

By looking at the total time elapsed for exchanging n bytes of data, we realize that up to 1 MB, the connection establishment is clearly the dominant cost. For 1 GB, the actual data transfer is more expensive.[4] We infer from this that iWARP/RDMA is only beneficial once a certain total amount of data is being transferred. Furthermore, as we have shown in the previous section, the buffers to be transferred must exceed a system-dependent minimal size to saturate the link; otherwise the *per-packet* rather than the *per-byte* costs prevail. This is confirmed again here by the *send data* values.

We have seen now that the connection setup itself is what causes the large time-to-first-byte. However, we have not uncovered the reason for it. We therefore time the individual steps of the connection establishment process from an application point of view. These are: setting the connection parameters (negligible), issuing the iWARP/RDMA connect call down to the subsystem (0.03 ms) and waiting for the connection established notification (201 ms). The last (and dominating) step includes the remote side buffer- and object creation and issuing the accept call (2.1 ms all together), the MPA handshake on the wire as well as moving the RNICs from TCP/IP mode into iWARP/RDMA mode. By packet analysis on the wire (using Wireshark), we have found the handshake and RTT delay to be negligible (in the order of TCP). However, moving the adapter into iWARP mode takes about 100 ms on each side which results in a total of the observed 203 ms.

In summary, our findings show that RDMA is clearly a bad fit for any application using many short-lived connections due to the comparably large connection setup cost.

4.3.2 Memory Region (De-)Registration

In our next experiment, we have identified the cost for registering and deregistering RDMA Memory Regions of different sizes.

[3]This does not mean that the remote application has seen a Work Completion for the Receive Work Request. It only guarantees that the remote RNIC has received, placed and acknowledged the data.

[4]We observe a higher connection cost for 1 GB because it includes the object- and buffer creation at the remote end which is a consequence of the API and our communication protocol.

Function `mr_bench(`*min, max*`)`: Memory Region registration benchmark.

```
1 for buf_size = min to max do
      /* map main memory                                          */
2     buffer = mmap(buf_size);
      /* time the memory registartion                             */
3     t_start = clock_gettime(CLOCK_REALTIME);
4     mr = ibv_reg_mr(protection_domain, buffer, buf_size, access_rights);
5     t_diff = clock_gettime(CLOCK_REALTIME) - t_start;
6     report(t_diff);
      /* deregister and unmap the memory again                    */
7     ibv_dereg_mr(mr);
8     munmap(buffer, buf_size);
9 end
```

Our benchmark core is sketched as pseudo code in the above listing. The MRs are registered on line 4. In our OpenFabrics RDMA subsystem, they are managed by the OFED user space library in a balanced search tree (red-black tree). When calling `ibv_reg_mr()`, a new MR reference is created and added to the tree. After that, the code traps into privileged mode where the user pages are mapped into the kernel virtual address space. In the next step, the underlying physical pages are allocated and pinned in main memory as necessary. Last, the page addresses are translated into bus addresses which are then registered with the RNIC for DMA.

Figure 4.3 shows the time required for registering Memory Regions of sizes between 1 B and 2 GB on a doubly logarithmic scale. As can be seen, the MR registration cost is constant up to and including 4 KB (page size) and increases linearly with the size of the MR (number of pinned pages) after that. The constant overhead for small buffers results from the rather long code path described above. After 4 KB, the dominating costs are page table lookups, walking and updating the involved data structures, translating the addresses and the like.

Since all the pages of the new MR must be pinned, they first need to be resident in physical memory. In the worst case, this means that each page of the MR causes a page fault (dashed line in Figure 4.3). To find out how much weight these page faults[5] carry in the context of MR registration, we have conducted a second test series with MRs whose underlying pages were already resident in physical memory before registration (marked with triangles in Figure 4.3). Beyond the size of a page, registering Memory Regions on pages which are not resident becomes significantly

[5]We only consider the cost of installing the page mapping. Potential disk I/O, in case the data has been swapped to disk, is not taken into account.

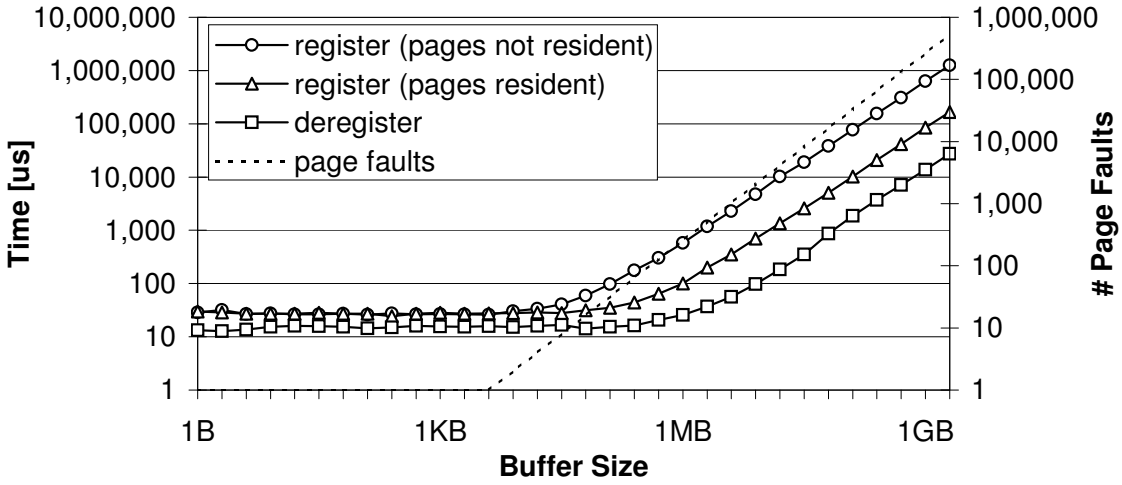


Figure 4.3: MR (de-)registration costs for resident as well as non-resident pages. Small buffers incur a constant overhead while large ones cause an overhead linear to the number of underlying pages. Deregistration is cheaper than registration for all sizes.

more expensive than registering an MR on resident pages. At a size of about 2 MB, the difference reaches almost an order of magnitude.

Huge pages can be used to reduce the total number of pages required to back a Memory Region of a given size. Since the overhead of registering a Memory Region increases with the number of pages involved, we expect to see a cost reduction when using pages of size 2 MB rather than the standard 4 KB. Our experiments have shown a reduction of the registration cost of up to 40% as compared to the standard page size (not shown in the chart). This is only true for Memory Regions which are larger than the size of a huge page, of course. However, the use of huge pages is not generally applicable.

Deregistration is a slightly simpler, inverse version of the registration code path. Since the pages of registered MRs are always pinned, page residency is not an issue here. As expected, Figure 4.3 shows that deregistration is significantly faster than registration (for all sizes). The explanation for this is that there is no need for address translations and that unpinning the pages is cheaper than pinning them. Up to and including 128 KB, the deregistration time is constant at $\sim$15 μs. For larger MRs, the time increases linearly with the buffer size as well. As pointed out before, deregistration of MRs which are no longer used is vital for system resource management because the underlying physical memory is pinned and not available for other processes.

We conclude that (de-)registering MRs induces a non-negligible hidden cost in terms of CPU load as well as delay and thus has a negative impact on the overall

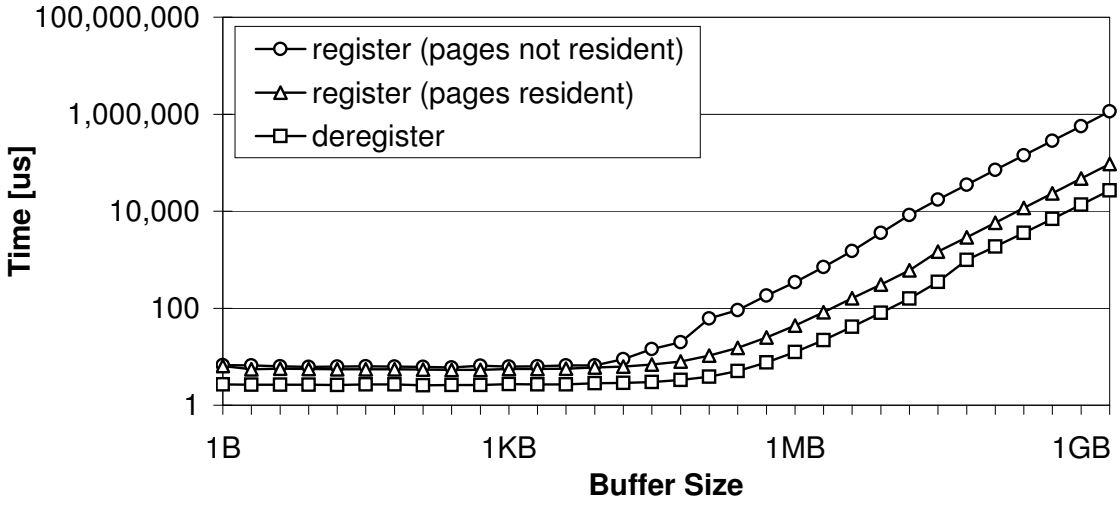


Figure 4.4: Softiwarp shows the same registration cost trend as the RNIC.

application performance. Yet, it is necessary for RDMA communication.

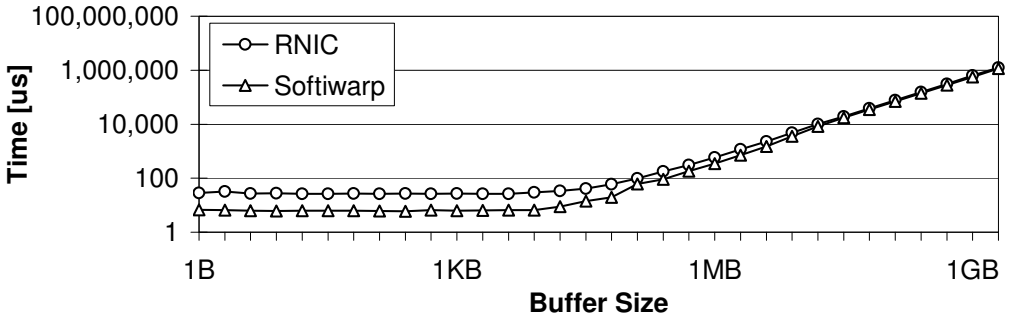
The Case for Softiwarp

While the above measurements were all taken on the Chelsio T3 RNIC, we will briefly review the results for the software-only iWARP solution: *Softiwarp*.

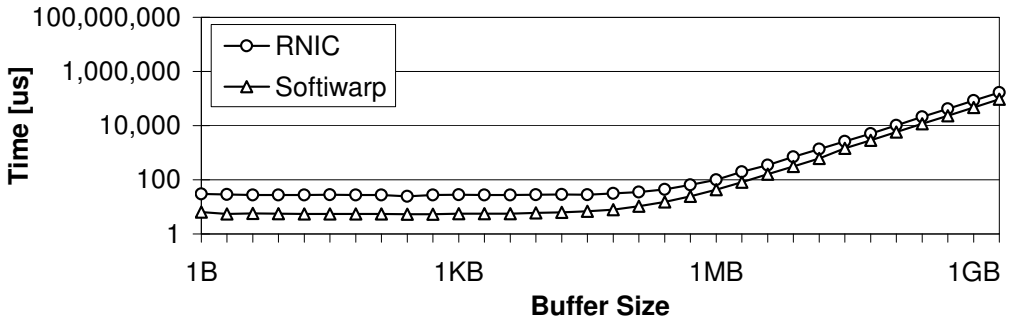
Figure 4.4 shows that the memory registration cost of Softiwarp follows the same trend as the one for the RNIC. There is a constant overhead for small buffers. After a certain size, the cost increases linearly with the number of involved pages. Again, registering buffers whose underlying pages are not yet resident are more expensive than the ones whose pages are installed. Furthermore, deregistration is always faster than registration.

Figure 4.5 compares the outcome of the three experiments one-on-one with the results from the RNIC. It can be seen that the registration cost on Softiwarp is slightly lower than the one for the RNIC. While the OFED MR management cost is incurred for both, Softiwarp does not involve any communication with the hardware which makes not only registration but also deregistration faster.

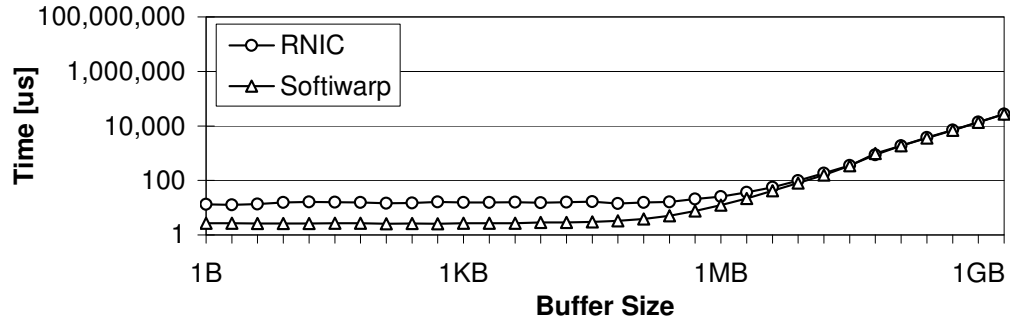
In the case of registering a buffer on non-resident pages, the page fault handling prevails for *large* Memory Regions resulting in an equal cost for Softiwarp and the RNIC (Figure 4.5(a)). A similar effect is visible for the deregistration experiment: for larger pages, the costs are roughly equal. When the pages are installed before the registration, however, the cost difference persists for all buffer sizes.



(a) MR registration causing page faults.



(b) MR registration on resident pages.



(c) MR deregistration.

Figure 4.5: Memory Region (de-)registration cost of Softiwarp compared to the cost incurred on the RNIC.

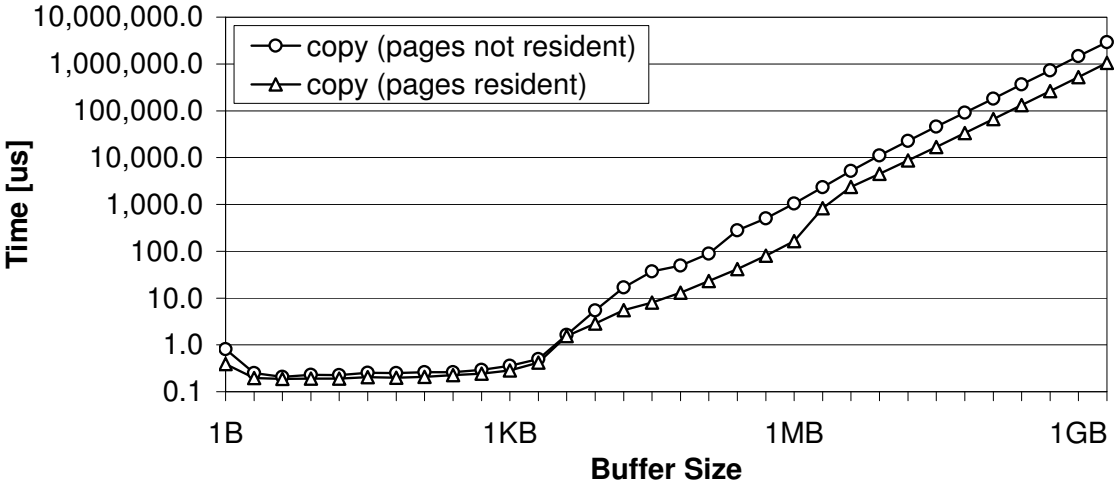


Figure 4.6: Memory copy costs for resident and non-resident pages. Constant cost for buffers up to half a page size (2 KB in our setup); linear increase for larger buffers.

4.3.3 Memory Copying

Having an application that cannot steer its output into an existing MR (e.g., coming from a mapped file) forces us to either register a new MR on the data (see previous section for the cost analysis) or copy the data into an existing MR if we want to use RDMA. We now consider the second option (copying). For that, we measure the pure copy performance of `memcpy()` on the same buffers as before, once with the pages resident in main memory and once causing page faults. Figure 4.6 displays the results of our experiments where the buffer size indicates the amount of data being copied.

Since `memcpy()` is a highly optimized function with a short code path, it has a very low overhead for buffers smaller than a page—the delay measured is actually dominated by the timer here. In our setup, up to 2 KB can be copied in under 1 μs which is about two orders of magnitude faster than MR registration. For buffers larger than 2 KB we see a picture which is similar to MR registration: The time increases linearly to the amount of bytes copied. As we will explore in Section 4.4, copying large buffers is significantly slower than registering them as new MRs. Page residency is also important for `memcpy()` but the performance improvement on large buffers is not as dramatic as in the case of MR registration.

Since `memcpy()` does not involve the RNIC, we expect its performance to be strongly dependent on the CPU as well as on the page size and cache sizes. Figure 4.7 confirms that for various systems. On an older Intel P4 1.8GHz, the copy delay is much larger than on a more recent Intel Xeon 2.66GHz for all buffer sizes.

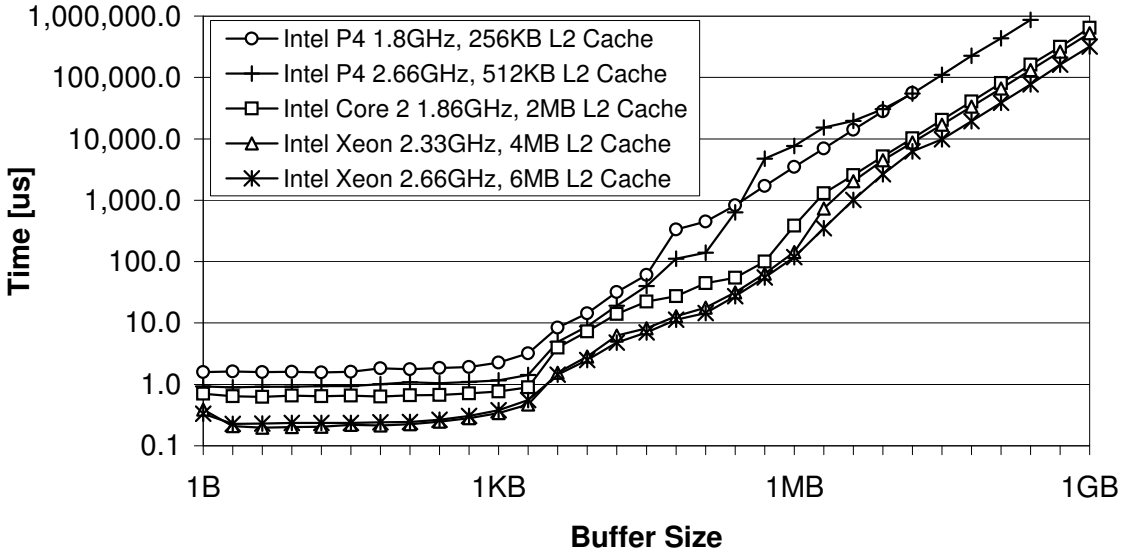


Figure 4.7: Memory copy performance on different systems. There is a strong correlation between the host processor speed and the copy performance.

4.4 Optimization Strategies

Based on our findings we now present several optimization strategies that reduce the overhead of RDMA communication.

4.4.1 Respect the Critical Buffer Size

In the case where we cannot steer our application output into an existing MR, we must either register the application output buffer as a new MR by means of `ibv_reg_mr()` or copy its content into an existing MR of appropriate size using `memcpy()` (see Section 4.3). Both approaches were used out of the box (i.e., no on-machine tuning was performed).

Figure 4.8 compares the delay of these two options and shows that MR registration has a significant overhead for small Memory Regions (<256 KB) as compared to `memcpy()` (up to several orders of magnitude!). After 256 KB however, it is much more efficient to register a new MR than it is to copy the application output. At about 4 MB (size of L2 cache), the delay difference reaches almost an order of magnitude in favor of the registration.

Furthermore, copying always induces 100% CPU load, whereas a reregistration (deregistration followed by registration) induces between 60% and 75% due to the additional hardware I/O wait time.

This leads to our first optimization: If buffer reuse is not possible but RDMA

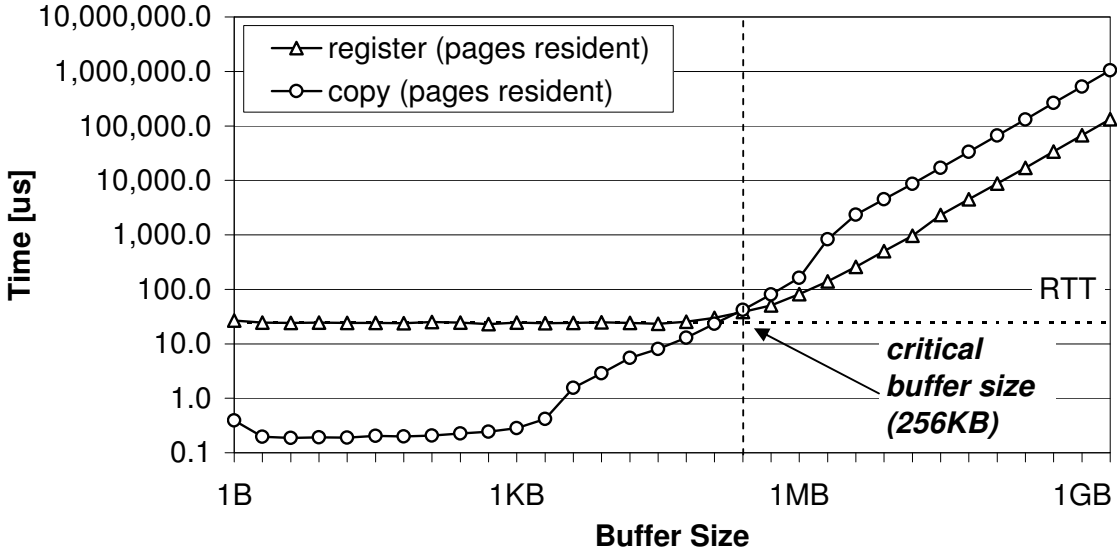


Figure 4.8: MR registration VS. copying with resident pages. While copying is significantly faster than MR registration for small buffers (below the critical buffer size), it is up to an order of magnitude slower for large buffers.

is still desired, buffers smaller than the critical size ought to be copied into existing MRs and larger buffers must be reregistered. Copying a large buffer (>2 MB in our case) is not only a lot slower than registering a new MR on it but it also consumes all the available CPU cycles and induces a higher load on the memory bus. In terms of cache pollution, registration is also preferred because it does not touch the data. As we will discuss later, this optimization matches typical communication buffer usage.

The important question now is what determines this critical size where registration outperforms copying. The answer is two-fold. The first aspect is the CPU performance: Since `memcpy()` is more CPU intensive than `ibv_reg_mr()`, running the above experiments on a slower CPU decreases the copy performance compared to registration, which results in a shift of the critical size towards smaller buffers. The second aspect which is even more serious is the page residency. Figure 4.9 shows that the registration of nonresident pages outperforms copying already for buffers larger than 32 KB which is a shift in favor of `ibv_reg_mr()` by a factor of 8. Using huge pages does not have a significant influence on the critical size as the cost reduction for `memcpy()` is about the same as for `ibv_reg_mr()`. When using Softiwarp rather than the RNIC, the critical buffer size also shifts towards smaller buffers (16 KB) because MR registration is faster on Softiwarp than on the RNIC, as shown before.

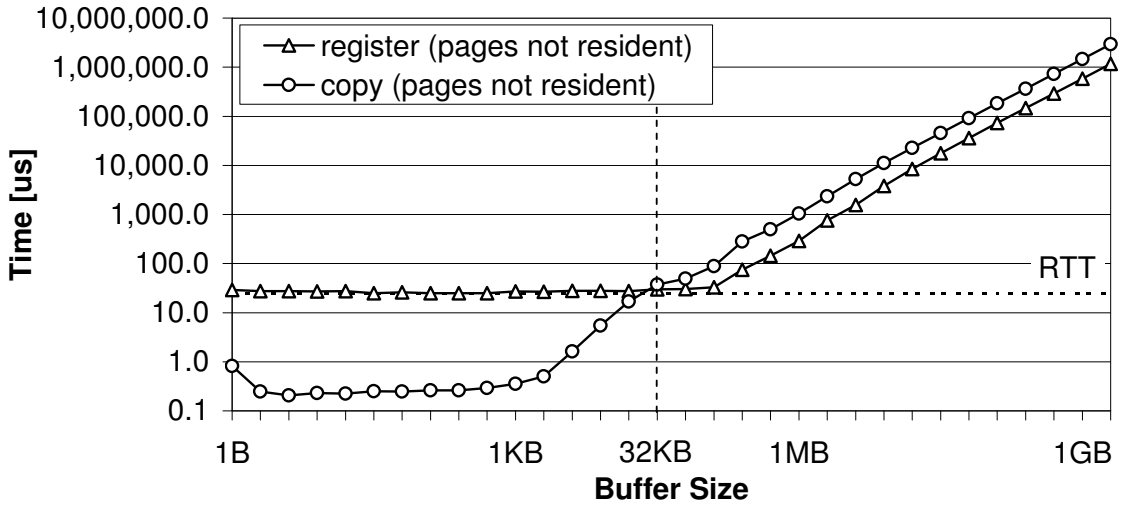


Figure 4.9: MR registration VS. memcpy with nonresident pages.

4.4.2 Overlap Buffer Management with Communication

The second optimization is to amortize the MR (de-)registration cost by overlapping it with waiting for a message from the remote host. The dotted horizontal line in the previous figures marks the round-trip time (RTT) between two RNICs through our 10 GbE fabric ($\sim$25 μs). It is a coincidence of our test system that the RTT is almost equal to the constant cost for registering small MRs—MR registration does not involve network communication. We can register a buffer on resident pages of up to 64 KB (or 8 KB in case of nonresident pages) within the RTT without inducing an overall protocol delay. MRs of up to 1 MB can be deregistered during RTT. The 64 KB for registration and 1 MB for deregistration are pessimistic lower bounds since our RTT does not take into account any application processing delay which typical real world protocols have.

4.4.3 Register Buffer on Resident Pages

Our third optimization is to design RDMA-based applications such that they register their MRs shortly after having touched or created the data—especially if they encompass more than one page—which reduces the expensive page fault processing during registration and therefore reduces the registration time by up to an order of magnitude. This also allows larger MRs to be registered during RTT.

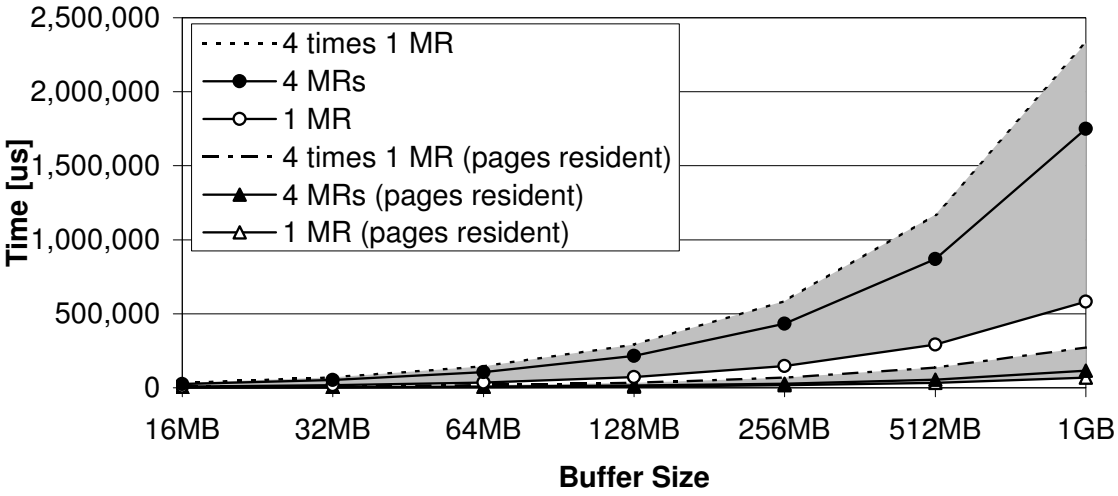


Figure 4.10: Parallel MR registration.

4.4.4 Parallel Buffer Registration and Applicability

Today, most machines are equipped with several CPUs and/or several cores. The question arises whether MR registration can benefit from that. Figure 4.10 depicts the latency for registering 4 MRs in parallel (one on each core). Again, we look at the case where the pages are resident (marked with triangles) and compare it with the one causing page faults. The dotted lines on top serve as reference showing the delay for sequential registration of the MRs on a single core and the solid lines at the bottom reflect the cost for registering a single MR. When comparing sequential with parallel MR registration, it is evident that the parallel registration can only profit from multiple cores when the pages are resident. Otherwise it is almost as expensive to register four MRs in parallel as it is to register them one after the other. We have found that the reason for this is the limited overall page fault rate of the system. Another motivation for the application to make sure the pages are resident before registration.

4.4.5 Suitability of the Optimizations

We now show why the proposed buffer management optimizations fit well with regard to real world protocols.

Large buffers typically contain the actual application data and are highly variable in size. Registering these large buffers as MRs, renders an equally large amount of the physically available main memory unusable for other processes due to the pinning requirement of RDMA buffers. Hence they should be deregistered when they are no longer needed. This matches our finding of reregistration being

cheaper for large MRs than copying.

Small buffers on the other hand are often used for exchanging control messages of constant size. By keeping them registered as MRs and refilling them with `memcpy()`, we induce a significantly lower delay and do not waste much memory even if they are currently unused. Since communicating hosts are often waiting for some kind of control messages from their peers before they can proceed with the protocol, it is vital that the delay for shipping these messages is low in order to get an efficient overall data exchange. An alternative approach is to transport these small control messages using plain TCP/IP but that results in the loss of packet ordering guarantees because of the extra socket which is undesirable in most cases.

Our experimental evaluation demonstrates clearly that a straight-forward explicit Memory Region management can degrade the overall application performance dramatically not only in terms of latency but also in terms of induced CPU load, cache pollution etc. For a good buffer management strategy—whenever buffer reuse is not an option—it is vital to respect the critical size where reregistration becomes more efficient. As we have shown, reregistering or copying the data according to the critical size results in a latency reduction of up to several orders of magnitude in both directions.

4.5 When is iWARP/RDMA beneficial?

RDMA over Ethernet offers a lower latency as well as a higher throughput than plain TCP/IP and even complements that with a close to idle CPU and reduced memory bus load that TCP/IP cannot provide. As we have discussed in the course of this chapter, there are, however, hidden costs in terms of the necessary buffer registration and connection setup.

Figure 4.11 repeats the *RDMA Write* experiment from Chapter 3 and demonstrates the performance penalty when the Memory Regions cannot be reused. The line charts represent the throughput and the bars the induced CPU load. The ideal scenario is shown in black: an application that can reuse its registered buffers extensively and therefore does not have to face the aforementioned costs. The CPU load is low and the link is saturated as soon as the per-byte cost dominates ($\geq$ 4 KB). The non-ideal scenario, where buffer reuse is not possible, is shown in gray and white. For the gray experiment, the communication buffers were constantly reregistered whereas for the white one, the buffers were registered once and thereafter refilled using `memcpy()`. Reregistration induces a slightly higher CPU load ($\times$1.7) than the ideal communication pattern and achieves a similar throughput ($-$8%). Copying, on the other hand, is significantly more expensive for large buffers (CPU load $\times$6) and faces a throughput reduction by up to 36%. We do not see a significant performance difference for the small buffers because

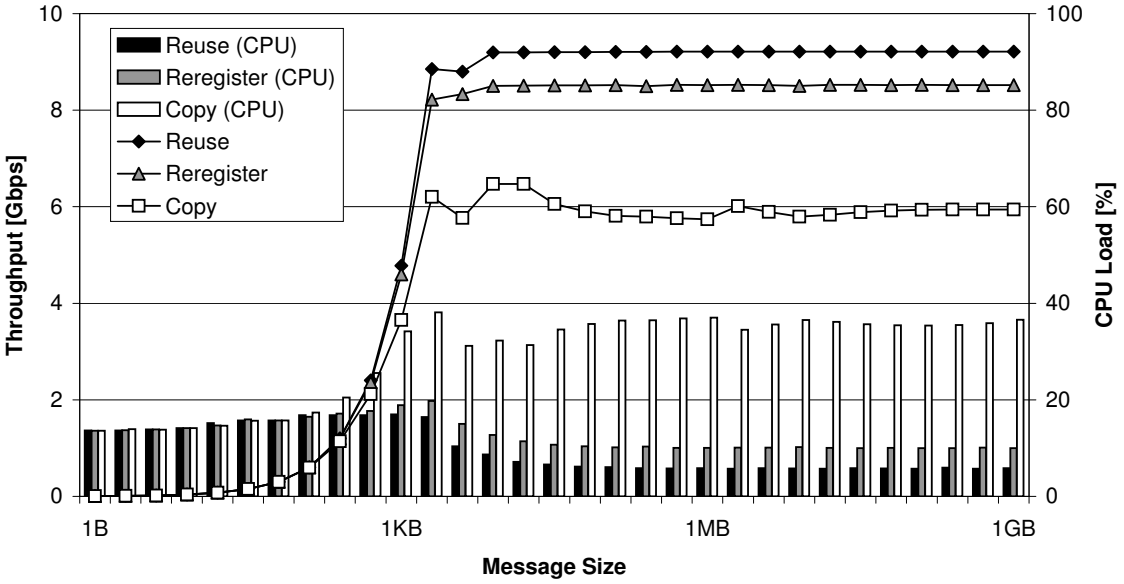


Figure 4.11: *RDMA Write* throughput for different buffering schemes. Only a *buffer reuse* strategy is able to fully leverage the link capacity while keeping the CPU overhead low. If reuse is not possible, reregistration is still preferable over copying for large buffers.

the buffer management overhead is hidden behind the per-packet processing and the transfer costs (cf. results from Chapter 3).

4.5.1 Critical Parameters

Despite all the benefits provided by iWARP/RDMA, there are applications that perform better when using plain TCP/IP due to the hidden costs presented in this chapter. In Tables 4.2 and 4.3, we address the open question as to when RDMA offers a benefit over traditional TCP/IP by listing the critical parameters and giving example applications for each parameter. In the second part of this thesis, we will look more closely into some of them and discuss a selection of real world applications in detail.

Table 4.2 lists the application characteristics which hint at a high performance improvement potential by applying iWARP/RDMA. As we have shown, the benefit provided by iWARP/RDMA is largest when the application in question is stable in terms of connections as well as buffers. Furthermore, it must require large amounts of data to be transferred across the network. Being able to utilize some of the RDMA specific features like the scatter-gather buffer referencing or the one-sided operations is not critical in terms of performance but might allow for a simpler

Application Characteristic	Example
large data transfers and large total volume	DB recovery, media streaming
able to reuse buffers	streaming, VoD
small data size variation	DB logging
long lasting connections	DB logging, streaming
data in main memory	HPC [NSL+08]
CPU intensive	HPC
utilize asynchronous interface	HPC, VoD
utilize scatter-gather list	HPC
utilize one-sided operations	streaming

Table 4.2: Application characteristics indicating that the application of iWARP/RDMA is likely to be beneficial. An ideal application has all of these characteristics.

Application Characteristic	Example
short time-to-first-byte required	DNS
short-lived connections	webserver [DW07a]
unpredictable msg size	RPC [MAFS03]
indirect buffers	Java apps

Table 4.3: Characteristics indicating limited or no benefit from applying iWARP/RDMA.

and less error prone protocol design and implementation. Table 4.3 lists strong indicators against RDMA. If any of them matches the application in question, extreme care has to be taken as iWARP/RDMA is likely to be outperformed by plain TCP/IP or other legacy protocols.

