Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Friedrich Seifert, Wolfgang Rehm (Eds.)

Selected Aspects of Cluster Computing

Preprint SFB393/01-12

Preprint-Reihe des Chemnitzer SFB 393

SFB393/01-12 **March 2001**

Contents

Author's addresses:

Friedrich Seifert
Wolfgang Rehm
TU Chemnitz
Fakultät für Informatik
D-09107 Chemnitz

`http://www.tu-chemnitz.de/informatik/RA/`

Abstract

Clusters gain more and more attention in the area of scientific computing since they are a low priced alternative to proprietary parallel computers. However, numerous problems on many levels still have to be solved in order to drive forward that development.

The present volume represents a collection of selected conference papers of our research group from the current project period and thereby gives an overview of various aspects of cluster computing, especially in the area of communication interfaces. We give an insight into all system levels starting at the hardware architecture via driver software and efficient message passing libraries up to performance analysis issues.

Note: The first article is written in German.

Keywords: Cluster computing, MPI, communication interfaces, Linux.

This page intentionally left blank.

Multiple Devices unter MPICH

Sven Schindler (svsc@informatik.tu-chemnitz.de)

Technische Universität Chemnitz
Fakultät für Informatik
Straße der Nationen 62, 09107 Chemnitz

Zusammenfassung

In dieser Arbeit wird die Möglichkeit vorgestellt, verschiedene Devices unter MPICH zu benutzen. Mittels dem für diesen Zweck entwickelten Multidevice ist es damit möglich, verschiedene Kommunikationsnetzwerke gleichseitig in einer MPI–Applikation zu verwenden. Es werden Aufbau und Protokolle des Multidevices vorgestellt. Es wird aufgezeigt, welche Änderungen notwendig sind um aus einem normalen ADI–2 Device ein, an das Multidevice passendes Subdevice zu erstellen. Für die Benutzung des Multidevices wurde ein Tool mit dem Namen mdconfig entwickelt, welches zum Abschluß vorgestellt wird.

1 Motivation

Der am Lehrstuhl Rechnerarchitektur der TU Chemnitz vorhandene Forschungscluster OSCAR benutzt zur Verbindung der einzelnen Rechner neben Fast Ethernet auch die Hochgeschwindigkeitsnetzwerke Myrinet und SCI. Zur Zeit existiert allerdings keine MPI–Bibliothek, die es erlaubt diese 3 Kommunikationsnetzwerke gleichzeitig zu benutzen. Dieser Mangel soll mit der vorliegenden Arbeit gelöst werden.

Für die Hochgeschwindigkeitsnetzwerke SCI und Myrinet wurden bereits für die MPICH–Bibliothek Anpassungen (Devices) entwickelt. Daher lag es Nahe, auf Basis der MPICH–Bibliothek ein Lösung zu schaffen, die auf den bereits existierenden Devices basiert.

2 Struktur von MPICH

MPICH (**M**essage **P**assing **I**nterface **CH**ameleon)[1] vom Argonne National Laboratory entstand zeitgleich zum Standardisierungsprozeß und verbreitete sich dadurch als Referenzimplementierung sehr schnell. Heute ist es die verbreiteste, freie MPI-Implementierung.

MPICH bietet, auf Grund der in der Abbildung 1 dargestellten Schichtenstruktur, die Möglichkeit, mit relativ geringen Aufwand eine MPI–Bibliothek für eine konkrete (noch nicht unterstützte) Kommunikationshardware zu erstellen.

Die gesamte MPI–Funktionalität wird in einen hardwareunabhängigen und in einen hardwareabhängigen Teil aufgespalten.

Der hardwareunabhängige Teil übernimmt folgende Aufgaben:

- Verwaltung von Gruppen, Kommunikatoren und Kontexten
- Erstellung und Verwaltung von MPI–Datentypen

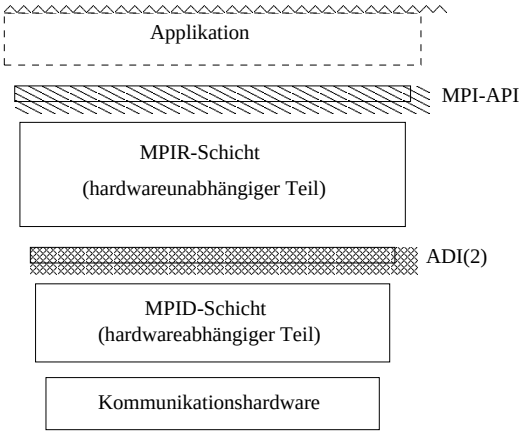

Abbildung 1: Struktur von MPICH

- Aufbau und Verwaltung von virtuellen Topologien

Optional steht eine Abbildung globaler Operationen auf Punkt zu Punkt Kommunikation zur Verfügung. Der hardwareabhängige Teil ist vor allem für die Realisierung der Punkt zu Punkt Kommunikation verantwortlich. Eigene Implementierungen der globalen Operationen können realisiert werden, falls diese eine Performancesteigerung zur Realisierung im hardwareunabhängigen Teil bieten.

Die Schnittstelle zwischen hardwareunabhängigen und hardwareabhängigen Teil wird als ADI (**A**bstract **D**evice **I**nterface)[2] bezeichnet. In der MPICH–Version 1.1.0 erfolgte die Einführung der ADI–2, auf die auch in dieser Arbeit Bezug genommen wird.

3 Das Multidevice

3.1 Struktur des Multidevices

Die Voraussetzung für den Entwurf des Multidevices war die volle Konformität mit der ADI–2 Schnittstellenspezifikation. Dies ist die Voraussetzung für die Verwendung des Multidevices in zukünftigen MPICH Distributionen.

