RZ 3764 (# 99774) 01/26/2010
Computer Science 21 pages

Research Report

Enabling Applications for RDMA: Distributed Compilation Revisited

Philip W. Frey,[1,2] Bernard Metzler,[1] and Fredy Neeser[1]

[1]IBM Research – Zurich
8803 Rüschlikon
Switzerland

[2]Swiss Federal Institute of Technology, Zurich (ETHZ)
8092 Zurich
Switzerland

Email: {phf,bmt,nfd}@zurich.ibm.com

Enabling Applications for RDMA: Distributed Compilation Revisited

Philip W. Frey[1,2], Bernard Metzler[1], and Fredy Neeser[1]

[1] IBM Research GmbH, Rueschlikon, Switzerland

{phf,bmt,nfd}@zurich.ibm.com

[2] ETH Zuerich, Switzerland

Abstract. Remote Direct Memory Access (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 zero-copy and direct data-placement techniques offered by RDMA. With the advent of *iWARP*, the RDMA technology is now available to the ubiquitous TCP/IP infrastructure and, thus, becomes interesting and relevant for legacy applications. In this paper, we present the conversion of the distributed C/C++ compiler *distcc* from sockets to the RDMA interface, and investigate the implications of using asynchronous and one-sided versus two-sided communication, RDMA-style buffer management, and notification mechanisms. We have designed a distcc extension that takes full advantage of asynchronous one-sided RDMA operations. By introducing a connection manager, we improve the communication efficiency and increase the scalability of distcc. Our tests with the RDMA-enabled distcc use a software-only iWARP implementation called *SoftRDMA*, which offers RDMA communication using conventional Ethernet adapters. The ideas and approach presented in this paper can be generalized to a wide range of applications.

Key words: RDMA, iWARP, zero copy, explicit buffer management, asynchronous communication, one-sided operations, SoftRDMA, tmpfs, distributed compilation, distcc

1 Introduction

Remote Direct Memory Access (RDMA) implements *direct data placement* which enables transferring data from the memory of one host directly into the memory of another without involving the CPUs in the transfer — extending the well-known local DMA model with network capabilities. In order to fully bypass the CPU 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. An RDMA-enabled network interface card

(RNIC) provides a hardware accelerated RDMA-ed network stack instance in addition to the conventional network stack in the operating system kernel. 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 new interconnect technologies available today (e.g., 10 Gb/s Ethernet [1], InfiniBand [2]) have caught up with the memory bandwidth available in modern architectures. While initially based on proprietary network technologies, the advent of TCP/IP-based RDMA, called iWARP, and the standardization of RDMA APIs (IT-API [3], RNICPI [4]) make RDMA suitable for legacy applications.

In this paper we explore how to convert legacy distributed applications to the RDMA communication model and exemplify the process with an iWARP-based distributed compiler. We describe the required application changes such as the adoption of asynchronous APIs and explicit buffer management to make efficient use of the RDMA semantics. Even though we advocate the use of hardware RDMA NICs, we did not have any available for our tests and therefore ran the experimental evaluation in software using conventional Ethernet adapters. To estimate the benefit of future RNICs, we used a dual core machine where one core was dedicated to network processing while the other was running the application. Since the software based RDMA implementation follows the standardized RDMA API, no porting effort is necessary to run the distributed compiler on RNICs later on.

The paper is organized as follows. Section 2 gives background information on RDMA by describing its basic semantics as well as some of the required OS infrastructure including the software implementation used for our experiments. In Section 3, we illustrate the process of enabling an application for RDMA by extending the distributed C/C++ compiler *distcc*. After an overview of the TCP-based distcc, we describe our RDMA extension called *rdistcc*. We then provide a discussion of typical RDMA connection management issues, advocate the use of a RAM disk for memory-to-memory file transfers in combination with RDMA and discuss our buffer management options. This is followed by the details of our rdistcc iWARP communication protocol. Section 4 presents the experimental evaluation of rdistcc for a small cluster. Section 5 lists related work and Section 6 presents our conclusions and an outlook.

2 RDMA Background

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). 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 [5]. This dilemma is aggravated by modern storage solutions, such as storage area networks (e.g., based on FibreChannel or iSCSI [6]) and file-based network-attached storage (e.g., NFS [7]). With such systems, high-bandwidth access to remote storage results in CPU- and memory-intensive network I/O. These distributed storage applications (e.g., iSER [8]) have been driving the development of modern interconnect technologies and the RDMA model [9].

2.1 RDMA Semantics

One-sided and Two-sided Operations: The transport-agnostic virtual interface architecture (VIA [10]) and later InfiniBand and iWARP have specified the RDMA architecture and communication semantics. 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 (see Figure 1).

Explicit Buffer Management: RDMA operations require an *explicit communication buffer management*: application buffers that are to be used as source or target of a remote DMA operation must be registered with the OS and the RNIC as *RDMA memory regions* (MR) first. Through memory pinning, this registration ensures that the buffer pages are resident in physical memory and that the correspondence between the buffer’s virtual address interval and the underlying page frames stays fixed. Only now the RNIC can access the communication buffers in physical memory without requiring any OS intervention (zero-copy and kernel bypassing).