4.6 Summary

Even though RDMA offers zero copy and kernel bypassing for efficient data transfers between remote hosts in terms of CPU load and memory bus bandwidth, it has hidden costs. Therefore RDMA is not equally well suited for all applications. We have argued why a low-overhead communication buffer management is key to the efficient use of RDMA and have presented a number of optimization strategies. For the case where the buffers cannot be reused, we have shown how the communication delay is reduced by up to a couple orders of magnitude if the critical buffer size is respected. Large buffers must be reregistered and small buffers refilled. The result is a significantly lower latency, less CPU load and reduced waste of memory. In that context, we have pointed out the importance of registering MRs on pages which are resident in main memory. Finally we have specified the application parameters which must be considered when assessing the use of RDMA for a concrete application.

4.7 Outlook

By now we have learned how to apply iWARP/RDMA and understand when it is able to provide a substantial benefit over TCP/IP. In the second part of this thesis, we will apply our insights and move from micro benchmarks to real applications which allows us to verify our applicability assessment in practice.

Part II

Enabling Applications for iWARP/RDMA

5 Distributed Compilation Revisited

In the second half of this thesis, we present a number of real-world examples where we have either extended an existing application with the iWARP/RDMA communication technology or designed a new application from scratch that demonstrates how RDMA can be leveraged. By this we

- verify our findings and claims presented in the first part and
- provide a comprehensible assessment based on concrete use cases which helps to answer the question of whether or not applying RDMA will be beneficial for a given application scenario.

We start the discussion with porting a distributed compiler from its TCP/IP communication mechanism to an iWARP-based one. The goal of this exercise is to illustrate the steps which are necessary to extend an existing distributed application (which is currently based on TCP sockets) with RDMA capabilities. Furthermore, we point out what the critical aspects are which need careful consideration in order to maximize the gain.

5.1 Introduction

Given the growing size of software packages that are distributed in source code, short compilation times are desired. This is particularly true for operating systems which are distributed in source code and have to be compiled from scratch (e.g., the

Gentoo Linux Distribution [gen]). One approach towards speeding up the overall compilation without changing the compiler core is to *parallelize* the process by using more than one CPU simultaneously on the local host. An extension of this concept, known as *distributed compilation*, is to add idle CPUs from *remote hosts*.

RDMA traditionally aimed at improving high-performance computing applications and storage-area networks because CPU cycles and memory bus load can be reduced significantly by applying the featured *zero-copy* and *direct data placement* techniques. With the advent of *iWARP*, the RDMA technology is now available to the ubiquitous TCP/IP infrastructure and, thus, becomes interesting and relevant also for legacy applications.

In this chapter we explore how to convert legacy distributed applications to the RDMA communication model and exemplify the process with the conversion of the distributed C/C++ compiler *distcc* [Poo04] from TCP sockets to the RDMA interface. We are going to extend it with iWARP/RDMA support, trying to limit the number of changes in the code while aiming for the largest possible performance improvement. The RDMA-enabled *distcc* is termed *rdistcc* in the following.

5.1.1 Contributions

Based on our findings presented in Chapter 4, we discuss how to extend *distcc* with RDMA capabilities such that it can take advantage of some of the features provided by RDMA. The ideas and approaches we present can be generalized to a wide range of applications (see subsequent chapters).

Client-driven Application Protocol. The original *distcc* protocol will be transformed from a peer-to-peer communication model to a client-driven one which relieves the server from some of his work and thus allows for better scalability. In this context, we assess the need for hardware support and point out the alternative of software-only RDMA communication.

RDMA-accessible Files. Since the input data to a compiler are typically files residing on a hard disk, we show how even file-based applications can profit from RDMA although it has been designed for transferring data residing in main memory. We present a way to do this without changing the way in which *distcc* accesses the data.

Connection Manager. As we have seen in Chapter 4, having a static environment with long-lasting connections is important when using RDMA since its initial connection setup is costly and complex. By introducing a connection manager, we improve the communication efficiency and increase the scalability of *distcc* which (in its original form) faces a lot of churn.

RDMA Aspects beyond Fast Data Transfers. Even though a common claim is that RDMA is only beneficial if the data to be transferred is large and if

10 Gb Ethernet or faster is used, we demonstrate with software-only iWARP over 1 Gb Ethernet and the distributed compiler (which transfers not that much data) that there is more to RDMA than raw data copy performance.

5.1.2 Chapter Overview

After this introduction, we continue with a short description of *distcc* followed by a revision of some relevant RDMA aspects (Section 5.2). Next, we discuss the iWARP/RDMA extension of *distcc* in detail (Section 5.3): we start by showing where we can improve the current design (Section 5.3.1), move on with real-world considerations (Section 5.3.2), show how to access files using RDMA (Section 5.3.3), discuss the explicit buffer management (Section 5.3.4), give an example of how to do an iWARP-based file transfer (Section 5.3.5), present our connection management strategy (Section 5.3.6) and finally discuss the network communication protocol. In Section 5.4, we evaluate the benefits with a number of experiments before we hint at further use cases of the presented techniques (Section 5.5) and discuss related work (Section 5.6).

5.2 Background

This section provides a short overview of how *distcc* operates and briefly revisits the aspects and features of RDMA which are relevant in this context.

5.2.1 *distcc* Overview

distcc is a well established, easy-to-use wrapper for the GNU Compiler Collection (*gcc*) [gcc] that enables remote compilation. It currently supports TCP and SSH connections. For the remainder of this discussion, we refer to the host that initiates and offloads compile jobs as the *master* and the hosts offering their CPU(s) as *slaves*. The master leverages the available CPU resources of the slaves for the compile process.

The *distcc* setup consists of a daemon running on each slave and a wrapped *gcc* on the master. The master does the pre-processing and final linking locally. Only the "source-code-to-object" translation is offloaded to the slaves. As the pre-processing and linking steps are performed locally, no header or library dependencies exist on the slaves nor are any changes to the Makefiles necessary.[1] Figure 5.1 depicts a simplified example of such a distributed compilation where the master offloads the translation of two source files into object files to the slaves.

[1]For more details on *distcc*, refer to `http://code.google.com/p/distcc/`

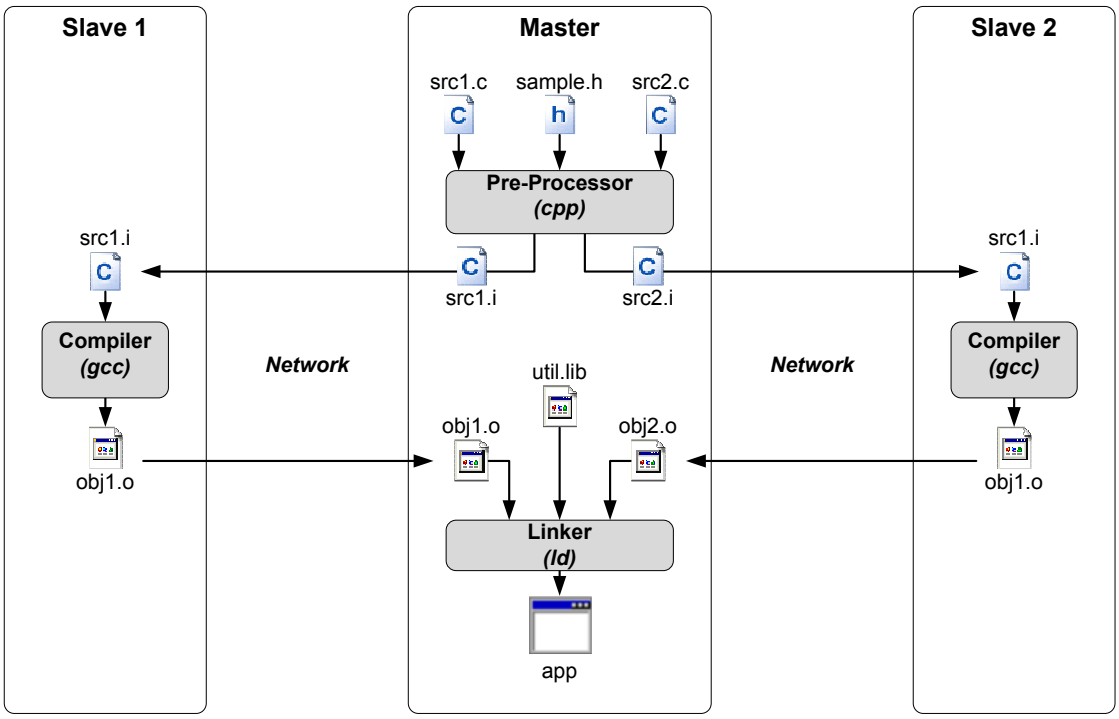


Figure 5.1: Example for a distributed compile job.

In the case of *distcc*, *NETWORK* stands for a legacy TCP/IP network. This is what we are going to replace with an iWARP connection.

The network communication protocol defined by *distcc* consists of four phases (see Figure 5.2): first, a TCP connection between the master and one of its slaves is established (*CONNECT* phase). After that the master sends the compilation instructions for the pre-processed source code followed by a file containing that code (*SEND* phase). The return path from the slave to the master is analogous, with the difference that now, first the outcome of the *gcc* invocation is sent followed by the resulting object file (*RECEIVE* phase). At the end of each compilation cycle, the TCP connection is closed (*DISCONNECT* phase).

5.2.2 Relevant Aspects of RDMA

We will now briefly revisit the RDMA features which are relevant for the upcoming discussion.

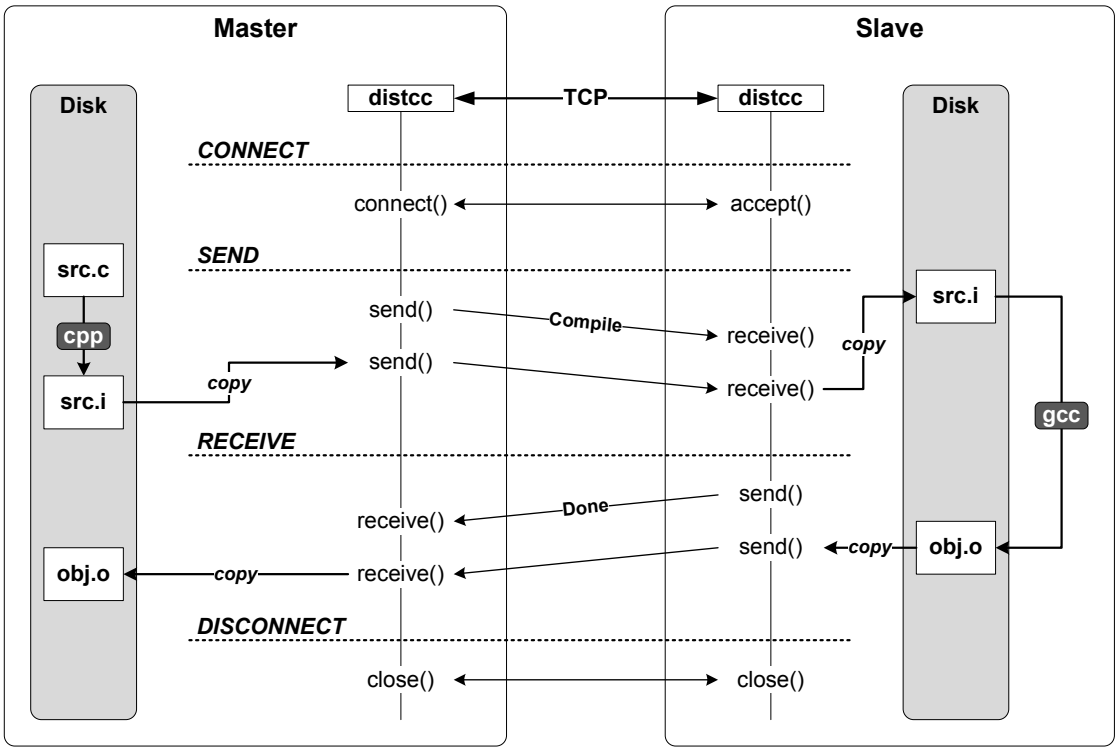
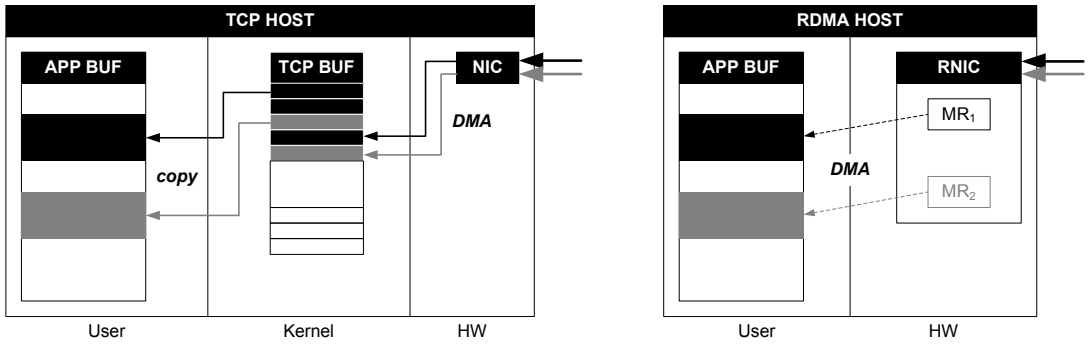


Figure 5.2: *distcc* network protocol.



(a) TCP socket buffers.

(b) RDMA direct data placement.

Figure 5.3: TCP socket buffer versus RDMA Memory Regions.

Direct Data Placement

Figure 5.3 depicts the difference between the flexible but more expensive socket buffers and the new RDMA-style data placement which we are going to apply.

In Figure 5.3(a), the inbound data is DMA'ed from the network card (*NIC*) into the intermediate socket buffer (denoted as *TCP BUF*). In a second step (when the user issues a TCP receive call), the data is *copied* from the kernel socket buffer into the user application buffer (denoted as *APP BUF*).

Figure 5.3(b) illustrates the Direct Data Placement approach. Thanks to the registered MRs, the RDMA-enabled NIC (RNIC) is able to directly DMA inbound data to the user application buffer without intermediate copying and without going through the operating system kernel. In Chapter 4, we have discussed the difficulties of this approach: the explicit communication buffer management. Memory Regions (MRs) have to be sized and registered before RDMA data exchanges are possible.

With the distributed compiler, we will see how that translates to real-world applications and learn that there are a number of factors to be considered when building the explicit memory management.

Asynchronous Operations

All operations (*Send*, *Receive*, *RDMA Read*, *RDMA Write*) are initiated and completed *asynchronously*. This means that the operations are described as *Work Requests* (WR) which are posted to Work Queues (i.e., to the *Send Queue* (SQ) or the *Receive Queue* (RQ)) where they are eventually, asynchronously processed by the RNIC.

The Work Queues are associated with *Completions Queues* (CQ) which are used to signal the outcome of the operation back to the application. In order to

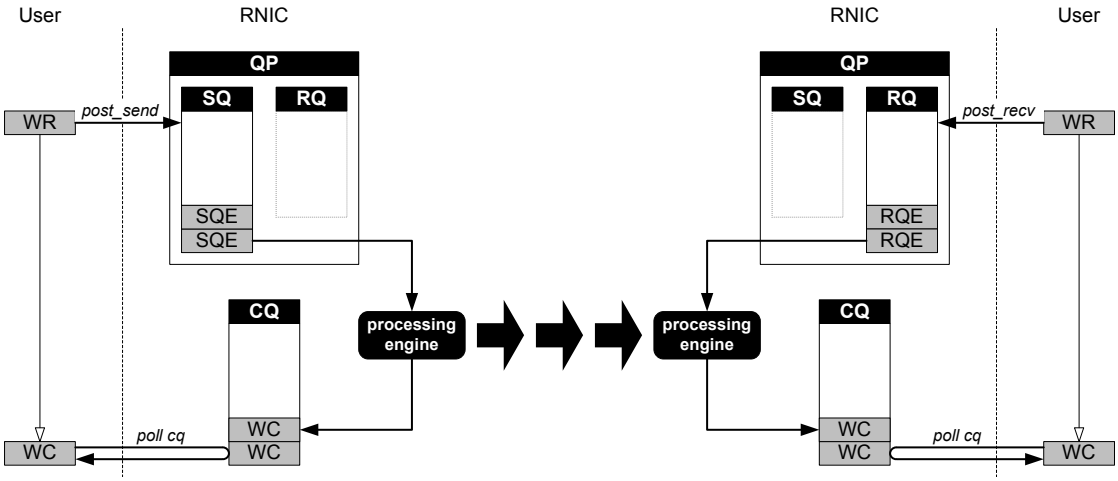


Figure 5.4: Asynchronous *Send/Receive* operations.

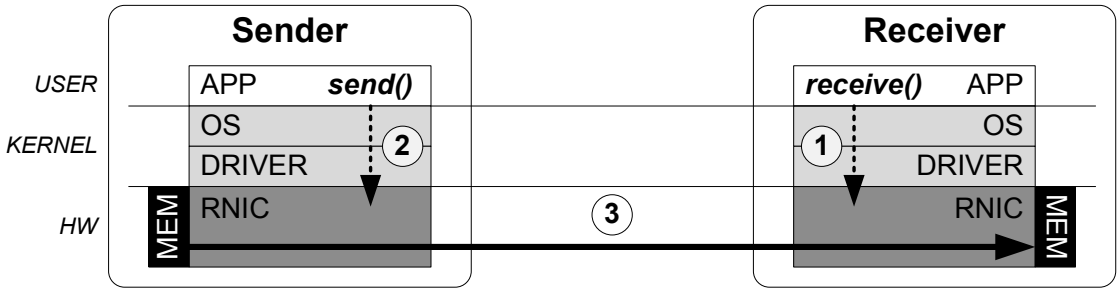
receive the Work Completions, the application may either wait for a respective event on the Event Channel (blocking) or poll the CQ at a certain interval (non-blocking). Figure 5.4 illustrates the asynchronous semantic at the example of a *Send/Receive* operation with CQ polling on the user level.

This API is different from the synchronous and blocking socket interface and the conversion is not always straight-forward. Due to the asynchronous completion, care has to be taken not to block the application yet not to waste CPU cycles for constant (unsuccessful) polling since we want to leverage the asynchronous nature in order to overlap computation (i.e., compiling source code or distributing compile jobs) with communication to get a performance improvement [BBC+03].

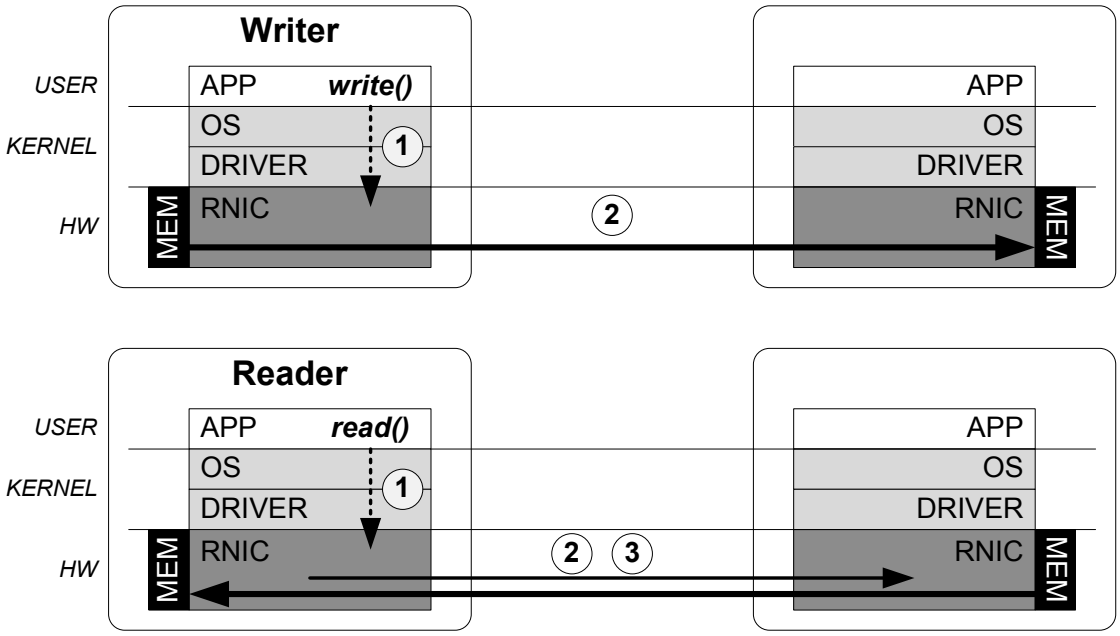
Two-sided versus One-sided Data Transfers

As mentioned in Chapter 2, RDMA defines a superset of the classical *Send/Receive* communication known from sockets. It introduces *RDMA Write* and *RDMA Read* as additional operations. An *RDMA Write* places local data into a remote application buffer whereas an *RDMA Read* fetches data from a remote buffer and writes it into a local one. In contrast to the *two-sided Send/Receive* communication—where the applications of both peers are involved in the data transfer—the *RDMA Write* and *RDMA Read* operations are *one-sided*. Only the application layer of the host issuing a one-sided operation is actively involved in the data transfer. At the remote host, the operation is handled by the RDMA device without application or kernel involvement. Figure 5.5 illustrates the two modes of operation.

Since socket-based communication is always two-sided, one might be tempted to simply map the socket calls to the *Send/Receive* operations. Even though



(a) Two-sided: *Send/Receive*.



(b) One-sided: *RDMA Write* and *RDMA Read*.

Figure 5.5: Two-sided versus one-sided operations.

this works in most cases, we will illustrate that the one-sided operations bear a potential beyond *Send/Receive*. It is thus important to consider the *semantics* of the data exchange operation when porting a legacy application to RDMA.

5.3 Extending *distcc* with iWARP/RDMA Capabilities

In this section, we will discuss in detail what changes are necessary in order to extend an existing (socket-based) application with iWARP/RDMA capabilities. In addition to that, we reason about our design decisions and motivate them based on our findings presented in the first part of this thesis.

5.3.1 How can *distcc* profit from RDMA?

The amount of data to be transferred between the master and a single slave is mostly small (up to a few megabytes per job). The network is thus unlikely to become the bottleneck. However, the CPUs on the master and the slaves are. The master is responsible for performing the following tasks: pre-process the source code, schedule the remote compile jobs, transfer the data and finally link everything together. The slaves on the other hand only translate the source code into binary format. We thus have an asymmetric work distribution—depending on the role of the host—in which the master becomes the bottleneck of the system when too many slaves are present.

Let the Slaves do the Work

The overall performance of the system is limited by the number of slaves the master is able to delegate jobs to. In order to reduce the master's CPU load, as much of the application logic as possible should hence be moved to the slaves. We address this problem by using *one-sided RDMA operations* because they enable a slave-driven communication protocol. As we will see, this results in a scalability improvement relative to *distcc*.

Reduce the In-Host Communication Overhead

The CPU cycles available for a distributed compile job can be increased both by adding more remote hosts to the setup as well as by taking unnecessary network processing load off the involved CPU(s). Naturally, remote compilation induces the overhead of distributing source code among the nodes offering their CPUs and of them returning the resulting object files back to the initiator. So to leverage the

total available CPU power from all hosts and to reach a high level of scalability, it is important to keep the in-host communication overhead small.

In the case of the distributed compiler, the following RDMA features thus aid in getting a better performance:

- overlap communication with computation by leveraging the *asynchronous communication interface*
- reduce the number of *CPU-driven copy operations* during the data transfers (see Figure 5.2)
- enable a client-driven protocol to take load off the central scheduler (master) with *one-sided RDMA operations*

5.3.2 RDMA Support in Practice

The master can reduce its CPU load using one-sided operations best if an RNIC (hardware acceleration) handles all the RDMA traffic. On the less-loaded slave side, an easily deployable, software-based iWARP solution (such as *Softiwarp* 3.2.2) with a conventional Ethernet NIC is sufficient. The slaves just need the RDMA semantics but not necessarily the hardware acceleration. With *Softiwarp*-enabled slaves, and a hardware accelerated master, the compute cluster can easily be extended up to many slaves while still remaining highly dynamic and flexible. By that, idle CPUs (e.g., workstations of a company) can be utilized for compilation.

However, in practice, machines are rarely equipped with RDMA hardware (only expensive servers are if at all). When using the original *distcc* application, performing a distributed compilation is possible with any ordinary machine. Furthermore, the role of the master should not be bound to a specific machine. In distributed compiling, the amount of data exchanged is typically not large enough to turn the memory bus of the machines into the bottleneck of the whole system. We thus argue that, in this scenario, we can replace the RNIC by dedicating a CPU to network processing while the other(s) are performing computation tasks. We therefore limit our experimental evaluation (Section 5.4) to the software iWARP module which allows RDMA-style communication on machines which are not equipped with an RNIC. *Softiwarp* gives us considerable flexibility in creating low-cost, RDMA/Ethernet-based compute clusters which is ideal for conducting RDMA experiments that focus more on the semantics rather than on pure performance.

For evaluations and experiments with hardware based iWARP, refer to the following chapters where we look at data volumes where dedicating CPUs to network processing can no longer replace an RNIC.

5.3.3 Making Files RDMA-accessible

The first thing that strikes as a bad fit for RDMA is that the input data for a compiler typically consists of files residing on a persistent storage device (i.e., a hard disk). Nevertheless, *rdistcc* can benefit from direct *memory* access by storing the files on a RAM disk mounted as a *tmpfs* [Sny90] volume (on all machines). The details are explained next.

Master Data Handling. At the beginning of the compile process, the source files are assumed to be stored on a hard disk drive attached to the master. We remember, that *distcc* needs to pre-process the source files locally before shipping them for remote compilation. We can exploit this step to make the files RDMA accessible without creating an additional overhead as follows. To pre-process a source file and prepare the result for later RDMA transfer to a slave, the pre-processor (*cpp*) invoked by *rdistcc* reads the source file (*src.c*) from the disk and writes the result (*src.i*) to a RAM disk (*tmpfs*) instead of back to the hard drive. We then use *mmap()* to create a shared mapping[2] of the pre-processed source file into the application address space and register it as an RDMA Memory Region.

Slave Data Handling. On the slaves, we always keep the source and object files in main memory to reduce file access times for the compiler. By using the *mmap*-approach, we eliminate the copy overhead for passing files between the *rdistcc* daemon (has memory view on the data) and the compiler (needs file view on the data). The object files resulting from compilation are written back to local memory.

Figure 5.6 illustrates the complete data path from source to object file the way it is implemented in *rdistcc*: first, the source (*src.i*) is placed in the RAM disk by the pre-processor. Next, the *src.i* is RDMA'ed to the remote host where the compiler transforms it into the object file (*obj.o*). Finally, the object is transferred back to the master node.

We argue that keeping everything in main memory on the slaves does not impose any restrictions for two reasons. First, because pre-processed source files are small in most cases (up to a few megabytes) and second, because a slave does only have to store a few source files at any time (equal to the number of parallel compile jobs offered to the master). It does not make sense for a slave to offer more compile jobs to the master than it is able to process in parallel (bound by the number of available CPUs). Hence, we do not consume a lot of memory altogether.

We illustrate that with a small example. Let us assume that our average slave is a 4-way CPU machine (thus offering 4 compile jobs). Further assume that the machine is equipped with 1GB of RAM (which is little nowadays) and that our average pre-processed file is 2 MB large and the average object file another 2 MB.

[2]It is vital to use a mapping which is shared to avoid the data being copied.

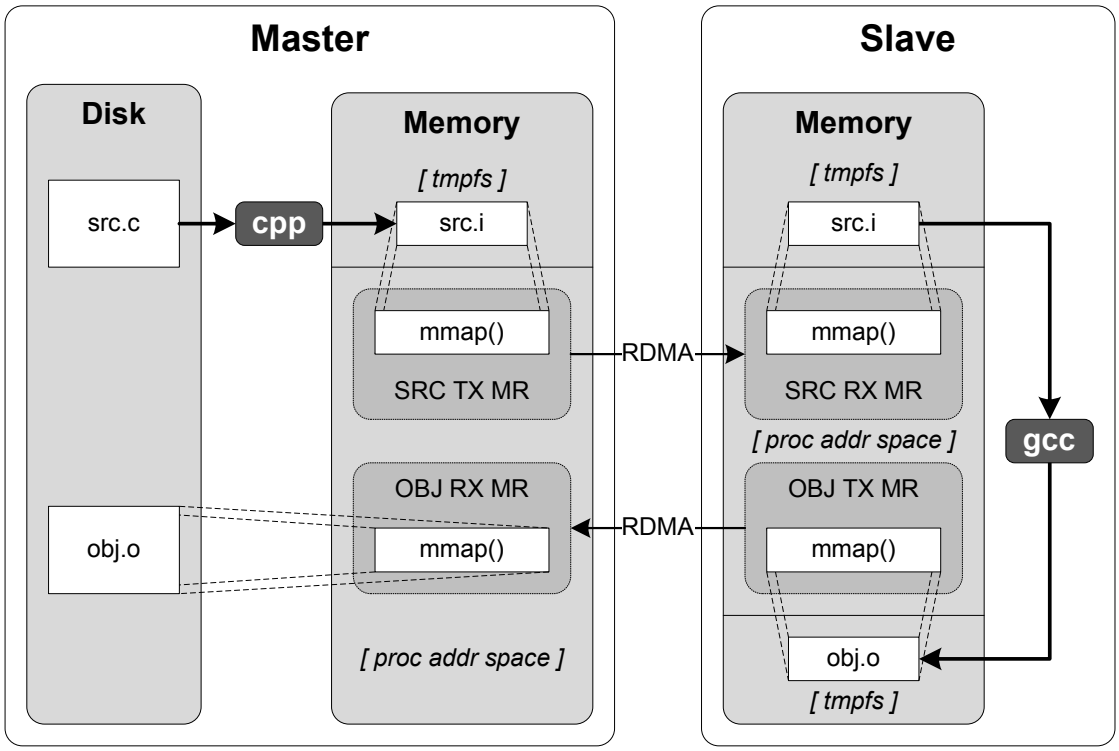


Figure 5.6: Data path for remote compilation.

In this (pessimistic) scenario, we have to store $4 \cdot (2 + 2)$ MB = 16 MB under full load which is a mere 1.6% of the total available memory. When using two more compile jobs than there are CPUs (which is suggested by the *distcc* author) we consume 2.3%. Compared with the memory used by *gcc* for the compilation, the data stored in memory by *rdistcc* is negligible.

If there is not enough memory available (unlikely), we can fall back to storing the data on disk at the slaves. Our implementation also supports RDMA transfers to and from hard disk instead of main memory. In that case, the mapping is file-backed and the *tmpfs* mount is not used. This comes of course with the drawback that the compiler can no longer directly operate on main memory but has to fetch the data from the disk.

Advantages of the Proposed Approach

The advantage of using a RAM disk in combination with a shared mapping is that *gcc* can access the data residing in main memory as if it were files on an ordinary file system—we thus do not need to modify *gcc*. At the same time, the RDMA subsystem can treat the (*mmap*'ed) file as a simple block of main memory. We

can thus associate a MR with the user address interval that corresponds to that block of memory and make it accessible for the iWARP subsystem.

In terms of performance, slow and therefore expensive transfers between disk and memory are reduced to two instances by using the RAM disks: first, we read the original source file from the disk at the master and second, we write the resulting object file back. Everything else is done in main memory.

Another advantage (which we do not exploit in our current *rdistcc* implementation) is that the files stored in such a way are *randomly accessible* not only by the local but also by the remote host.

5.3.4 *rdistcc*'s RDMA Memory Region Management

Now that the data is stored in memory, we need to investigate how to best register that memory with the RDMA subsystem.

When designing an iWARP-based application, the buffer management is key to its success. Sometimes it is not even possible to manage the buffers such that the application is able to profit from RDMA. This is, for instance, the case for applications with short-lived connections where the buffers exist only for a short time and thus have to be (re-)advertised often (i.e., for each new connection). If these buffers are also small, the management overhead will soon outweigh the zero-copy benefit [DW07a,FA09]. The setup of the RDMA MRs induces a delay because the memory has to be pinned and registered with the subsystem as explained earlier.

We will now discuss our MR management options in the case of *rdistcc*. As Figure 5.6 shows, we need two separate MRs on each host: one for the pre-processed source (SRC TX MR and SRC RX MR) and a second one for the object file (OBJ TX MR and OBJ RX MR). We have seen in Chapter 4 that the preferred way of using MRs is to allocate them at the beginning of the program execution and to reuse them thereafter without the need for reregistration or refilling by copying. Unfortunately, this is not feasible in the case of *distcc* because we can force neither the pre-processor (*cpp*) on the master nor the compiler (*gcc*) on the slaves to write their output to a predefined physical memory address without modification (they only have the file-system view on the data). Also, the destination address argument for the mapping is only a hint. There is no guarantee that the data is going to be mapped to that address. When using the shared mapping, this is almost impossible and would (in most cases) cause the data to be copied. We therefore have the options of either creating a MR once at the beginning and refilling it for each file by copying or, alternatively, we can de-register it when a transmission is done and register a new one for the next file.

There are thus two aspects that need to be considered for our MR management: the *memory footprint* (due to the pinning) and the *registration cost*. If we are

refilling the MR, we need to make sure that it is large enough to hold the largest file we ever need to transmit. This is not feasible because a software project, for which distributing the compile process makes sense, consists of a large number of files and checking the size of all of them might take too long. Also, it is hard to predict the size of the output from the pre-processor and the compiler. Furthermore, refilling is a great waste of memory in projects where we have only a few large files but a lot of small files because the memory stays pinned as long as it is registered.

Based on the findings presented in Chapter 4, small buffers should be copied while it is faster to reregister large buffers. For Softiwarp, the critical size where it becomes cheaper to register the buffers is at about 16 KB.

Considering this, we have decided to manage the MRs as follows: for the SRC TX MR[3] and the OBJ TX MR[4] (see Figure 5.6), we reregister the MRs for each file. Refilling would limit the scalability of the master due to the large memory footprint resulting from over-provisioning for large files. The buffer re-advertisement does not cost us much here because we can piggy-back it on the compile instructions (see Section 5.3.7). The delay induced by that results from opening an existing file followed by mapping it into the application address space and registering it as a MR with the RDMA subsystem. With TCP, the data would have to be copied from the user space memory into kernel socket buffers which is slower for files larger than 16 KB. The OBJ RX MR[5] is also reregistered in each iteration with the additional cost of allocating an empty target file (*create_mmap* column). This makes sense because the object files are written back to the hard drive on the master. If the master would take the approach of refilling existing MRs, the total amount of locally available memory would soon become its bottleneck and prevent it from being able to deal with a large number of slaves. For the SRC RX MR[6] we can register the MR once and reuse it. This is the optimal use case. As argued in the last section, it does not impose much memory dissipation on the slaves.

In this example, we see that (in real-world applications) there is more to the Memory Region management discussion than the registration versus memcpy delay. Sometimes, the greater circumstances (e.g., the communication protocol) do only allow one way of performing the MR management. At other times, the two options end up yielding a near-equal performance. We find a bit of both in our case.

[3]source transmit Memory Region
[4]object transmit Memory Region
[5]object receive Memory Region
[6]source receive Memory Region

5.3.5 Transferring the Files using iWARP

Since all the data are in memory and the necessary RDMA buffers (MRs) are in place, we can now transfer the payload between the master and slave machines using RDMA.