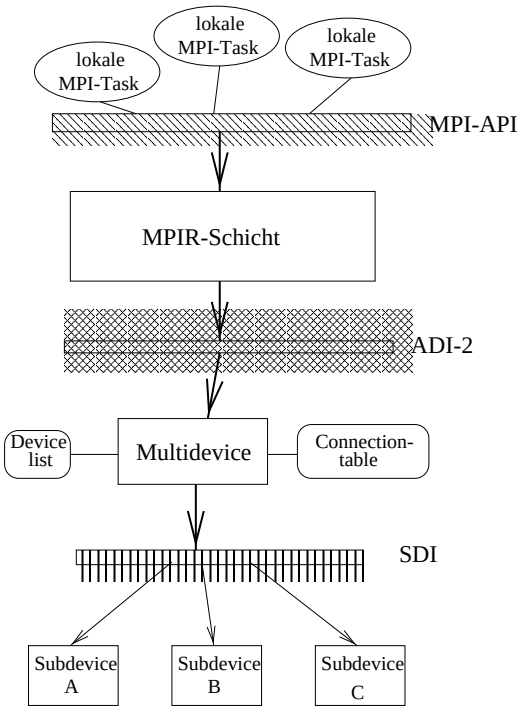

Abbildung 2: Struktur des Multidevices

Für das Verständnis der im folgenden vorgebrachtren Struktur ist es notwendig, sich den Aufbau einer MPI–Applikation auf z.B. einem Cluster of Workstations vorzustellen. Eine solche Applikation besteht zumeist aus mehreren Rechnern auf denen wiederum mehrere lokale Tasks arbeiten. Für Austausch der Daten der einzelnen Tasks sollte im lokalen Fall shared memory und über Rechnergrenzen hinaus hoffentlich zur Verfügung stehende Hochgeschwindigkeitsnetzwerke genutzt werden.

Die Abbildung 2 zeigt die Einbindung des Multidevices in die MPICH. Das Multidevice ist aus der Sicht des hardwareunabhängigen Teils ein “normales” ADI–2 Device und stellt beliebig vielen Subdevices das SDI (**S**ub**D**evice **I**nterface) zur Verfügung. Das SDI ähnelt stark der ADI–2, so daß die Erstellung eines neuen Subdevices aus einem “normalen” ADI–2 Device mit relativ geringem Aufwand möglich ist. Genauere Ausführung zu Subdevices findet man in Abschnitt 4.

Für die Funktion des Multidevices sind die Strukturen Devicelist und Connectiontable unbedingt erforderlich. Da erst zur Laufzeit feststeht, welche Subde–

vices überhaupt genutzt werden, können diese Subdevices erst zur Laufzeit geladen werden. Aus diesen Grund ist

es erforderlich, in der Devicelist Zeiger auf alle Funktionen der genutzten Subdevices zu halten. Die Connectiontable beinhaltet die gesamte Verbindungsinformation, d.h. es wird angegeben welches Subdevices für die Kommunikation zwischen 2 bestimmten MPI–Tasks verwendet werden soll. Das Gewinnen der Informationen für Devicelist und Connectiontable erfolgt mittels eines knotenspezifischen Konfigurationfiles (siehe Abschnitt 5).

Als Subdevice stehen für den lokalen Fall ein Multithreadingsubdevice und über Rechnergrenzen hinaus ein SCI–Subdevice zur Verfügung. Ein Subdevice für Myrinet[3] ist in Arbeit.

3.2 Die Initialisierung

Zu Beginn der MPI–Applikation unter MPICH ist nur eine MPI–Task aktiv, die mit Hilfe eines Prozessgruppenfiles für den Start aller weiteren MPI–Tasks verantwortlich ist. Diese erste aktive MPI–Task soll im folgenden als Master bezeichnet werden. Der Ablauf dieser Initialisierung gestaltet sich dabei folgendermaßen:

1.Schritt: Lesen des Prozeßgruppenfiles (nur Master) Der Master liest aus dem Prozeßgruppenfile die Namen der an der MPI–Applikation beteiligten Knoten sowie die Anzahl der auf jeden Knoten zu startenden MPI–Tasks.

2.Schritt: Starten eines Prozesses auf jedem Knoten (nur Master) Danach startet der Master auf jedem beteiligten Knoten eine MPI–Task (im folgenden Client genannt). Dieser Task wird ein Parameter mitgegeben, welche sie eindeutig als Nichtmaster ausweist, um zu vermeiden, daß endlos Prozesse erzeugt werden. Beim Start der Clients erfolgt der Aufbau einer Socketverbindung zwischen Master und jedem Client, die in der weiteren Initialisierung benötigt wird.

3.Schritt: Verteilen der Netzdaten In diesem Schritt sendet der Master, unter Nutzung der im vorherigen Schritt erzeugten Socketverbindung, die aus dem Prozessgruppenfile gewonnenen Daten an alle Clients. Damit besitzen nach diesem Schritt alle MPI–Tasks dieselben Informationen.

4.Schritt: Einlesen des Konfigurationsfiles Nun werden aus den knotenspezifischen Konfigurationsfiles die Daten für Devicelist und Connectiontable eingelesen. Das Erstellen der Konfigurationsfiles erfolgt durch das Tool mdconfig (siehe Abschnitt 5).

5.Schritt: Laden der Subdevices Nachdem durch das Einlesen der Konfigurationsfiles feststeht, welche Subdevices benötigt werden, können nun die Funktionszeiger der Subdevices bestimmt und in die Devicelist eingetragen werden. Damit ist ab hier ein Zugriff auf die Subdevices möglich.

6.Schritt: Starten aller benötigten lokalen MPI–Tasks Nun ist es an der Zeit, die anderen lokalen MPI–Tasks zu starten. Dieses erfolgt durch die Initialisierungsfunktion des lokalen Subdevices. Dadurch wird in diesem Schritt auch das lokale Subdevice initialisiert.