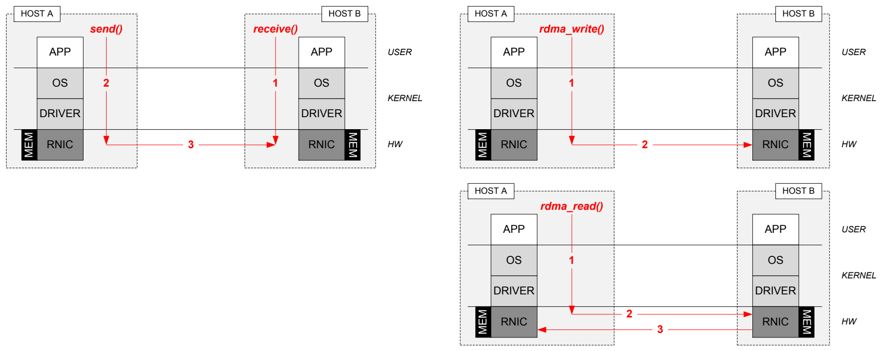

Fig. 1: Two-sided versus one-sided operations

During RDMA data transfers each MR is identified through a unique ID, called *STag*. Figure 2 depicts the difference between RDMA style data placement with explicit buffers and TCP with the flexible but more expensive socket buffer.

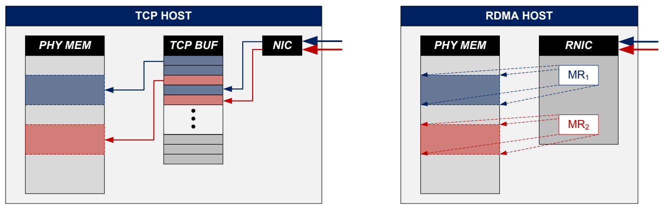

Fig. 2: TCP socket buffer versus RDMA memory regions

Asynchronous API: All operations (Send, Receive, RDMA Read, RDMA Write) are initiated and completed *asynchronously*. They are described as *work requests* (WR) which are posted to local work queues. The queues are asynchronously processed by the RDMA stack (e.g., the RNIC). Each communication endpoint has two of these work queues: a *send queue* (SQ) and a *receive queue* (RQ). Together they are referred to as *queue pair* (QP). The QP is associated with a *completion queue* (CQ) through which the RDMA stack reports back on the completion status of processed work requests. In order to get the completions, the application thread may either wait for an expected work completion event on the event channel (blocking) or poll the CQ at a certain interval (non-blocking). The IT-API [3] is an example for a high-level programming abstraction of this. Figure 3 illustrates the asynchronous semantic for the send and receive operations.

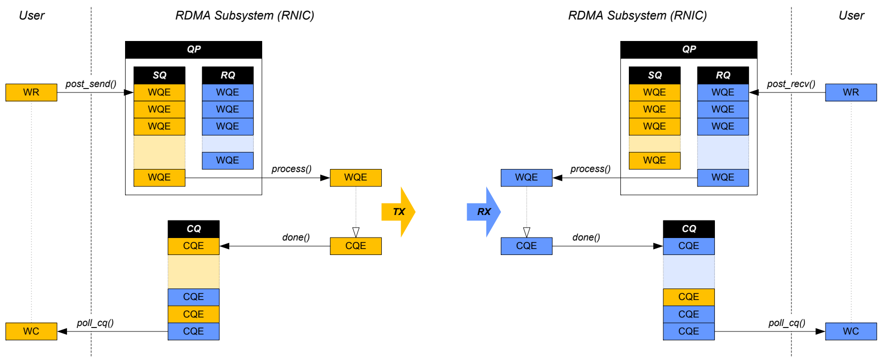

Fig. 3: Asynchronous send/receive operations

2.2 RDMA Transport: iWARP

The Internet Engineering Task Force (IETF) has specified a set of companion protocols for RDMA over TCP/IP: *MPA* [11], *DDP* [12] and *RDMAP* [13]. The topmost protocol is RDMAP which 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. MPA finally serves as an auxiliary layer that transports the discrete DDP packets over a TCP stream. This so called iWARP [3] stack enables RDMA connectivity over low-cost, Ethernet-based network infrastructures.

As 10 Gigabit/s Ethernet (10GbE) is available today, iWARP competes with highly specialized network technologies. Think of iWARP as Ethernet extended with the features of InfiniBand. Based on the economies of scale, multi-gigabit Ethernet together with iWARP have the potential to become the unifying interconnect in the data center, relegating proprietary technologies such as InfiniBand, Myrinet, and Quadrics to niche markets [1]. Another very promising property of 10GbE is its extended operating distance of 40 km, which allows a complete rethinking of physical data-center layout and its integration with the Internet.

While some vendors already offer complete iWARP hardware solutions, OS support for iWARP is appearing more slowly. The interface between the OS and the RDMA subsystem is rather complex and was originally described only semantically in terms of the *RDMAC Verbs* [14]. OS vendors have jointly standardized this interface as RNICPI [4]. The industry-driven OpenFabrics [15] effort implements its own version of a transport-independent RDMA subsystem for Linux (both iWARP and InfiniBand are supported).

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

2.3 iWARP Software Implementation

We have based all our RDMA experiments on the IBM Research *SoftRDMA* iWARP stack due to the lack of RDMA hardware (RNIC). While iWARP stack processing is typically offloaded to dedicated hardware, a pure or partial iWARP software implementation can still be very useful for creating low-cost, RDMA/Ethernet-based experimental compute clusters. The main advantage of SoftRDMA for us is that it allows the use of RDMA and the migration of applications to RDMA on systems without RNIC hardware — a conventional Ethernet NIC is sufficient.

When RNICs are available, a mixed setup consisting of RNICs together with SoftRDMA is particularly attractive for asymmetric networking applications, where one side has no RNIC but plenty of compute cycles to spend while the other side depends on the use of an RNIC for efficient data transfers at aggregate throughputs that are high in relation to the CPU instruction rate (e.g., 100 MB/s on a 2 GHz Pentium core).

3 Use Case: Enabling a Socket-Based Distributed Compiler for RDMA

Given the growing size of software packages that are distributed in source code, short compilation times are generally desired. One approach for 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* (e.g., *distcc* [16]), is to use idle CPUs from remote hosts.