Function `tx_file` demonstrates in simplified pseudo code how a file is transferred using a Write operation on our iWARP library. We illustrate the reregistration approach. The following steps are required:

1. map the file into the application address space and register it as a MR with the RDMA subsystem [lines 1–2]

2. create and post a Work Request to the Send Queue [line 3]

3. fetch the Work Completion from the associated Completion Queue (CQ) [line 4][7]

Thanks to our iWARP library, we do not have to set up a scatter/gather list and populate the Work Request. Instead, we simply call `post_write_lmr()` with the source and destination buffers as well as the offsets within them (0) and the size of the data transfer. Since we wait for the completion of the operation on line 4, we need to flag the *RDMA Write* operation as being signaled to trigger the generation of a Work Completion.

Function `tx_file`(*file_descr, size, dst_addr, dst_stag*): RDMA data exchange.

```
   /* create shared, memory backed mapping                          */
1  buffer = mmap(size, MAP_SHARED|MAP_ANONYMOUS, file_descr);
   /* register the buffer as a RDMA MR                              */
2  iw_lmr_register(buffer, size, access, src_lmr, iw_ctx);
   /* create the transmit Work Request and post it to the SQ        */
3  post_write_lmr(src_lmr, 0, dst_rmr, 0, size, IBV_SEND_SIGNALED,
   iw_ctx);
   /* wait for the corresponding Work Completion                    */
4  await_completions(IW_SCQ, 1, wc, iw_ctx);
```

[7]The third step is omitted if the Work Request is configured not to generate a Work Completion.

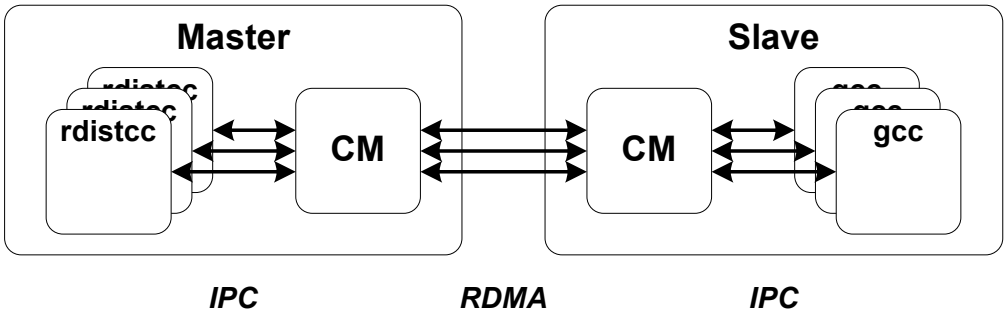


Figure 5.7: Connection manager.

5.3.6 Connection Management

The last piece of the puzzle before presenting the final network communication protocol is the connection management. As mentioned earlier, establishing an iWARP connection is much more expensive than establishing a TCP connection because iWARP adds three more protocols on top of the TCP stack and additional programming objects are needed. A conventional *distcc* master establishes a new TCP connection to one of its slaves for each source file which is to be compiled remotely and closes it again after having received the result. This obviously does not scale for iWARP. We thus introduce a *connection manager* (CM) that keeps the RDMA connections open while the *rdistcc* master is submitting individual compile jobs. The CM acts as a persistent RDMA proxy for the *rdistcc* processes (cf. Figure 5.7) and takes care of the RDMA network transfers (posting Work Requests, managing remote buffer information, etc.).

Each master process first connects to its host-local CM using standard *inter-process communication* (IPC). To reduce intra-host data copying between an *rdistcc* process and its local CM, an *rdistcc* process only provides the CM with path information for locating the pre-processed file rather than sending the whole file through the IPC channel. The CM then takes care of the actual RDMA data transfers and executes the network communication protocol.

Figure 5.8 depicts an example of a complete *rdistcc* setup including the interaction between the application logic and the CM. The CM is implemented as a thread pool with a fixed number of threads per slave. One thread cycle encompasses exactly one complete protocol cycle (see next section). The number of slaves involved as well as the number of remote compile jobs is set before starting *rdistcc*. This fixed thread allocation is efficient and light weight at run time and matches the static setup for which *distcc* is designed.

Overall, the CM minimizes the connection management overhead and simplifies the *rdistcc* application logic. Having individual threads to process the communication simplifies the connection handling and leads to a reliable and early error

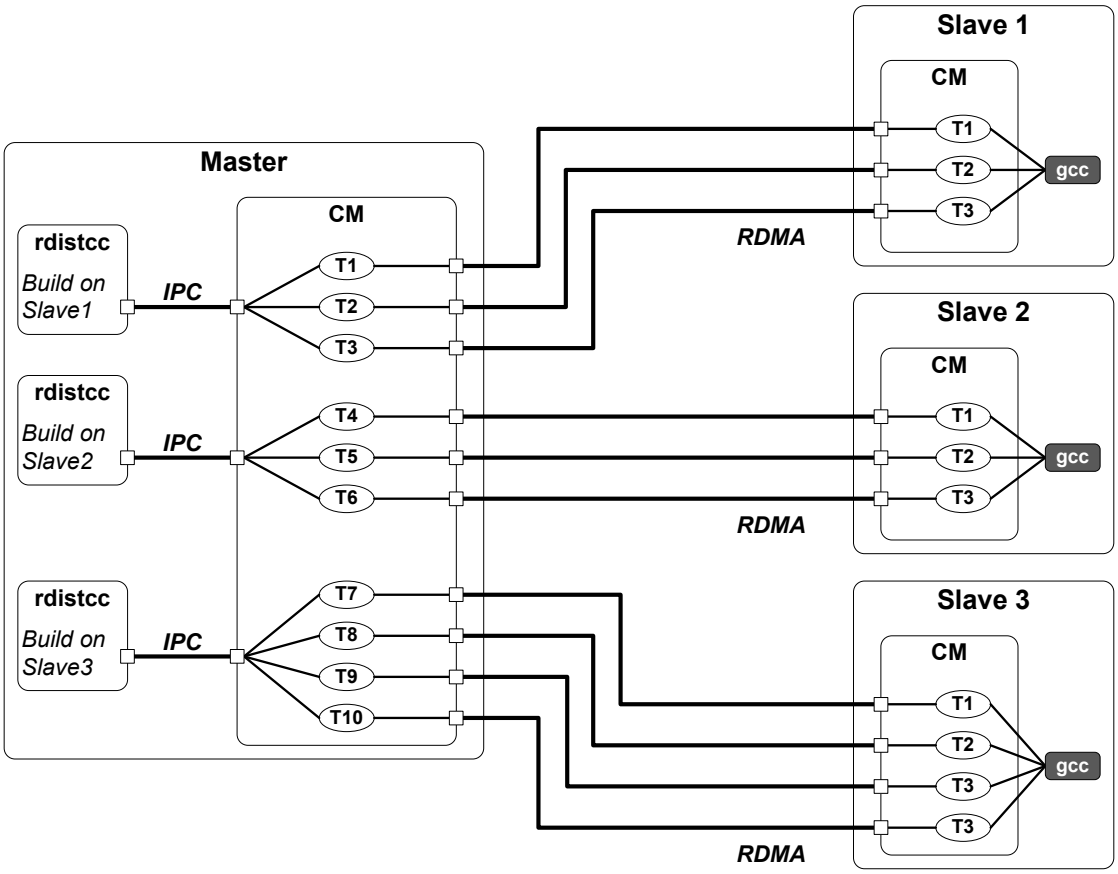


Figure 5.8: *rdistcc* processes, threads and their associations.

detection. Without the CM, the iWARP connection setup overhead as discovered in Chapter 4 would render RDMA useless by removing all its advantages.

5.3.7 Application Protocol for iWARP

We will now look at how it all fits together by transforming the TCP-based *distcc* protocol into an RDMA-based one.

As hinted earlier in this chapter, the decision of how to utilize the one-sided and two-sided operations can have a significant impact on the benefit of using RDMA. It can make a difference in protocol complexity as well as in load distribution between communicating peers. The advantage of the one-sided communication is that only the application layer of one side has to actively participate in the data transfer. The drawbacks are that a buffer advertisement is necessary beforehand and that the other side will not know when the data transfer has completed. Two-sided *Send/Receive* operations are thus typically used for exchanging control

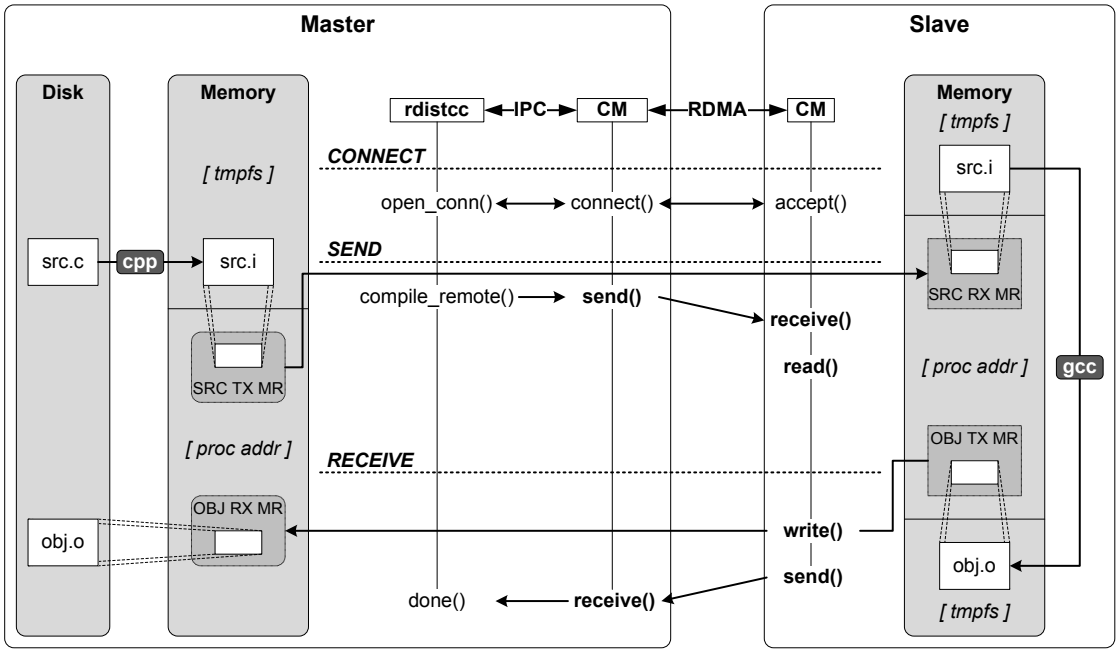


Figure 5.9: iWARP based master-slave protocol

messages whereas the one-sided operations are preferred for transferring the actual data.

The protocol proposed in Figure 5.9 involves both, two-sided (*Send/Receive*) and one-sided (*RDMA Write* and *RDMA Read*) operations and is partitioned into the three phases as follows:

In the *CONNECT* phase, the *rdistcc* process connects to its CM and sends an IPC request for opening an RDMA connection to a given slave. If that RDMA connection already persists, the ICP call returns immediately. Otherwise, the local CM connects to the corresponding remote CM of the given slave and establishes a new RDMA connection. The RDMA connections are kept open as long as they are needed (see Section 5.3.6).

Next, the *SEND* phase is executed for the file which is scheduled to be compiled by the given slave. In this phase, the master provides the slave with the following control information: the compilation instructions (the exact *gcc* command), the *STag* for the MR where the pre-processed source data resides (SRC TX MR) and the *STag* for the MR where the compiled object file should be written to (OBJ RX MR). This data exchange is done using a two-sided *Send/Receive* operation. By using this two-sided communication, the slave is notified when the control information has arrived and can immediately fetch the actual data (the pre-processed file) by using the one-sided *RDMA Read* operation. Once the data has been read, the slave starts the *gcc* compilation process to produce the object file.

After completing the compilation according to the instructions received, the slave places the resulting object file in the master's memory using an *RDMA Write* and issues a *Send* operation to notify him of the completion. We denote this as the *RECEIVE* phase. When the master receives this *Send* message, the object file is already present in his buffer and ready for being linked.

As can be seen from Figure 5.9, the master involvement is small—it only needs to map the source file into a MR and inform the slave about it through the local CM. The slave then takes care of the data transfer and compilation. Meanwhile, the master can schedule other compile jobs. This enables even slow masters to use many slaves in parallel. Such a slave-driven protocol cannot be designed with two-sided operations like the ones that TCP offers.

We argue that, despite the increased complexity due to the explicit buffer management, our protocol does not impose much overhead compared to the original one because first, the memory mapping and MR registration overhead is not larger than copying the *gcc* or *cpp* input/output between the socket buffers and the application address space and second, the size of the buffer information we add to the existing control messages is negligible.

5.4 Experimental Evaluation

In this section, we assess the value of our software-based RDMA extension in terms of performance. For that, we conduct a small cluster experiment on our RDMA-enhanced *rdistcc* by compiling the Linux kernel 2.6.22 in a distributed fashion on 1 to 14 slaves controlled by a single master.

The nodes are connected via 1 Gb Ethernet enhanced with software-based iWARP (*Softiwarp*). The slave machines feature a 1.8 GHz Pentium 4 processor and 3 GB of RAM whereas the master has a 3.2 GHz Pentium Xeon dual core processor. On the master, we have bound the network processing as well as the application to the same core by setting the CPU affinity accordingly.

Figure 5.10 shows the overall wall-clock execution time measured for compiling the Linux kernel on 1 to 15 machines. The 1-machine case is added as reference and refers to a local-only compilation on the master. In the following, we explain the four different experiments and evaluate their outcome.

5.4.1 Data Residing on Memory versus Hard Disk Drive

We have started the experiment series by compiling the kernel using *distcc* (over TCP) with all files residing on disk. The wall-clock execution time is shown as *TCP(disk)* in Figure 5.10. The bar indicated with *TCP(memory)* represents the outcome of the second experiment where the files reside on the RAM disk rather

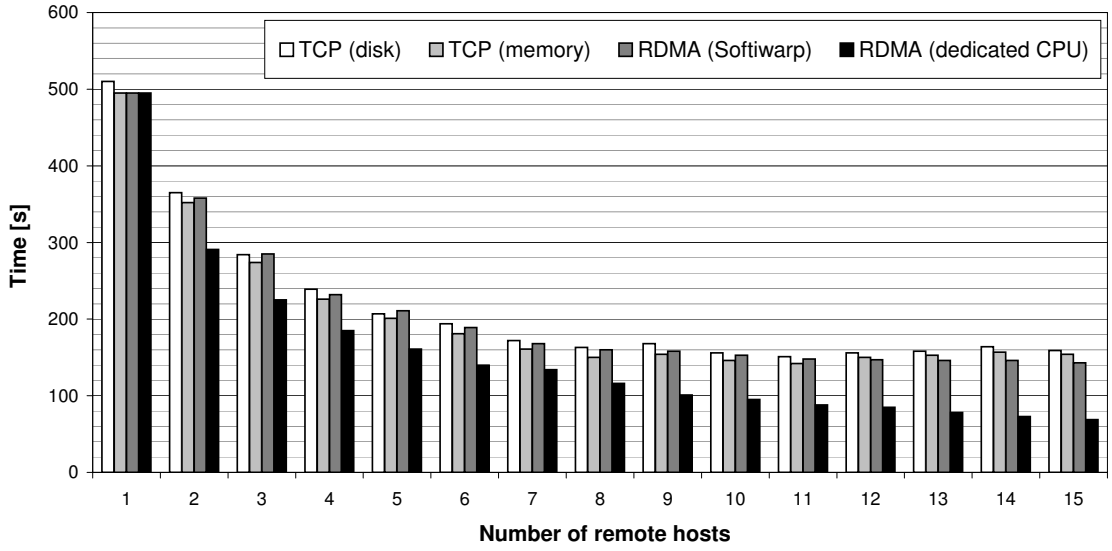


Figure 5.10: Compiling the Linux kernel 2.6.22 in a distributed fashion using the TCP as well as the software RDMA transport.

than on the hard disk drive. We find that reading and writing the data from/to the hard disk does not significantly impact the overall compile time (compared to having the data in memory) because the distribution and source code translation take much longer than the disk I/Os.

In all but the first experiments, the compiler input- and output data reside on RAM disks (*tmpfs* volumes) in order to assure that the performance gain is not the result of *rdistcc* keeping everything in main memory where *distcc* would normally keep it on disk.

5.4.2 TCP versus RDMA

After having seen that storing the data on a RAM disk does not affect the outcome a lot, we now wonder whether the RDMA extension is able to increase the performance.

Given that the current *Softiwarp* implementation is based on TCP kernel sockets and thus can not achieve true zero-copy, *distcc* over TCP (denoted as *TCP(memory)*) and *rdistcc* over *Softiwarp* (see *RDMA(Softiwarp)*) yield a similar performance. Even though Softiwarp processes three additional network layers (*MPA*, *DDP* and *RDMAP*), the overhead is negligible compared to plain TCP. Regarding scalability with respect to the number of slaves, *distcc* over TCP scales only up to 7 slaves and *rdistcc* over *Softiwarp* does not do much better which is what we expect due to the missing hardware acceleration.

We conclude that simply replacing TCP with software-driven RDMA is not enough to reduce the wall-clock clock execution time of our compiler.

5.4.3 Dedicating a Core to RDMA Stack Processing

With today's trend towards multi-core and many-core machines, one (or several) cores can be spared for RDMA processing [Mog03] as long as the memory bus is not the bottleneck. To leverage RDMA on the master system which has no RNIC, we are now dedicating one core to RDMA processing in software while the other is running the *rdistcc* master application. The slaves remain unchanged. While this setup is similar to running *rdistcc* on a single-core machine that is equipped with an RNIC, running the application and *Softiwarp* on different cores in fact leads to a lower performance (than if an RNIC was used) due to poor data locality resulting in frequent cache misses. Nevertheless, the use of 14 slaves now gives a two-fold reduction of the total compilation time compared to TCP. The system can be expected to scale further when more slaves are added.

One can rightly argue that the same performance increase can be expected when dedicating a single core to TCP rather than RDMA. While this is true for applications (like the compiler) which do not transfer a large amount of data, it is no longer feasible when the local memory bus starts to limit the throughput as we will see in Chapter 7. Furthermore, having the RDMA extension in place allows us to replace the software iWARP stack with an RNIC if need be (e.g., when many more slaves are available).

5.4.4 Who Would Need an RDMA-enabled NIC?

To assess the need for an RNIC, we measure the CPU load distribution with *distcc* over TCP (see Table 5.1). We find that the master spends a large amount of CPU cycles on local data copying and network processing (summarized as *data transfers*), since it communicates with all the slaves in parallel and establishes new TCP connections for each file it transmits. We therefore argue that it would make sense to equip the master with an RNIC so that the hardware can handle the data transfers. The CPU cycles saved by this can then be assigned to management, pre-processing and linking tasks which are also quite CPU intensive but cannot be hardware accelerated.

At the slaves on the other hand, we find that almost 80% of the CPU load is caused by *gcc* processes. The bottleneck here is clearly not the communication stack and expensive RNICs are therefore not necessary. The relatively high idle value further indicates that the slaves still have enough processing power to do RDMA in software and shows that the TCP *distcc* master is not able to keep them all busy. This is what sets the limits in terms of scalability. The effect is

Node	Operation	CPU time [%]
master	data transfers	59.2
	management tasks	28.0
	pre-processor and linker	12.4
	idle	0.5
slave	compiler	79.8
	idle	18.9
	data transfers	2.1

Table 5.1: CPU load distribution for TCP-driven *distcc* on master and slave machines.

even stronger if an encrypted communication channel (i.e., SSH) is used due to the crypto overhead as well as the more expensive initial connection setup.

5.4.5 Conclusion

The experimental evaluation has shown, that RDMA can also be beneficial when transferring comparably small amounts of data. Furthermore, we have found it to be useful to improve the scalability of asymmetric, centralized systems which depend on a single node.[8]

The main benefits which we have identified are:

- the enablement of *client-driven protocols* (here: slave-driven) to improve the overall system scalability
- *fewer synchronization points* (thanks to one-sided operations) which interrupt the main work flow
- the *overlapping of communication and computation*

However, the benefit is not as dramatic as in the case of a real RNIC and large data volumes (see following Chapters).

5.5 RDMA File Access - Further Considerations

The proposed technique of creating a shared file mapping into the application address space and thereafter associating it with a Memory Region for remote

[8] Another example of that kind which uses real RDMA-enabled hardware can be found in the subsequent chapter.

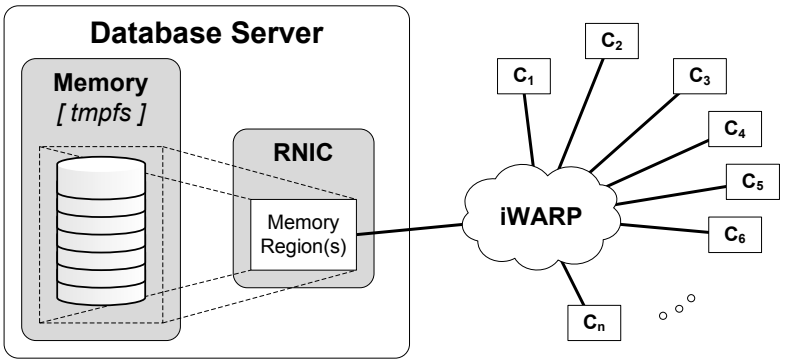


Figure 5.11: Distributed database using the RDMA file transfer mechanism.

DMA is by no means limited to the compiler extension. It could for example also be used for accelerating the file transfer protocol (FTP) or similar applications.

The advantage of the proposed file transfer mechanism is not only the zero-copy data transmission but also the enablement of true remote random access as well as the hardware support through RNICs. A similar, alternative approach is the kernel *sendfile* mechanism. However, *sendfile* does not provide remote random access and copy avoidance is only possible on the transmit side.[9]

The distributed compiler does not make use of the fact that the remote file is accessible in a true random fashion. This could be interesting for applications like a distributed, file-based database (cf. Figure 5.11) where random access to some small parts of a large remote database file is desired without having to copy the whole file over the (possibly slow) interconnect. This example illustrates that iWARP might also qualify for low throughput environments (e.g. the Internet) due to its features beyond zero-copy performance improvement.

5.6 Related Work

5.7 Summary

Taking distributed compilation provided by *distcc* as an example, we have illuminated the various aspects that need to be considered when enabling a legacy application for iWARP/RDMA: buffer and Memory Region management, the asynchronous interface, one-sided versus two-sided operations and connection management. Based on that, we have discussed how distributed compilation can be accelerated with RDMA. Through the use of the one-sided RDMA operations, we

[9] A more detailed comparison between *sendfile* and iWARP zero-copy can be found in Chapter 6.

have demonstrated that iWARP can be attractive for enhancing legacy applications that are currently based on the synchronous socket interface even if they do not transfer large volumes of data.

5.8 Outlook

We have argued that distributed computing systems using an RNIC at the busy nodes and *Softiwarp* together with conventional Ethernet NICs at the other nodes are performance- and cost-effective (in terms of network processing) for applications with different CPU load distributions—depending on the role of the node. Compared with plain TCP, the use of iWARP therefore allows busy core nodes with an RNIC to serve more edge nodes in parallel. In the following Chapter, we will illustrate and verify that claim at the example of distributing high-definition multimedia content in an on-demand fashion where a high service quality and good scalability are the main concerns.

6

Server-Efficient High-Definition Media Dissemination

Having illustrated the *changes* which are necessary to extend a socket-based application with iWARP capabilities, we now investigate a use case which focuses on the *performance benefits* of applying iWARP/RDMA with *dedicated hardware*. In this chapter, we present a method for serving a large volume of data to a (potentially large) number of clients in a way which is efficient for the server in terms of the number of clients it is able to provide the content to without taking a loss in the offered service quality. To do that, we leverage the *zero-copy* and *OS bypassing* techniques provided by RDMA-enabled NICs (RNICs). As we will see, dedicating one or several CPU cores to network processing is no longer a valid replacement for RNICs when it comes to high data rates.

6.1 Introduction

The way how the Internet is used has changed dramatically in the past few years. Today, about 23% of the world's population are connected to the Internet—most of them through broadband access. This number as well as the bandwidth offered by the Internet service providers (ISPs) are increasing rapidly [intc]. As a consequence, web content is no longer static but has become dynamic and far richer.

Especially *Video-on-Demand* (VoD) services such as YouTube[1], Amazon VoD[2] or AOL Video[3] are becoming more and more popular.

With the widespread availability of broadband connections, the quality of the media provided by VoD- as well as streaming services increases constantly. Even though today most videos are still encoded in *standard definition* (SD) with a rather low bit rate, large Internet service providers (ISPs) already foresee *high-definition* (HD) media to become the predominant format in the near future.

According to Comcast, the next step in the evolution of Internet content will not only be a shift from *standard-definition* towards *high-definition* media but also from *broadcast* towards *unicast* services [Sax]. This trend towards unicast is attributed to the request for *true on-demand content* as opposed to streams which are simply broadcast in loops on different channels.

In the following discussion, we hence focus on the server-side impact of high-definition media content distribution over high-speed unicast channels. An increasing number of clients requesting media in HD which requires much higher bit rates poses a number of challenges for the server infrastructure.

6.1.1 Challenges

According to Plagemann et al. [PGHA00], *high-definition video-on-demand* (HD VoD) poses two orthogonal service requirements: a *large data throughput* to deliver the high-definition content in due time and a *low latency* as well as a *high responsiveness* from the servers to support convenient, interactive media control. Transmitting HD media over unicast channels to an increasing number of users renders meeting these requirements quite complex for two main reasons.

- First, the content servers must be able to sustain a *higher aggregate data throughput* than before (the bit rate of a HD video compressed with the popular H.264 codec [WSBL03] is roughly 10x larger than that of its SD equivalent).

- Second, the clients expect convenient, interactive control over the content *at all times* (pause, skip or rewind as well as switching to another movie). This *unpredictable behavior* of each client makes it virtually impossible for the server to prefetch data efficiently and predict the aggregate service rate. The problem is even amplified when the media content is encoded with a variable bit rate (VBR).[4]

[1] http://www.youtube.com

[2] http://www.amazon.com/gp/video/ontv/start

[3] http://video.aol.com

[4] The technique of encoding in a variable bit rate is often found in practice because it allows for a reduction of the average bit rate and thus helps in saving some of the network bandwidth.

6.1.2 Problem Statement

Scalability in existing VoD systems is achieved simply by adding more servers and setting up load balancing mechanisms. While this is straightforward, it increases the running costs of the server infrastructure due to higher server and network maintenance costs, as well as larger power consumption and cooling demands. The problem we address in this chapter is *how to improve the scalability of a single HD-video server* (i.e., its ability to serve a larger number of clients) such that the demand of the HD VoD service on the server infrastructure is minimized. Such an improvement involves removing the overhead encountered in current systems. For this purpose, we propose a novel protocol based on iWARP/RDMA.

6.1.3 Contributions

The contributions of this chapter are three-fold.

- First, we show why traditional media dissemination mechanisms (i.e., *RTP over UDP* and *HTTP over TCP*) are inadequate for serving content at high bit rates by analyzing the performance and scalability of existing VoD solutions and identifying their bottlenecks through a number of experiments.
- Second, we present a *novel media dissemination method* for large bit rates based on a zero-copy protocol stack implementation on dedicated iWARP/RDMA hardware. We then prove through extensive experiments that our method increases server-side scalability and removes most of the overhead caused by current approaches while offering efficient VCR-like media control. For that, we compare our solution with existing ones as well as with the kernel *sendfile* mechanism and a *TCP-offload engine*.
- Third, we outline an extension to our new protocol to support *real-time live media streams*.

6.1.4 Chapter Overview

The chapter is structured as follows: Section 6.2 highlights the iWARP/RDMA benefits in the context of HD media distribution and reviews two popular streaming transports. Section 6.3 compares UDP- with TCP-based application protocols that enable VoD services and explains two TCP acceleration methods. Section 6.4 presents a novel iWARP/RDMA-based protocol for video-on-demand as well as streaming services and evaluates the proposed solution by comparing it to legacy systems. Section 6.5 summarizes the benefits and drawbacks. Section 6.6 lists related work before Section 6.7 finally summarizes this chapter.

6.2 Background

In this section, we briefly outline the iWARP/RDMA benefits with regard to HD media dissemination and review the two most popular streaming mechanisms used today.

6.2.1 RDMA Benefits

In bypassing the operating system and eliminating intermediate copying across buffers (thanks to the direct data placement feature), RDMA reduces the CPU cost of large data transfers as well as the end-to-end latency which is precisely what we need to minimize the server load in the VoD scenario.

Similar to the compiler presented in the last chapter, we will be able to benefit from the asynchronous nature of the communication API. As we will see in Section 6.4, we can have a number of outstanding one-sided *RDMA Read* Work Requests on the clients which allows us to

- overlap shipping of the data with encoding or decoding of the media and
- fetch the required data from the server without interrupting its current task(s).

In this scenario, the Memory Region management is straight forward and does not cause any overhead other than a slightly delayed start of the playback. Furthermore, we do not need a connection manager or similar mechanism to handle the iWARP connections because the setup is static and the connections are (typically) long lasting. Offering HD VoD services thus seems to be a perfect match for RDMA.

RDMA over Wide Area Networks

Since iWARP makes RDMA available over Ethernet, we can use it to transfer HD media across any Ethernet-based network. Even though unlikely in the Internet of today (due to bandwidth constraints), it is conceivable for the future to attach an RDMA-enabled media content server directly to a 10 GbE wide area network (WAN) backbone serving its HD content to a (potentially large) number of clients. The application scenario for serving data over 10 GbE today would rather be a local area network (e.g., in a company). Since we use iWARP we can later on also deploy the setup on the Internet without any changes.

The clients do not need to be equipped with an RNIC because they do not receive that much data—the hotspot is the server. Therefore, in a realistic scenario, the clients would run *Softiwarp* on ordinary Ethernet NICs while the server should

be equipped with an RNIC. Having a software RDMA solution is critical to the real-world applicability of our proposal for cost reasons.

6.2.2 Prevalent VoD Transports

We will now review two legacy protocol stacks which are currently used to disseminate media data in an on-demand fashion.

RTP over UDP

A common approach for transferring media streams over a network is to use the Unreliable Datagram Protocol (UDP) accompanied by the Real-Time Transport Protocol (RTP) [SCFJ03] which defines a standardized packet format for audio and video. In order to feed out-of-bound information back to the streaming source, the Real Time Control Protocol (RTCP) is added. Finally, the Real-Time Streaming Protocol (RTSP) provides control over the media to the user and completes the VoD setup over UDP.

Since RTP was designed with focus on streaming media and other real-time data, it seems like a good match for VoD. The motivation for using unreliable connectionless services such as UDP for video streaming is that media streams typically tolerate data loss better than delay and therefore the higher reliability of TCP is not needed. With UDP, unnecessary retransmissions of data (which is not needed anymore because it would arrive too late) are avoided. Therefore, missing parts of the streams are skipped rather than stopping the stream and waiting for the retransmissions which leads to a more pleasing viewing experience. Contrary to common wisdom, however, this method is not at all suited for disseminating media at *high bit rates* as we will see in Section 6.3.2.

HTTP over TCP

YouTube, for example, as one of the main sources of Internet media streams today, has chosen HTTP-based video dissemination. Even though HTTP is based on *connection-oriented TCP* and was not specifically designed to meet media-streaming requirements, the combination offers some important advantages:
Interoperability. HTTP assures good interoperability in the sense that no special software is needed. Any state-of-the art web browser will do for receiving the stream.
Firewalls. HTTP port 80 is allowed on most firewalls whereas other ports (potentially used by RTP) are often blocked.
Server Efficiency. HTTP is able to outperform RTP in terms of server efficiency as we will demonstrate in Section 6.3.3.

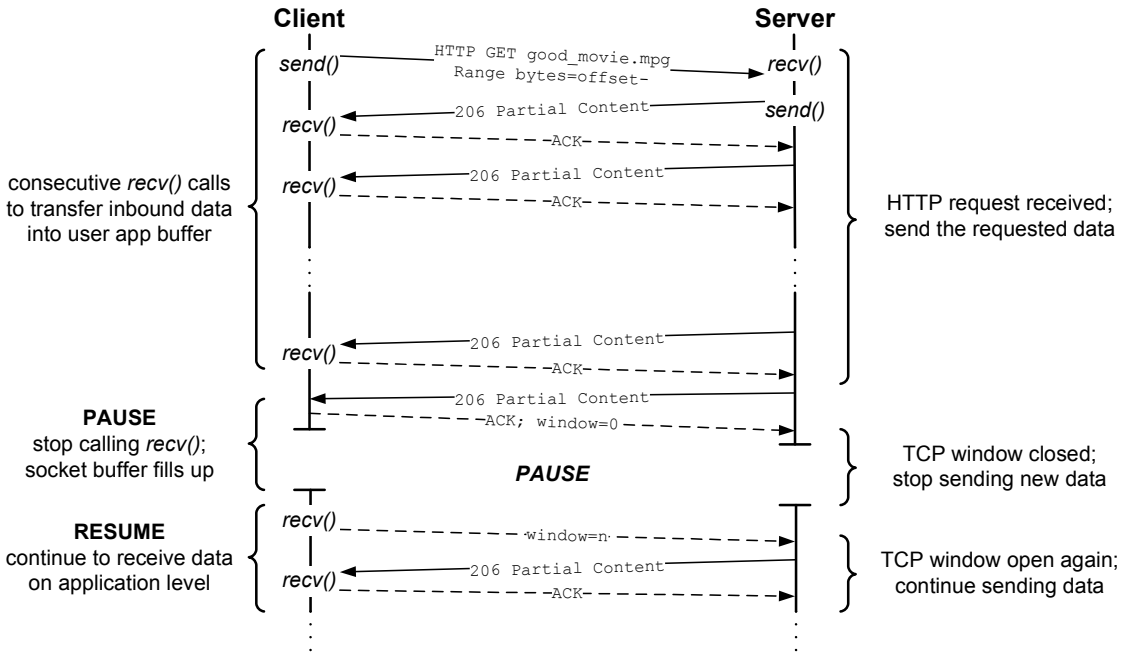


Figure 6.1: The TCP flow control mechanism implicitly enables the client to pause the media stream (without connection teardown) by stopping to call *recv()*. As a consequence, the receive socket buffer fills up, the TCP window closes and the server stops transmitting new data.

Reliability. The TCP reliability feature can be desirable for highly compressed media where the loss of a key frame can cause severe playback disruption (given that the retransmission completes in time).

Media Control. The TCP flow control mechanism implicitly allows client-driven media control (see Figure 6.1). Not reading new data from the socket and thus delaying *TCP window updates* causes the sender to stall the stream. Seeking is done by simply sending an updated `HTTP GET` request with the desired new offset. There is no need for a set of companion protocols as in the case of RTP/UDP.

6.3 Assessment of Current Systems

Before looking at our iWARP/RDMA-based media dissemination mechanism, we evaluate current existing video dissemination solutions in practice through a series of experiments. We focus on solutions that use RTP and HTTP as their transport.

As defined by the problem statement, we strive for server-side scalability improvements. Our main criterion of media distribution efficiency is the maximum number of clients a single server is able to serve concurrently without service

degradation such as frame losses or late frames.

6.3.1 Experimental Setup

We start the evaluation by describing the setup used throughout the experiments.