7.Schritt: Initialisieren der globalen Subdevices (eine Task pro Knoten) Als nächstes werden die globalen Subdevices initialisiert. Diese Initialisierung darf nur durch eine Task je Knoten erfolgen. Andere MPI–Tasks werden durch die Subdevices nicht mehr gestartet. Zur Parameterübergabe kann die im Schritt 2 angelegte Socketverbindung genutzt werden.

8.Schritt: Anlegen und Initialisieren der globalen Daten Zu guter Letzt werden noch die globalen Daten initialisiert. Dazu gehören vor allem Daten der MPIR–Schicht, wie z.B. **MPIR_shandles** und **MPIR_rhandles**, die von allen MPI–Tasks angelegt wird. **MPID_MyWorldRank** sollte von allen MPI–Task initialisiert werden, während **MPID_MyWorldSize** für alle MPI–Tasks gleich ist und somit nur einmal pro Knoten benötigt wird.

3.3 Kommunikationsprotokolle

Die im folgenden genutzte Einteilung in blockierende und nichtblockierende Kommunikation ist bereits durch die MPI–API[4] vorgegeben. Die Einteilung in normales Empfangen und Empfangen von **MPI_ANY_SOURCE** geschieht auf Grund der Tatsache, daß das Empfangen von **MPI_ANY_SOURCE** bei den meisten MPI–Implementierungen einige Probleme verursacht(so auch hier).

3.3.1 Blockierendes normales Senden/Empfangen

Das normale blockierende Senden/Empfangen ist der einfachste Fall. Das Multidevice wählt mit Hilfe der Connectiontable das zu benutzende Subdevice. Diese Auswahl erfolgt durch den Rank des Kommunikationspartners und evtl. durch die Größe der Message. Auf Empfängerseite ist zu beachten, daß vorher der lokale Rank des Senders in den globalen Rank umgerechnet werden muß. Danach wird, mit Hilfe der Devicelist, die entsprechende Funktion des gerade ermittelten Subdevices gerufen.

3.3.2 Nichtblockierendes normales Senden/Empfangen

Bei nichtblockierender Kommunikation wird ein sogenannter Request benutzt. Er enthält alle zur Kommunikation notwendigen Daten, wie z.B. Tag, Größe der Nutzerdaten, Empfänger usw. Dieser Request ist die einzige Möglichkeit, die Kommunikationsoperation nach ihrer Initialisierung zu referenzieren. Mit Hilfe dieses Requests kann abgefragt werden, ob die Kommunikationsoperation bereits beendet ist. Für diese Abfrage ist es notwendig, im Request zu speichern, welches Subdevice benutzt wurde. Danach wird wie im vorhergehenden Fall die entsprechende Funktion des ermittelten Subdevices aufgerufen.

3.3.3 Blockierendes normales Senden/Empfangen von MPI_ANY_SOURCE

Das normale blockierende Senden wurde bereits im Abschnitt 3.3.1 beschrieben. Vorausgesetzt es sind mehrere Subdevices auf einem Knoten aktiv, ist es beim Empfangen einer Message von **MPI_ANY_SOURCE** nicht möglich zu bestimmen, welches Subdevice zu benutzen ist. Die ADI–2 bietet mit der Funktion **MPID_Iprobe** die Möglichkeit, nichtblockierend zu testen, ob eine bestimmte Message sofort empfangen werden kann oder nicht. Jedes Subdevices besitzen eine analoge Funktion. Das Multidevice testet nun mit diesen Funktion reihum, ob die Message empfangen werden kann. Dies geschieht solange, bis ein Subdevice die Empfangsbereitschaft signalisiert. Danach wird beim empfangsbereiten Subdevice die Empfangsoperation ausgelöst.

3.3.4 Nichtblockierendes normales Senden/Empfangen von MPI_ANY_SOURCE

Das nichtblockierende Senden funktioniert wie im Abschnitt 3.3.2 beschrieben. Anders als im vorhergehenden Fall darf die Empfangsroutine nicht warten, bis ein passendes Send vorliegt, so daß ein komplizierterer Mechanismus benutzt werden muß. Die Abbildung 3 zeigt den grundlegenden Ablauf eines nichtblockierenden Empfangs von **MPI_ANY_SOURCE**.

Zuerst testet das Multidevice mittels **Iprobe**, ob sofort durch eines der Subdevices empfangen werden kann. Ist dies der Fall, wird der Request dem empfangsbereiten Device zugeordnet, und der Empfang verläuft wie in Abschnitt 3.3.2 beschrieben.

Ist der sofortige Empfang nicht möglich, dupliziert das Multidevice den Request. Es initialisiert den gemeinsamen Mutex der Duplikate und reiht den Originalrequest sowie dessen Duplikate in die AnyQueue[1] ein. Der Originalrequest wird dem Multidevice zugehörig markiert. Danach ruft es die Empfangsfunktion des Subdevices mit

[1] Die AnyQueue enthält alle Requests, die auf eine Empfangsoperation von **MPI_ANY_SOURCE** verweisen