We have chosen distcc on Linux as an example to show how socket-based legacy applications can profit from RDMA. Distcc is designed for static local-area network (LAN) environments where there is not a lot of churn. As we will see this is important when using RDMA since its initial connection setup is quite costly and complex. The original distcc protocol can be very nicely transformed from a peer-to-peer like communication model to a client-driven one which relieves the server from some of his work. From an application point of view, our new protocol simplifies the communication. We show with our example how even file-based applications can profit from RDMA although RDMA has been designed for transferring chunks of memory. We do this without changing the way in which distcc (`gcc` and `cpp` in particular) operates on the data. Often it is claimed that the use of RDMA is only justified if the data to be transferred is large and if 10 Gb Ethernet or faster is used. We demonstrate with software-based 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.

3.1 DISTCC Overview

Distcc is a well established, easy-to-use gcc wrapper that enables remote compilation. It currently supports TCP and SSH connections only. For the remainder of this discussion, we refer to the host that initiates and offloads compile jobs as *master* and a host offering its CPU(s) as *slave*. A master is using the CPU resources of one or several slaves for each compilation process.

The distcc setup comprises a daemon running on each slave and a wrapped `gcc` on the master. The ability of the master to specify a set of slaves enables an implicit failover mechanism: if a file cannot be compiled on a slave in due time, the file is processed locally, and the next file will be sent to a different slave. The master does the preprocessing as well as the final linking locally and offloads the source code to object translation to the slaves. As the preprocessing and linking steps are performed locally, no header or library dependencies exist on the slaves nor are any changes to the Makefile necessary.

The communication protocol defined by distcc consists of four steps: first, a TCP connection between the master and one of its slaves is established. After that the master sends the compilation instructions for the preprocessed source code followed by a file containing that code. 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. At the end of each compilation cycle, the TCP connection is closed again (see Figure 4).

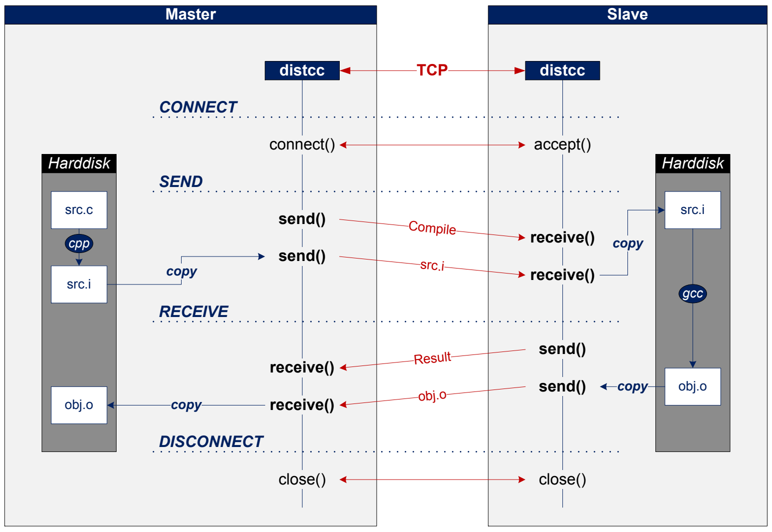

Fig. 4: Distcc network protocol

3.2 How Can a Distributed Compiler Profit From RDMA?

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. To leverage the total available CPU power and reach a high level of scalability, it is important to keep this communication overhead small.

The CPU cycles available for the actual compilation can be increased both by moving compile jobs to remote hosts across the network as well as by taking unnecessary network processing load off the CPU(s). As described in Section 2, RDMA relieves the host's CPUs of executing network-related code and permits overlapping computation with communication through its asynchronous interfaces. In addition to that, RDMA offers one-sided operations which enable client-driven protocols and eliminates the in-host copy overhead depicted in Figure 4 thanks to its zero-copy technique.

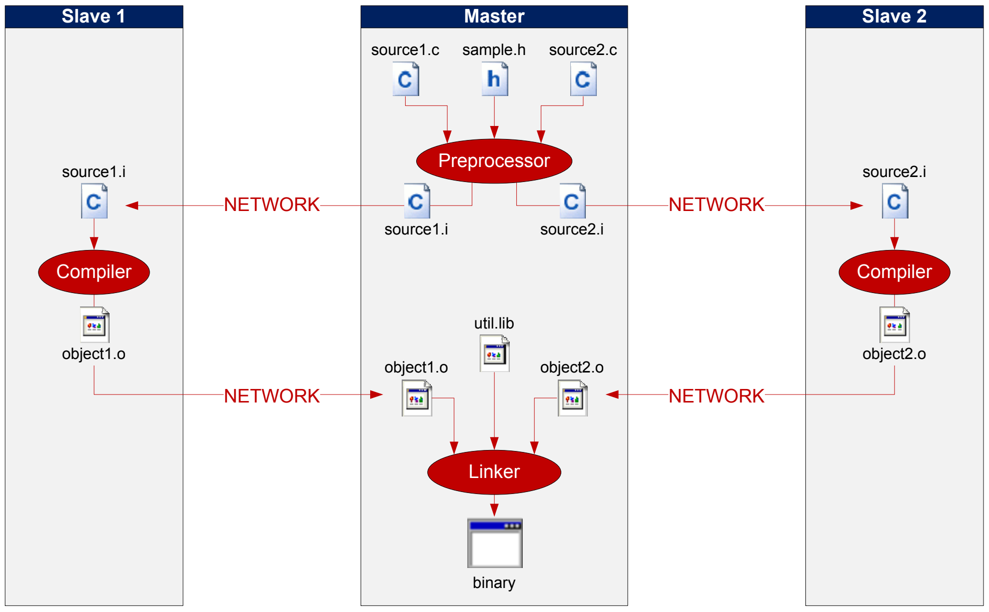

Fig. 5: Example for a distributed compilation

Figure 5 depicts a simplified example of a distributed compilation where the master node is offloading the translation of two source files into object files to some slaves. 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 in our rdistcc example. As we will see, this replacement is not trivial due to the semantic discrepancies between sockets and RDMA.