Service Parameters

The videos used for the experiments are encoded with the predominant codec for HD media (i.e., H.264). We first look at the influence of streaming at *different bit rates* resulting from the various resolutions (see Figure 6.2). We start with SD resolution (480p) at an average of 1 Mbps and go up to full HD (1080p) requiring 8.7 Mbps on average for our test movie. Next, we investigate the impact of the data access pattern by the clients. As we will see, it can make a difference if the clients watch the stream sequentially from the beginning to the end or if they perform random jumps across it.

We have also analyzed the influence of other service parameters, such as offering varying numbers of movies, movies of different lengths, or changing the client-side cache size. Compared with the bit rate and the access pattern, none of the other parameters had a significant impact on scalability and are therefore not discussed any further.

In order to avoid disk access overheads affecting the results, we have preloaded the complete test data set into the main memory at the source (i.e., the server). In reality, this imposes a limitation with respect to the amount of data a server is able to provide to its clients at any point in time. In our final discussion, we will point out two ways to address this issue. For now, we assume that the main memory of our server machines is large enough to hold the entire media data.

Distribution Network

We have examined the performance of the existing VoD solutions on a 10 Gb switched Ethernet fabric. Our test bed consists of an IBM BladeCenter containing six HS21 BladeServers. Each of them is equipped with a quad core Intel Xeon CPU (2.33 GHz), 8 GB of main memory and a Chelsio T3 RDMA-enabled 10 GbE NIC. The BladeServers are running a Fedora Linux 2.6.27 kernel with the OpenFabrics Enterprise Distribution v1.4 [ofe] for iWARP support. One BladeServer acts as the server (media source) and the other five are connecting to it as clients (media sinks). Each client machine runs up to 200 client application instances concurrently to simulate a total of 1000 physical clients.

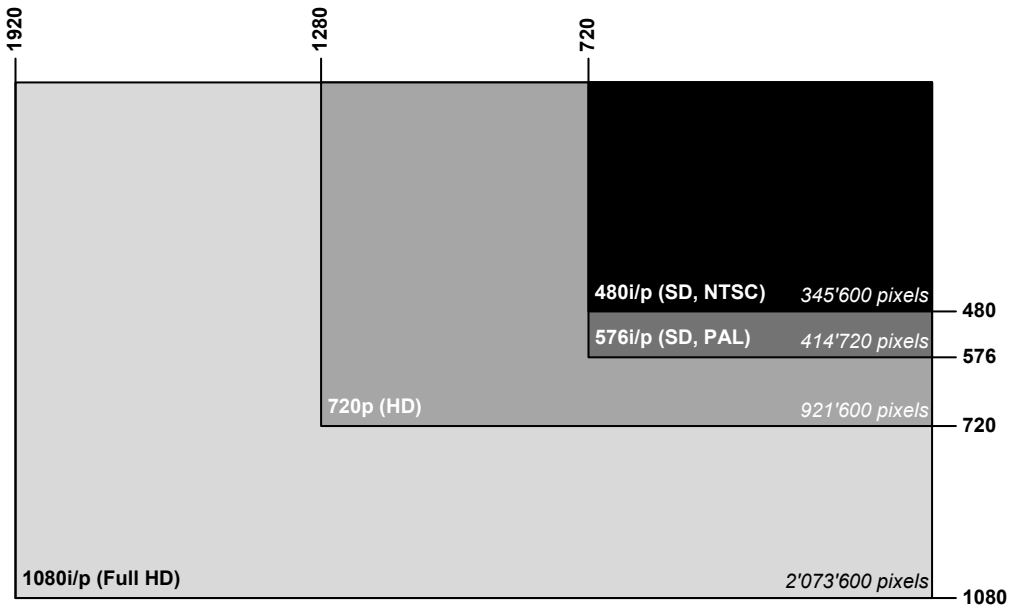


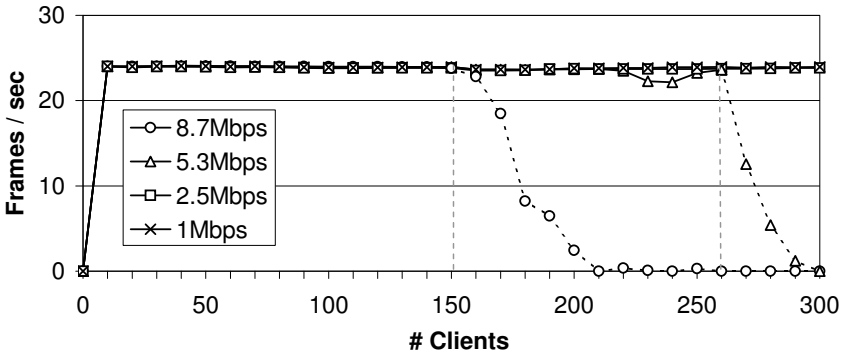
Figure 6.2: Video resolutions ranging from 480x720 (SD, NTSC) to 1080x1920 (full HD). The visual real-estate is a factor 6 larger for full HD as compared to SD.

6.3.2 RTP-based Systems

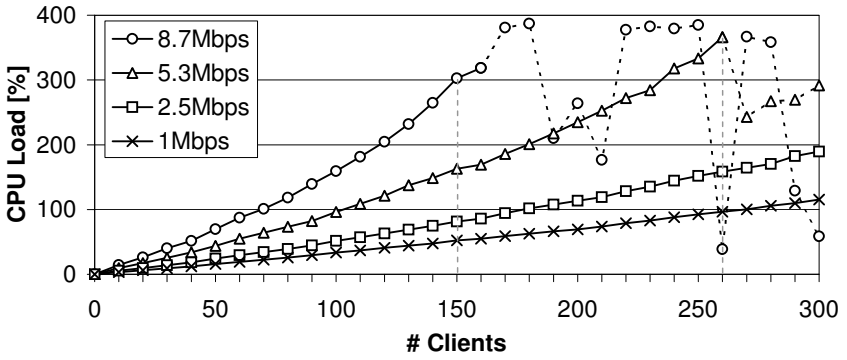
In our first test, we explore the scalability of *RTP over UDP* for various bit rates with special focus on the high bit rates needed for HD media content. We have conducted the experiments with the two most prevalent media servers for Linux that offer VoD services using RTP: The open source *VideoLAN Server* [vid] and the *Darwin Streaming Server* [dss]. The following charts reflect the measurements from the VideoLAN software only because its performance is not significantly different from the Darwin Streaming Server.

We have conducted our first experiment on the setup described above with up to 300 clients.[5] Figure 6.3(a) reflects the client's viewing experience by the number of received frames. The stream features 24 frames per second. In order to get a smooth playback, all the frames must be received in time. As can be seen, the media encoded with the low bit rates required for SD media (i.e., 1 Mbps and 2.5 Mbps) are successfully received by all the 300 clients (and less). When moving towards the higher bit rates, however, the number of successfully serviced clients drops to 260 for 5.3 Mbps and 150 for 8.7 Mbps. As we are running on a 10 Gbps network, the observed service rates indicate that only about 15 % of the available bandwidth is actually utilized. The erratic shape of the curves beyond

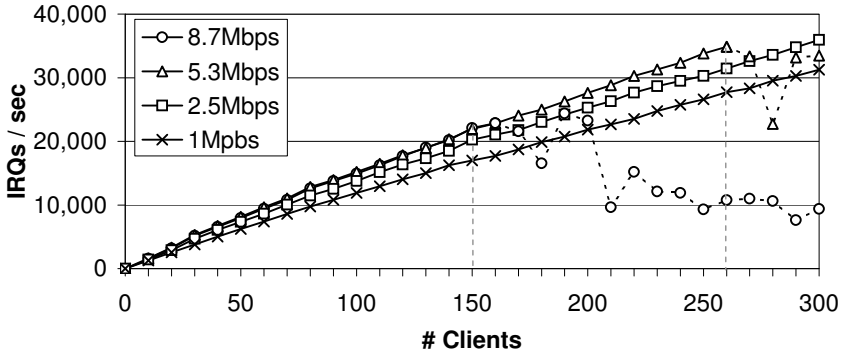
[5]Up to 60 client application instances are running on each of the 5 client machines.



(a) Successfully (in time) received frames. The number of clients which can (concurrently) receive the stream drops with an increasing bit rate.



(b) Scalability is limited by the CPU power. The maximum number of concurrent clients corresponds with maximum CPU utilization.



(c) RTP over UDP causes a lots of interrupts (especially at high bit rates) causing cache thrashing and inefficient CPU utilization.

Figure 6.3: RTP over UDP shows poor scalability on the high bit rates required for the (full) HD resolution.

the scalability limit (dashed parts of the plots) is caused by nondeterministic, non-reproducible behavior of the server and suggests severe service degradation.

Figure 6.3(b) reveals the reason for it: we are bound by the CPU(s)[6] of the server. The moment where the CPUs reach maximum utilization corresponds quite precisely with the start of the observed service degradation. When investigating the reason for the high CPU load using *oprofile* [opr], we have found that most of the CPU cycles are spent in *copying data* and *RTP packetizing*.

Another effect of RTP-based high speed communication can be found in Figure 6.3(c). A large number of interrupts are thrown. This leads to numerous context switches and cache pollution which is undesirable [MB91]. Also here, the curves depend on the bit rate.

On one hand, RTP/UDP suffers severely when streaming at high data rates. On the other hand, we have found that RTP has the advantage of not being susceptible to non-linear data access patterns. Even with relatively high jump rates (i.e., a random jump by each client every 30 seconds), the number of serviceable clients remains the same (not shown). As we will see in the upcoming section, HTTP suffers a lot more in that respect.

We conclude that RTP over UDP is not well-suited for high bit rates as it is not able to make good use of the network resources at hand due to excessive CPU usage.

6.3.3 HTTP-based Systems

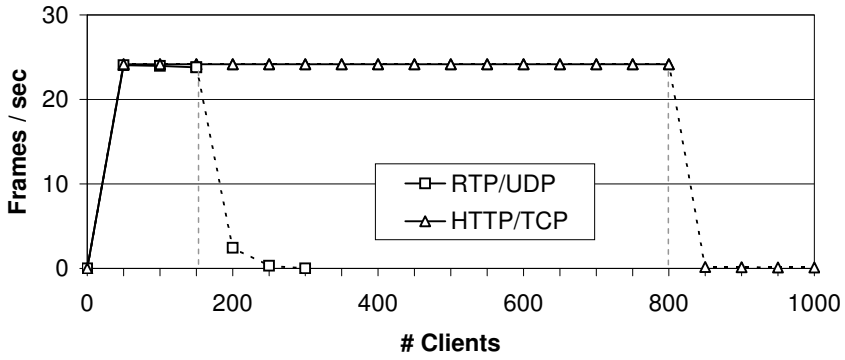
In the preceding section, we have shown that RTP/UDP does not manage to fully utilize the 10 GbE link because of the large data copying overhead when streaming at high bit rates. We now repeat the above experiment with the 8.7 Mbps data rate using HTTP/TCP and compare the result with our previous findings. Again, the key performance metric is how well the server scales in terms of the *number of concurrent full HD streams* it can provide. For our experiments, we have chosen the prevalent HTTP server by Apache [apa] with the multi-processing worker module[7] which is needed to be able to stream to more than 150 clients in parallel as each data stream needs to be handled by its own thread. To obtain a fair comparison with RTP, we first use the *plain kernel TCP stack* without *sendfile* support and without TCP-offloading. Figure 6.4 shows the result compared to RTP over UDP.

As can be seen in Figure 6.4(a), the link can still not be saturated. We can only provide our service to 800 out of the 1000 clients—20% of the 10 GbE bandwidth are still unused.

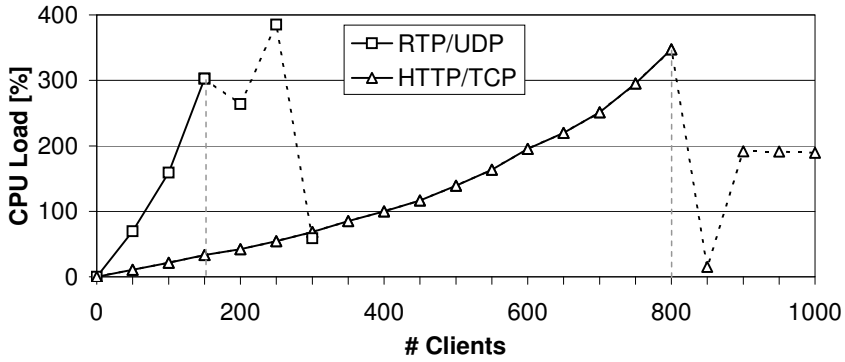
However, as shown in Figure 6.4(b) the CPU limit (where all 4 CPUs are busy)

[6] We denote full CPU utilization of the 4 cores as 400 % in the Figure.

[7] http://httpd.apache.org/docs/2.0/mod/worker.html

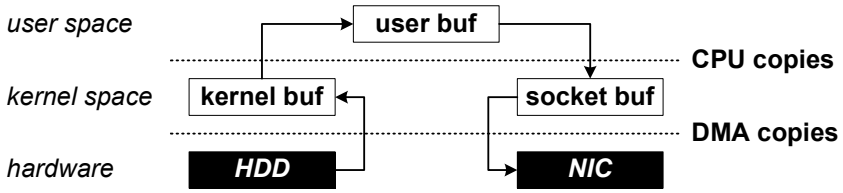


(a) RTP/UDP can serve full HD content to up to 150 clients in parallel while HTTP/TCP can handle up to 800 clients.

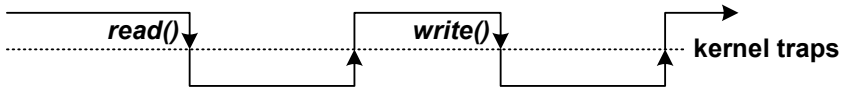


(b) Also HTTP/TCP suffers from a high CPU utilization. Both mechanisms face an exponential load increase relative to the number of clients.

Figure 6.4: HTTP/TCP on Apache scales more than 5 times better than RTP/UDP on VideoLAN using the full HD bit rate (8.7 Mbps).



(a) The data on the hard disk drive (HDD) is first read into a user buffer through a temporary kernel buffer. Thereafter, it is sent out through the network interface card (NIC) by copying the data through yet another kernel buffer (the socket buffer).



(b) The data is fetched into user space by means of a *read()* operation before it is sent onto the network with a *write()* call.

Figure 6.5: Transmitting data residing on a hard disk drive through user space over the network requires a number of DMA- as well as CPU-driven copy operations and several context switches.

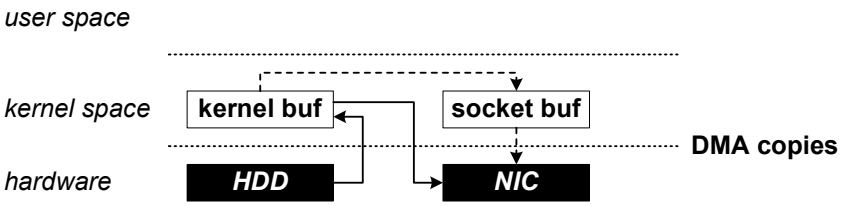
is reached much later with HTTP/TCP than with RTP/UDP. This is a result of the reduced packetizing overhead and the simpler connection management. Yet, the server CPU load increases exponentially (as in the case of RTP/UDP), indicating bad scalability properties when moving to even higher bandwidths (e.g., by inserting several 10 GbE network adapters into the server).

In order to reduce the server load we now apply the Linux *sendfile* mechanism and add a *TCP-offload engine* (TOE).

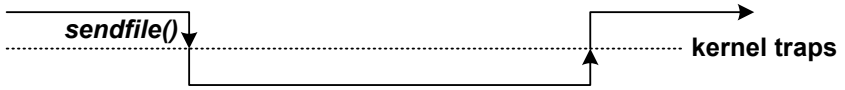
Kernel Sendfile Support

The high server side CPU load observed in the HTTP/TCP test is primarily due to massive data copying on the data source (server). Figure 6.5 illustrates the process followed to send data stored on a hard drive by using the normal *read()/write()* combination: The data is first read from disk through a temporary kernel buffer into a user buffer; then it is written back into another kernel buffer and finally copied to the network interface card (NIC) (cf. Figure 6.5(a)). Overall, the process is expensive because it involves 2 DMA copies, 2 CPU copies and several context switches (Figure 6.5(b)).

Since version 2.2, the Linux kernel offers the *sendfile* system call. In contrast to *read()/write()*, it allows the data to be sent directly from the temporary kernel buffer onto the network without going through user space. If the utilized NIC is equipped with a real DMA engine (as is the case for our Chelsio T3 adapter), the



(a) The detour through user space as well as the CPU-driven copy operations are avoided. The socket buffer is populated with pointers for transmitting the data residing in the kernel buffer.



(b) The data transmission using *sendfile()* requires only one system call.

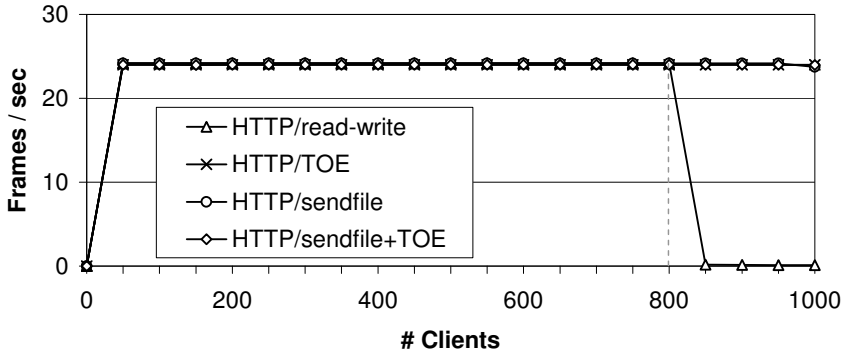
Figure 6.6: The *sendfile()* mechanism is more efficient than the *read()/write()* sequence as it passes the data from the storage medium to the network card entirely in kernel space and without CPU-driven copy operations.

CPU copies are avoided altogether. Furthermore, the numbers of context switches and necessary system calls are reduced. Figure 6.6 illustrates the process.

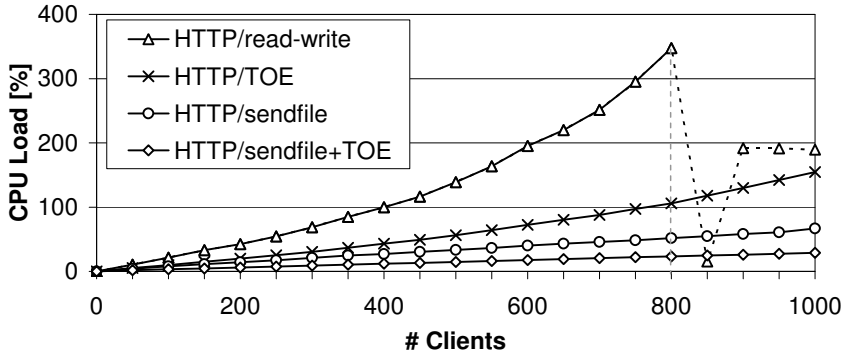
The advantage of *sendfile* is that it does not require changes in the software or application protocols used. Furthermore, since it is based on files, a massive storage cluster (e.g., RAID) can be used to store all the content, and it is still possible to do zero-copy data transmission on the server side.

In order to see the impact of the *sendfile* copy avoidance mechanism on the scalability, we have reconfigured our Apache web server to apply this mechanism and re-ran the above VoD experiment. Figure 6.7 shows the outcome: The number of clients the server can handle is now limited by the link capacity while inducing only 70% CPU load on one core for serving all the 1000 clients (denoted as *HTTP/sendfile* in the figure).

Even though, *sendfile* offers a great performance benefit over the classical *read()/write()* approach, it has also some weaknesses. First of all, the *sendfile* mechanism can only be applied at the sending side. While this not a problem in our one-to-many communication scenario, it is a drawback for high throughput data exchange on a single link as the receiving side faces a higher overhead in traditional communication systems (e.g., TCP/IP). Second, *sendfile*—as the name suggests—can only transfer files from the file system onto the NIC. There is no support for direct memory access to user level buffers and the like. Last, the interrupt rate is still high causing cache pollution.



(a) By using either a *TCP-offload engine* (TOE) or the kernel *sendfile* mechanism, all 1000 clients can be served and the link becomes the limiting factor.



(b) Both, the *TOE* as well as the *sendfile* mechanism bring down the server CPU load significantly (*sendfile* does a better job because it eliminates the intermediate data copies on the sender). The best performance is achieved if both mechanisms are combined.

Figure 6.7: By combining the kernel *sendfile* mechanism with a *TCP-offload engine*, the server-side CPU load can be reduced significantly causing the network bandwidth to become the bottleneck.

Offloading the TCP/IP Stack

In addition to the *sendfile* mechanism, we can offload the TCP/IP stack processing from the OS kernel onto dedicated hardware—a so called *TCP-offload engine* (TOE).

As mentioned in Chapter 2, TOE's were an early attempt to reduce the communication overhead of the TCP/IP stack. Figure 6.7(b) shows why they were not that successful (denoted as *HTTP/TOE*): even though a TOE runs on expensive, dedicated hardware, it consumes more CPU cycles than the *sendfile* approach—which does not require special hardware—because it is not able to eliminate the intermediate data copy steps. While the TOE approach is able to serve the full HD stream to all the clients, it still requires 155 % CPU load (more than twice as much as *sendfile*).

However, combining *sendfile* together with the TOE results in a CPU load reduction by half as compared to *sendfile* alone. Yet, the induced load still linearly increases with the number of clients (denoted as *HTTP/sendfile+TOE*).

We conclude that HTTP over TCP is able to saturate the 10 GbE link with moderate CPU load—which is linearly increasing with the number of clients—if the kernel *sendfile* mechanism together with a TCP-offload engine are used. Without these TCP enhancements, the CPU load increases exponentially. HTTP is then not much better than RTP/UDP in terms of server side scalability.

Clients Interacting with the Stream

So far, we have only considered passive clients that, once the stream has started, do not issue any further control commands. As we have to provide VCR-like control over the stream to be *true on-demand*, we need to investigate the implications of client control commands on the server. Pausing and resuming was illustrated in Figure 6.1. In a real-world scenario, this operation does not cause much load on the server. Seeking and jumping, on the other hand, does as users typically issue these commands with a much higher frequency.

In this section, we look at the impact of the clients performing random jumps within the media at given intervals. We have each client perform a jump to a random position within the stream every 30 or 60 seconds[8] and compare the CPU load with clients performing no jumps at all. Figure 6.8 depicts the result for plain TCP (Figure 6.8(a)) as well as for *sendfile*-enhanced data dissemination (Figure 6.8(b)).

In the case of plain TCP, the server is no longer able to serve 800 clients but only 600 or 450 when the clients jump every 60 or 30 seconds, respectively. The *sendfile*-enhanced server shows a similar behavior. In order to serve 1000 clients,

[8]The clients are not synchronized—they do not all jump at the same time.

the software now induces 170 % or 230 % CPU load for the 60 or 30 seconds jump intervals, respectively.

In summary, we state that the more the users interact with the stream, the more load is induced on the server machine. In the next section, we will present our iWARP/RDMA approach and discuss it in comparison with RTP/UDP and HTTP/TCP.

6.4 Server-Efficient Media Dissemination with iWARP

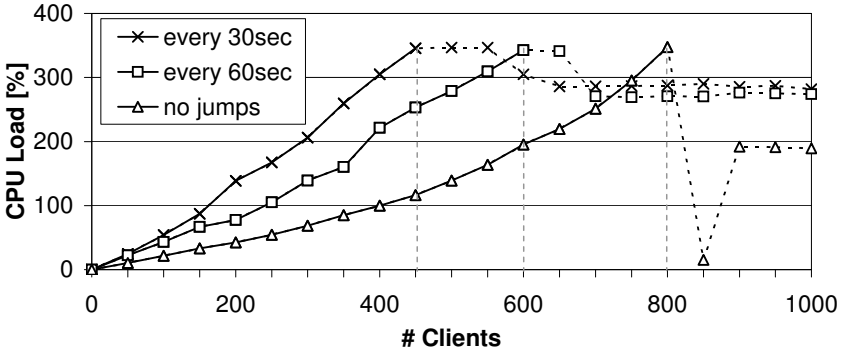
While the aforementioned *sendfile* and *TOE* mechanisms significantly improve server performance, an iWARP/RDMA-based approach potentially eliminates *all* server CPU involvement during video data dissemination. This section proposes a VoD network protocol based on iWARP, evaluates the performance of an actual implementation of the proposed protocol and discusses the benefits of the RDMA semantics for VoD systems.

6.4.1 iWARP/RDMA-based VoD Protocol

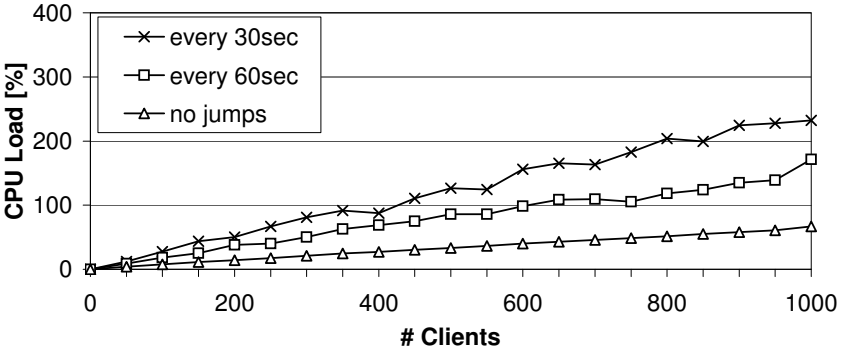
The purpose of our protocol is to reduce the server load to the minimum while fulfilling the client-side VoD requirements in terms of bandwidth and media access semantics (i.e., a bandwidth large enough to transport the high bit rate required for *full HD* and a low latency for *interactive stream control*).

RDMA Connection Management

The proposed communication protocol, depicted in Figure 6.9, starts with the clients opening an iWARP connection to the server. The connection setup allows a small amount of private data to be attached to both, the connection request and the connection response message. The clients use the private data of the connection request to select the movie, whereas the server attaches a buffer descriptor of the requested movie to its connection response. The buffer descriptor is a triplet consisting of the starting address of the buffer holding the movie, the length in bytes and a steering tag (STag) which uniquely identifies the RDMA source Memory Region. After the connection establishment phase, the clients have all the information needed to fetch the movie using the *RDMA Read* operations. In contrast to the compiler case presented in the previous chapter, we do not need a connection manager since the connections are expected to be long lasting and the initial setup cost is amortized over the playback interval.



(a) HTTP using *read()/write()*. The CPU load reaches its maximum earlier if the clients do random jumps through the media stream. A true video-on-demand solution must offer this kind of control functionality and should not be negatively affected by it.



(b) HTTP using *sendfile*. Also in this case, the CPU load increases significantly (i.e., a factor 3 in the 30 sec case) when the clients perform random jumps.

Figure 6.8: Having the clients perform random jumps rather than consuming the media stream in sequence adds a significant load to the server in the HTTP-based solutions. Even the *sendfile* is not able to avoid this overhead.

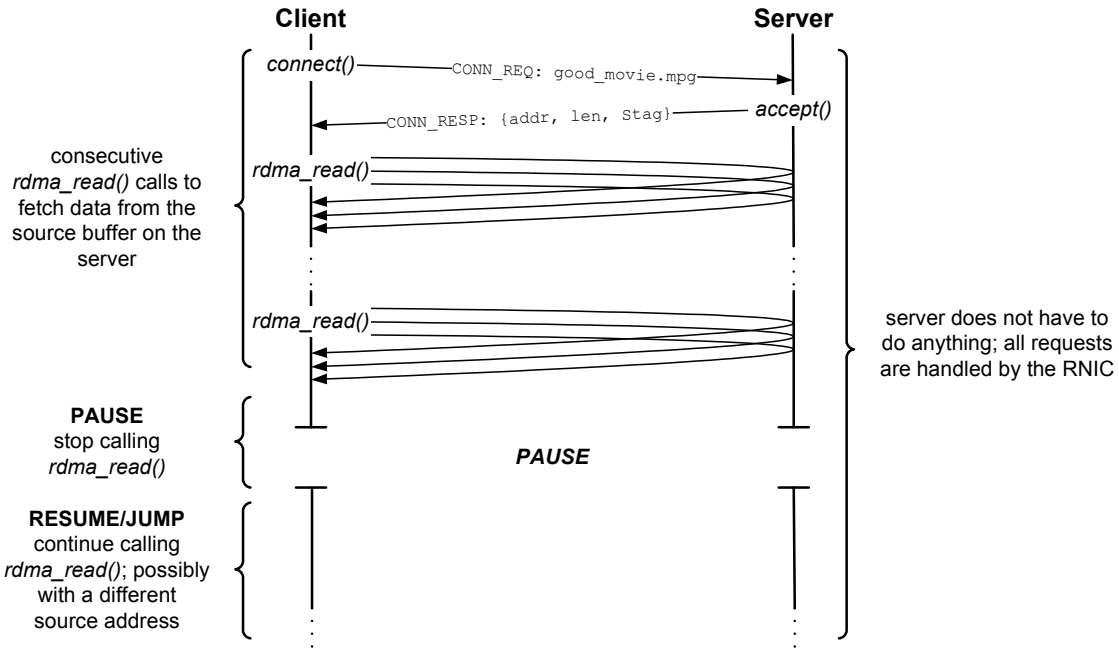


Figure 6.9: Client-driven RDMA protocol.

Transferring the Payload

Semantically, we are given the choice between *RDMA Read*, *RDMA Write* and *Send/Receive* to implement the actual media data transfers. From a protocol perspective, the key difference between a streaming service and video-on-demand is that the former is push-based while the latter is pull-based. Therefore, having the clients issue *RDMA Read* operations is the most natural way of implementing the data exchange for VoD. As this operation is one-sided, it enables a *completely client-driven protocol* offering efficient VCR-like media control at minimum server load. Figure 6.9 illustrates our simple VoD protocol. We will discuss later in this chapter (Section 6.4.4) that *RDMA Read* in combination with *Send/Receive* operations are also well-suited for implementing push-based live-stream data dissemination.

As we have seen in Chapter 3, RDMA operations perform best—low CPU load while being able to provide the maximum throughput—for data transfers larger than a certain *minimum size*. This minimum data transfer size in our setup was found to be 8 KB. We thus always request at least 8 KB from the server in our VoD extension.

Since the entire movie is statically available in the buffer advertised by the server, the clients simply need to issue continuous *RDMA Read* operations from different source offsets in order to get the data required for playback. This protocol

is a good example to demonstrate the benefit of the one-sided operations: once the client has the remote memory information, the server CPUs are not involved in the data dissemination anymore as there are no synchronization points in the protocol apart from the connection setup and teardown.

Integrating the iWARP Protocol into Existing VoD Systems

Due to the client-driven protocol, the server implementation is simple. We have thus designed it from scratch as a stand-alone application.

On the client side, we suggest two ways of integrating the iWARP communication:

- either by implementing a client-local proxy which translates the RDMA communication into HTTP or RTP or
- by extending the client implementation with RDMA code.

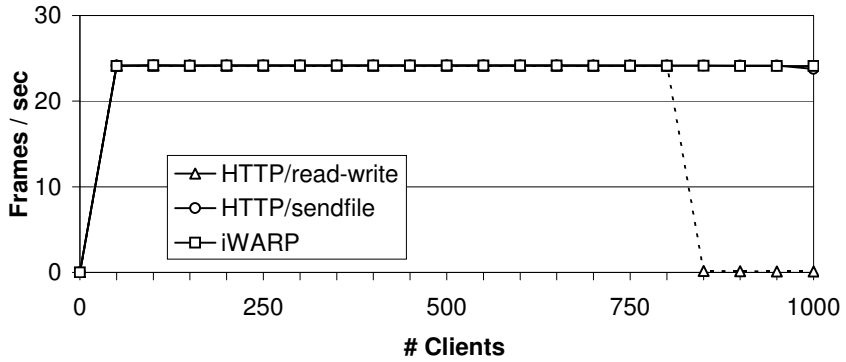
The proxy has the advantage, that any media playback software (which understands HTTP or RTP) can be used unmodified. The minor drawback is that the proxy requires the data to be copied. However, this is just a minor issue because the data rates at the clients are low.

For the evaluation presented next, however, we have extended the VideoLAN client with RDMA capabilities to minimize the number of components involved and to get an upper bound for the scalability.

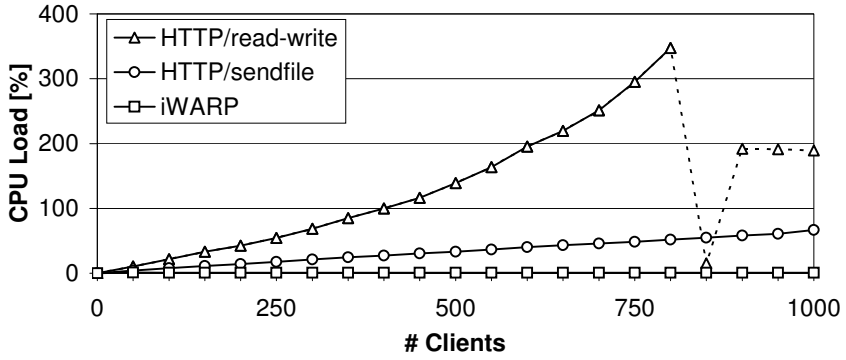
6.4.2 Protocol Performance Evaluation

In this section, we analyze the performance of our iWARP-based system in terms of server overhead and scalability.

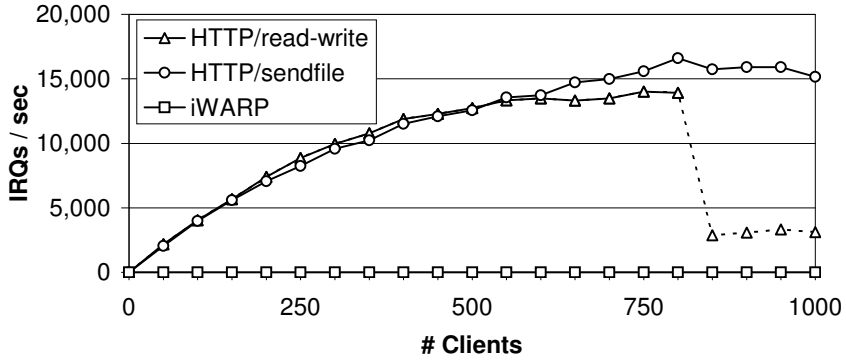
Figure 6.10 shows the results of our protocol compared with plain HTTP (denoted as *HTTP/read-write*) and HTTP with *sendfile* support (denoted as *HTTP/sendfile*) offering *full HD content*. We have limited the experiment to 1000 clients, as this is enough to fill the 10 GbE link. Figure 6.10(a) indicates that our solution is capable of serving the required data to all clients in time by saturating the link. Figure 6.10(b) reflects the server's CPU load. During data transfer, plain HTTP induces an exponential load and HTTP with *sendfile* a linear load. Our RDMA protocol, in contrast, induces no load at all, indicating good scalability. Independent of the local CPU performance, the RNIC itself is able to saturate the link by processing the *RDMA Read Requests* in hardware. In addition to avoiding data being copied, using an RDMA protocol with an RNIC completely avoids interrupts from the NIC since the CPU is not involved in protocol processing (see



(a) In terms of scalability, iWARP is just as good as *sendfile*. The link is the bottleneck: the clients get all data without playback disruption.



(b) HTTP shows a CPU load increase linear to the number of clients while iWARP incurs a constant, negligible overhead only.