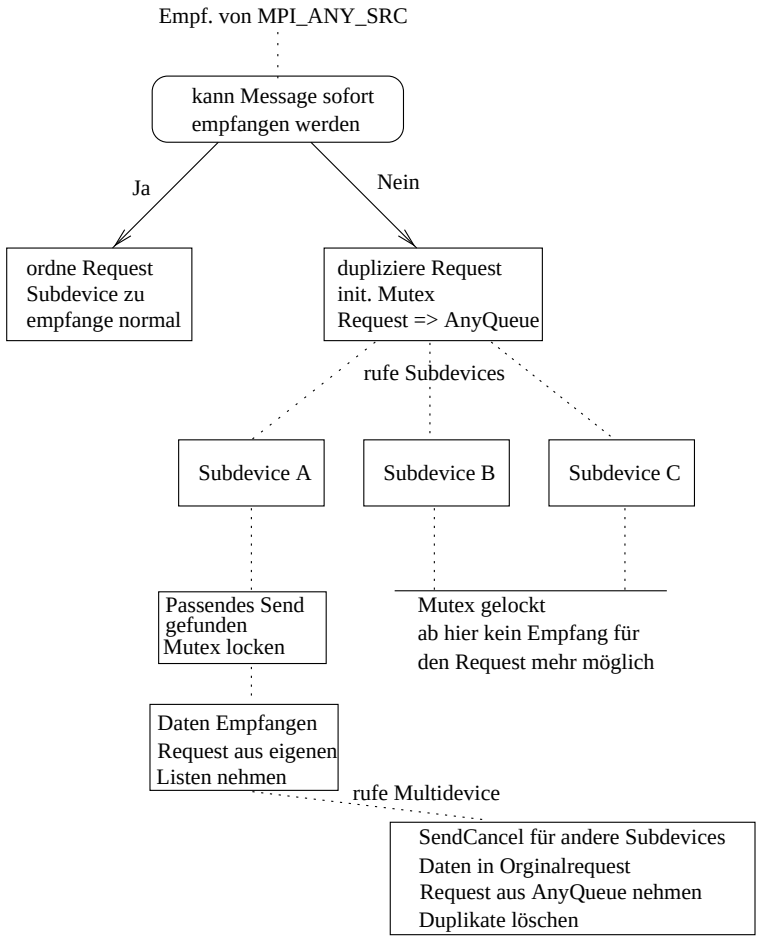

Abbildung 3: Ablauf des Empfangs von MPI_ANY_SOURCE

den Duplikaten auf. Jedes Subdevice besitzt nun ein anderes Duplikat des Originalrequests und den gemeinsamen Mutex.

Will nun ein Subdevice den Request bearbeiten, d.h. passende Daten empfangen, so muß es zuerst den Mutex locken, um damit anzuzeigen, daß der Request bereits bearbeitet wird. Schlägt das Locken des Mutex fehl, so muß das Subdevice nach einem anderen passenden Request suchen. Nach dem Locken des Mutex empfängt das Subdevice die Daten und füllt die Statusstruktur des Requests. Danach löscht es den Request aus seinen internen Listen und ruft die Multidevicefunktion **MULTI_CancelAnyRequest** auf. Diese Funktion löscht mit Hilfe von **RecvCancel**[2] die Duplikate aus den Queues der Subdevices. Danach werden die Statusdaten in den Originalrequest kopiert und der Orignalrequest als complete markiert. Nun sind noch die Requests aus der AnyQueue zu entnehmen und die Duplikate zu löschen.

3.4 Systemmessages und indirekte Kommunikation

Bei der in diesem Abschnitt vorgestellten indirekten Kommunikation handelt es sich um ein Konzept, welches noch nicht in der Software realisiert wurde. Unter indirekter Kommunikation wird im folgenden eine Datenübertragung über einen Zwischenknoten verstanden.

Ein Beispiel dafür ist in der Abbildung 4 zu erkennen. Hier soll eine Message von einer MPI–Task auf Knoten A zu einer MPI–Task auf Knoten C übertragen werden. Es wird angenommen, daß keine Verbindung zwischen den

[2]Diese Funktion entspricht der ADI–2 Funktion **MPID_RecvCancel**

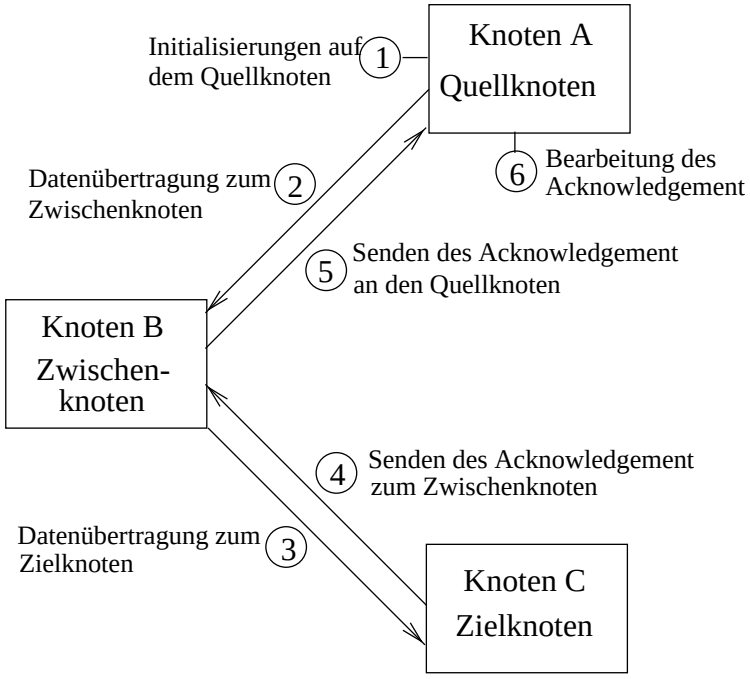

Abbildung 4: Ablauf einer indirekten Kommunikation

Knoten A und C existiert bzw. eine solche Verbindung langsamer als die Übertragung von A nach B und von B nach C ist.

Beim Senden der Message an Knoten B (mit der Bitte um Weiterleitung) entsteht das Problem, daß auf Knoten B die Message nie erwartet wird. Dadurch wird eine Art der einseitigen Kommunikation notwendig. Diese wird in Form der Systemmessages realisiert.