For rdistcc, the networking share of the overall CPU load strongly depends on the host's role. The master is typically very busy distributing compile jobs and collecting the results.

The slaves on the other hand, just wait for new input from the master whenever they are not compiling. 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 therefore be moved to the slaves. We address this problem by the use of one-sided RDMA operations (RDMA Read and RDMA Write). They enable a slave-driven communication protocol which minimizes network I/O processing at the master's CPU and results in improved scalability relative to distcc.

The master can reduce its CPU load using one-sided operations only if an RNIC (hardware acceleration) handles all that RDMA traffic. On the less-loaded slave side, an easily deployable, software-based iWARP solution (such as SoftRDMA) with a conventional Ethernet NIC is sufficient. With SoftRDMA-enabled slaves, and a hardware accelerated master, the compute cluster can easily be extended by many slaves while still remaining highly dynamic and flexible. Using this approach, idle CPUs (e.g., workstations of a company) can be utilized for compilation.

3.3 Applying RDMA Without Changing the Current Application Semantics

Although compilation involves files that usually reside on disk, rdistcc can still benefit from direct *memory* access because we store the files on a RAM disk mounted as *tmpfs* [17]. By doing this, slow and therefore expensive transfers between disk and memory are eliminated. Moreover, local copies within main memory can be avoided when using a mapping which is shared. With that we get two views on each file: for the compiler it looks like an ordinary file residing on the file system but for the RDMA subsystem it is just a block of main memory. We can now associate a memory region (MR) with the user address interval that corresponds to that block of memory to make it accessible for the iWARP transport. This solution has the advantage that no `gcc` modification is needed and that the remote hosts have remote random access to the files.

Before we can map a file as described above, we have to get it from the persistent storage onto the tmpfs mount. On the master side of the distributed compiler, the source files are typically stored on a hard disk. Remember that distcc needs to preprocess the source files locally before shipping them for remote compilation: We exploit this step to get files into main memory without creating any overhead. To preprocess a source file and prepare the result for later RDMA transfer to a slave, the `cpp` preprocessor invoked by rdistcc reads the source file from the disk but writes the result (`*.i` file) to our tmpfs mount (memory) instead of back to disk. Next, we use `mmap()` to map the preprocessed source file into the application address space and register it as an RDMA memory region, denoted as SRC TX MR [4] in Figure 6. The preprocessed source file is now ready for RDMA transport.

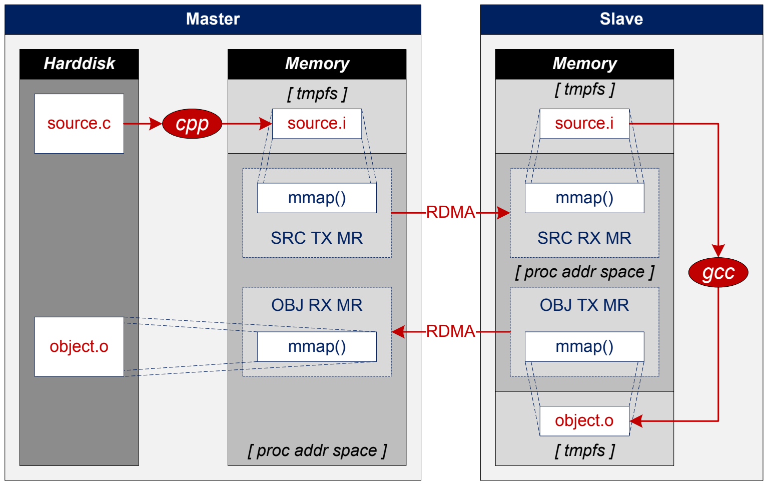

Fig. 6: Data path for remote compilation

On the slaves, we keep both the source and object files in main memory to reduce file access times for the compiler. By using the same mmap-approach as on the master, we eliminate the copy overhead for passing files between the rdistcc process (has MR view on data) and the compiler (gcc operates on ordinary files). The object files resulting from compilation are written back to local memory. Figure 6 depicts the entire data flow of rdistcc from source code to target object files.

We argue that keeping everything in main memory on the slaves does not impose any restrictions because preprocessed source files are small in most cases (< 1 MB). As it does not make sense for a slave to offer more compile jobs to the master as it is able to process in parallel (bound by the number of available CPUs), we do not consume a lot of memory altogether. Let us assume that our average slave is a 4-way CPU machine with 1GB of RAM, offering 4 compile jobs and that our average preprocessed file is 2 MB large. Even with the memory consumed by the compiled object file, say another 2 MB, we need no more than 16 MB under full load on average which is a mere 1.5% of the total available memory. When using two more compile jobs than there are CPUs (which is suggested by the distcc author) we are consuming 2.3%. Compared with the memory used by gcc for the compilation, the data stored in memory by rdistcc is negligible.

For projects with much larger source files or on machines with many more CPUs and very little memory, it makes sense for the slaves to put the data on harddisk drives. Our implemen-

[4] source transmit memory region

tation also supports RDMA transfers to and from harddisk 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.

3.4 RDISTCC Buffer Management

As Figure 6 shows, we need two separate memory regions on each host: one for the preprocessed source and another for the object file. The creation of these memory regions induces a delay because the memory has to be pinned and registered with the RDMA subsystem as explained before. Table 1 shows the delays measured on our SoftRDMA test machines for various sizes. The machines are equipped with an Intel®Pentium 4 CPU with 1.80 GHz, 2 MB Cache and 3 GB of RAM.