(c) The *sendfile*-enhanced HTTP solution does not reduce the context switch rate compared to classical *read()/write()* data transfers. iWARP, on the other hand, does not cause any interrupts as the RNIC handles all data transfer requests autonomously.

Figure 6.10: Our proposed iWARP solution scales up to the link capacity without inducing an overhead in terms of CPU load or interrupts.

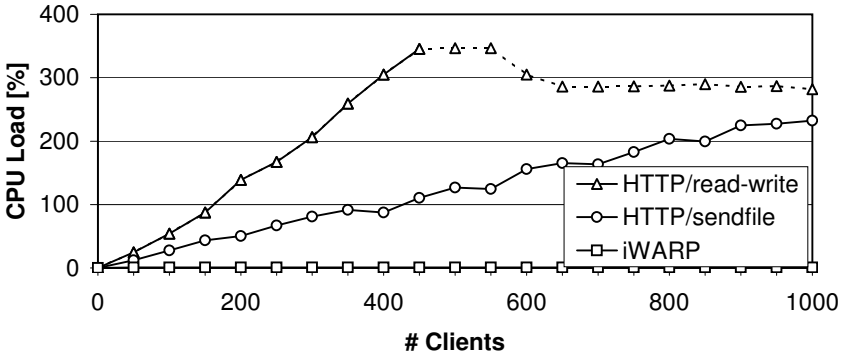


Figure 6.11: Random jumps every 30 seconds on full HD content. VCR-like media control such as jumping to a different position of the media does not cause any overhead when using iWARP. HTTP, however, suffers with increasing client interaction.

Figure 6.10(c)). This is highly desirable as it reduces the context switching rate significantly and therefore leads to a lower cache pollution.

Also, the server does not suffer from the clients exercising their media control features. Figure 6.11 depicts the impact of all clients performing random jumps within the media stream every 30 seconds. While the HTTP-based systems incur a much higher CPU load, the iWARP solution does not suffer at all thanks to the RNIC.

This could not be achieved to the same extent with a software-based iWARP stack running on the server as it is based on the kernel TCP implementation and would thus suffer in a similar way as the HTTP systems. At the client side, however, the data rate is low and it makes perfect sense to apply *Softiwarp* there.

6.4.3 In-Band VCR-like Media Control

Our fully client-driven video streaming protocol is a server-efficient way of providing VCR-like media control without requiring explicit feedback- or control messages. All efforts to control the data flows are performed at the client side, thus freeing the server from almost all application protocol processing. This is reflected in Figures 6.10(a) and Figures 6.10(b): The CPU load induced on the server is negligible while all clients receive their requested data in time. The simple protocol thus provides a number of advantages:

- Each client is free to autonomously read any amount from any position within the advertised buffer whenever new data for playback is needed. If several movies are available on a server, a client can watch any of them by simply switching to another buffer.

- The server does not need to keep track of which client is watching which movie—our server is *stateless* while disseminating data.
- An expensive stream control protocol with feedback loop as well as synchronization or packetizing overhead (as in RTP) are avoided altogether, thereby reducing not only the server load but also the overhead on the network itself.
- The server-side overhead is minimal (cf. Figure 6.10(b)) because the RNIC hardware takes care of the data transfer. It processes the inbound *RDMA Read Requests* and sends back the requested data through corresponding *RDMA Read Response* messages without requiring operating system intervention. The CPU of the server is only needed for connection establishment and tear down.
- In contrast to the *sendfile* approach, copies are avoided not only on the server but also on the clients. This feature is particularly interesting as it allows us to extend our protocol to support even real-time streams.

6.4.4 Live Streaming as a Special Case of VoD

Besides VoD, the other popular media dissemination scenario is live streaming. There, the video content is not prerecorded but generated in real time, for example by a video camera that continuously writes its output to local memory. The key difference is that in a VoD environment the client is free to choose when to fetch the next part of the video since it does not change on the server. Live streaming, on the other hand, is driven by the media source at the server, and the media buffer is continuously overwritten. In that context, a high server-side CPU availability is desirable since the data typically needs to be processed (e.g., encoding/compression) before it is transmitted over the network. The *sendfile* approach cannot directly be applied to this scenario as the data would have to be written to a file first. With our iWARP-based VoD protocol on the other hand, we can provide an efficient zero-copy solution for live streams.

For such an extension of the protocol, we look at live streaming as a special case of VoD where the user does not interact with the stream apart from starting and stopping it. The server's video data production must now be synchronized with client data consumption (*RDMA Read*): The server may do that by sending periodic notification messages through *Send/Receive* operations. Not sending the data itself but updates about data availability has a number of advantages from a server perspective. As these notifications are sent to all clients, each client can choose the stream it currently wants to receive without inducing any coordination or tracking overhead at the server. A push-based scheme, on the other hand, would require the server to keep track of which client is receiving which streams

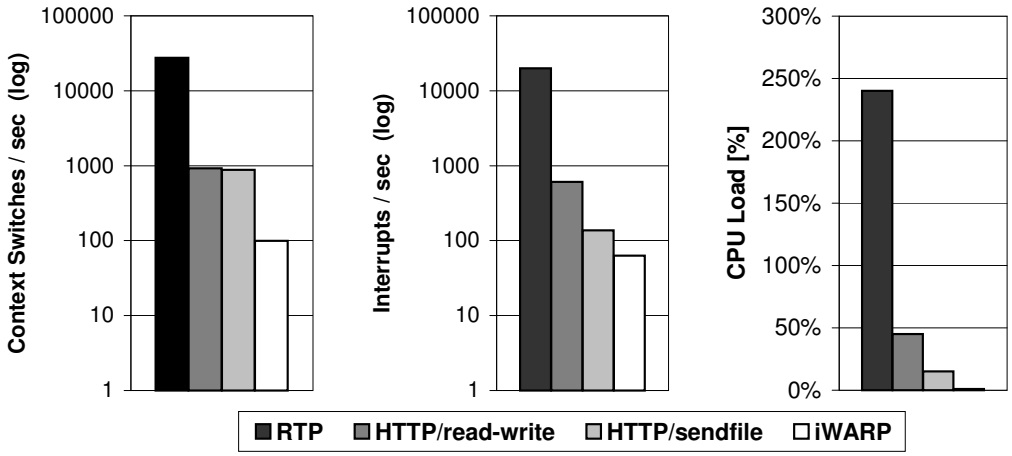


Figure 6.12: Direct protocol overhead comparison (200 clients at 5.3 Mbps).

and transmit the payload data accordingly. The amount of link bandwidth wasted is also small as the size of the notification messages is negligible compared to the payload of the stream.

6.5 Discussion

A VoD server offering HD media based on RTP suffers from a high interrupt- and context switching rate as well as a CPU overhead which is exponential to the number of clients served. By using HTTP instead, the overhead can be reduced significantly (see Figure 6.12). Applying the *sendfile* mechanism to HTTP brings the CPU load down to a linear increase with the number of clients, which is efficient enough to saturate the 10 GbE link as long as the NIC is equipped with a true DMA engine. With our iWARP-based protocol, we can reduce the context switching overhead again by an order of magnitude compared with HTTP and bring the CPU load down to a small constant.

Furthermore, to allow a parallel serving of all clients, HTTP requires each stream to be processed by a separate thread. This results in a potential waste of system resources and necessitates an increased number of context switches. In contrast, when using RDMA, a single thread is sufficient as the connection multiplexing is performed by the RNIC.

Our iWARP-based VoD solution using an RNIC is able to significantly reduce the interrupt- and the context switching rate not only on the server but also on the client. This is highly desirable as it leads to a much lower cache pollution, thus improving local application processing performance. Furthermore, a fully offloaded RDMA stack eliminates the copy overhead on both sides, which is important when

streaming data at even higher rates (e.g., at several Gbps per client).

The drawback of the RDMA solution is that the server must be equipped with an RNIC. In addition to that, also the clients must have RDMA stack support—but here *Softiwarp* may be sufficient. As the RDMA semantic is different from the common socket API, major application adaptations are needed. Furthermore, the physical memory is the limiting factor for the total size of the data to be transmitted. However, this limitation can be circumvented by applying local buffer replacement strategies (e.g., a local pyramid broadcast [VI96]) or by attaching the server to a RamSan system [ram], which offers up to several hundred gigabytes of DDR memory accessible through RDMA.

An important advantage of using RDMA is the possibility to combine client-server control type interaction with the data transfer operation itself. By issuing *RDMA Reads* at appropriate offsets, the client is able to seek through the data set without frequent tear down and re-establishment of the data channel. The server application is not even involved when the client changes the movie playback position. Using a socket-based approach, each seek would close the current TCP stream on both sides and re-open the media with the new offset. Another strong server scalability advantage of the RDMA approach is the complete avoidance of a dedicated control channel between the server and each client, which is otherwise typically implemented by just another peer-to-peer socket connection.

6.6 Related Work

Already in the early 90s, Fall and Pasquale suggested solutions to shortcut data paths within the operating system [FP93]. They proposed *splice*, a new UNIX system call, which allows moving data between two file descriptors while avoiding copies between kernel- and user address space. Using *splice* helps to reduce CPU load as well as the number of context switches and is closely related to the *sendfile* optimization we have analyzed. The same authors showed the applicability of the *splice* system call to continuous-media playback over UDP transport [FP94]. As we have seen, UDP used to be a valid choice for media transmissions at low data rates, but it has to be reconsidered when moving to the high bit rates required by full HD media content.

Another copy avoidance solution for on-demand media servers is presented by Halvorsen et al. [HJS+02]. A shared Memory Region is used between the hard drive where the media data reside and the network interface in order to create a specialized zero-copy data path from the storage medium to the communication system. This shared buffer is created at stream setup time and remains statically allocated throughout the transmission. The proposed zero-copy mechanism only assures that the data is not copied until it is handed over to the network

subsystem. The further steps are outside the scope of this work. Although their goal is similar, our solutions are fundamentally different. They suggest UDP as a suitable transport layer and restrict their applicability to non-live media content. Furthermore the data is sent out only in fixed periodic intervals, whereas we offer a true on-demand solution with a VCR-like media control.

Also Miller et al. [MKT98] have presented an I/O system design targeted at data streaming applications. Data streaming is discussed on a variety of levels, not limited to network communication. A global buffer cache is suggested to achieve zero-copy data propagation (through reference passing) within the operating system.

Further extensive research has been conducted on how to distribute media over the Internet based on the assumption that the available bandwidth is limited [SFLG00, NZ02]. Wu et al. [WHZ+01] and Plagemann et al. [PGHA00] have presented detailed discussions of the different challenges posed by designing mechanisms and protocols for Internet streaming services. In our work, we assume the bandwidth to be sufficient and investigate server-side limitations.

Peer-to-peer (P2P) driven streaming systems are in the focus of many research groups in order to address the issue of sending media data to a large number of nodes over the (relatively) limited bandwidth on the Internet. While iWARP could be used to replace the in-kernel TCP stack of most P2P systems used on the Internet, we argue that it does not make sense for two main reasons. First, RDMA is only beneficial once the throughput is high enough—this is not yet the case in today's Internet communication infrastructure. Second, every peer would have to be equipped with an RNIC to profit from the hardware acceleration which is rather costly. P2P systems scale up to overlay networks beyond our 1000 clients. If we need to support more simultaneous clients, we can readily plug more RNICs into our server as we are not limited by the CPU. At some point, however, the memory bus starts to become the bottleneck (the next chapter will present an example where this is the case). A second option (which is prevalent in practice today) to increase the scalability is to add more server machines.

6.7 Summary

We have analyzed and shown by experiment why server-side copy avoidance together with a client-driven protocol are key to achieving good scalability when offering HD media content from a single server to a large number of clients. By proposing an iWARP-based application protocol, we have shown how an efficient VRC-like media control can be implemented for video-on-demand as well as real-time streaming services and demonstrated its significant performance improvements. Finally we have highlighted its advantages and trade-offs compared to the

sendfile zero-copy mechanism offered by the Linux kernel as well as *TCP offload engines* implemented in hardware.

6.8 Outlook

Having demonstrated the potential of iWARP/RDMA on the example of providing data at a high aggregate rate from a single server machine to large number of clients (1-to-many scenario), the next chapter will focus on a different topology where we have a peer-to-peer like setup with high data rates between any two communicating nodes. We will also switch the application domain again and present an example of how iWARP/RDMA can be leveraged to accelerate distributed database processing.

7

The *Data Roundabout* - High-Speed Networks for Distributed Database Processing

The last application domain in this thesis for which we are going to assess the benefit of using iWARP/RDMA is *distributed databases*. With regard to the previous two chapters, the world of databases has quite different characteristics which provide further insight and understanding of the value added by iWARP/RDMA. The particular database architecture which we are going to present in this chapter is no longer based on client/server communication but follows the peer-to-peer principle where each communicating host is facing a roughly equal work load. As we are continuously pumping a large volume of data through the peer-to-peer overlay, we see the benefit no only with regard to CPU utilization but also in terms of a reduced memory bus contention. Last but not least, this chapter hints at the potential of applying iWARP/RDMA to cloud-style communication environments.

7.1 Introduction

With great *distributed compute power* at everyone's fingertips, either in terms of real hardware or provided by a cloud infrastructure, user expectations have grown high. Even complex ad-hoc queries on the data are expected to be answered in interactive time by automatically and optimally utilizing the power of the resources

at hand.

In this work, we look at a particular part of the challenge to meet these expectations. We apply iWARP/RDMA with its high-speed characteristics to process database queries entirely in distributed main memory in order to reach throughput rates that are beyond what commodity disks or conventional networks can provide.

Unlike in most previous distributed DB architectures, network communication is no longer to be avoided at all cost as data transfers have become significantly more efficient—in terms of latency and throughput—and do no longer induce much overhead on the hosts. Therefore, rather than trying to avoid communication at all cost, we leverage the available bandwidth.

Thanks to the efficient, hardware accelerated network communication offered by the RNICs, fundamentally new distributed database architectures can be considered. We study the opportunities offered from the perspective of the core database algorithms using a simple topological network structure: we propose the *Data Roundabout*, a ring-shaped network consisting of several machines. Each of them stores a portion of the complete data set in main memory and continuously lets it circulate around the ring.

The broader context of our work is the *Data Cyclotron* project, a joint effort between CWI Amsterdam and ETH Zurich to explore non-traditional architectures to cope with the ever-increasing requirements from large-scale business intelligence and eScience applications. Given the availability of RDMA, our approach is to rethink distributed database processing and consider the network as our friend, not as an enemy to be evaded at all cost.

7.1.1 State of the Art

The technology behind most distributed database systems today dates back to early prototypes in which the underlying assumptions and the approaches taken are largely a consequence of the network environments at that time. Most importantly, in the early days network communication was fairly slow (3 Mb/s were considered a "high-speed" network) and thus treated as a major cost factor in distributed query processing, if not the only one considered at all [BGW+81]. In a distributed setting, the primary goal of join processing techniques, was to avoid network communication, often at the expense of additional CPU work [BC81, ML86].

Another consequence of the slow networks from back then is the generic architecture that has become pervasive in distributed query processing: all data is partitioned over available network hosts (often only few of them) and remains there mostly static. A notable exception are the scalable distributed data structures [LNS96], which adapt to the arrival of data. Queries, by contrast, are shipped between hosts during query processing, usually along with state information or intermediate query results. This processing model is a good fit for classical work-

loads, where most queries are known in advance (the data can be partitioned accordingly) and involve only few, simple join predicates.

Today, roughly three decades later, the hardware landscape and application demands have changed significantly. Even commodity networks provide extremely high throughput and low latency and, thanks to hardware acceleration, incur only negligible communication cost. Real-time data mining or business intelligence applications have shifted the challenges in distributed large-volume data processing towards complex queries [DDF+09] and reflect an increasing importance of ad-hoc queries. Particularly the former class of queries often depends on functionality beyond foreign-key lookups, such as band or similarity joins.

Another shift is driven by economic forces. In the spirit of *cloud computing*, large installations of commodity off-the-shelf systems are becoming preferred over few high-performance machines. Cost effectiveness, fault tolerance, and scalability are achieved by adding and removing machines on-demand. Cloud-style operational models no longer require the dedication of machines for keeping specific data or performing specific tasks. Instead, they strive for trivial replacement, addition, or removal of network hosts as well as a low overall system complexity.

7.1.2 Problem Statement

The aim of the *Data Roundabout* architecture is to provide such cloud-style behavior for distributed database query processing. In particular, we strive for a fully decentralized design featuring

- a low management overhead,
- good scalability and
- an effective resource utilization.

The ultimate goal is to create a flexible, self-managing system which offers low query response times even for complex ad-hoc queries.

The *Data Roundabout* architecture is not limited to a specific query type. We nevertheless restrict the discussion presented in this chapter to processing join operations with special focus on data sets which are too large to fit into the main memory of a single machine and thus either have to be stored on slower secondary storage or distributed among a number of interconnected nodes forming a network.

7.1.3 Contributions

The main contributions of this chapter are two-fold:

- First, we present the *Data Roundabout*—a novel distributed database architecture based on iWARP/RDMA—that is able to exploit the computing resources at hand offering good performance and scalability characteristics. We review the design and implementation with the RDMA limitations, discussed in Chapter 4, in mind. A thorough performance evaluation of the *Data Roundabout* is provided.

- Second, we add a number of join algorithms on top of the *Data Roundabout* to assess the benefit for database query processing. Any traditional (single-host) join algorithm can be run in our distributed setup, relying on the fast interconnects to transfer the data. In another in-depth evaluation, we analyze and illustrate the implications of the *Data Roundabout* based join execution depending on the problem type and compare it with local join algorithms. Also, we assess the iWARP/RDMA benefit by contrasting it with plain TCP/IP.

7.1.4 Chapter Overview

This chapter is organized as follows. In the upcoming section, we provide some background on how large joins are processed in conventional setups and list aspects which make RDMA interesting for distributed database processing. Section 7.3 then introduces and evaluates our *Data Roundabout* transport before we add a real database operation (the join) to it in Section 7.4. The benefits with respect to distributed join processing on the *Data Roundabout* are then assessed in Sections 7.5 and 7.6. Section 7.7 provides a look into the neighborhood and Section 7.8 concludes the chapter.

7.2 Background

This section provides background information on distributed join processing which motivates our *Data Roundabout* design presented later in the chapter. We also reason about how RDMA can be beneficial in the distributed database context.

7.2.1 Processing Large Joins in Distributed Main Memory

As mentioned in the introduction, we look at the problem of calculating the join result $R \bowtie S$ for the input relations R and S with focus on the particular case where the relations R and S are too large to fit into the main memory of a single machine but are small enough be kept in some form of distributed main memory spread across a number of machines. So instead of storing the relations on the (slow) local

hard disk(s), we split them into roughly equally sized chunks and distribute those among a cluster of machines[1] connected through an iWARP/RDMA network.

Before going into the details of the *Data Roundabout* architecture, we motivate our design by illustrating how a large join operation can be executed in traditional systems.

Small Joins on a Single Machine

Let us start the discussion with the assumption that all input data *fit into the memory* of a single machine in order to get a point of reference for the distributed solution discussed subsequently.

Leaving pre-processing costs (e.g., hashing or sorting) aside, the time to perform a hash or merge join $R \bowtie S$ on a single machine can be as small as

$$(|R| + |S|) \cdot \textit{in-memory join throughput}$$

where $|R|$ and $|S|$ denote the sizes[2] of R and S, respectively. In practice, the *in-memory join throughput* often gets close to the physical bandwidth of the underlying host memory bus.

Here and in the following, we disregard the costs for the materialization of the result. Independent of the join processing technique, it would amount to $|R \bowtie S| \cdot \textit{bandwidth of main memory}$. In order to validate the output of our algorithms while maximizing the amount of available memory for holding the input relations, we only *count* the tuples matching the join predicate.

Large Joins on a Single Machine

When the input relations are larger than the available main memory, any single-host algorithm has to resort to secondary storage as temporary buffer (typically a hard disk drive). Chances are that the best way of processing the join is then to use a *block nested loops join*. It reads in turn (sufficiently large) *chunks* of each relation into main memory for performing the join until the whole data set has been processed (see Algorithm 7.1):

The available amount of main memory determines the *chunk size* for the input relations. This means that for a given buffer size M_S which holds a chunk $S_i \in S$, we will perform $n = \lceil |S| / M_S \rceil$ iterations of the outer loop. For every iteration of the outer loop (i.e., n times), we read the whole relation R into memory to execute the inner loop (the actual join) on the current chunk S_i. This results in a disk

[1] In the following discussion, the terms *machine*, *host* and *node* are used interchangeably.

[2] The size is either the number of tuples within the relation or the physical size in bytes of the relation.

Algorithm 7.1: Block Nested Loops Join

in : relations R and S (both residing on disk)
out: $R \bowtie S$

```
1 foreach block S_i ∈ S do
2     read S_i from disk ;
3     foreach block R_j ∈ R do
4         read R_j from disk ;
5         compute R_j ⋈ S_i in memory ;
```

I/O cost of $(n \cdot |R| + |S|)$.[3] Since n is proportional to $|S|$, the total disk I/O cost incurred to evaluate $R \bowtie S$ is roughly proportional to

$$(|R| \cdot |S|) \cdot \textit{disk throughput}.$$

Compared to the I/O cost for small joins we now face the product of the relation sizes instead of the sum. Furthermore, the *disk throughput* is significantly lower than the *memory bus bandwidth*. Being forced to use the disk as intermediate buffer is thus *highly undesirable*.

Large Joins on Multiple Machines

One way of reducing the disk I/O is to parallelize the outer loop of Algorithm 7.1 across multiple hosts. With n hosts available, we need to run only one outer loop iteration on each host and thus leverage the total available main memory. To this end, we have to provide each host H_i with its respective piece S_i of the input relation S and with the *entire* relation R.[4] The necessary network transfers are illustrated in Figure 7.1. Each host receives its share of S plus the full content of R in order to calculate its sub-result $R \bowtie S_i$. Note that since we compute the join in a distributed way now, the join result $R \bowtie S$ ends up as a fragmented relation, distributed over all nodes.

With this approach, we trade the network load for less disk I/O since network I/O is significantly cheaper than disk I/O these days.[5] The total I/O cost that the sender H_s has to bear in this approach is

$$(n \cdot |R| + |S|) \cdot \textit{network throughput}.$$

[3] R and S are assumed to be residing on secondary storage already and we do not account for that extra I/O cost to initially write them to disk.

[4] We assume that this is done from some host H_s which acts as the data source. The algorithm could also trivially be adapted to fetch the source data from one of the processing hosts H_i.

[5] While a typical value for disk bandwidth is ≈100 MB/s (or a multiple in RAID configurations), modern interconnects can provide 1.25 GB/s (10 Gb Ethernet) and beyond.

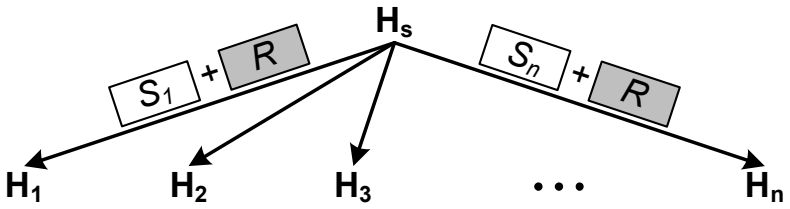


Figure 7.1: Distributed block nested loops join. The data source H_s sends to each host H_i the respective chunk S_i of S as well as the full relation R.

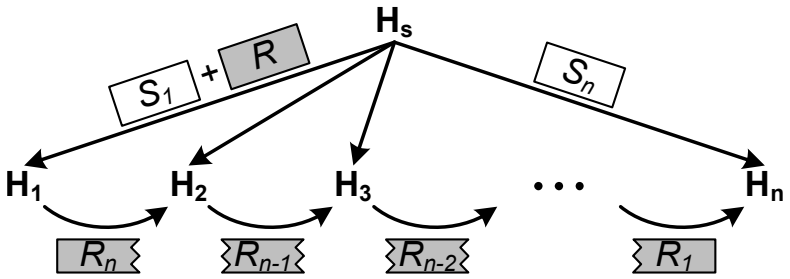


Figure 7.2: Chaining all processing hosts H_i decreases the network bottleneck at the data source and leverages the available inter-host bandwidth.

The benefit of this approach compared to the previous one (residing to secondary storage) is two-fold. First, we are bound by the (fast) *network throughput* rather than by the (slower) *disk throughput*. Second, the join processing is parallelized across all machines. We will see the practical effects of this in the evaluation presented later in this chapter.

Unfortunately, transmitting the inner join relation R multiple times can cause a serious bottleneck at host H_s. In the following section, we present a first optimization to address this issue.

A Smarter Way to Parallelize

We can decrease the bottleneck at the data source by taking advantage of the available network bandwidth *between* the processing hosts H_i. We can do this by *chaining* all H_i together as illustrated in Figure 7.2.

In this configuration, H_s sends the chunks $S_i \in S$ to each respective host as before. The relation R, however, is now also split into chunks $R_j \in R$ and sent only to the *first* processing host H_1. There, we evaluate the local fragment of the join $R \bowtie S_1$ by calculating $R_j \bowtie S_1$ for all $R_j \in R$, and *forward* the chunks R_j to the next processing host H_2 using the network link $H_1 \rightarrow H_2$, and so forth. R is split into chunks R_j for two reasons. First, because the whole relation might be too large to fit into the main memory of a single machine and second, for leveraging

the pipeline. The subsequent hosts in the chain cannot start the processing until they receive some part of R. Thus the join execution can only run in parallel when all the machines involved have their S_i plus some R_j ready to be joined. Hence, each node H_i ($i < n$) now executes Algorithm 7.2 (H_n simply drops all pieces R_j after processing).

Algorithm 7.2: Smarter Distributed Join

in : relations R and S_i (where R is received in chunks R_j)
out: $R \bowtie S_i$ ($R \bowtie S$ is available distributed across all hosts)

```
1 receive S_i from H_s ;
2 foreach block R_j received either from H_s or H_{i-1} do
3 |   compute R_j ⋈ S_i in memory ;
4 |   forward R_j to host H_{i+1} ;
```

The total network traffic on H_s (which still remains the bottleneck in terms of network I/O volume) is now reduced to

$$(|R| + |S|) \cdot \textit{network throughput}.$$

The Algorithm 7.2 has another advantages (besides the I/O load reduction of the source) compared to the data distribution model depicted in Figure 7.1: the join execution can be better parallelized thanks to the pipelined data distribution of the individual chunks R_j as there is no contention on a single link.

Yet, there is a small penalty that we might have to pay: to reach the last node in the chain (H_n), the chunks of $R_j \in R$ now have to be propagated across all $n-1$ hosts sitting in front of H_n. However, it depends on the implementation details whether the penalty is actually observable or not because also in the approach discussed in the previous section, we have to serve each host H_i (including H_n) with data. The lowest propagation delay is achieved by swapping lines 3 and 4 in Algorithm 7.2. As we will discuss in the next section, iWARP/RDMA allows us to overlap the join processing (line 3) with the data propagation logic (line 4) and thus can hide the communication delay in certain cases [BBC+03].

In Section 7.3, we describe how our *Data Roundabout* approach pushes this parallelization effort even further to support entire query plans to be executed in a distributed fashion. Before that, we cast some light on the potential benefits of iWARP/RDMA with respect to distributed databases.

7.2.2 RDMA Benefits for Distributed Databases

Application Requirements

In the context outlined above, we face requirements that are different from the ones discussed in the previous chapter on media dissemination. The most apparent one is that the distributed DB application domain is not as sensitive to *jitter* and *delay* as is media streaming. Therefore, we consider a network topology which is not client/server (or master/slave) based but follows the peer-to-peer principle. Furthermore, all hosts H_i (think clients in Chapter 6) here need to receive almost the *same data*. In the distributed join processing approach illustrated in the previous sections, each host H_i requires its individual share S_i plus the (much larger) common relation R whereas the clients in the video-on-demand application can be at arbitrary playback positions within the stream and thus require different data sets.

Efficiently Shipping Large Data Volumes over the Network

Both application scenarios (databases and streaming) involve large amounts of data to be transferred between communication partners. In the database scenario, the data volume of each transfer can be in the order of the size of the available host memory—if the data set was smaller, we would execute the database operations on a single host. We have illustrated that resorting to secondary storage is less attractive than distributing the data across a number of hosts which are connected to a high-throughput network. The iWARP/RDMA infrastructure at hand offers 10 Gbps which is faster than commodity disk throughput.

For the motivation of this work, it is important to realize that the *zero-copy* and *direct data placement* techniques offered by iWARP/RDMA allow us to ship these large data volumes efficiently (in terms of host involvement). We would like to recall the rule of thumb here where 1 Gbps of data throughput requires about 1 GHz of CPU power when using conventional TCP/IP communication. The RDMA benefit compared to TCP/IP will be assessed in the upcoming Section 7.6.

The second benefit of RDMA besides CPU cost savings is that it also significantly reduces the memory bus load as the data is directly transferred to/from its location in main memory using intra-host DMA. Therefore, the data crosses the memory bus only once per transfer. Conventional, socket-based communication may lead to noticeable *contention* on the memory bus under high network I/O. Adding additional CPU cores to the system is thus *not* a replacement for RDMA. While this was not critical in the systems presented in the previous chapters, it is vital for the *Data Roundabout* as we will see in the evaluation.

For these two reasons, the *Data Roundabout* does largely depend on the availability of RNICs because we run in a peer-to-peer setup where we expect a roughly

equal work load on each host in the ring. *Softiwarp* is hence not an option here. Thanks to the RNICs, the overhead on the CPU(s) as well as on the memory bus is thus significantly reduced[6] and we have most of the local resources still available for database operation execution even in the case where we utilize the network heavily.

Hiding the Data Transfers by Overlapping them with Computation

The join processing on each host is performed on subsets of the initial relations (i.e., $\forall j \in \{1..n\}$ execute $R_j \bowtie S_i$) in order to produce the sub-result $R \bowtie S_i$. As mentioned, Algorithm 7.2 allows the overlapping of steps 3 and 4 when RDMA is used as the underlying transport. For that, we forward[7] the current chunk R_j to the next host H_{i+1} (line 4) *while* we compute the join (line 3). RDMA does readily provide us with the asynchronous interface enabling the overlap of communication (forwarding chunks R_j) and computation (joining chunks R_j with S_i) thanks to the queue-based interaction between the verbs consumer and the underlying provider (see Chapter 3).

7.3 The *Data Roundabout* Transport

As a starting point to explore novel architectures for their potential to meet the requirements mentioned in the beginning of this chapter, we propose the *Data Roundabout* which consists of a (potentially large) number of commodity systems which are connected over high-speed RDMA connections to form a *logical storage ring* structure. As we strive for a decentralized mode of operation, each node only communicates with its two immediate neighbors in the ring and all the data is forwarded in the same direction, say clockwise. We assume the combined main memory of all participating hosts to be large enough to hold the *hot set* of the database in a distributed fashion; other data may be kept in slow, distributed disk space. Figure 7.3 shows a *Data Roundabout* of size six (i.e., one that consists of six hosts).

A fundamental difference to classical distributed systems is that we keep the queries and their state local and move base data over the network instead. In fact, we keep the data *circulating* in the ring *continuously*. Queries remain local to one or more nodes and pick necessary pieces of data as they flow by in the ring. The intuition for this stems from the spinning disks. The *Data Roundabout* is different from a disk in that it has not just one but potentially many read/write

[6] This is especially true for large data transfers of static size.

[7] Forwarding the data does not mean that we delete it locally. We only delete it once we have calculated the join sub-result.

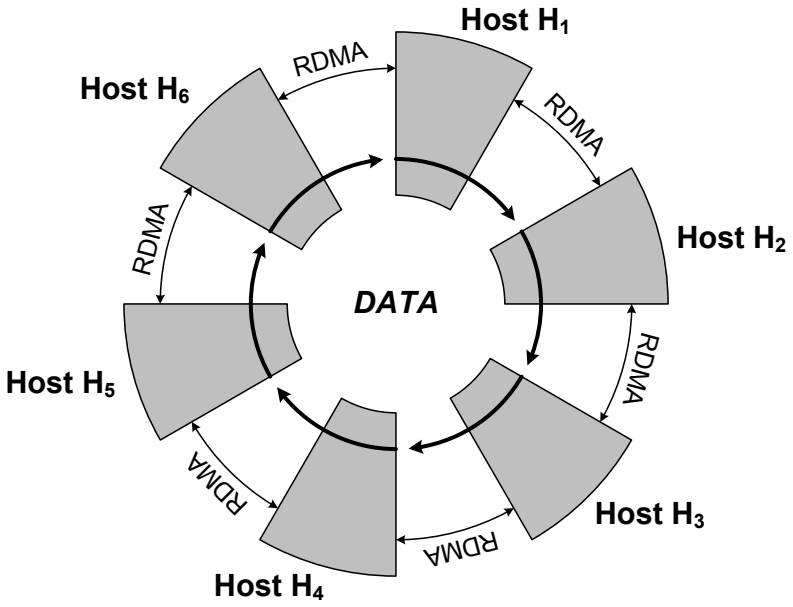


Figure 7.3: The *Data Roundabout*. Hosts H_i are organized as a ring, connected by high-speed RDMA links. The data flows continuously around the ring.

heads (each participating node is one). Furthermore, the time of circulation as well as the total capacity can be changed by expanding or shrinking the *Data Roundabout* size. With an increasing size, we also have more processors available for the query evaluation. Furthermore, as each node plays an equivalent role in the overall system, there is no bottleneck node (or every node is a bottleneck node) which is important for achieving good scalability.

Taking full advantage of the idea of rotating data, however, requires certain care in the algorithm design.

7.3.1 Considerations for Applying RDMA

As we have shown in Chapter 4, not every application can take full advantage of iWARP/RDMA. Rather, applications have to respect the characteristics of RDMA to take full advantage of the hardware-accelerated transport. We briefly repeat the ones relevant for the current context.

First, all buffers (for receiving and for sending data) have to be *sized* and *registered* with the network card *before* starting an RDMA-based data exchange. This enables the network interface card to access the application memory through its DMA engine without any involvement of the operating system. The registration process is rather CPU intensive as we have seen in Chapter 4 since it involves several address translations and because the memory must be pinned and protected

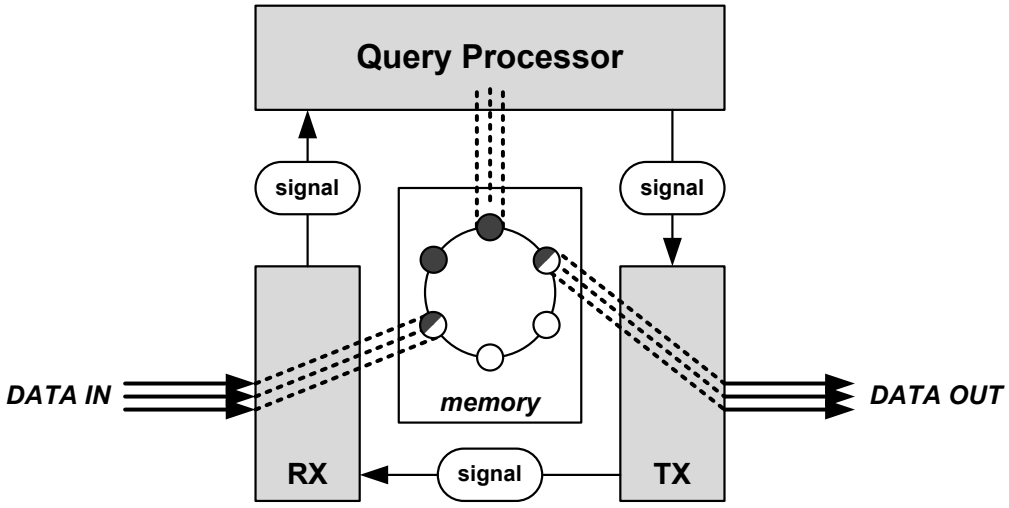


Figure 7.4: Inner workings of a *Data Roundabout* node. The data is stored in a ring buffer residing in main memory. Three entities operate on the data in a coordinated fashion: the *Query Processor*, the *receiver* (RX) and the *transmitter* (TX).

from being swapped out to disk. Whenever speed is the major concern, the cost of registration renders on-demand allocation and registration of memory buffers undesirable. In the upcoming section, we will present a buffer management scheme which is initialized in the setup phase and then remains fixed throughout the course of operation to minimize the buffer (re-)allocation overhead.