Eine Systemmessage ist eine spezielle, nicht von der Applikation initiierte Message. Sie ist einseitig, d.h. sie wird explizit nur auf der Senderseite ausgelöst, der Empfang erfolgt implizit. Der Mechanismus der Systemmessage ähnelt dem der Active Messages[5]. Eine Systemmessage wird auf Senderseite durch ein negatives Tag[3] gekennzeichnet. Hat ein Subdevice eine solche Systemmessage erkannt, ist es verpflichtet diese sofort zu übertragen. Da der Empfang implizit erfolgt, d.h das Subdevice auf Empfängerseite keinen Receiverequest für Systemmessages erhält, muß das Subdevice einen Empfangsbuffer zur Verfügung stellen. Nach Beendigung des Empfangs wird der Systemmessagehandler des Multidevices aufgerufen. Der Systemmessagehandler führt eine durch das Tag gekennzeichnete Aktion aus. Nach Beendigung des Systemmessagehandlers kann der Empfangsbuffer durch das Subdevice freigegeben werden. Als Voraussetzung für das erfolgreiche Arbeiten von Systemmessages müssen Subdevices einen eigenen Thread zum Senden und Empfangen besitzen, da sonst der sofortige Empfang und somit das sofortige Aktivieren des Systemmessagehandlers nicht gewährleistet werden kann[4].

Im MPI–2[6] enthaltene Funktionalität wie z.B. dynamisches Prozeßmanagement wird Konzepte wie Systemmessages erforderlich machen. Die ebenfalls in MPI–2 enthaltene einseitige Kommunikation kann ebenfalls durch dieses Konzept realisiert werden, wenn kein gemeinsamer Speicher zur Verfügung steht.

Der Ablauf einer indirekten Kommunikationoperation ist in der Abbildung 4 dargestellt. Für die indirekte Kommunikation muß die Requeststruktur um eine Semaphore erweitert werden. Es wird weiterhin eine IndirectQueue angelegt, die alle noch nicht beendeten, indirekten Kommunikationsoperationen enthält.

Im **1. Schritt** wird der Request in die IndirectQueue gestellt (im blockierenden Fall wird zuerst ein solcher Request

[3]Tags haben innerhalb einer MPI–Applikation einen Wertebereich von 0 bis **MPI_MAX_TAG**

[4]Das liegt daran, daß das Subdevice sonst nur beim Aufruf durch das Multidevice aktiviert wird. Erfolgt ein solcher Aufruf nicht, kann es zur Blockade des MPI–Systems kommen.

erstellt). Die Semaphore wird initialisiert und später benutzt, um auf die Beendigung der Sendoperation zu warten. Das Multidevice packt die Nutzdaten in einen Buffer und stellt diesem Buffer einen Header mit Rank, Zieltask und originalem Tag voran.

Das Versenden dieses Buffers zum Zwischenknoten erfolgt **2. Schritt** mittels einer Systemmessage. Diese Systemmessage erhält ein spezielles negatives Tag und als Ziel den Rank einer MPI–Task auf dem Zwischenknoten. Nach der Übertragung durch ein entsprechend ausgewähltes Subdevice wartet beim blockierenden Senden die Senderoutine des Multidevices an der Semaphore auf Ankunft des Acknowledgements und somit auf die Beendigung der Operation bzw. kehrt beim nichtblockierenden Senden sofort zurück.

Nach dem Empfang der Daten wird auf dem Zwischenknoten im **3.Schritt** der Systemmessagehandler aktiv. Dieser erkennt anhand des Tags, daß es sich um eine indirekte Message handelt. Er kopiert die Nutzdaten in einen Buffer[5] und legt einen Request für die normale, nichtblockierende Sendeoperation zum Zielknoten an. Dieser Request enthält dieselben Informationen wie auf dem Quellknoten. Er wird nachfolgend an die IndirectQueue (die Semaphore hat hier keine Bedeutung) angehangen. Danach löst der Systemmessagehandler die nichtblockierende Übertragung der Daten aus.

Nach dem Empfangen der Message auf dem Zielknoten wird im **4.Schritt** ein Acknowledgement mittels einer Systemmessage versendet. Im nichtblockierenden Fall ist für das Versenden dieser Systemmessage das entsprechende Subdevice verantwortlich.

Nach dem Empfang des Acknowledgements auf dem Zwischenknoten wird in **5. Schritt** der im 3. Schritt angelegte Request aus der IndirectQueue entfernt und freigegeben. Danach erfolgt das Senden des Acknowledgements als Systemmessage zum Quellknoten.

Im **6. Schritt** entfernt der Systemmessagehandler den im 1. Schritt angehängten Request aus der IndirectQueue. Danach wird im blockierenden Fall die Semaphore gepostet und somit die wartende MPI–Task wieder aktiv. Zu guter Letzt gibt die aktivierte MPI–Task den Request wieder frei. Im nichtblockierendem Fall wird der Request als complete markiert.

Dieser Mechanismus ist sehr aufwendig, zeigt aber eine Möglichkeit für die Realisierung von indirekter Kommunikation. Da aber neben erhöhten Aufwand auf Quell– und Zielknoten auch eine erhöhte Last auf dem Zwischenknoten erzeugt wird, soll vor dem Einsatz der indirekten Kommunikation Notwendigkeit und Sinn überporüft werden.

4 Erstellung eines Subdevices aus einem “normalen” ADI-2 Device

Dieser Abschnitt befasst sich mit den für ein Subdevice notwendigen Änderungen. Die Darstellung bezieht sich auf ein “normales ADI–2 Device, wie z.B. in [2] beschrieben.

Der fundamentale Unterschied zwischen einem normalen Device und einem Subdevice besteht in der Tatsache, daß ein Subdevice threadsafe entwickelt werden muß. Dies beruht auf der Tatsache, daß bei Benutzung von Threads als lokale MPI–Tasks diese verschiedenen Threads gleichzeitig Funktionen eines Subdevices nutzen können. Als Folge davon sind Datenstrukturen, die von verschiedenen Threads beschrieben werden können, durch Mutex zu schützen. Hier ist ein möglichst feingranulares Locking anzustreben, um unnötige Wartezeiten zu vermeiden. Durch das Locken verschiedener Datenstrukturen kann die Gefahr von Deadlocks entstehen.