MR SIZE	**open_mmap** [μs]	**create_mmap** [μs]	**memcpy** [μs]
16 KB	33	72	31
64 KB	56	151	132
256 KB	151	412	574
512 KB	292	752	1159
1024 KB	537	1453	2353
2048 KB	1039	2782	4786

Table 1: MR registration delays

The first column (**open_mmap**) indicates the time needed for opening an existing file, mapping it into the process address space and registering it as a new memory region. For the second column (**create_mmap**) we first created a new empty file and stretched it to the desired size before mapping it and registering it as a MR. We have added the **memcpy** column as reference (no mapping or MR registering was done there).

We have the options of either create a memory region once at the beginning and reuse it for each file or we can deregister it when a transmission is done and register a new one for the next file. We are facing a tradeoff between minimizing the overall delay (reuse) and minimizing the memory footprint (reregister). If we are reusing the memory region, we need to make sure that it is large enough to hold the largest file we need to transmit. This is a great waste of memory in projects where we have a few really large files and a lot of small files: keep in mind that the memory of the MR is pinned as long as it is registered. This is especially critical for the master since it needs two MRs per slave. Furthermore, MR reuse makes only sense when the data does not have to be copied from some intermediate buffer into the target MR within the host. In our case this means that we would have to make sure that the cpp and gcc always write their output to the same physical memory address. With several simultaneous cpp and gcc processes, this is not feasible.

Considering this, we have decided to manage the MRs as follows: for the SRC TX MR [5] and the OBJ TX MR [6] (see Figure 6), we reregister the MRs for each file. The delay induced 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 (**open_mmap** column in Table 1). With TCP, the data would have to be copied from the user space memory into kernel socket buffers which is even for small files up to four times slower than our zero-copy approach (see **memcpy** column). The OBJ RX MR [7] is also reregistered in each iteration with the additional cost of allocating an empty target file (**create_mmap** column). If the master would reuse its 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 [8] we can register the MR once and reuse it - the most efficient way in terms of latency. As argued in the last section, this does not impose much memory dissipation on the slaves.

Since all the data are in memory and the necessary RDMA buffers (MRs) are in place, we can now transfer the payload between master and slave using RDMA. For each remote compilation we have two slave-driven RDMA operations: the transfer of the preprocessed source file to the slave and of the compiled object file back to the master.

Listing 1.1 demonstrates in simplified pseudo code how a buffer can be transferred using the IT-API for RDMA. It consists of the following steps:

- register a buffer with the RDMA subsystem [lines 1-3]
- post a work request to the QP [lines 5-9]
- fetch the work completion from the CQ [10-11]

The third step is omitted if the work request is configured not to generate a work completion.

Listing 1.1: RDMA data exchange

```
1 char *buf = mmap(LEN);                               // create buffer
2 it_lmr_handle_t lmr_tx;
3 it_lmr_create(buf, LEN, &lmr_tx);                    // register buffer
4 /* fill the buffer... */
```

[5] source transmit memory region
[6] object transmit memory region
[7] object receive memory region
[8] source receive memory region

```
it_lmr_triplet_t segment;                              // create segment on buffer
segment.addr.abs = buf;
segment.length   = LEN;
segment.lmr      = lmr_tx;
it_post_rdma_write(segment, LEN, dst_addr, dst_stag);    // post work request
it_event_t event_sq;                                   // wait for work completion
it_evd_wait(evd_sq_hdl, TIMEOUT, &event_sq);
```

3.5 RDISTCC Connection Management

Establishing an iWARP connection is more expensive than establishing a TCP connection. As explained in Section 2.2, iWARP adds three more protocols on top of the TCP stack. We start with a TCP connection which is then upgraded to an iWARP connection by performing an MPA handshake which consists of a 2-way message exchange where the connection parameters are negotiated. The setup also requires more programming objects at each end which have to be instantiated first. We experienced an average TCP connection setup delay (including allocation of the socket and 3-way handshake) of 136 μs on a connection with a network delay of 41 μs. Establishing an RDMA connection induces an average delay which is almost ten times as big (1152 μs including object creation as well as TCP and MPA handshakes).

A conventional distcc master establishes, for each source file which is to be compiled remotely, a new TCP connection to one of its slaves and closes it again after having received the result. That does not scale very well for iWARP, as it is unnecessary and rather expensive. We have thus introduced a *connection manager* (CM) that keeps RDMA connections open while the rdistcc master is submitting compile jobs. Our CM minimizes connection management overhead and improves overall scalability. Furthermore, it takes care of RDMA-specific transport mechanisms (posting work requests, managing remote buffer information etc.) which simplifies the rdistcc application logic.

The CM is implemented as a thread pool with a fixed number of threads per slave (Figure 7). The number of slaves involved as well as the maximum number of remote compile jobs (slave threads) is set before starting rdistcc. This fixed thread allocation is very efficient and light weight at run time and matches the static setup for which distcc is designed. One thread cycle encompasses exactly one complete protocol cycle. This simplifies connection handling and leads to a reliable and early error detection.

In our rdistcc extension, the CM acts as a persistent RDMA proxy for the rdistcc processes (Figure 8). Every master process scheduled for remote compilation 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 to an absolute minimum, an rdistcc process merely

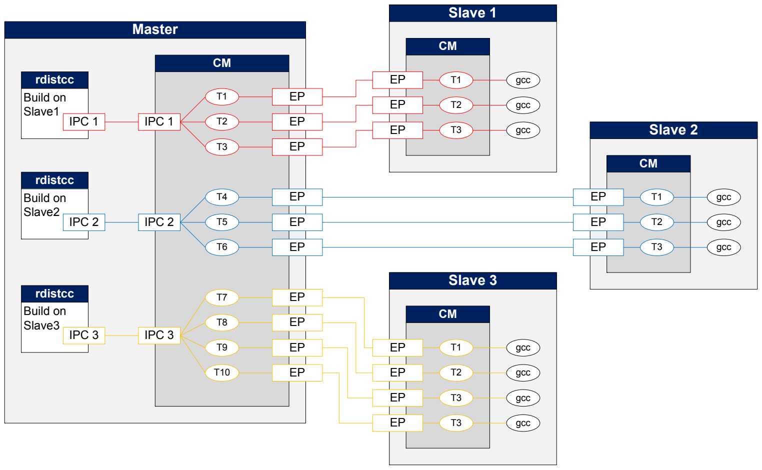

Fig. 7: Processes, threads and their associations

provides the CM with path information for locating the preprocessed files rather than sending the whole files through the IPC channel. The CM then takes care of the actual RDMA data transfer.