Second, each data transfer is initiated by posting Work Requests to the RNIC, a control task that still has to be performed by the CPU. To keep the resulting CPU overhead low, it is desirable to transfer the data in *large chunks* which requires fewer Work Requests to be posted. Also the RNIC itself is able to handle large data transfers more efficiently than small ones (cf. Chapter 3).

7.3.2 The *Data Roundabout* Design on RDMA

In the following, we introduce the design of the individual *Data Roundabout* nodes. The description applies to all the nodes as none of them is assigned a special role or task. A high-level picture of the inner workings of a *Data Roundabout* node is given in Figure 7.4.

Data Buffer

Considering the aforementioned memory registration limitations, we have designed each node in the *Data Roundabout* to be equipped with a statically allocated *ring*

of memory buffers to hold the data rotating in the ring (see Figure 7.4).

All of the ring buffer elements are *sized and registered in the beginning* so that they can be reused at join execution time. By this, we eliminate the need for memory registration at runtime and thereby achieve an optimal transport efficiency. Furthermore, each of these buffer elements is of the exact same size (across all the nodes within a given *Data Roundabout*) to reduce the control overhead. However, this means that we sometimes send too much data (in the case where a buffer element is not completely filled). At a bandwidth of 10 Gbps, it pays off (up to a certain extent) to send (possibly) invalid data at the end of to the payload rather than investing another round-trip time to negotiate the correct size of the payload to be transferred. As RDMA works best on large buffers, we always transfer a whole ring buffer element and not a single tuple, for instance (cf. Figure 3.20).

In-Host Data Propagation

The data propagation within the hosts has been designed in an asynchronous way involving the following three entities: a *query processor*, a *receiver thread* and a *transmitter thread*. In Figure 7.4, these entities are referred to as *Query Processor*, *RX* and *TX*, respectively.

The *query processor* is responsible for evaluating the queries and operates on one ring buffer element at a time. When it has finished processing the current buffer, it asks for that buffer to be forwarded by the *transmitter* and continues with the next buffer while the *transmitter* is forwarding the processed data. If the next buffer has already been filled by the *receiver*, the *query processor* can start processing it immediately. Notifications between these three entities (e.g., when a new buffer has been filled or an old one has been sent out and has become ready for the next iteration) is implemented through signals.

Overlapping communication and computation is a key part of the *Data Roundabout* architecture because it hides the data propagation delay of the network. Furthermore, since the CPU and memory bus overhead caused by RDMA communication is low, the *query processor* is not hindered by the concurrent data transfers. Also, this asynchronous thread composition helps to smoothen differences in processing times and jitter on the network because there are usually some buffers before and after the one that the query processor is currently working on.

Forwarding the Data

The data propagation is driven by the query processor. We have therefore designed it in a push-based manner where each node pushes the processed data to the next node in the ring. Since we are facing a unidirectional data flow, we can not

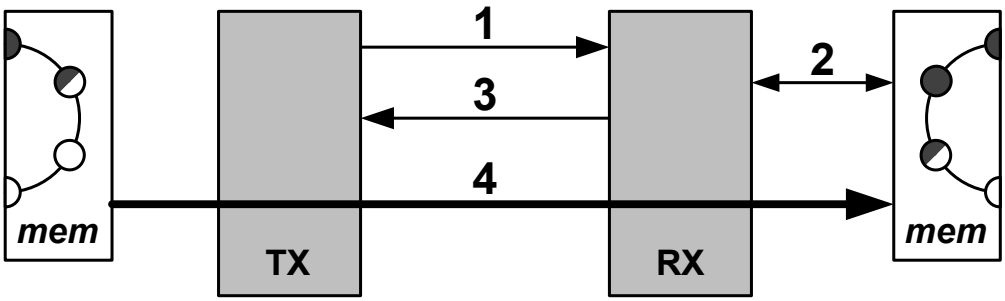


Figure 7.5: Link protocol applied when forwarding a ring buffer element. The *transmitter* requests the data propagation before sending the buffer to assure flow control.

piggyback ring buffer status updates back to the previous node. Thus, there is no implicit knowledge of how many buffers are available at the next node. In order to prevent the receiving node from running out of ring buffer elements, each node asks for permission before forwarding a buffer and delays the data transmission until it gets the confirmation for buffer space being available. This results in the three-way link protocol depicted in Figure 7.5: first, the *transmitter* requests to send the next buffer, then, the *receiver* checks the local ring buffer and confirms the request upon buffer availability. Finally, the *transmitter* forwards the payload from the buffer using an RDMA data transfer.

While this data exchange pattern seems rather cumbersome, it provides an implicit flow control mechanism and can thereby operate easily in a fully decentralized fashion. As we will see in the evaluation of the *Data Roundabout*, the performance penalty is negligible when transferring the data in chunks which are large enough.

The envisioned data propagation scheme requires some meta data to be attached to each buffer. In order to avoid mixing the payload with this meta data (which would render the database operation logic more complicated), we utilize the scatter-gather feature offered by RDMA in combination with dedicated *control buffers* (see Figure 7.6). This means that each data buffer is implicitly accompanied by a control buffer which can carry data-specific information such as where the data is originated or what type of data it is. Thanks to the scatter-gather feature, these control buffers can be of any size and reside anywhere within main memory (however, they all have to be of the same size in our setup).

With regard to the RDMA data transfer operation, we have decided to use *Send/Receive* rather than *RDMA Read* or *RDMA Write* because the performance is about the same (Chapter 3) and the two-sided semantic is better suited for the aforementioned control buffer exchange. Each node can thus decide on its own where inbound payload and control information are sent out from and where

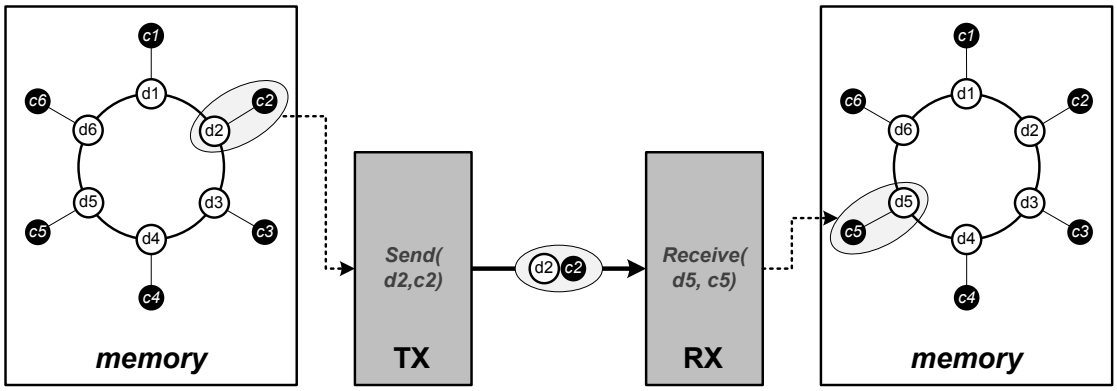


Figure 7.6: The *Data Roundabout* utilizes the scatter-gather mechanism provided by RDMA to automatically attach control information to each data buffer. dX refer to data buffers and cX designate the corresponding control buffers.

they are received to. Applying one-sided communication would be unnatural and more complicated in this scenario because additional buffer advertisements would be necessary. We thus find that the one-sided operations are not beneficial in all cases (see Chapter 6 for reasons to use one-sided communication).

Our *Data Roundabout* design satisfies the high-level requirements sketched in the beginning of this chapter. The ring is built from commodity systems and its design and data flow pattern are deliberately kept simple, in order to ease maintenance and scalability. Hence, a *Data Roundabout* system can trivially be extended or shrunken, depending on CPU and/or main memory demand. Furthermore, we do not partition data based on a priori workload knowledge, which lets us seamlessly handle ad-hoc queries.

7.3.3 *Data Roundabout* Performance Characteristics

We now look closer into what is feasible in practice with the *Data Roundabout* transport. Before assessing the potential for evaluating database queries, we run a micro benchmark on the ring network itself with the goal to identify the critical parameters that lead to efficient utilization of the resources at hand. We are particularly interested in the cost incurred in terms of CPU load due to the network communication as well as in the delay and throughput of different ring configurations.

Test Environment

Our experiments use *Data Roundabout* instances of up to six network hosts (IBM HS21 BladeServers), which is the maximum number of RDMA-equipped machines

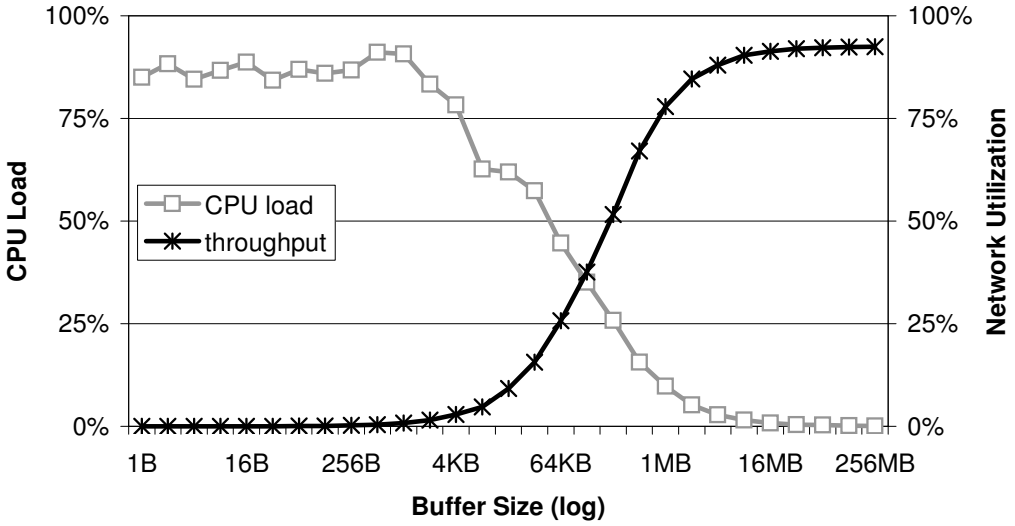


Figure 7.7: Good resource utilization requires large ring buffer elements.

we currently have available. The machines and the RNICs are the same as the ones we have used in the previous chapters. Each of them has a quad core Intel Xeon CPU running at 2.33 GHz, 32 KB L1 data cache and 32 KB L1 instruction cache, 4 MB unified L2 cache and 6 GB of main memory. The BladeServers are running Fedora Core 9 with a vanilla 2.6.27 Linux kernel.

Throughput

In order to get a significant performance advantage from distributing the queries among several machines, we must be able to provide the CPU core(s) with enough data to prevent them from stalling. The amount of data we can get to the cores per second is limited by the throughput of our ring. According to the network hardware, we can achieve up to 10 Gbps, full duplex.

Figure 7.7 shows the network utilization together with the CPU load induced induced by the transport using ring buffer elements of various sizes. It can be seen, that for small buffer elements, we are not able to make good use of the link and furthermore are wasting most of the precious CPU cycles on the communication due to the much higher frequency of Work Requests being posted to the RNIC as well as due to the three-way link protocol presented in the previous section. As soon as the buffers are larger than a critical minimal size (about 4 MB in our setup), we utilize more than 90 % of the high-speed link while investing only few CPU cycles (less than 3 %) in the data transfer. Compared to the figures presented in Chapter 3, we see that the CPU load for small buffer sizes is significantly higher and the link is saturated only for larger buffers. In the big picture, however, the

tendency is the same and the performance penalty paid due to the three-way link protocol is negligible when the buffers are large enough.

This corresponds to the earlier finding that the ring buffer elements must have a certain minimal size before we can leverage the computing and communication resources at hand. Almost all of the CPU cycles are then available for query processing and the data is being transferred at link speed yielding the lowest overall transport delay. This is a good match for our scenario as we design for workloads that are too large to fit into the main memory of a single host which means that we have large data volumes circulate the ring.

In summary, the above implies that our decentralized, simple design is able to make good use of the communication and computation resources at hand under conditions which are met in our application scenario.

Loop Delay

The next issue to address is the optimal size of the *Data Roundabout* ring in terms of the number of nodes. The *lower bound* is given by the size of the hot set of the data: as we want to keep it all in distributed main memory, we must have sufficient nodes such that the capacity of their combined main memory is sufficient.

In order to learn about the consequences of increasing the ring size and to reason about the upper limit we run the following experiment: we let the data perform a full loop around the ring and measure the overall delay until the data returns to its original sender—we refer to that as the *loop delay*. It describes the minimum time a node has to wait before having seen the complete rotating data set exactly once.

Table 7.1 shows the loop delay for *Data Roundabout* rings of sizes between 2 and 6 nodes (not yet including any query processing). We have repeated the experiment with data set sizes (contributed per node) ranging from 1 MB to 1 GB. The result is as expected: every nodes increases the delay proportionally to the ring size difference since we perform the data propagation in a store and forward fashion. Naturally, the loop delay also increases with the total amount of circulating data because each node forwards the whole set.

In terms of the upper bound, the above table confirms the expectation that it is best to use the minimum number of nodes necessary to hold the hot set. It results in the smallest loop delay and, therefore, in the lowest response time. However, the table does not yet include the query processing time. Having the query processing distributed over more nodes might result in a lower per-query execution time and thus in a lower overall response time. The query execution is taken into account in the evaluation part of the upcoming section.

We conclude that the size of the *Data Roundabout* leaves us with a trade-off: Having more nodes allows us to use more resources for the query processing but

	1 MB	16 MB	256 MB	1 GB
2 Nodes	2.2 ms	28.6 ms	453 ms	1808 ms
3 Nodes	3.3 ms	44.1 ms	699 ms	2794 ms
4 Nodes	4.3 ms	58.5 ms	925 ms	3697 ms
5 Nodes	5.4 ms	73.6 ms	1165 ms	4654 ms
6 Nodes	6.5 ms	88.0 ms	1394 ms	5567 ms

Table 7.1: Loop delay increases are proportional to the *Data Roundabout* size.

on the other hand also increases the loop delay of the data which determines the lower bound for the overall response time.

Our *Data Roundabout* transport is now ready to be extended with database queries. In the following, we are going to analyze and discuss its value for evaluating joins on large data sets.

7.4 Revolutionary Distributed Join Processing on the *Data Roundabout*

To effectively exploit the throughput opportunities offered by the *Data Roundabout* architecture, algorithms on top of the transport layer have to adhere to a rather stringent data flow pattern. We present a strategy that provides this data flow pattern and enables us to compute arbitrary database joins in distributed main memory over input data of arbitrary size.

In this section, we describe the inner workings of our join processing approach on the *Data Roundabout*. We start by providing a high-level view and then continue with a number of selected algorithms. The benefit assessment will follow in Section 7.5.

7.4.1 Problem Scenario

As introduced in Section 7.2, our focus is on the evaluation of a binary join $R \bowtie S$, where both input relations are assumed to be too large to fit into the local memory of a single machine. R and S together fit conveniently into the *distributed* memory of a *Data Roundabout* network, however.

The join $R \bowtie S$ is computed in a fully distributed fashion and its result is again available as a distributed table. As such, the join output can readily be used as input to subsequent processing in a larger query plan.

Although our experiments focus on equi-joins—thus demonstrate how the *Data Roundabout* can be combined with efficient in-memory algorithms such as hash or

sort-merge joins—we are not bound to equality predicates. Modern application classes could use this flexibility to accelerate band joins or similarity joins, for example.

The approach presented in Section 7.2.1 has minimized the amount of network I/O necessary to process input data that originates from a single host (the data source H_s). In practice, however, this data may already be spread across the nodes (e.g., when coming from an earlier evaluation of a distributed join).

In the following, we thus assume that, prior to join processing, both input tables are distributed over the network hosts H_i. We do not care how the data is distributed, but we assume the distribution to be reasonably even. This assumption is readily provided, for instance, in recent database prototypes for cloud infrastructures such as HadoopDB [ABPA+09].

7.4.2 The Join Operation

The principle of how to perform joins on the *Data Roundabout* is illustrated in Figure 7.8: one of the two relations, say S (partitioned into sub-relations S_i), is kept *stationary* on each node during processing while the fragments of the other relation, say R (partitioned into sub-relations R_j), are *rotating* in the *Data Roundabout*. Like in the distributed join evaluation approach presented earlier (cf. Figure 7.2), the processing hosts H_i are forwarding the chunks R_j in one direction. The main difference is that the last host H_n is connected back to the first host H_1. This closed loop allows the data to circulate around the ring several times if need be.

All ring members (hosts H_i), join each fragment R_j flowing by against their local piece of S (S_i) *locally* using a commodity in-memory join algorithm. The result of each $R_j \bowtie S_i$ is accumulated at every H_i and becomes part of the overall join result. After one revolution of R, all hosts H_i have seen the full relation R and have thus computed the partial join results $R \bowtie S_i$. Since the S_i are a partitioning of S, the full join result $R \bowtie S$ is now available as a distributed table spread across all H_i (ready for further processing, as mentioned above).

The task of the *Data Roundabout* transport layer is to efficiently move the rotating relation R around the network. As illustrated earlier in Section 7.3.2, the *transmitter* and *receiver* asynchronously move the pieces $R_i \in R$ in and out of the hosts, attempting to keep the *query processor* busy at all times.

7.4.3 A Selection of Join Algorithms

Within a full revolution of input relation R, all possible combinations of fragments R_j and S_i of R and S, respectively, are co-located on some host once and then combined to produce $R_j \bowtie S_i$ (as such, our join operates similar to a block nested

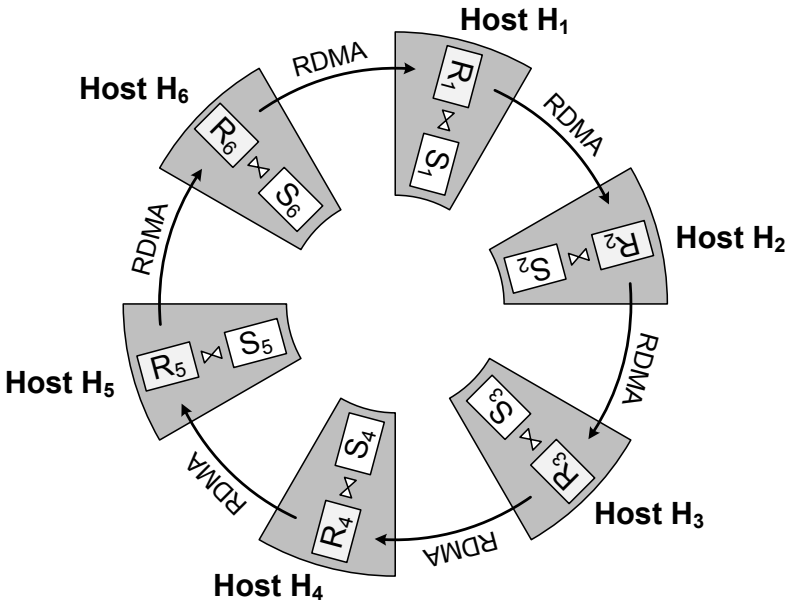


Figure 7.8: *Revolving join*: Relation R circulates in the ring while S remains stationary. Each node H_i calculates all joins $R_j \bowtie S_i$.

loops join [FGKT09]). The *Data Roundabout* can thus be used with arbitrary implementations of $\bowtie$ and support arbitrary join predicates.

Since we strive for fully distributed in-memory processing, we focus on join algorithms that are known to perform well in main memory-based setups. We have therefore ported the sophisticated MonetDB implementations of the *partitioned hash join* and *sort-merge join* to our *Data Roundabout* setup. The hash-based join algorithm inherently provides support only for equi-joins, while our implementation of the sort-merge join can also handle band joins. For all other join predicates (e.g., similarity joins), our system falls back to an implementation of *nested loops join*.

We have chosen to present these three join algorithms because each of them reveals some interesting insight about the implications of using the ring network-structure for query processing.

Partitioned Hash Join

All of the index-based join algorithms which we are going to discuss, operate in two phases. In the first phase, the *setup phase*, the index structure for the data is generated. There is no inter-node communication in this phase. During the second phase, the *join phase*, the data is rotating and the join result is being calculated based on the index.

Our implementation of *partitioned hash join* is derived from the *radix join* algorithm [MBK02] as found in the most recent distribution of the MonetDB system.[8] The implementation is carefully tuned to exploit the cache characteristics of modern CPU hardware, including the size of the on-chip L2 cache and the size of an L2 cache line.

During the *setup phase* we partition all input data and create hash tables on the partitions of the stationary in-memory join argument S_i. The subsequent *join phase* then scans partitions of R_j and probes into hash tables of the S_i partitions.

Each host partitions the two local input chunks R_j and S_i into fragments $r_{j,k}$ and $s_{i,k}$ using the same hash function on their join keys. The goal is to achieve a partitioning where each piece $s_{i,k}$ of the stationary chunk S_i and an associated hash table fit into the L2 CPU cache. Such a partitioning makes the subsequent join phase (that uses a standard hash join to scan $r_{j,k}$ and probe into a hash table on $s_{i,k}$) particularly cache-efficient, since all hash probes can be handled fully by the L2 cache. For details refer to the work by Manegold et al. [MBK02].

The join phase of our *partitioned hash join* can straightforwardly exploit the parallelism provided by modern multi-core systems by computing the disjoint $r_{j,k} \bowtie s_{i,k}$ and $r_{j,l} \bowtie s_{i,l}$ on separate CPU cores.

Sort-Merge Join

Sort-merge join operates in two phases as well. The *setup phase* here involves sorting both input fragments by their join keys. During the *join phase*, the sorted fragments are scanned in parallel and "merged" by aligning matches or skipping forward on mismatches. Although sorting incurs an additional cost over the simpler partitioning in *partitioned hash join*, the join phase of *sort-merge join* favors an even more cache-efficient (strictly sequential) access pattern and can be implemented to readily support band joins or inequality predicates.

Much like in MonetDB, our implementation relies on an efficient implementation of `qsort` in the C library, and we leverage available parallelism by sorting both input fragments (R_i and S_i) in parallel. The join phase also runs multi-threaded: We split the R_j into a number of non-overlapping sub-partitions ($r_{j,k}$) equal to the number of cores in the system. Individual threads then join the stationary S_i with one such piece of R_j.

Nested Loops

As a fall-back option and for assessing also non-indexed joins on the *Data Roundabout*, we have implemented a simple nested loops join. For the nested loops algorithm, there is no *setup phase* and the *join phase* is essentially a sequential

[8] Available since release `Nov_2009`.

scan over both relations. This yields a complexity in the order of $\mathcal{O}(|R| \cdot |S|)$. The advantage of this simple nested loops join is that it allows for similarity joins. Furthermore, it features interesting scalability properties with increasing computing resources as we will see in the evaluation.

7.4.4 Interacting with the Revolving Join

The descriptions of the algorithms above assumed that only a single join $R_j \bowtie S_i$ had to be evaluated. Such an evaluation involves the execution of both join phases: hashing/sorting and joining.

In practice, our join implementations see the same input data over and over again since each host H_i calculates its result $R \bowtie S_i$ by calculating $R_j \bowtie S_i$ for all R_j flowing by in the ring. It thus makes sense to invoke the setup phase of either join implementation only once, then re-use its output during the full execution of the join. The effort spent in the setup phase is then amortized over multiple executions of the join phase. We can do so by sending access structures or reorganized data (sorted or partitioned) over the *Data Roundabout* transport layer.

This is an instance where we can exploit the bandwidth provided by our RDMA transport mechanism. Rather than investing CPU cycles to reduce network traffic—the common strategy in existing systems—we spend some network capacity to save CPU work. Sometimes, the *Data Roundabout* system may thus suggest a different balance between the efforts spent on pre-processing and query evaluation. We will assess and discuss such trade-offs in more detail based on experimental evidence in the following section.

7.5 Experimental Assessment of the Revolving Joins

This section shows the *Data Roundabout* in action performing joins. A discussion of its characteristical features is provided. The test environment is the same as in Section 7.3.3.

7.5.1 Distributing the Join Evaluation

First, we investigate how well we are able to leverage the resources at hand with the *partitioned hash join*. To that end, we have generated input relations R and S that are just about large enough to fit into the main memory of a single machine (140 million tuples per relation with 12 bytes per tuple;[9] this results in a total

[9]The 12 bytes consist of 4 bytes for the tuple ID (join key) plus 8 bytes random payload.

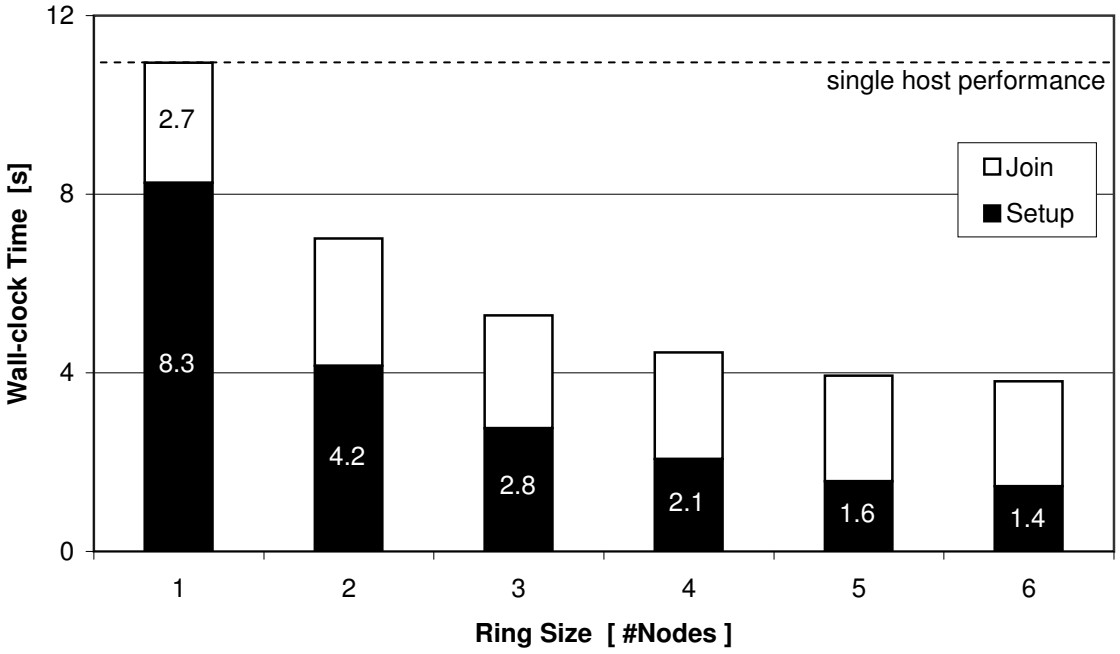


Figure 7.9: Joining a fixed data set on an increasing number of nodes.

data volume of 2×1.6 GB). The 4-byte join key is populated with uniformly distributed integer numbers.

Figure 7.9 shows the execution times observed when computing the join $R \bowtie S$ on a single host and moving on to a distributed evaluation with up to six network hosts on the *Data Roundabout*. In the distributed case, we have spread all the data evenly across all network hosts before performing the join.

The most apparent observation is that the distribution of the join considerably reduces the overall join processing time. Another particular observation is the non-existence of synchronization time, i.e., the overhead imposed by the iWARP/RDMA transport. Both observations indicate good resource utilization properties.

***Data Roundabout* Overhead.** A design goal of *Data Roundabout* is to leverage RDMA such that network communication can be fully overlapped with data processing. Our measurements confirm that, indeed, the *Data Roundabout* is able to fully hide the network cost and perform all communication parallel to the actual join processing.

Network processing will only cause an effect on the observable execution speed if the *query processor* finishes its task significantly faster than the *Data Roundabout* is able to provide new data. As we will show later, this effect can be observed in our implementation of sort-merge join.

Setup Cost. The separation into the two processing phases (*setup* shown in black and *join* shown in white in Figure 7.9) illustrates where the runtime improvement comes from: distributing the generation of the hash table for the stationary relation S cuts down the time spent in the setup phase relative to the number of participating nodes. Distribution over six *Data Roundabout* hosts, for instance, reduces the setup cost by a factor of about six (8.3 s for single-host execution vs. 1.4 s on six hosts).

Join Cost. The total amount of time spent in the *join phase*, however, is *not* improved by the distribution; a behavior that might seem surprising at first. The reason why we do not benefit during the join phase in this configuration can be explained with the particular characteristics of a hash join. During the join phase, the local hash joins scan their current piece of the rotating relation (R_j) and perform a hash lookup (into their S_i) for each tuple. Given a reasonably "friendly" configuration (a proper hash function and rare hash collisions), the cost of a hash lookup is *independent* of the size of the stationary relation S_i.

During a full run of such a join, each participating host will scan all pieces $R_j \in R$—hence, the entire relation R—exactly once. The total cost of the join phase is thus *independent* of the number of network hosts:

$$\text{cost}\,(R \bowtie S_i) \cong |R| \cdot \text{cost per hash lookup.} \qquad (\star)$$

Highly skewed data invalidates the assumption of rare hash collisions. In the next section, we illustrate how this affects the performance of the join phase.

Skewed Input

The previous experiment was based on hash-friendly, uniform key distributions. However, real-world use cases rarely follow perfect uniformity but exhibit various flavors of skew. We explore the effect of skewed input data on the *Data Roundabout* mechanism by generating input tables according to a Zipf distribution with varying Zipf factors z.

For various z factors we have generated input relations of size $|R| = |S| =$ 412 MB (36 million 12-byte tuples). For each generated instance we run the join $R \bowtie S$ once on a single host and once on a ring that consists of six hosts. Figure 7.10 reports the execution times that we measured for the *join phase* of our partitioned hash join. We omit the setup phase in this graph since it is unaffected by the data skew.

For Zipf factors of $z = 0.6$ and greater, the exponential increase of the number of *duplicates* in the data sets begins to have a noticeable effect on the execution time of our in-memory hash join. This is not a surprise: the increasing number of *hash collisions* lets hash join slowly degrade towards a nested loops-style evaluation.

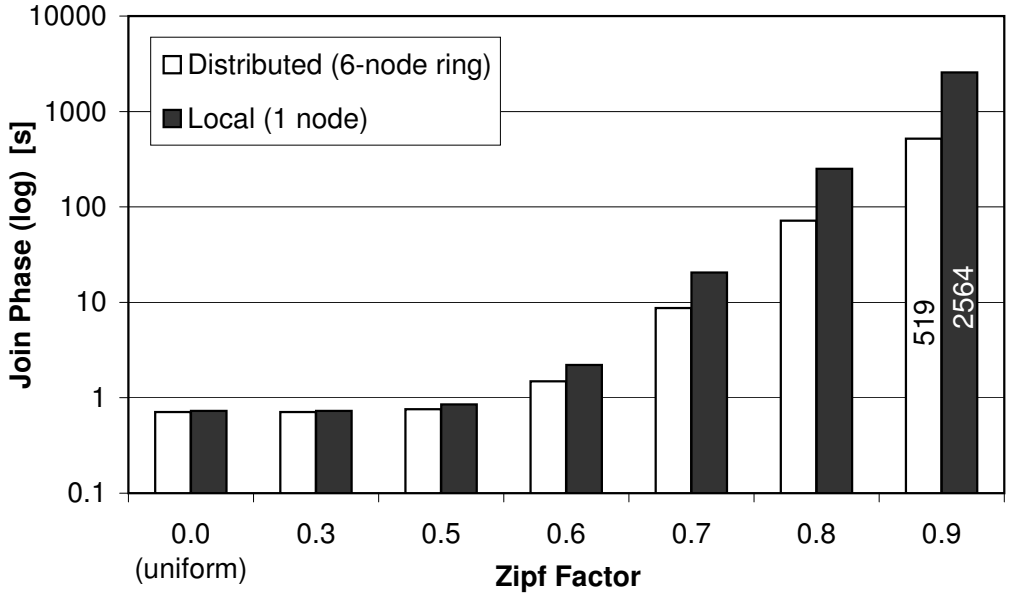


Figure 7.10: Join phase on skewed data.

The distributed join (white bars) can handle the increasing skew appreciably better. While, in line with our previous experiments, the processing of uniformly distributed data cannot benefit from a parallel execution, Figure 7.10 shows a five-fold advantage of the *Data Roundabout* join for input data with a skew of $z = 0.9$.

The benefit stems from the following: the local ring buffer mechanism of *Data Roundabout* balances differences in the execution speeds of the participating hosts. Thus, a host that is stuck in a chunk of data with a high number of duplicates will not immediately slow down the rest of the ring. A host in the *Data Roundabout* will only have to wait once it has fully consumed all the data in its local ring buffer.

Nested Loops

In addition to the index-based hash join, we now use the nested loops algorithm on the *Data Roundabout* as an example of a non-indexed join (we fall back to this join algorithm in case our index-based ones cannot answer the query). As outlined in Algorithm 7.1, the runtime of the nested loops depends on the size of *both* relations R and S. Each node has to scan the whole rotating relation (all $R_j \in R$) for each tuple from its stationary relation S_i in order to calculate $R \bowtie S_i$ which results in the following cost:

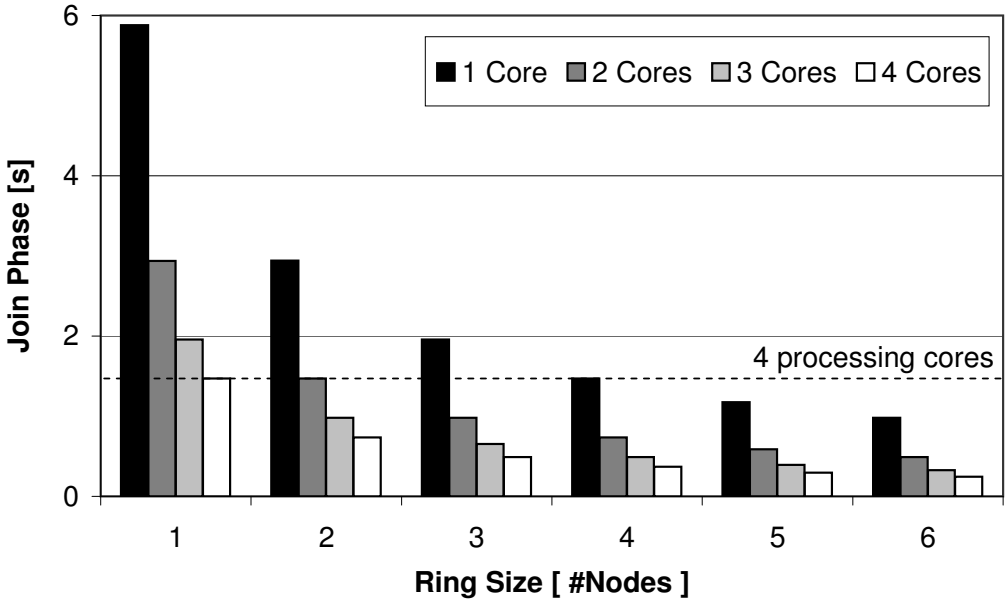


Figure 7.11: Join phase using the nested loops algorithm. The duration of the join phase is proportional to the number of involved CPUs (local and remote).

$$\text{cost}\,(R \bowtie S_i) \cong |R| \cdot |S_i|.$$

Same as for the hash join, the factor $|R|$ is independent of the size of the *Data Roundabout*. However, the relation S is split among all nodes (and CPUs) which results in a cost reduction proportional to the amount of utilized compute resources.