4.1 Die Initialisierung

Bei der Initialisierung tritt Unterschied zwischen Subdevices für lokale bzw. für globale Kommunikation zu Tage. Während ein lokales Subdevice für den Start der lokalen Tasks verantwortlich ist (siehe Abschnitt 3.2 Schritt 6),

[5] Dies ist leider notwendig, um die Empfangsroutine des Subdevices nicht unnötig zu blockieren. Der Header wird bei dieser Kopieroperation abgeschnitten.

dürfen Subdevices für globale Kommunikation keine weiteren MPI–Tasks starten.

Der Hauptunterschied des lokalen Subdevices zum Vorläufer ADI–2 Device besteht in der Tatsache, daß im lokalen Subdevice keine globalen, MPIR–Schicht relevanten Datenstrukturen[6] mehr angelegt und initialisiert werden. Diese Aufgabe übernimmt das Multidevice.

Für die Entwicklung eines Subdevices für globale Kommunikation sind größere Änderungen notwendig. Die Initialisierung beginnt mit dem Problem festzustellen, auf welchen Knoten das Subdevice eigentlich aktiv ist. Für diese Aufgabe wird durch das Multidevice eine Hilfsfunktion mit dem Namen **GetDeviceNodeList** zur Verfügung gestellt. Ein ADI–2 Device startet, nachdem ihm die beteiligten Knoten bekannt sind, auf den jeweiligen Knoten einen Prozeß und hat dabei die Möglichkeit, dem Prozeß und damit natürlich auch dem Device benötigte Parameter mitzugeben. Da ein Subdevice die Prozesse nicht mehr selbst startet, muß es ein anderer Mechanismus zur Parameterübergabe gefunden werden. Ist ein Subdevice auf dem Masterknoten aktiv, so kann die Parameterübergabe mittels der bei der Initialisierung des Multidevices erzeugten Sockets erfolgen. Ist auf dem Masterknoten das Subdevice nicht aktiv, muß z.B. eine eigene Socketverbindung aufgebaut werden.

4.2 Senden und Empfangen

Für lokalen Subdevices sind für die Kommunikation fast keine Änderungen notwendig. Es ist allerdings zu beachten, daß die Nummerierung der lokalen MPI–Tasks nicht mehr bei 0 beginnen muß.

Da sich mehrere lokale MPI–Tasks ein globales Subdevice “teilen”, muß über die Verwaltung der Kommunikationsrequests bei globalen Subdevices nachgedacht werden. Es sind folgende Lösungen möglich:

1. Alle Kommunikationsrequests werden weiterhin in einer gemeinsamen Struktur gehalten. Das hat zur Folge, daß, wie schon erwähnt, die gemeinsamen Datenstrukturen vor gleichzeitigem Zugriff geschützt werden müssen. Damit ist bei dieser Variante nur ein sehr grobes Locking möglich, nämlich das Locken der gesamten Kommunikationsrequests.
2. Für jede lokale MPI–Task wird eine eigene Struktur zum Verwalten der Kommunikationsrequests angelegt. Damit sind gleichzeitige Zugriffe weitestgehend ausgeschlossen.

Die zweite Variante ist zu bevorzugen, da daß feingranularere Locking zu Performancevorteilen gegenüber der ersten Variante führt. Bei der ersten Variante besteht zudem die erhöhte Gefahr von Deadlocks.

Für den Fall, daß indirekte Kommunikation zum Einsatz kommen soll, ist es notwendig die im Abschnitt 3.4 beschriebenen Systemmessages zu implementieren. Dabei ist die Bedingung zu beachten, daß das Subdevice als Thread laufen muß.

4.3 Anbindung zum Multidevice

Das Subdevice muß eine Funktion mit dem Namen <*Subdevicename*>**_LoadDevicePtr** zur Verfügung stellen. Diese Funktion füllt einen Eintrag der Devicelist mit den Zeigern der Subdevicefunktionen. Diese Funktion wird bei der Initialisierung des Multidevices (im 5.Schritt) aufgerufen.

5 Konfiguration des Multidevices

Wie schon in den vorhergehenden Abschnitten erwähnt, sind neben den Daten, die das Prozeßgruppenfile bietet, weitere knotenspezifische Daten notwendig. Da es in einem großen System nicht nur aufwendig, sondern auch

[6] Dies betrifft vor allem **MPIR_shandles**, **MPIR_rhandles**, **MPID_MyWorldRank** und **MPID_MyWorldSize**.

sehr fehleranfällig ist, für jeden Knoten von Hand ein Konfigurationsfile zu schreiben, wurde dieser Prozeß mit dem Tool mdconfig (Der Name steht für **M**ulti**d**evice **Config**urator) automatisiert.