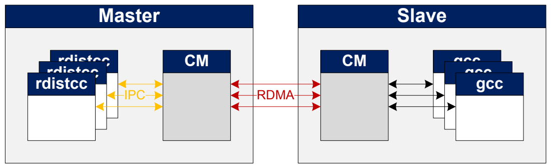

Fig. 8: Connection manager

3.6 RDISTCC Protocol for iWARP

When designing an iWARP-based application, the buffer management is key to its success. Sometimes it is not possible to manage the buffers in a way in which the application can profit from the RDMA semantics. This is the case for applications with short-lived connections where the buffers exist only for a short time and have to be advertised very often (e.g., for each new connection). If these buffers are also small, the management overhead will soon outweigh the zero-copy benefit.

Another important design point is the decision of how to utilize the one-side and two-sided operations. This can make a difference in protocol complexity as well as in load distribution between communicating peers. The advantage of one-sided communications is that only the application layer of one side has to actively participate in the data transfer. The drawbacks are that a buffer specification exchange is necessary beforehand and that the other side will not know when the data transfer is complete. Two-sided send/receive operations are typically used for exchanging small control messages whereas the one-sided operations are preferred for transferring large data sets.

We will now look at how it all fits together by transforming the TCP-based distcc protocol into an RDMA-based one. The protocol proposed in Figure 9 involves both two-sided (Send/Receive) and one-sided (RDMA Write/RDMA Read) operations and is partitioned into the following three phases.

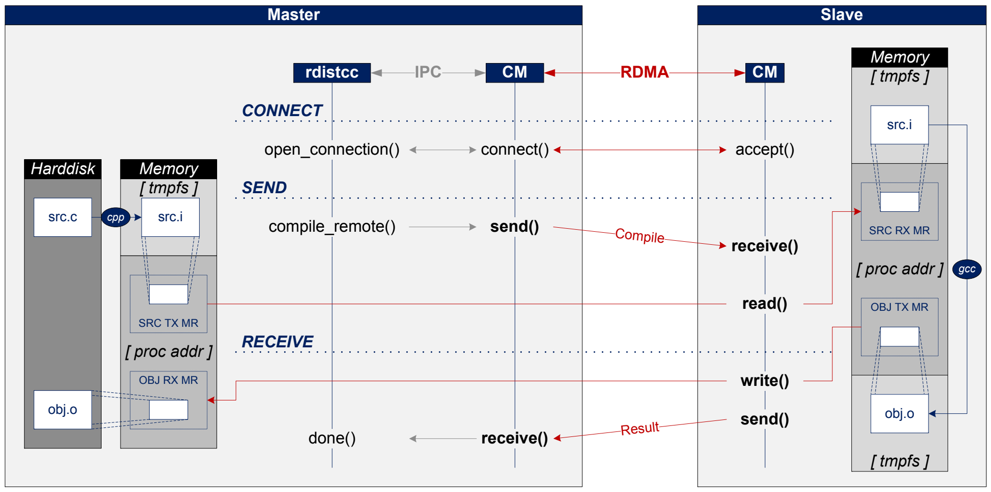

Fig. 9: iWARP based master-slave protocol

In Phase 1, the RDMA connection between the local and the remote CM is established if necessary. We do this only once for each slave and keep the connections open as long as they are needed (see Section 3.5).

Phase 2 is then executed for each remotely compiled file: the master starts by sending the compilation instructions and information about the preprocessed source file to the slave, along with the SRC STag and OBJ STag, which give the slave remote access to the SRC TX MR and OBJ RX MR respectively. Upon receiving such a request, the slave fetches the source file into its SRC RX MR using an RDMA Read and initiates local compilation. The master can perform other tasks in the meantime until notification from the slave.

After completing 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 to notify him of the completion. We denote this as phase 3. At the time the master CM receives this Send message, the object file is present in his buffer. The number of context switches involving the application and the CPU load on the master are kept to a minimum because the data transfer is slave driven and the master CM is only notified once the entire object file is locally available and ready for further processing.

As can be seen from Figure 9, the master involvement is very small — it merely needs to map the source file into a memory region 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 any overhead compared to the original one. First, the memory mapping and MR registration overhead is not bigger 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.

4 Experimental Evaluation

We conducted a small cluster experiment for performance measurements on our RDMA-enhanced rdistcc, compiling the Linux kernel 2.6.22 as reference. The cluster setup consisted of one master and 14 slaves connected via 1 Gb/s Ethernet. All machines were equipped with a 1.8 GHz Pentium 4 processor and 3 GB of RAM. To get a fair comparison between the TCP and RDMA solution, the compiler input- and output data were on tmpfs mounts in all cases. Like this we assure that our performance gain is not just because rdistcc keeps everything in main memory as opposed to standard distcc.

While future experiments might use an RNIC on the master and a software-based RDMA solution on the slaves for the reasons stated below, our current rdistcc tests are based entirely on SoftRDMA, because of the lack of RNIC availability at the time the tests were conducted. SoftRDMA 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 than on pure performance. It is not always necessary to equip a host with expensive RDMA hardware — in cases where a server faces a high aggregate throughput which is distributed to a large number of clients, the clients just need the RDMA semantics but not necessarily the hardware acceleration that the server needs.