Figure 7.11 confirms this claim for relations of size $|R| = |S| = 1.4$ GB where each tuple is again 12 bytes large. We find that the join phase execution time is indeed proportional to the number of CPU cores involved. A particular observation is that it does not matter whether the CPU cores are local or at a remote host. In the case highlighted in the figure, for instance, we see that the join execution time is equal for any 4-core configuration (be it 1 node with 4 cores, 2 nodes with 2 cores each or 4 nodes with 1 core each). This observation indicates that the *Data Roundabout* essentially behaves as if it was one large machine (enough main memory and many CPUs).

We conclude from the hash (indexed) on uniform as well as skewed data and from the nested loops (non-indexed) join, that the improvement which can be expected from distributing the join execution on a *Data Roundabout*can result from the setup as well as from the join phase and depends solely on the algorithm used. In general, the *Data Roundabout* seems to have good scalability properties

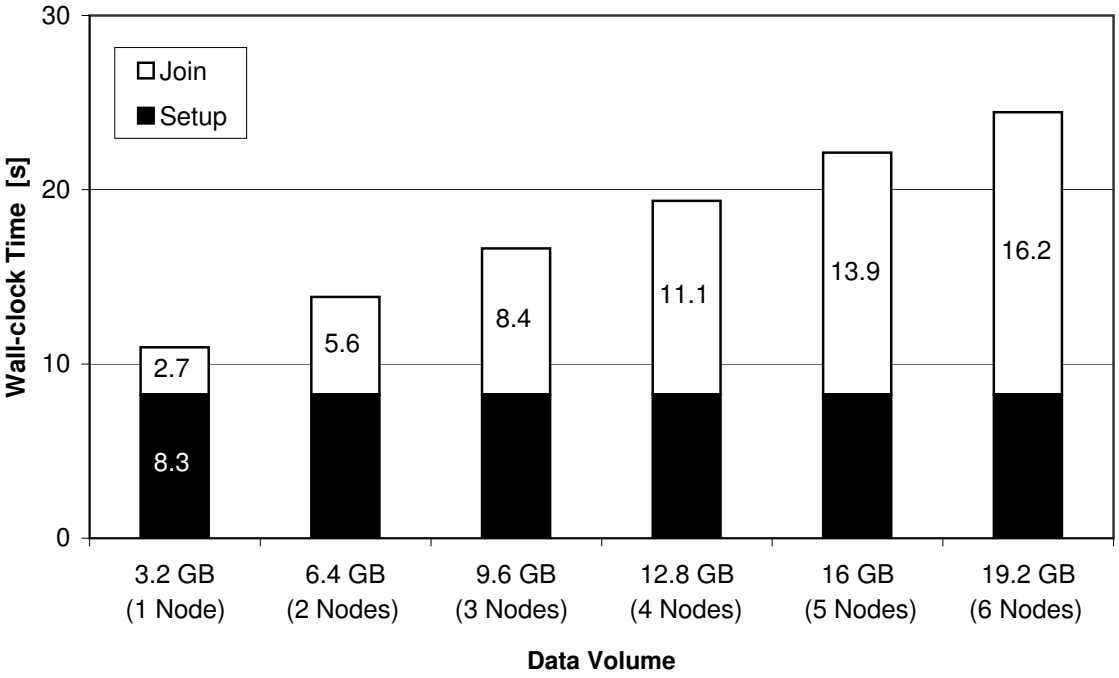


Figure 7.12: Each node adds 3.2 GB to the data set. The time spent in the join phase scales linearly with the size of R while the setup cost remains constant.

(at least up to the extent to which we hardware for experimenting) and the network communication is well hidden. This confirms our claim that thanks to RNICs, the network does not have to be avoided at all cost.

7.5.2 Large In-Memory Join

After having demonstrated the advantage of the *Data Roundabout* over local execution in terms of performance improvement, we now move on to problem sizes where a local execution is no longer feasible.

The primary purpose of performing a join operation on the *Data Roundabout* is to be able to evaluate *large* join instances that could not be evaluated on a single host without resorting to (slow) secondary storage. We verify this capability by scaling up the problem size, while simultaneously distributing the problem over more hosts H_i (we keep the per-host data volume constant at 2×1.6 GB; filled with uniform data). Figure 7.12 illustrates the resulting processing times for the hash-based join on data volumes of up to 19.2 GB.

Distributing the hash phase now leads to a size-independent setup cost. This is because we distribute the task such that the per-host data volume remains constant. The time spent in the join phase now increases linearly with the size

of the input data (or, more precisely, with the size of the rotating relation R). This confirms our assessment of the join phase cost for the hash join, as given by Equation ($\star$).

Scalability. The important outcome of the experiment is that thanks to the *Data Roundabout* we are indeed able to process large problem sizes purely in distributed main memory. A single machine with a large enough memory might have achieved comparable throughput during the join phase but would have had to pay a significantly higher price for the setup (up to n times higher on as compared to a *Data Roundabout* of size n). Furthermore, while the amount of memory addressable by a single host is severely limited[10] [intb], the *Data Roundabout* can be trivially scaled up to large configurations.

We conclude that the *Data Roundabout* makes *distributed memory* available in a simple and resource-efficient way to processing queries of arbitrary size without resorting to secondary storage.

Performance Benefit. The achieved hash-join throughput according to Figure 7.12 is 0.29 GB/s for a single machine and 0.78 GB/s for a *Data Roundabout* of size six (an improvement by a factor of about 2.7). The throughput can be increased by adding more machines which results in the setup cost being split accordingly. When there are many machines, the setup cost becomes nearly negligible and we get close to the upper bound of the throughput which is given by the theoretical link capacity of 1.25 GB/s. This performance (already on our six node setup) is hard to match when resorting to secondary storage.[11]

7.5.3 Sort-Merge Join: Setup Cost vs. Join Cost

We are now going to analyze an alternative index-based join algorithm: the sort-merge join.

The join phase characteristics of sort-merge join resemble those of the partitioned hash join as shown before. Sorting, however, incurs a significantly higher cost than the generation of the partitioned hash tables, which is why we see much higher setup costs in Figure 7.9 (partitioned hash) than in Figure 7.13 (sort-merge). As can be observed in Figure 7.13, this leads to significantly longer overall execution times for small *Data Roundabout* configurations. However, the cost is also decreasing linearly with an increasing number of nodes.

Figure 7.14 reveals that the higher setup cost slightly pays off during the join phase (white bars). Merging two sorted tables yields a cache-friendly, strictly sequential data access pattern. In the case of our largest join configuration (19.2 GB

[10] Even the modern Intel i7/Nehalem CPUs are limited to 64 GB of physical memory.

[11] A recent Serial ATA 3 drive has a theoretical maximum sequential read bandwidth of 0.6 GB/s.

distributed over 6 hosts), this cut down the time spent in the join phase from 16.2 s (partitioned hash) to 6.4 s (sort-merge), a more than two-fold advantage.[12]

As pointed out earlier, the setup cost is a one-time investment from which—in contrast to a single-host execution—the in-memory join steps can benefit several times. How often a join execution takes advantage of the higher investment in the setup phase depends on the size of the *Data Roundabout* ring. High setup costs are better amortized if the join is executed on larger rings.

Thus, the use of the *Data Roundabout* based join may suggest a different balance between the effort spent in the setup phase and its resulting performance in the join phase. For the two particular implementations of the partitioned hash join and the sort-merge join, we expect the latter to overpass the former in *Data Roundabout* configurations from ≈ 30 nodes upwards (i.e., for data volumes $\gtrsim$ 100 GB) which means that it makes sense to invest (once) in a higher setup cost in order to reduce the join cost (occurred several times) when the data and number of networked hosts are large enough.

Kim *et al.* [KSC09] have recently studied the trade-off between hash and sort-merge joins with highly tuned implementations of both algorithms and found similar performance already on a single host. Sort-merge join is then likely to be the better choice already for *Data Roundabout* configurations of small sizes.

"Synchronization" Cost

In contrast to our observations on the partitioned hash join (Section 7.5.1), Figure 7.14 shows that the join phase of the sort-merge join is too fast to fully hide the cost of network communication. The time shown in gray is the time that the *join threads* now spend waiting for new data to arrive through the *Data Roundabout* transport layer (we say they *synchronize* with the *Data Roundabout* layer).

The performance that we observe indicates that we are hitting the limits of the physical 10 Gbps transport layer. For a full join evaluation, the entire relation R has to be pumped once through each participating host. For the 6-host configuration in Figure 7.14, this means that $|R| = 9.6$ GB of data crossed each *Data Roundabout* link in 6.4 s + 2.3 s = 8.7 s, corresponding to a throughput of 1.1 GB/s, which is close to the theoretical maximum of the underlying 10 Gb Ethernet transport.

In contrast to the partitioned hash join, we are able to saturate the link already with a roundabout of size six when using an algorithm with a join phase which is as efficient as the one of the sort-merge join.

In order to improve performance using this join algorithm, we would have to equip each node with another RNIC to double the available network bandwidth.

[12] Even with the new "sync" time considered (2.3 s), the advantage is still a factor of 1.8.

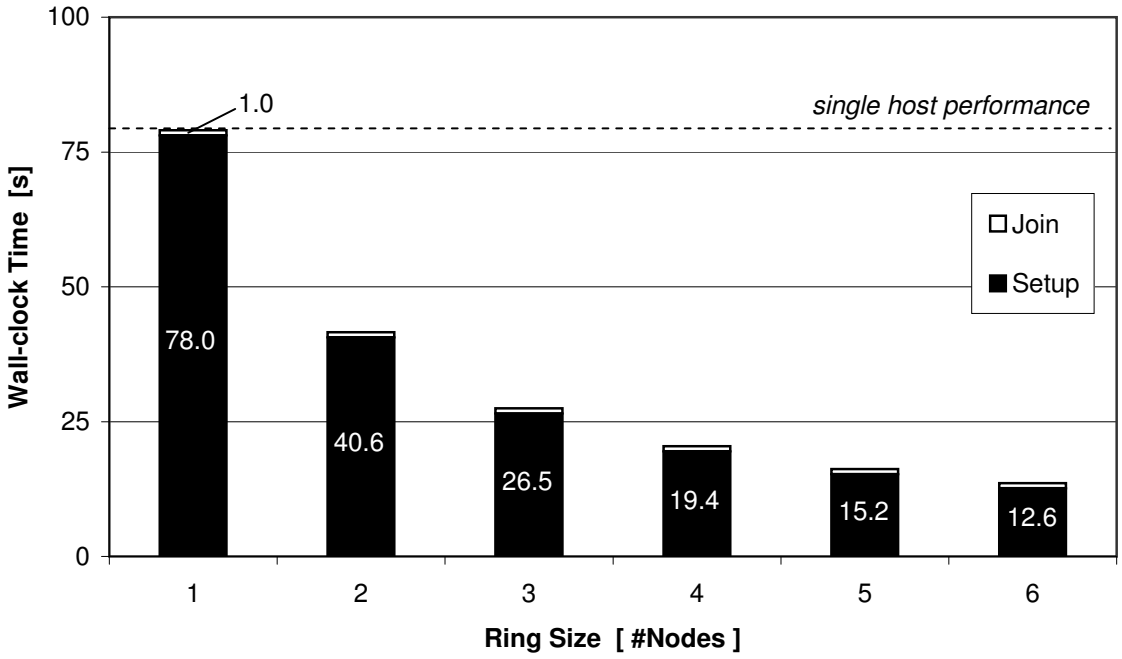


Figure 7.13: Sort-Merge Join: Joining a fixed data set on an increasing number of nodes.

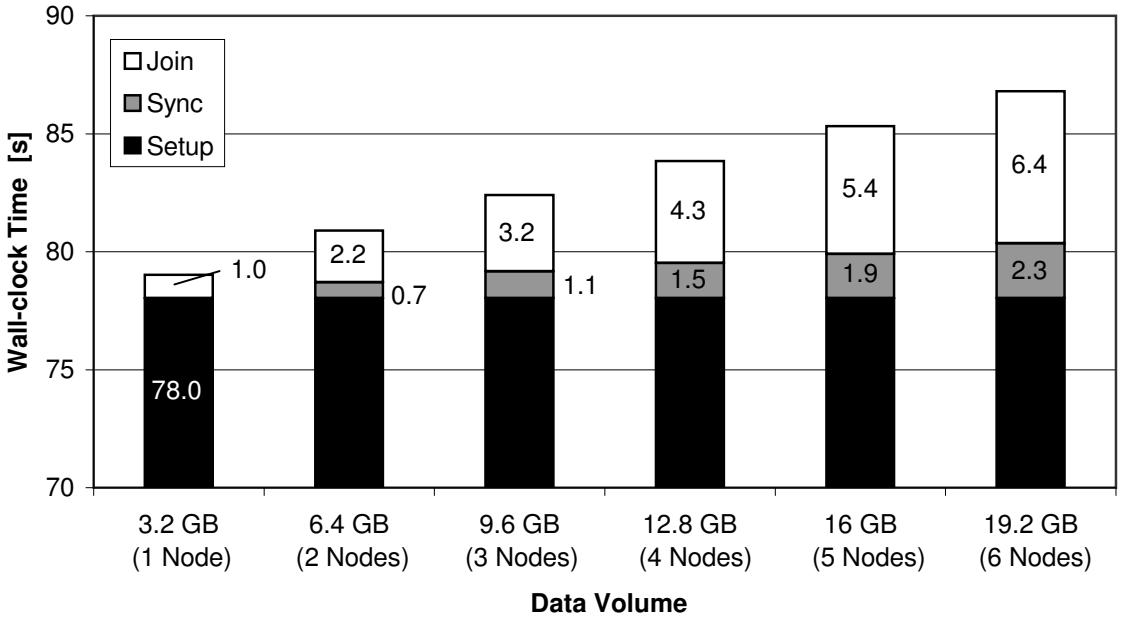


Figure 7.14: Sort-Merge Join: Each node adds 3.2 GB to the data set.

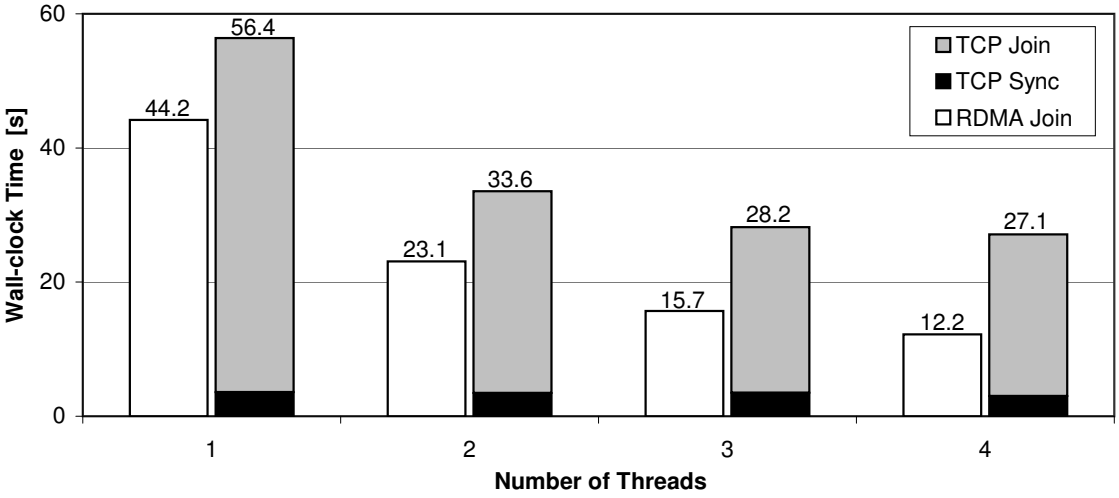


Figure 7.15: Partitioned hash join on a *Data Roundabout* of size six with a total table data of 13.4 GB. RDMA versus TCP with a varying number of *join threads*.

7.6 Is RDMA Beneficial At All?

We have presented the *join/Data Roundabout* pair running on top of a hardware-accelerated RDMA transport layer. For assessing the added value of RDMA, we now replace the iWARP/RDMA layer with ordinary TCP/IP and rerun the join experiment.

To this end, we generate data instances of sizes $|R| = |S| = 600$ million tuples with 12 bytes per tuple (corresponding to a total data volume of 2×6.7 GB) and distribute the evaluation of $R \bowtie S$ over a *Data Roundabout* of size six (as before). We run the join first with RDMA support and then with the standard socket mechanism provided by the Linux kernel. That is, we change the *transmitter* and *receiver* of the *Data Roundabout* transport to use `send` and `recv` calls instead of their RDMA counterparts. Since this obviously causes additional load on the available CPU cores, we have configured the join algorithm to allocate a varying number of cores for join processing, in order to have the remaining cores available for network communication. When using only two *join threads*, for instance, two CPU cores should always remain available for the two communication entities.

Figure 7.15 shows the execution times for the *join phase* of our partitioned hash join (the setup phase is independent of the transport mechanism and we therefore omit it in the comparison). The RDMA-based *Data Roundabout* outperforms the TCP-based one in all configurations. We also observe that even though the transport is multi-threaded, the TCP approach (in contrast to RDMA) is not able to fully hide the synchronization time. Surprisingly, RDMA is also better in the case where only 1 core is computing the join and three cores are available for the

	Throughput RDMA	Throughput TCP	Improvement Factor
1 Thread	0.30 GB/s	0.24 GB/s	1.3
2 Threads	0.58 GB/s	0.39 GB/s	1.5
3 Threads	0.85 GB/s	0.47 GB/s	1.8
4 Threads	1.10 GB/s	0.49 GB/s	2.2

Table 7.2: Throughput achieved using several degrees of local parallelism. The higher the parallelism, the greater is the benefit of using RDMA.

data propagation. This is due to the fact that RDMA not only saves CPU cycles by avoiding the intermediate data copies but also reduces the context switch rate due to the queue-based communication style. Hence, the two RDMA transport entities are only interrupted upon completion of the whole transfer and not every time some piece of data is ready for delivery. This results in little disturbance of data processing operations and thus in a low cache pollution.

With TCP sockets, on the other hand, the *receiver* must bring (copy) the data from the socket buffer into the application memory on its own. So the *receiver* is frequently interrupted. This *indirect* data placement costs CPU cycles and causes many more context switches. Furthermore, the incoming data might not completely fit into the socket buffer, causing the transport to stall until the *receiver* has copied the data from the socket buffer into the application memory. While we could utilize this behavior in the previous chapter to pause the video stream, it is counterproductive in our current scenario.

Yet, the largest performance benefit between RDMA and TCP results when using all four cores for the join processing (cf. Table 7.2). *Join threads* and *communication threads* now all compete for the available CPU cycles, pollute each others caches, and cause a large number of context switches. The benefits of the cache-efficient join algorithms are mostly annihilated.

Another important aspect of iWARP/RDMA with respect to the four-thread configuration is the reduced traffic on the memory bus. As we pointed out in Chapter 2, the data crosses the memory bus only once between the network and the host memory when using the direct data placement provided by RDMA. TCP on the other hand requires three to four crossings in either direction. In terms of the hash join, we face roughly the following load on the bus for each rotation step:

read the stationary chunk S_i
read the rotating chunk R_j
send R_j to the next host
receive R_{j+1} from prev. host

As $|R_j| = |S_i| = 1.1$ GB and since we are running on a *Data Roundabout* of

	CPU Load RDMA	CPU Load TCP
1 Thread	25 %	31 %
2 Threads	50 %	59 %
3 Threads	76 %	84 %
4 Threads	100 %	86 %

Table 7.3: CPU load during the join phase of the hash join. 100 % refers to all four cores being completely busy. TCP is not able to keep all CPUs busy.

size six, this means for RDMA that a total of about

$$6 \cdot (4 \cdot 1.1 \text{ GB}) = 26.4 \text{ GB}$$

cross the memory bus in 12.2 s resulting in a load of 2.2 GB/s. In the case of TCP/IP, the steps for sending and receiving induce *each* about $4 \cdot 1.1$ GB yielding a total of

$$6 \cdot (2 \cdot 1.1 \text{ GB} + 2 \cdot 4.4 \text{ GB}) = 66 \text{ GB}$$

in 27.1 s which is equivalent to a memory bus traffic of 2.4 GB/s. We have measured a memory bus bandwidth of 3.5 GB/s in our systems which means that the TCP-based join execution could not reach the 12.2 s from the RDMA-based one even if the CPUs were fast enough (the memory bus would have to sustain 5.4 GB/s).

Adding more CPUs is hence not an alternative to RDMA: in the case where all cores are processing the join (using TCP), the total CPU utilization reaches only about 86 % (Table 7.3) which indicates that adding further CPUs would not yield an improvement. RDMA, on the other hand, incurs a CPU load which matches the number of cores that are computing the join indicating that it is able to fully utilize the available compute resources.

While the higher memory bus load certainly contributes to the slow-down of the TCP/IP based partitioned hash join on the *Data Roundabout*, it can become the limiting factor for join algorithms (such as the sort-merge join) where more time is invested in the setup phase to speed up the join phase. We have found this confirmed: while RDMA yields a join-throughput improvement by a factor of 2.2 for the partitioned hash join, we have found an increased factor of 3.2 for sort-merge where the performance benefit of the join phase can only be leveraged if the memory bus is not congested.

7.6.1 *Data Roundabout* Characteristics Summary

The *Data Roundabout* essentially provides the necessary infrastructure to leverage existing in-memory query evaluation techniques to the processing of large data

sets in a distributed environment.

Leveraging Main-Memory Resources. The main effect of the *Data Roundabout* is the efficient use of available main memory resources in a multi-host setup. In many cases, this is going to make the join processing viable at all, when no single host would be available to perform the full join locally.

Applicability. A virtue of the *Data Roundabout* is that it does not depend on any particular pattern that supported query types would have to satisfy. As such, due to its scalability properties, the *Data Roundabout* can also be applied to tackle problems that are not amenable to any of the existing (often hash-based) optimization strategies simply by throwing a lot of hardware at them.

In-Memory Query Processing. Likewise, the *Data Roundabout* is oblivious of the algorithm that is used to implement the in-memory query evaluation. As a consequence, the use of the *Data Roundabout* will not always yield the same benefit. The resulting CPU load, for instance, will benefit those in-memory query implementations best that would show poor scaling otherwise (e.g., nested loops joins).

7.6.2 Going Really Large - An Outlook.

Taking all of the above into consideration, we argue that iWARP/RDMA is not only beneficial for the *Data Roundabout* but it is key to its success. If we plan to deal with *really large* tables (i.e., Terrabytes), we need many nodes to hold the data. According to our findings from above, this means that the setup cost becomes small compared to the join cost. Hence, we should utilize a join algorithm like the sort-merge join which has a cache-efficient, streamlined join phase and invest somewhat more into the index setup. As we have seen, the network will eventually become the bottleneck. We can add more RNICs to the machines and run at multiples of 10 Gbps. The next bottleneck is then either the memory bus or the CPUs. However, if we did not utilize RNICs, we would be bound by the memory bus already with a single NIC and would never be able to achieve the throughput of the RDMA-based *Data Roundabout*. In addition to that, we can not at all profit from future, even faster networks.

7.7 Related Work

We kept the design of the distributed join processing and the *Data Roundabout* transport layer deliberately simple. As such we feel that many of the ideas presented in this work would blend well with existing research and with some of the recent developments in hardware technology. The availability of a fast transport

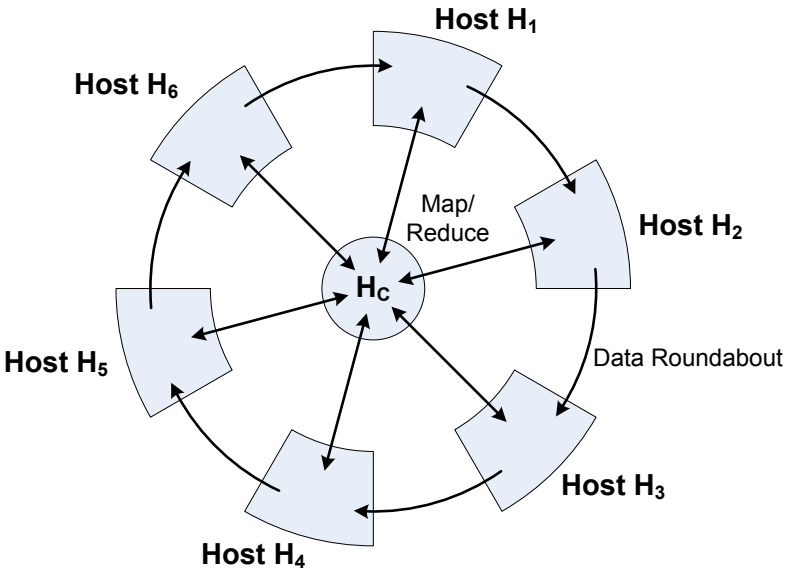


Figure 7.16: MapReduce architecture enhanced with *Data Roundabout* based join processing.

mechanism eliminates much of the urgency to reduce network transfer volumes as was the primary goal of earlier work [BC81, ML86, VG84].

Our spinning join setup resembles the *DataCycle* [BGH+92] or the *Broadcast Disks* [AAFZ95] systems that put significant effort into properly scheduling data on the transport stream. Integrating the ideas of that work into the *Data Roundabout* system is part of the ongoing *Data Cyclotron* effort [GK10] and has already inspired a number of design decisions in the evolving system prototype.

More recent work includes new systems designed for cloud environments. While systems built on MapReduce-style architectures (such as the recently proposed HadoopDB [ABPA+09]) can achieve excellent scale-out for certain types of queries, they still lack a convincing means to perform arbitrary joins *across* the pre-assigned data partitions. The *Data Roundabout* approach could fill this gap (see Figure 7.16) and enable the vision of a distributed true-SQL system.

On the technology side, joins on the *Data Roundabout* could be an interesting application for Intel's emerging *I/O Acceleration Technology (I/OAT)* [ioa]. With help of the *Direct Cache Access* (DCA) feature of I/OAT, network controllers can place data directly into CPU caches. As we have shown in Section 7.3.1, the *Data Roundabout* works well already with RDMA transfer units of around one megabyte, small enough to be loaded straight into caches. This might not only help to cut down transport latencies, but also yield an even further reduction of main memory bus contention.

Finally, we would like to relate our work to the *systolic systems* developed in

the 1980s. Systolic systems are composed of a network of processors with a simple rhythmical (hence the term "systolic") data flow in-between. Although Kung and Leiserson [KL78] had small-scale, on-chip processing units in mind when they presented the first "systolic algorithms," some of the observations made at the time may still be applicable to a *Data Roundabout* ring.

7.8 Summary

In this chapter, we have presented an example of how the bandwidth offered by modern networks with iWARP/RDMA capabilities can be exploited for distributed database processing. For that, we have proposed the *Data Roundabout* architecture which organizes processing nodes in a ring shaped network. We have further illustrated the peculiarities of the *Data Roundabout* by running distributed join queries on large data sets on top of it. The *Data Roundabout* has promising characteristics also in other settings [GK10, Ker08].

With the *Data Roundabout* based join, large database joins can be processed as *in-memory joins* by taking advantage of the distributed main memory in a cluster system. The system becomes CPU-limited instead of bound by disk or network I/O. Other than in a centralized system, the capacity of a *Data Roundabout* storage ring can be scaled up trivially, making it possible to process input data of arbitrary size. In line with the idea of cloud computing, such scaling may even be performed at runtime, based on application workload demand.

The effect of distributing CPU load depends on the particular query problem and on the algorithm chosen to perform intra-host evaluation. We have shown that critical and CPU-intensive sub-tasks, such as hash generation or joins over skewed data, can benefit best from the *Data Roundabout* mechanism.

Last but not least, we have shown and argued that it is key to have an RNIC which offers hardware-accelerated direct data placement in order to fully leverage the available high-performance communication infrastructure.

8 Conclusion

Following "Moore's Law", computing power per machine doubles every two years on average. However, network technology performance has recently grown at a much faster pace. Because of this trend and the unavoidable overhead in common TCP/IP stack implementations, an increasing share of a host's processing power is dedicated to pure network I/O and therefore unavailable to application processing. In the course of this thesis, we have investigated the suitability of Remote Direct Memory Access (RDMA) to address this problem. RDMA is a mechanism whereby data can moved efficiently between the application memory of the local and remote computer. In bypassing the operating system, RDMA significantly reduces the CPU cost of large data transfers and eliminates intermediate copying across buffers, thereby making it attractive for implementing distributed applications.

In the first part of this work, we have provided an overview of the research which eventually led to what we refer to as RDMA today. We have then presented a number of projects which aimed at leveraging RDMA in different application domains. The discrepancies among the final benefits of applying RDMA were a key motivator for the investigations presented in this thesis. They led us to address the following question, "Under what circumstances is RDMA able to provide a substantial benefit and what do we gain with it?". In order to answer these questions, we have first carried out extensive experimental investigations of the RDMA subsystem and its interfaces. Thereafter, we have built a selection of real world applications on RDMA in order to verify our findings and gain further insight.

iWARP: RDMA over Ethernet

Today, the RDMA data transfer model is available on top of two popular fabrics: the proprietary InfiniBand as well as standard Ethernet. In 2007, the IETF has defined a set of companion protocols, termed *iWARP*, to enable RDMA communication over Ethernet. Even though Ethernet cannot quite offer the same performance as InfiniBand, its standards-based interface, support for legacy infrastructure and lower cost provide a significant benefit. We have therefore decided to focus our research on the use of iWARP. However, many findings are valid for RDMA in general and therefore also for InfiniBand.

With iWARP, RDMA is no longer limited to HPC environments. As it runs on the ubiquitous Ethernet, which already today offers a throughput of 10 Gigabit/s as well as a fairly low latency, iWARP is a good candidate for simplifying data center infrastructures through fabric consolidation—Ethernet could thus be used for LAN/NAS, SAN as well as HPC. iWARP is also becoming interesting for increasing the performance of legacy applications which are currently communicating through TCP sockets.

iWARP/RDMA Benefits

RDMA reduces the I/O overhead within communicating hosts by avoiding intermediate copying of the data (*zero-copy*) and by removing the OS from the critical data path (*kernel bypassing*)—both of these concepts are fundamentally different from the approaches taken by the TCP socket abstraction. The immediate advantages are:

1. significant savings in CPU cycles (on both sides)
2. reduced memory bus load (factor 2–4)
3. lower context switch rate
4. lower power consumption for data intensive communication

Apart from these immediate performance benefits, RDMA has a few more appealing features:

- one-sided RDMA operations (*RDMA Write* and *RDMA Read*) in addition to the two-sided *Send/Receive* communication
- an asynchronous interface between the application and the network adapter
- scatter/gather buffers

With one-sided operations, only the application of the host initiating the data exchange is involved in the data transfer. At the remote machine, the RNIC handles the requests in hardware. The advantage of this communication style is that it allows for truly client-driven applications, whereby most of the processing tasks are taken away from the (single) server and distributed among the (large number of) clients. While the one-sided operations do not provide a better performance than *Send/Receive*, they allow for a significant improvement in terms of scalability for 1-to-n communication scenarios. This is not feasible to the same extent with TCP as we have seen in our high-definition media dissemination system.

The asynchronous interface between the application and the RNIC enables overlapping of communication with computation which is vital for hiding the communication delay of the network and for making efficient use of the compute resources. In the *Data Roundabout* project, for instance, we were able to minimize the overall query response time by hiding the network delay behind the query execution. Being able to leverage this feature provides a clear advantage over TCP—RDMA might be of limited use for applications that cannot profit from it.

Scatter/gather buffers, finally, provide an effective means for separating the payload from meta data such as control- or head information. However, this feature is not available for the destination of *RDMA Writes* as well as for the source of *RDMA Reads*—those are limited to single, continuous buffers.

Drawbacks of iWARP/RDMA

Besides the numerous advantages of RDMA, we have identified several problems and difficulties which must be taken into account when assessing the overall benefit of the technology for a certain application. These are in short:

1. the RDMA API is radically different from the socket interface (necessitates fundamental changes of the application code)

2. every communication buffer must be registered with the RNIC (costly)

3. registered memory is blocked for other applications

4. for each inbound *Send*, there must be a Receive Work Request pending (necessitates synchronization between peers)

5. buffer of Receive Work Request must be of appropriate size for data from inbound *Send* message (not always possible in advance)

6. one-sided operations need prior buffer advertisement (requires another round trip)

7. no implicit notification of remote peer when one-sided operation has finished (yet another round trip for explicit notification message)

8. expensive RNIC hardware is needed in most cases

First of all, the interface to the RDMA subsystem is radically different from sockets. Unfortunately, it is not just different but also much more complex and error-prone because a lot of the responsibility, which was hidden by the socket interface, has been moved to the application developer with RDMA.

The most critical of these responsibilities is the communication buffer management which has to be performed explicitly at the application level. All RDMA operations (including *Send/Receive*) are executed on *preregistered buffers only* which means that every memory segment which is to be used as source or destination buffer of an RDMA data transfer has to be registered with the RNIC in advance. As we have shown, this registration process is quite costly because it follows the slow control path through the operating system and triggers updates of various kernel data structures. Furthermore, address translations from (user) virtual to physical bus addresses are necessary. What is more, the size of such a registered buffer (Memory Region, MR) cannot be changed anymore—a deregistration followed by a fresh registration is necessary. This has a number of drawbacks in practice. On one hand, it is often difficult or even impossible to determine the appropriate buffer sizes in advance. On-demand buffer registration, on the other hand, has a negative impact on the performance at runtime. This cost is often overlooked. To make things worse, the registration not only costs time but also blocks the underlying memory for other applications—over-provisioning is thus only possible up to a certain extent.

The fixed size of Memory Regions is particularly critical for *Receive* operations because they are consumed from the RNIC *Receive Queue* in FIFO order and not according to matching destination buffer sizes. The RDMA specification dictates that exactly one Receive Work Request must be consumed for each inbound *Send* message. In practice, this means that the destination buffer targeted by the next Receive Work Request must provide enough space for placing the entire payload of the next inbound *Send* message.

A related issue is the fact that there must be at least one Receive Work Request on the Receive Queue whenever an inbound *Send* message arrives. An empty Receive Queue results in an error and immediate termination of the connection. The need for always having the appropriate Receive Work Requests ready for matching inbound *Send* messages necessitates a sophisticated synchronization between communicating peers—while the performance penalty of this synchronization is often low, it complicates the communication protocol.

There are also some issues with regard to the one-sided operations. First of all, the buffer information must be exchanged by means of a *buffer advertisement*

before a remote DMA operation (i.e., *RDMA Read* or *RDMA Write*) can be executed. The necessary information consists of three parts: the address, length and STag of the remote buffer. Because the STag is generated by the RDMA subsystem, a re-advertisement is necessary whenever the buffer is reregistered (e.g., after resizing). Depending on the communication protocol and network topology, such a re-advertisement might necessitate additional round trips or it might even be impractical (e.g., in the *Data Roundabout* where the data flow is strictly unidirectional).