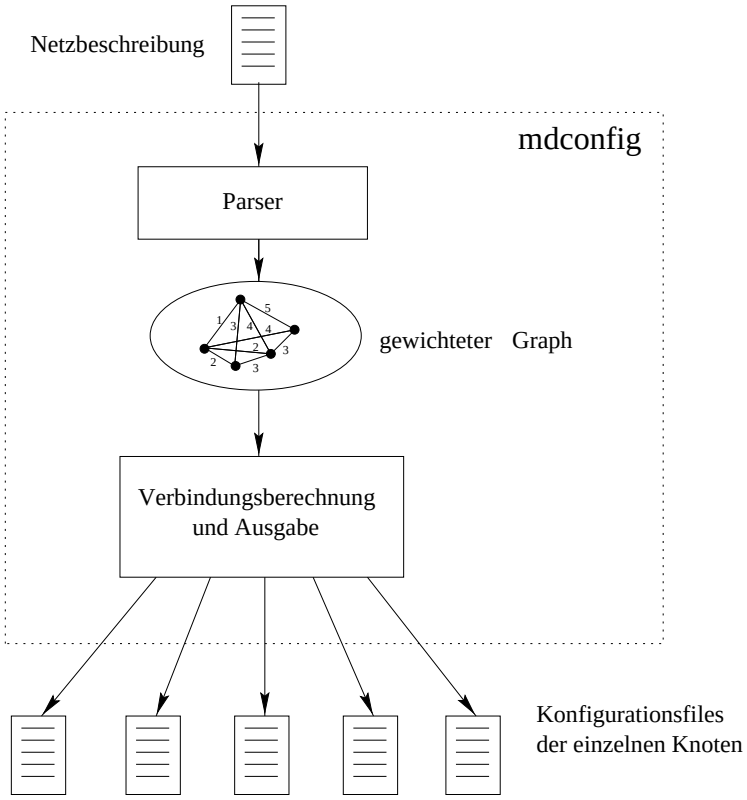

Abbildung 5: Arbeitsweise von mdconfig

Die Arbeitsweise des Tools ist in der Abbildung 5 dargestellt. Es erhält seine Eingabe in Form einer globalen Netzbeschreibung. Diese Beschreibung beinhaltet die Namen der zu benutzenden Subdevices, sowie alle existierenden Verbindungen. Aus diesem Eingabefile erstellt ein Parser einen gewichteten Graphen aus dem wiederum mit Hilfe eines leicht modifizierten Dijkstraalgorithmus die schnellsten Verbindungen und die dabei zu benutzenden Subdevices ermittelt werden. Die so gewonnenen Information werden in rechnerspezifischen Files abgespeichert.

In der Netzbeschreibung können Netzwerkparameter für unterschiedliche Messagegrößen angegeben werden. Damit ist es möglich, verschiedene Subdevices fúr verschiedene Messagegrößen zu verwenden. Die Benutzung indirekter Kommunikation kann mit Angabe eines Wertes für die Umsetzung einer Message auf dem Zwischenknoten gesteuert werden.

Eine genaue Beschreibung der Konfigurationssprache befindet sich in [7].

Literatur

[1] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High–Performance, Portable Implementation of the MPI message–passing standard. Argonne National Laboratory and Mississippi State University, 1996.

[2] William Gropp and Ewing Lusk. MPICH Working Note: The Second–Generation ADI for the MPICH Implementation of MPI. Argonne National Laboratory, Mathematics and Computer Science Division, 1996.

[3] Carsten Dinkelmann. Implementierung einer effizienten MPI–Schnittstelle für Myrinetkarten auf der Basis von Fast Messages. Diplomarbeit, WH Zwickau, FB Physikalische Technik/Informatik, 1999.

[4] Message Passing Interface Forum. MPI: A message–passing interface standard Vers. 1.1. http://www.mcs.anl.gov/mpi/standard.html, June 1995.

[5] Thorsten von Eiken, David E. Culler, Seth Copen Goldstein and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architekture*, 1992. available from `http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps`

[6] The MPI Forum. The MPI message-passing interface standard Vers. 2.0, May 1998.

[7] Sven Schindler. Entwurf und Implementierung eines ADI–2 Multidevices. Diplomarbeit,TU Chemnitz, Fakultät für Informatik, Lehrstuhl Rechnerarchitektur, March 1999.

A new generic and reconfigurable PCI–SCI bridge

Mario Trams, Wolfgang Rehm

{mtr,rehm}@informatik.tu-chemnitz.de

Technische Universität Chemnitz
Fakultät für Informatik
Straße der Nationen 62, 09111 Chemnitz

Abstract— **SCI becomes more and more accepted in the community of parallel computing, especially in case of Cluster Computing. At the moment Dolphin ICS is currently the leader in SCI Link Chip design as well as in PCI–SCI bridge manufacturing. Although raw performance of PCI–SCI products increased a lot over the last years, the basic architecture of this hardware has not changed. However, there are several disadvantages with todays PCI–SCI bridges such as unhandy memory management and the missing facility to realize protected user–level DMA. From our point of view, the last one is one of the most important feature to add into a conventional PCI–SCI architecture since this can increase general system throughput by a significant amount.**

In this paper we want to present a new PCI–SCI bridge we are currently about to build up. Further we describe our new architectural concepts helping to improve current SCI architecture for cluster computing.

Similar to the PCI–SCI bridges developed at the CERN and TU–Munich our bridge is mainly based on reconfigurable FPGAs. Hence, it is a *generic and reconfigurable* **hardware which can take up a lot of logical designs and therefore may not only be suitable for our ideas. So this paper may also be interesting for hardware developers and not only for SCI users.**

Keywords— **PCI–SCI Hardware, Message Passing, Virtual Interface Architecture, Protected User–Level DMA**

I. History and Motivation

Back in 1997 we got two PCI–SCI bridges developed by the CERN (Switzerland) RD24 project [5]. Our intention was to pursue the partly implemented FPGA design as well as to see what we can implement to speed up our applications.

In fact, we revised one of the FPGAs (the PCI–FPGA) [17] on this card so that it is able to act as a PCI Master on the PCI bus now. During this work we got not only a lot of experiences in FPGA design (as base for any hardware development) but also in general design of PCI–SCI interface hardware.

When this work was finished the next step initially planned was to improve the DMA engine of this card. But at this time we saw some limitations of the CERN PCI–SCI bridge detaining us from realizing new ideas. In addition, a lot of things in the appropriate hardware market have changed: the LC–2 came out, new larger and faster FPGAs came out etc. Therefore we decided to develop a new PCI–SCI board with latest technologies. The principle architecture of this bridge should be equal to the CERN's one, but with some small but important changes.

Because companies such as Dolphin are able to produce their own very high integrated and ultra–fast custom chips (ASICs), it is very difficult for us to be competetive with their products. Therefore it's not a major goal to beat their hardware in absolute performance characteristics. Our target is more on architecture issues which may flow into commercial products later. However, we're looking forward for a good and powerful design, since architectural features can increase end–performance at a higher level than it can be achieved by increasing the clock frequency or so.

The work presented in this paper is sponsored by the SMWK/SMWA Saxony ministries (AZ:7531.50-03-0380-98/6). It is also carried out in strong interaction with the project GRANT SFB393/B6 of the DFG (German National Science Foundation).