Given that the current SoftRDMA implementation is based on TCP kernel sockets and thus can not achieve zero-copy, comparing distcc over TCP and rdistcc over SoftRDMA with fewer than 11 slaves shows that our protocol in combination with SoftRDMA induces very little overhead. Regarding scalability with the number of slaves, distcc over TCP scales only up to

7 slaves and rdistcc over SoftRDMA does not do much better which is what we expected due to the missing hardware acceleration.

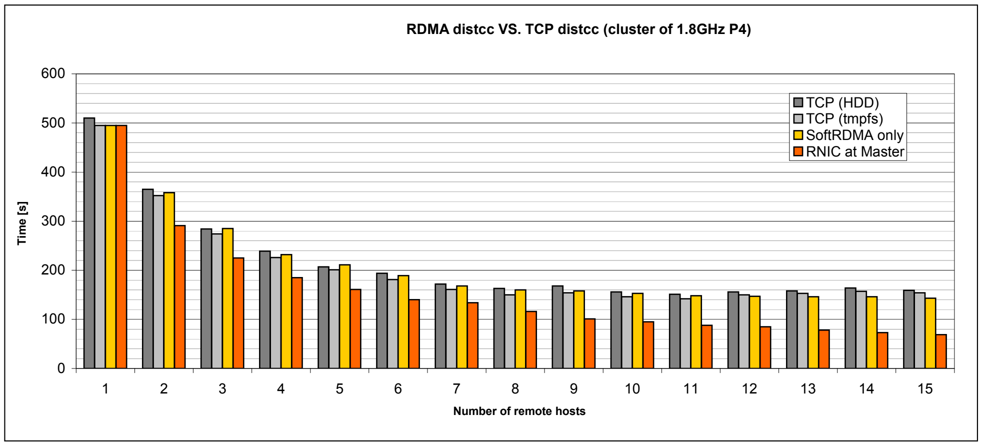

Fig. 10: Test results

Figure 10 shows the measured compilation time for TCP distcc with the data on a harddisk drive as well as on a tmpfs mount, rdistcc with SoftRDMA on all nodes and the (estimated) compilation time for rdistcc with an RNIC at the master and SoftRDMA at the slaves (from left to right on the chart).

To assess the need for an RNIC on the slaves and on the master, we measured the CPU usage distribution with distcc over TCP (Tables 3 and 2). We have found that the master spends a large amount of CPU cycles on local data copying and network processing, since it communicates with all the slaves in parallel and establishes new TCP connections for each file it transmits. We therefore argue that it makes sense to equip the master with an RNIC so that the hardware can handle the data transfer and eliminate the copy overhead. The CPU cycles saved from network processing can now be assigned to management tasks such as selecting the slaves, checking the results etc. which are also quite CPU intensive and cannot be hardware accelerated.

At the slaves on the other hand, we found that 80% of the CPU load was caused by `gcc` processes. The bottleneck here is clearly not the communication stack and expensive RNICs are therefore not required when transforming the application to RDMA. The high idle value further indicates that the slaves have enough processing power to do RDMA in software. It also shows that the TCP distcc master is not able to keep them all busy which is what sets the limits in terms of scalability. The situation is even worse when SSH is used due to the encryption overhead and the more expensive connection setup.

operation	**CPU time** [%]
cpp and ld	12.35
management tasks	28.00
idle	0.50
data transfer	59.15

Table 2: Load distribution on master

operation	**CPU time** [%]
compiler (gcc)	79.75
idle	18.85
data transfer	2.10

Table 3: Load distribution on slaves

With today's trend towards multicore and manycore machines, one (or several) cores can be spared for RDMA processing [18]: to estimate the compilation time for rdistcc with an RNIC at the master, we replaced the master machine with an off the shelf dual-core machine of similar power. One core was dedicated to RDMA processing in software while the other was running the rdistcc master application. The slaves remained unchanged. While this setup is similar to running rdistcc on a single-core machine that is equipped with an RNIC, running the application and SoftRDMA on different cores in fact leads to a pessimistic estimate 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, and the system can be expected to scale further when more slaves are added. We also observe that the speedup achieved with RDMA is not due to fact that the data is residing on memory instead of on disk.

5 Related Work

Offloading network processing and enabling socket based applications for RDMA has been demonstrated in a different application domain by extending the Apache webserver with RDMA capabilities [19] — the extension has been implemented as an Apache module using the Ohio software iWARP stack. The idea is similar to that of rdistcc: one server deals with a lot of clients and needs all its CPU cycles to manage some content. Enabling the HTTP protocol for RDMA requires a small addition to the header. The performance evaluation indicates that Apache can only benefit from the extension under high load and when the clients are requesting large files ($>$ 1MB). This is because a new RDMA connection has to be established for each visiting client and possibly for each request in case the connection cannot be kept alive. As we have shown, RDMA connection management is quite expensive. A typical communication scenario of a webserver is that of many different clients requesting some (typically small) files and leaving afterwards which means that the RDMA connections are short lived and a considerable effort is put into connection setup and tear down compared to the data transfer. The buffer management challenges they face are that reusing a per-client buffer results in expensive memory copy operations and does not scale on the server side. Dynamic registration on the other hand adds the cost for the registration and later deregistration for each data transfer. The Apache webserver example is designed to be used over the Internet and therefore strongly depends on TCP/IP based RDMA. It presents a use case of RDMA over wide area networks.