A second issue of the one-sided operations is the fact that there is no way of implicitly notifying the remote host about the completion of a data transfer—one-sided operations only generate a Work Completion locally. For instance, a host providing some data to others for reading has no way of knowing when the others have finished the data transfers. A subsequent *Send/Receive* synchronization message exchange is thus necessary, inducing another round trip delay and increasing protocol complexity.

Last but not least, the full potential of iWARP/RDMA can only be realized when RDMA-enabled NICs are in place—they are still much more expensive than ordinary Ethernet NICs, however.

Application of RDMA in Practice

When we have started to write benchmarks and applications involving RDMA communication, we have realized that there are environments in which RDMA does not provide the expected performance advantage over TCP. If the aforementioned issues are not addressed carefully, RDMA loses all its performance advantages. We have therefore identified a number of optimizations which make a substantial difference in the overall performance of RDMA based applications. The optimizations fall into the following four categories: *application enablement*, *buffer management*, *data transfers* and *connection management*.

Enhancing Application Enablement. iWARP/RDMA communication requires both end points to be equipped with RDMA-capable network adapters (RNICs) which are still quite expensive. To enable iWARP communication over ordinary low-cost Ethernet adapters, we have proposed a software-only RDMA solution, termed *Softiwarp*. Softiwarp allows for mixed setups consisting of hardware- and software enabled RDMA because it is wire-compatible with the RNICs. Although we do not achieve the same performance and overhead reduction as a true RNIC, Softiwarp makes RDMA attractive for a whole range of applications for which RDMA would otherwise not be an option. An example of this is our high-definition media dissemination system where a significant number of low cost clients, running Softiwarp, allow the single server to leverage its RNIC(s).

Second, the iWARP/RDMA API on which the industry has agreed is rather

cumbersome and error prone to program against. As another enhancement, we have therefore proposed a user library that eases iWARP/RDMA application development significantly—particularly for programmers which are not (yet) familiar with all the details of the RDMA specification. We have shown that our library induces essentially no overhead compared to the original interface. Yet, all the functionality provided by the original API is preserved while the application development has become much easier as our interface is more intuitive and hides many tedious details. In particular, we simplify the connection- and buffer management as well as the initiation of RDMA data transfer operations.

Buffer Management Optimizations. RNICs can only execute their data transfer operations on application buffers which are registered as Memory Regions (MRs). The cost of the registration process increases linearly with the number of pages involved. Hence, for an application to profit the most from RDMA, it has to be able to reuse its buffers during operation. Yet, such reuse is only possible if the application can be designed to output all its data directly to that fixed user virtual memory address interval where the MR is situated. Furthermore the data set must always be of the same size or else either memory is wasted or the transfer fails because the MR is not large enough. If this is not possible, the application must either *copy* the data locally into an existing MR or *register* the data as a new MR on the fly.

We have found that only a combined approach is able to keep the overhead low: while it is faster to copy small data sets (smaller than the critical size), it is significantly more efficient to reregister larger buffers. The reasons for reregistering large buffers rather than copying the data are the following:

- shorter delay (up to an order of magnitude)
- fewer CPU cycles necessary (RNIC performs some of the tasks)
- almost zero load on the memory bus (the data itself is not touched)
- low data cache pollution
- MRs can be deregistered after use (memory is not blocked)

By experiment, we have shown that a straight-forward MR management can degrade the overall application performance dramatically. For an effective buffer management strategy, it is vital to respect the critical buffer size where reregistration outperforms copying—the resulting latency reduction can be of up to several orders of magnitude. Other factors like the buffer re-advertisement and the communication protocol have to be considered as well when designing the strategy.

Whenever possible, buffers should be registered after their underlying pages have been installed in order to benefit from parallel registration on SMP systems.

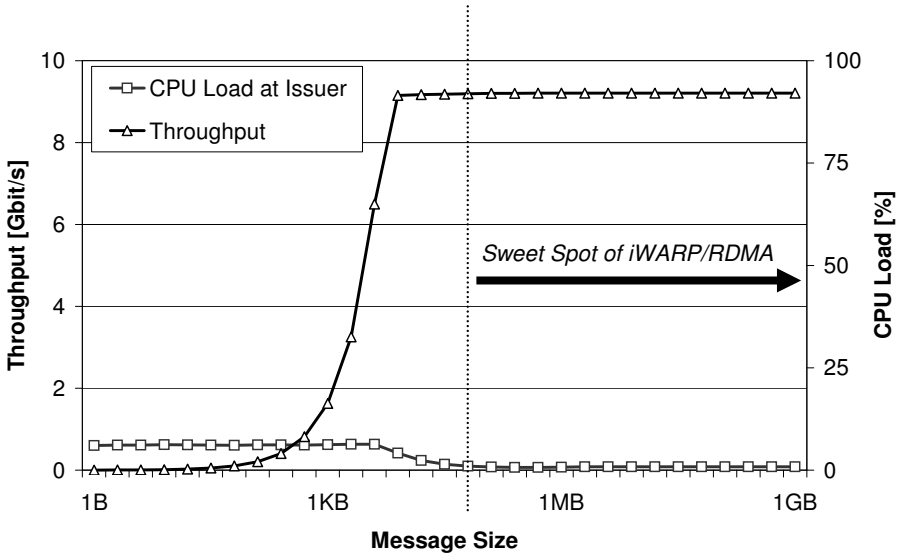


Figure 8.1: iWARP/RDMA is most effective in shipping large data chunks. A high throughput is achieved at negligible CPU load.

Also, the registration delay can be hidden by overlapping it with communication (e.g., while waiting for a response from the peer).

Data Transfer Considerations. Not only the buffer management but also the communication strategy must be chosen carefully. The first thing we have observed is that the one-sided operations offer the same performance as the two-sided ones—they can thus be chosen with focus on the appropriate semantics. Yet, one-sided operations induce neither CPU load nor context switches on the remote machine.

The sweet spot of RDMA communication is the transfer of massive data in large chunks as Figure 8.1 shows: the maximum throughput is reached with negligible CPU load. For small messages, on the other hand, the benefit is marginal because RNICs are designed to reduce the per-byte rather than the per-packet cost. For small messages, a separate TCP channel can be used but care has to be taken with regard to race conditions because the message ordering guarantee is lost.

In order to reduce the overall latency and minimize the application involvement, data should be shipped using unsignaled Work Requests whenever possible. Furthermore, the communication protocol should be designed with as few synchronization points as possible.

Connection Management. When considering the application of iWARP/RDMA, the duration of individual connections has to be taken into account. We have shown that the setup of an iWARP connection is an order of magnitude slower than the establishment of a TCP channel. As a result, iWARP has a significantly larger time-to-first-byte than TCP which makes it highly unsuitable for

applications facing short-lived connections (e.g., a webserver). In some cases, a connection manager can be deployed to keep the iWARP connections open. We have demonstrated this with the distributed compiler extension.

Real World Application Examples

In part II of this thesis, we have applied our findings and optimizations to the following three real world applications: a distributed compiler, a high-definition media dissemination server and the *Data Roundabout*.

First, we have shown the various aspects that need to be considered when enabling a legacy TCP-based application for iWARP/RDMA at the example of *rdistcc*, the distributed C/C++ compiler extension. In particular, we have demonstrated that, in most cases, there is no straight forward mapping from sockets to the verbs interface due to the explicit buffer management. The attempt to hide the RDMA API behind a socket abstraction annihilates some of the performance as was shown in the case of SDP because a socket application is not aware of the buffer registration constraints. Also, we have introduced the RDMA connection manager for reducing the connection establishment overhead. Last, we have presented an elegant and generally applicable way for making files accessible through RDMA. In terms of performance, we have seen that the use of Softiwarp can improve application performance when dedicating a core to network processing. However, this is only feasible as long as the memory bus is not congested.

Subsequently, we have focused on the applicability of RNICs for shipping large amounts of data. To that end, we have built an iWARP-based media server offering high-definition content on demand to a substantial number of clients. We could show a significantly improved scalability when using iWARP rather than TCP or UDP even when using the sendfile mechanism and a TCP-offload engine. The reasons for the improvement are the following:

- no control channel is needed: the one-sided *RDMA Read* operation allows for effective in-band VCR-like media control at minimum server load
- the server is stateless (apart from the connection management)
- copy avoidance not only on the sender but also on the receiver
- a single thread is sufficient to handle all clients; the connection multiplexing is performed at hardware level
- frequency of client interaction does not affect the server

The media server is a good example for demonstrating the advantages of one-sided operations because the protocol involves no synchronization points and only

a small number of buffer (re)advertisements (usually just one). The clients can autonomously read any amount of data from any position within the advertised buffer. Finally, we have also shown how the dissemination scheme can be easily transformed from video-on-demand to live streaming.

Last but no least, we have applied the iWARP technology to the database domain by building the *Data Roundabout*. Thanks to iWARP, we could leverage the high-speed network and with that exploit the available distributed compute resources. In essence, we traded an increased network load for a simple yet flexible communication scheme causing virtually no management overhead. The gain thanks to RDMA is many fold:

- leverage the available network bandwidth rather than trying to minimize the amount of data to be transferred
- save CPU cycles due to the reduced data management overhead which enables good scalability as well as effective utilization of the main memory and CPUs available across the network
- hide network latency by overlapping communication with computation
- scatter/gather buffers for separation of payload from control information; payload is not polluted with meta data which simplifies data processing
- exploit available bandwidth for sending access structures (e.g., hash tables) along with the data; amortize work invested in building the structures
- significant savings on the memory bus leading to more bandwidth being available for query execution
- the network is faster than disks; we can bring more data to the CPUs

The use of RNICs results in a significant reduction of the interrupt rate and thus in a lower cache pollution. Furthermore, the involvement of the CPUs in network I/O as well as the memory bus load are very low on the sending as well as on the receiving endpoints. These facts allow for heavy computation (as needed for query execution) in parallel with data exchanges. This is neither possible with TCP nor with Softiwarp. Hence, we find that assigning many CPUs to network I/O is not a replacement for hardware-accelerated iWARP/RDMA—the reduction of the memory bus load thanks to the zero-copy mechanism offered by the RNICs is essential.

Alternatives to iWARP/RDMA

As we have documented, RDMA has a lot of advantages but also some drawbacks and limitations. The most severe issues are the need for explicit buffer management and the new interface which necessitate profound application changes. While kernel TCP is not an alternative to RDMA for excessive data transfers due to the high resulting CPU load, the memory bus traffic and the high interrupt rate, its performance can be improved by adding a *TCP-offload engine* (TOE). The advantages are that part of the work is offloaded to dedicated hardware while the application interface is preserved. However, the intermediate copying cannot be avoided which means that the memory bus traffic is still high. In most cases, the performance improvement over kernel TCP is not significant enough to justify the additional hardware.

As a second alternative, there is the *sendfile* mechanism provided by most recent Linux kernels. The advantages are that some of the intermediate copying is avoided on the transmit side and that no special hardware is required. Yet, the receive side still incurs the copy overhead. This is particularly problematic because the receive side faces the higher load due to the irregular nature of data arrival. Furthermore, the *sendfile* mechanism is limited to files.

We have seen at the example of the high-definition media server that a *TOE* in combination with *sendfile* can offer a significant improvement over plain TCP which might be sufficient in some situations. The added value of RDMA is thus not always large enough to compensate for the additional cost and development effort.

Summary

We conclude that an application profiting from full RDMA performance has the following characteristics: It lives in an environment where there is little or no churn (negligible connection setup costs), it can reuse its buffers (MRs) extensively and transfers a lot of data. Furthermore, it is able to overlap communication with computation (asynchronous interface) and can make use of one-sided operations (remote computer does not need to be notified; few synchronization points in the protocol). On the other hand, an application that faces a lot of churn, operates on unpredictable, highly varying buffer sizes and depends on a short time-to-first-byte might perform better by using plain TCP.

Hence, not every application can profit from RDMA but those that can profit, profit significantly. Table 8.1 summarizes our findings and strategies presented in this thesis which are key for realizing the potential of RDMA on the application level.

Finding	Consequence
iWARP has a large setup delay.	RDMA is not suitable for applications which face a lot of churn. Keep RDMA connections open whenever possible.
There is a minimum total amount of data necessary for RDMA to pay off.	The application should transfer a large amount of data (at least 1 GB).
RDMA requires a minimum transfer unit size.	The data should be transferred in few large chunks rather than in many small ones to leverage the link capacity and keep the processing cost low.
MR (de-)registartion is costly.	Reuse buffers whenever possible. Overlap registration with protocol synchronization- or other idle time.
MR reuse is not always possible.	Copy small MRs and reregister large ones according to the critical buffer size.
Copying data causes CPU load and memory bus traffic.	Only copy small buffers for which reregistration would be more expensive.
MR reregistration necessitates buffer readvertisement.	Take cost (and feasibility) of readvertisement into account when designing the buffer management strategy and communication protocol.
MR registration causes page faults.	Register buffers whose pages are already resident: it is faster and we can leverage multicore systems.
Registered buffers block memory for other applications.	Reregister large buffers on demand and deregister them as early as possible.
Packet ordering guarantee is lost when using a separate TCP channel for control messages.	Use RDMA also for small control traffic to avoid race conditions.
RDMA provides an asynchronous interface.	Overlap communication with computation to hide network latency.
RDMA Read and *RDMA Write* operations are one-sided. The remote application does not get notified about the data transfer completions.	These operations are of limited use for protocols with many synchronization points.
One-sided operations do not offer a better performance than two-sided ones.	Select the operation type which fits best semantically.

Table 8.1: RDMA limitations and consequences for the application developer.

Bibliography

[AAFZ95] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast Disks: Data management for asymmetric communication environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 199–210, 1995.

[ABC+98] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng Hong Lim, Kenneth M. Mackenzie, and Donald Yeung. The mit alewife machine: Architecture and performance. In *Proceedings of the International Symposium on Computer Architecture*, pages 509–520, 1998.

[ABGS97] Ed Anderson, Jeff Brooks, Charles Grassl, and Steve Scott. Performance of the CRAY T3E multiprocessor. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–17, 1997.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, pages 922–933, 2009.

[apa] Apache HTTP Server. `http://projects.apache.org/projects/http_server.html`.

[BB03] Christian Bell and Dan Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 198, 2003.

[BBC+03] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An evaluation of current high-performance networks. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 10, 2003.

[BBJP06] Pavan Balaji, S. Bhagvat, Hyun-Wook Jin, and Dhabaleswar K. Panda. Asynchronous zero-copy communication for synchronous sockets in the sockets direct protocol (sdp) over infiniband. In *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, page 303, 2006.

[BC81] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28:25–40, 1981.

[BC02] Philip Buonadonna and David Culler. Queue pair IP: A hybrid architecture for system area networks. *ACM SIGARCH Computer Architecture News*, 30:247–256, 2002.

[BcFB+07] Pavan Balaji, Wu chun Feng, Sitha Bhagvat, Dhabaleswar K. Panda, Rajeev Thakur, and William Gropp. Analyzing the impact of supporting out-of-order communication on in-order performance with iWARP. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing*, page 35, 2007.

[BCS06] Mark Baker, Bryan Carpenter, and Aamir Shafi. An approach to buffer management in Java HPC messaging. In *Proceedings of the 6th International Conference on Computational Science*, pages 953–960, 2006.

[BDFL96] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li. Protected, user-level DMA for the SHRIMP network interface. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 154–165, 1996.

[BGH+92] Thomas F. Bowen, Gita Gopal, Gary Herman, Takako Hickey, Kuo C. Lee, William H. Mansfield, John Raitz, and Abel Weinrib. The datacycle architecture. *Communications of the ACM*, 35:71–81, 1992.

[BGW+81] Philip Bernstein, Nathan Goodman, Eugene Wong, Christopher Reeve, and James Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6:602–625, 1981.

[BJM+96] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the hamlyn sender-managed interface architecture. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 245–259, 1996.

[BJVP05] Pavan Balaji, Hyun-Wook Jin, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. Supporting iWARP compatibility and features for regular network adapters. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005.

[Bla96] Trevor Blackwell. Speeding up protocols for small messages. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 85–95, 1996.

[BNV+04] Pavan Balaji, Sundeep Narravula, Karthik Vaidyanathan, S Krishnamoorthy, Jiesheng Wu, and Dhabaleswar K. Panda. Sockets Direct Protocol over InfiniBand in clusters: Is it beneficial? In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 28–35, 2004.

[BSL07] Brian W. Barrett, Galen M. Shipman, and Andrew Lumsdaine. Analysis of implementation options for MPI-2 one-sided. In *Proceedings of the 14th European PVM/MPI Users' Group Conference*, 2007.

[BSP04] Pavan Balaji, Hemal V. Shah, and Dhabaleswar K. Panda. Sockets vs RDMA interface over 10-Gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *Proceedings of the 2004 Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies*, 2004.

[BSR06] Nathan L. Binkert, Ali G. Saidi, and Steven 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*, pages 315–324, 2006.

[CCC97] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture Specification, Version 1.0. 1997.

[CER+07] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU Aligned Framing for TCP Specification, 2007.

[CGY01] Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End-system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39:68–74, 2001.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27:23–29, 1989.

[CLRC+03] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 196–208, 2003.

[CR02] Don Cameron and Greg J. Regnier. *The Virtual Interface Architecture (First Edition)*. Intel Press, 2002.

[CvE98] Chi Chang and Thorsten von Eicken. A software architecture for zero-copy RPC in Java. Technical report, 1998.

[CvE00] Chi-Chao Chang and Thorsten von Eicken. Javia: A Java interface to the virtual interface architecture. *Concurrency - Practice and Experience*, 12:573–593, 2000.

[DAPP93] Peter Druschel, Mark B. Abbott, Michael A. Pagals, and Larry L. Peterson. Network subsystem design: A case for an integrated data path. *IEEE Network*, 7:8–17, 1993.

[Dav90] Bruce S. Davie. Host interface design for experimental, very high-speed networks. In *Proceedings of Compcon Spring 90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 102–106, 1990.

[Dav91] Bruce S. Davie. A host-network interface architecture for ATM. *ACM SIGCOMM Computer Communication Review*, 21:307–315, 1991.

[DDF+09] Nicolas Dieu, Adrian Dragusanu, Francoise Fabret, Francois Llirbat, and Eric Simon. 1000 tables inside the from. *Proceedings of the VLDB Endowment*, pages 1450–1461, 2009.

[DDW05] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. Design and implementation of the iWARP protocol in software. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems*, pages 471–476, 2005.

[DDW07] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. iSER storage target for object-based storage devices. In *Proceedings of the 4th International Workshop on Storage Network Architecture and Parallel I/Os*, pages 107–113, 2007.

[DIFL96] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software support for virtual memory-mapped communication. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 372–281, 1996.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM symposium on Operating Systems Principles*, pages 189–202, 1993.

[dss] Darwin Streaming Server. `http://developer.apple.com/opensource/server/streaming/`.

[DW05] Dennis Dalessandro and Pete Wyckoff. A performance analysis of the Ammasso RDMA enabled Ethernet adapter and its iWARP API. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, pages 1–7, 2005.

[DW07a] Dennis Dalessandro and Pete Wyckoff. Accelerating web protocols using RDMA. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications*, pages 205–212, 2007.

[DW07b] Dennis Dalessandro and Pete Wyckoff. Memory management strategies for data serving with RDMA. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*, pages 135–142, 2007.

[DWB+93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 7:36–43, 1993.

[DWM06] Dennis Dalessandro, Pete Wyckoff, and Gary Montry. Initial performance evaluation of the NetEffect 10 Gigabit iWARP adapter. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, pages 1–7, 2006.

[EM95] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 196–205, 1995.

[FA09] Philip W. Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, 2009.

[FBB+05] Wu Chun Feng, Pavan Balaji, C Baron, Laxmi N. Bhuyan, and Dhabaleswar K. Panda. Performance characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the 13th Symposium on High-Performance Interconnects*, pages 58–63, 2005.

[FGKT09] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the 5th International Workshop on Data Management on New Hardware*, pages 27–33, 2009.

[FGKT10] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. A spinning join that does not get dizzy. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, pages –, 2010.

[FHH+03] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, 2003.

[FHMA09] Philip W. Frey, Andreas Hasler, Bernard Metzler, and Gustavo Alonso. Server-efficient high-definition media dissemination. In *Proceedings of the 19th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 49–54, 2009.

[FM99] Michail D. Flouris and Evangelos P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Cluster Computing, Special Issue on I/O in Shared-Storage Clusters*, 2:281–293, 1999.

[FMN10] Philip W. Frey, Bernard Metzler, and Fredy Neeser. Enabling applications for RDMA: Distributed compilation revisited. Technical report, IBM Research RZ3764, January 26, 2010.

[FP93] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *Proceedings of the Winter 1993 USENIX Conference*, pages 327–333, 1993.

[FP94] Kevin Fall and Joseph Pasquale. Improving continuous-media playback performance with in-kernel data paths. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 100–109, 1994.

[gcc] GCC, the GNU Compiler Collection. `http://gcc.gnu.org`.

[gen] Gentoo Linux. `http://www.gentoo.org`.

[Geo] Johann George. A tour of the Linux OpenFabrics stack. `https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/01_Events/Event_presentations/2006_OpenFabrics_Workshop/05_A_Tour_Of_The_OpenFabrics_Stack.pdf`.

[Geo06] Patrick Geoffray. A critique of RDMA. `http://www.hpcwire.com/features/17886984.html`, 2006.

[GK10] Romulo Goncalves and Martin Kersten. The Data Cyclotron query processing scheme, 2010.

[GP03] Brice Goglin and Loic Prylli. Transparent remote file access through a shared library client. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1131–1137, 2003.

[HCPR] Jeff Hilland, Paul Culley, Jim Pinkerton, and Renato Recio. RDMA Protocol Verbs Specification, Version 1.0. `http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf`.

[HJ92] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. *ACM SIGPLAN Notices*, 27:111–122, 1992.

[HJS+02] Pal Halvorsen, Espen Jorde, Karl-André Skevik, Vera Goebel, and Thomas Plagemann. Performance tradeoffs for static allocation of zero-copy buffers. In *Proceedings of the 28th Euromicro Conference*, 2002.

[HZH+07] Wan Huang, Hongwei Zhang, Jin He, Jizhong Han, and Lisheng Zhang. Jdib: Java applications interface to unshackle the communication capabilities of InfiniBand networks. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, pages 596–601, 2007.

[ibt] InfiniBand trade association. `http://www.infinibandta.org`.

[inta] 10 Gigabit Ethernet technology overview. `http://www.intel.com/network/connectivity/resources/doc_library/white_papers/pro10gbe_lr_sa_wp.pdf`.

[intb] Intel 64 and IA-32 architectures software developer's manual, volume 1. `http://www.intel.com/products/processor/manuals/`.

[intc] World Internet stats. `http://www.internetworldstats.com/stats.htm`.

[ioa] Accelerating high-speed networking with Intel I/O acceleration technology (white paper). `http://www.intel.com/network/connectivity/vtc_ioat.htm`.

[KCH+07] M. Ko, M. Chadalapaka, J. Hufferd, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA), 2007.

[Ker08] Martin Kersten. The database architecture jigsaw puzzle. In *Proceeding of the 24th IEEE International Conference on Data Engineering*, pages 3–4, 2008.

[kJC96] Hsiao keng Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 21–21, 1996.

[KL78] Hsiang T. Kung and Charles E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282, 1978.

[KOH+98] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark A. Horowitz, Ashish Gupta, Mendel Rosenblum, and John L. Hennessy. The stanford FLASH multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, pages 485–496, 1998.

[KP93] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM Conference on Communication Architectures, Protocols and Applications*, pages 259–268, 1993.

[KP96] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, pages 817–828, 1996.

[KSC09] Changkyu Kim, Eric Sedlar, and Jatin Chhugani. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment*, pages 1378–1389, 2009.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the connection machine cm-5. In *Proceedings of the 4th Annual ACM symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[LC95] Lok T. Liu and David E. Culler. Evaluation of the intel paragon on active message communication. In *Proceedings of the Intel Supercomputer Users Group Conference*, 1995.

[LNS96] Witold Litwin, Marie Neimat, and Donovan Schneider. LH* - a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21:480–525, 1996.

[LPA09] Jiuxing Liu, Dan Poff, and Bulent Abali. Evaluating high performance communication: A power perspective. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 326–337, 2009.

[LWK+03] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th International Conference on Supercomputing*, pages 295–304, 2003.

[MAF+02] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer, Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv Wickremesinghe, and Eran Gabber. Structure and performance of the direct access file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 1–14, 2002.

[MAFS03] Kostas Magoutis, Salimah Addetia, Ra Fedorova, and Margo I. Seltzer. Making the most out of direct-access network attached storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 189–202, 2003.

[Mar02] Evangelos P. Markatos. Speeding up TCP/IP: Faster processors are not enough. In *Proceedings of the 21st IEEE International Performance Computing and Communications Conference*, pages 341–345, 2002.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, 1991.

[MBK02] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions Knowledge and Data Engineering*, 14:709–730, 2002.

[MC99] Alan M. Mainwaring and David E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. *ACM SIGPLAN Notices*, 34:119–130, 1999.

[MKT98] Frank W. Miller, Pete Keleher, and Satish K. Tripathi. General data streaming. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 232–241, 1998.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 149–159, 1986.

[MNF] Bernard Metzler, Fredy Neeser, and Philip W. Frey. Remote direct memory access (RDMA) host software. `http://www.zurich.ibm.com/sys/rdma`.

[MNF09] Bernard Metzler, Fredy Neeser, and Philip W. Frey. A software iWARP driver for OpenFabrics. In *OpenFabrics Alliance Sonoma Workshop*, 2009.

[Mog03] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 5–5, 2003.

[MRB+06] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the Mellanox InfiniBand software stack. In *Proceedings of the 12th International Euro-Par Conference*, pages 124–133, 2006.

[MSG04] Kostas Magoutis, Margo Seltzer, and Eran Gabber. The case against user-level networking. In *Third Workshop on Novel Uses of System Area Networks*, 2004.

[MZ08] Aravind Menon and Willy Zwaenepoel. Optimizing TCP receive performance. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 85–98, 2008.

[NBF07] G Narayanaswamy, Pavan Balaji, and Wu Chun Feng. An analysis of 10-Gigabit Ethernet protocol stacks in multicore environments. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*, pages 109–116, 2007.

[NCTP07] Ranjit Noronha, Lei Chai, Thomas Talpey, and Dhabaleswar K. Panda. Designing NFS with RDMA for security, performance and scalability. In *Proceedings of the 2007 International Conference on Parallel Processing*, page 49, 2007.

[NGSH05] Sai B. Narasimhamurthy, Prabhanjan C. Gurumohan, Shesha Sreenivasamurthy, and Joseph Y. Hui. Quanta data storage: An information processing and transportation architecture for storage area networks. *Selected Areas in Communications, IEEE Journal on*, 23:2032–2040, 2005.

[NMF10] Fredy Neeser, Bernard Metzler, and Philip W. Frey. SoftRDMA: Implementing iWARP over TCP kernel sockets. *IBM Journal of Research and Development. Special Issue on Network-Optimized Computing*, 54:5:1–16, 2010.

[NSL+08] Sundeep Narravula, Hari Subramoni, Ping Lai, Ranjit Noronha, and Dhabaleswar K. Panda. Performance of HPC middleware over InfiniBand WAN. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 304–311, 2008.

[NTSP02] Jarek Nieplocha, Vinod Tipparaju, Amina Saify, and Dhabaleswar K. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Workshop on Communication Architecture for ClustersProceedings*, 2002.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The j-machine multicomputer: An architectural evaluation. *ACM SIGARCH Computer Architecture News*, 21:224–235, 1993.

[NZ02] Thinh Nguyen and Avideh Zakhor. Distributed video streaming over Internet. In *Proceedings of the ACM/SPIE Conference on Multimedia Computing and Networking*, pages 186–195, 2002.

[ofe] OpenFabrics Enterprise Distribution (OFED). `http://www.openfabrics.org/`.

[OHH09] Li Ou, Xubin He, and Jizhong Han. An efficient design for fast memory registration in RDMA. *Journal of Network and Computer Applications*, 32:642–651, 2009.

[opr] OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net`.

[Pak08] Scott Pakin. Receiver-initiated message passing over RDMA networks. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2008.

[PF01] Ian A. Pratt and Keir Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. In *Proceedings of the 20th INFOCOM Annual Join Conference of the IEEE Computer and Communication Societies*, pages 67–76, 2001.

[PGHA00] Thomas Plagemann, Vera Goebel, Pal Halvorsen, and Otto Anshus. Operating system support for multimedia systems. *The Computer Communications Journal*, 23:267–289, 2000.

[PMB09] Stavros Passas, Kostas Magoutis, and Angelos Bilas. Towards 100 Gbit/s Ethernet: Multicore-based parallel communication protocol design. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 214–224, 2009.

[Poo04] Martin Pool. distcc, a fast free distributed compiler. In *Proceedings of linux.conf.au*, 2004.

[Pos81] J. Postel. Transmission Control Protocol, 1981.

[RA07] Mohammad J. Rashti and Ahmad Afsahi. 10-Gigabit iWARP Ethernet: Comparative performance analysis with InfiniBand and Myrinet-10G. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[RAC97] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 257–274, 1997.

[ram] Ramsan-5000. `http://www.superssd.com/products/ramsan-5000/`.

[RB03] Allyn Romanow and Stephen Bailey. An overview of RDMA over IP. In *Proceedings of the 1st International Workshop on Protocols for Fast Long-Distance Networks*, 2003.

[Rec03] Renato J. Recio. Server I/O networks past, present, and future. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 163–178, 2003.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 325–336, 1994.

[RMC+07] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification, 2007.

[RMI+04] Greg J. Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *IEEE Computer*, pages 48–58, 2004.

[RMTB05] A. Romanow, J. Mogul, T. Talpey, and S. Bailey. Remote Direct Memory Access (RDMA) over IP Problem Statement, 2005.

[Sax] Vik Saxena. Bandwidth drivers for 100 g Ethernet. `http://www.ieee802.org/3/hssg/public/jan07/Saxena_01_0107.pdf`.

[SBB+07] Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on InfiniBand: Efficient network buffer utilization at scale. In *Proceedings of the 14th European PVM/MPI Users' Group Conference*, 2007.

[SBM+05] Sayatan Sur, Uday K. Bondhugula, Amith R. Mamidala, Hyun-Wook Jin, and Dhabaleswar K. Panda. High performance RDMA based all-to-all broadcast for InfiniBand clusters. In *Proceedings of International Conference on High Performance Computing*, 2005.

[SCFJ03] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications, 2003.

[SCR+03] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) Version 4 Protocol, 2003.

[SFLG00] Klaus Stuhlmüller, Niko Färber, Michael Link, and Bernd Girod. Analysis of video transmission over lossy channels. *IEEE Journal on Selected Areas in Communications*, 18:1012–1032, 2000.

[SMS+04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI), 2004.

[Sny90] Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn European UNIX Users Group Conference*, pages 241–248, 1990.

[SP00] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. *ACM SIGCOMM Computer Communication Review*, 30:309–319, 2000.

[SPRC07] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct Data Placement over Reliable Transports, 2007.

[ST93] Jonathan M. Smith and C. Brendan S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7:44–52, 1993.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Professional, 1994.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol, 2000.

[Tan02] Andrew S. Tanenbaum. *Computer Networks (4th Edition)*. Prentice Hall, 2002.

[TG03] Rajeev Thakur and William Gropp. Improving the performance of collective operations in mpich. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI User's Group Meeting*, pages 257–267, 2003.

[The] The Interconnect Software Consortium. Interconnect Transport API (IT-API) Version 2.1. `https://www.opengroup.org/icsc/uploads/40/9143/IT-API-V2.1.pdf`.

[TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1:554–565, 1993.

[TOHI98] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 308–314, 1998.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed

computing. In *Proceedings of the 15th ACM symposium on Operating Systems Principles*, pages 40–53, 1995.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[VG84] Patrick Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9:133–161, 1984.

[VI96] Subramaniyam R. Viswanathan and Tomasz Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia Systems*, 4:197–208, 1996.

[vid] VLC media player. `http://www.videolan.org`.

[VM03] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63:853–865, 2003.

[WC00] Matt Welsh and David E. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency - Practice and Experience*, 12:519–538, 2000.

[WHZ+01] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Ya qin Zhang, and Jon M. Peha. Streaming video over the Internet: Approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*, 11:282–300, 2001.

[wir] Wireshark Network Protocol Analyzer. `http://www.wireshark.org`.

[WM87] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5:97–120, 1987.

[WSBL03] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13:560–576, 2003.

[WSBM06] Tim S. Woodall, Galen M. Shipman, George Bosilca, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings of the 13th European PVM/MPI Users Group Meeting*, pages 76–85, 2006.

[WW05] Pete Wyckoff and Jiesheng Wu. Memory registration caching correctness. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, pages 1008–1015, 2005.

[ZHH+07] Hongwei Zhang, Wan Huang, Jizhong Hanand, Jin Heand, and Lisheng Zhang. A performance study of Java communication stacks over InfiniBand and Giga-bit Ethernet. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, pages 602–607, 2007.

Curriculum Vitae

Personal Data

Name	Philip Werner Frey
Date of birth	January 4, 1981
Citizenship	Swiss

Philip Frey is a member of the System Software group in the Systems Department at IBM Research Zurich. He received an M.S. degree in Computer Science from the Swiss Federal Institute of Technology (ETH) Zurich in 2006. He subsequently joined IBM, where he works on host system enablement for Remote Direct Memory Access (RDMA) in software. Related to the PhD studies he is pursuing at the Systems Group at ETH Zurich, he works on experimental evaluations for assessing the RDMA benefits for novel- and legacy distributed applications. He is author or co-author of various papers and inventor or co-inventor of three filed patent applications.

Education and Degrees

2006 – 2010	PhD student Swiss Federal Institute of Technology (ETH) PreDoc IBM Research Zurich
2001 – 2006	Master of Science ETH in Computer Science Swiss Federal Institute of Technology (ETH)
1994 – 2001	Matura Type B (English and Latin) Kantonsschule Raemibuehl, Zurich

Journals, Conferences and Workshops

Fredy Neeser, Bernard Metzler, and Philip W. Frey. SoftRDMA: Implementing iWARP over TCP kernel sockets. *IBM Journal of Research and Development. Special Issue on Network-Optimized Computing*, 54:5:1–16, 2010.

Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. A spinning join that does not get dizzy. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, 2010.

Philip W. Frey, Bernard Metzler, and Fredy Neeser. Enabling applications for RDMA: Distributed compilation revisited. *Technical report, IBM Research RZ3764*, January 26, 2010.

Bernard Metzler, Philip W. Frey and Animesh Trivedi. Softiwarp - Project Update. In *OpenFabrics Alliance Sonoma Workshop*, 2010.

Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the 5th International Workshop on Data Management on New Hardware*, pages 27–33, 2009.

Philip W. Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, 2009 (Best-Paper Award).

Philip W. Frey, Andreas Hasler, Bernard Metzler, and Gustavo Alonso. Server-efficient high-definition media dissemination. In *Proceedings of the 19th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 49–54, 2009.

Bernard Metzler, Fredy Neeser, and Philip W. Frey. A software iWARP driver for OpenFabrics. In *OpenFabrics Alliance Sonoma Workshop*, 2009.