The Ohio Supercomputer Center has presented an alternative software implementation of the iWARP protocol stack [20]. They provide a kernel-space and a user-space version together with a set of wrapper functions that follows the OpenFabrics Verbs. The Ohio stack suggests its own, non-standardized API which makes applications that are built on it less portable. They claim that their implementation is wire compatible with RDMA hardware (Ammasso RNIC) and that they do not require kernel modifications. Concerning the performance, CRC calculation in software results in a latency increase by a factor of two. The maximum throughput on 1Gb Ethernet of about 920 Mb/s can only be reached with packets of size 16 KB or larger and with CRC disabled. SoftRDMA on the other hand achieves the same throughput already with 2 KB packets (or 4 KB if CRC is enabled). The CPU load induced by the Ohio stack is considerably higher than with SoftRDMA as well, especially at the receiver side.

6 Conclusions and Outlook

When carefully designed, RDMA-based protocols can relieve the CPU from the heavy network I/O processing needed to saturate today's high-speed interconnects and save the cycles for other tasks such as data processing.

The technique we proposed of creating a shared file mapping into the application address space and thereafter associating it with a memory region for remote DMA is by no means limited to the compiler extension. It could for example also be applied for accelerating the file transfer protocol (FTP) or similar applications. 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 in applications like distributed, file-based databases where a 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. With this solution we have demonstrated that iWARP also qualifies for low throughput environments (e.g. the Internet). The main advantages of using `mmap` in combination with a MR instead of some upper layer protocol (e.g. `sendfile` extension) for enabling true random access are its simplicity and direct hardware support. In the case of the database mentioned before, this is needed when a lot of clients access the database file concurrently. Some further research on update propagation and synchronization will be necessary if write access to the database file is granted to the clients for example. Figure 11 illustrates the idea.

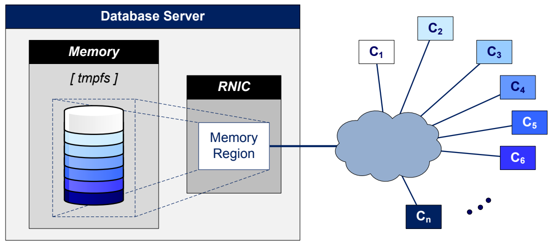

Fig. 11: Database mapping

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 shown in detail how distributed compilation can be accelerated with RDMA. Through the use of the one-sided RDMA operations, we 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 that much data.

We have argued that clusters using an RNIC at the busy nodes and SoftRDMA together with conventional Ethernet NICs at the other nodes are performance- and cost-effective for applications with different CPU load distributions — depending on the role of the node — for network- and application processing. Compared with plain TCP, the use of iWARP therefore allows busy core nodes with an RNIC to serve more edge nodes in parallel.

While, in our example, iWARP already provides a two-fold performance improvement over plain TCP using 1 Gb/s Ethernet, more dramatic improvements are expected from upcoming applications using 10 Gb/s Ethernet, particularly from applications requiring a high average data rate with comparatively limited processing needs per byte of data.

References

1. Intel Corporation. 2003. *10 Gigabit Ethernet Technology Overview.* http://www.intel.com/network/connectivity/resources/doc_library/white_papers/pro10gbe_lr_sa_wp.pdf.
2. InfiniBand Trade Association. 2004. *InfiniBand Architecture Specification Volume 1, Release 1.2.*
3. Interconnect Software Consortium, The Open Group. 2005. *Interconnect Transport API (IT-API), Version 2.1.*
4. Interconnect Software Consortium, The Open Group. 2005. *RNIC Programming Interface (RNICPI), Version 1.0.*
5. Clark D., Jacobson V., Romkey J., Salwen H. 1989. *An Analysis of TCP Processing Overhead.* IEEE Communications Magazine, June 1989 - Volume 27, Number 6.

6. Satran J., Meth K., Sapuntzakis C., Chadalapaka M., Zeidner E. 2007. *Internet Small Computer Systems Interface (iSCSI)*. IETF RFC 3720.

7. Callaghan B.,Pawlowski B., Staubach P. 1995. *NFS Version 3 Protocol Specification*. IETF RFC 1813.

8. Ko M., Chadalapaka M., Hufferd J., Elzur U., Shah H., Thaler P. 2007. *Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA)*. IETF RFC 5046.

9. Romanow A., Mogul J., Talpey T., Bailey S. 2005. *Remote Direct Memory Access (RDMA) over IP Problem Statement*. IETF RFC 4297.

10. Compaq Computer Corp., Intel Corp., Microsoft Corp. 1997. *Virtual Interface Architecture (VIA) Specification*.

11. Culley P., Elzur U., Recio R., Bailey S., Carrier J. 2007. *Marker PDU Aligned Framing for TCP Specification*. IETF RFC 5044.

12. Shah H., Pinkerton J., Recio R., Culley P. 2007. *Direct Data Placement over Reliable Transports*. IETF RFC 5041.

13. Recio R., Metzler B., Culley P., Hilland J., Garcia D. 2007. *A Remote Direct Memory Access Protocol Specification*. IETF RFC 5040.

14. Hilland J., Culley P., Pinkerton J., Recio R. 2003. *RDMA Protocol Verbs Specification, Version 1.0*.

15. OpenFabrics Alliance. http://www.openfabrics.org/.

16. Pool M. 2003. *distcc, a fast free distributed compiler*. Whitepaper.

17. Snyder P. 1990. *tmpfs: A Virtual Memory File System*.

18. Mogul, J.C. 2003. *TCP offload is a dumb idea whose time has come*. Hewlett-Packard Laboratories, Palo Alto, CA.

19. Dalessandro, D., Wyckoff, P. 2007. *Accelerating Web Protocols Using RDMA*. Proceedings of Network Computing and Applications (IEEE NCA'07).

20. Dalessandro, D., Devulapalli, A., Wyckoff, P. 2005. *Design and Implementation of the iWARP Protocol in Software*. Proceedings of Parallel and Distributed Computing Systems (PDCS'05).