II. Reasons for a new bridge

Our primary focus is on message passing applications following the standardized *Message Passing Interface* (MPI [14]). The reason for this focus is that we are part of a research poject at our university called *Numerical Simulation on Massive Parallel Computers* (SFB 393). Mathematicans and physicists involved in this project use MPI for their FEM *Finite Elements Method* simulation software, and so our job is to provide appropriate optimized compute servers beginning at the MPI level down to the low hardware layers.

A. DMA Functionality

Although the primary intention of SCI is on distributed shared memory applications, its potentially high speed and low latency is also well suitable for message passing — that's well known. But although with SCI shared memory communication very low latencies and high bandwidths can be achieved (as shown in [2] for instance), the problem here is that the CPU participates active on the data transfer. E.g. it copies data from local memory to imported SCI memory. First, this consumes expensive CPU cycles with stupid copy operations. And second, it decreases in principle maximum achievable bandwidth, because a send is not a real zero–copy send (read data from local memory into CPU register; write data from CPU register into remote memory).

As a work–around for this problem a PCI–SCI bridge typically offers a DMA engine for larger block transfers. However, at the moment Dolphin's PCI–SCI bridges offer only a more conventional DMA engine which can't be used to realize some kind of protected user–level DMA. Thus, every DMA transfer must pass the operating system kernel as a managing instance. This in turn wastes also CPU time and increases transfer latency.

As we already presented in [19] we made some analysis on Dolphin's cards in comparing SCI shared memory performance with DMA performance not only in view of their absolute bandwidth values. The most important analysis was to see how much the CPU is slowed down during both communication methods.

Or in other words: How much time is available for CPU calculations when there's some data to be transferred.

As base for this analysis three points were taken:

- bandwidth of SCI shared memory writes
  We assumed 82MB/s over all message sizes starting at 64Bytes (see also *write–only* cases described in [2]).
- bandwidth of SCI DMA dependent on message size
  We used our measured Ping–Pong curve here (max. 50MB/s).
- CPU slow–down during active DMA engine
  We used the worst case performance loss here which was about 15% when the CPU cache was nearly not used.

Based on these facts it was possible to develop a roundabout analysis of the available CPU time in case of shared memory and DMA as function of the message size.

Because Dolphins cards (D310) showed a less bandwidth for DMA than SCI shared memory, the following scheme for determining the available CPU time can be applied:

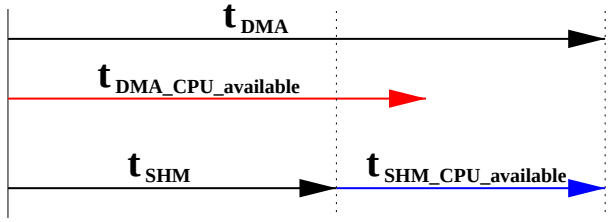

Fig. 1. Comparison Scheme

To transfer a dedicated message size using DMA, this takes a relatively long time. During this time the CPU can work in parallel, but not at full speed (only with $100\% - 15\% = 85\%$). Therefore

$$t_{DMA\_CPU\_available} = 0.85 \cdot t_{DMA}$$

To transfer the same message size using SCI shared memory it takes not so much time, but the CPU is not available in parallel. But since $t_{SHM} < t_{DMA}$, compared with DMA there's still some time remaining after the transfer. Therefore

$$t_{SHM\_CPU\_available} = t_{DMA} - t_{SHM}$$

Figure 2 demonstrate the graphical representation of these simple formulas.

As it can be seen, the switching point were Dolphin's DMA engine becomes more affordable in this view is at surprisingly low 128Bytes. But note that these measurements expect a user–level control of the DMA engine which is in practice not possible with Dolphin's cards.

Note that this analysis is not very accurate [19], and the real switching point probably has to be moved to slightly larger message sizes. But the result gives a raw idea about the behavior. For more precision the DMA mechanism has to be analyzed in very fine grained steps which is not trivial.

B. Memory Management Issues

Another completely different aspect is the SCI memory handling. Dolphin's bridges can handle SCI page sizes of 512kB

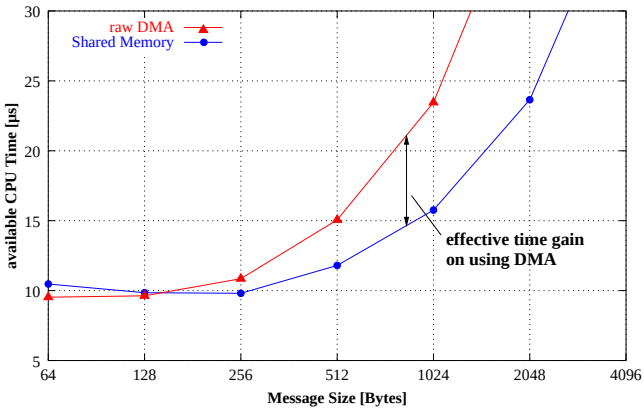

Fig. 2. CPU time available during data transfer

only. This implies that if someone wants to export some memory, he has to do this in a granularity of 512kB and the exported 512kB pages must be aligned to a 512kB boundary. However, allocating relatively large and aligned memory areas is momentarily not supported by common operating systems such as Linux as we use it. Otherwise, if it would be supported, this would tend to a hard memory fragmentation over the time.

To solve this problem, we use in case of Linux the so–called *Bigphysarea–Patch* enhanced by the PC^2 at Paderborn [3], which is an extension to the Linux memory management. With this patch it is possible to reserve an amount of dedicated consecutive memory locations for special purposes — such as memory to export into SCI space. To avoid unwanted memory accesses of the PCI–SCI bridge to sensitive locations, the bridge is set up to allow accesses only to the dedicated memory region.

The problem with this configuration is that it makes the handling with MPI applications very difficult. Especially in view of zero–copy transfer operations. Because data transfers can happen on the reserved memory region only, this would require the MPI applications to use special `malloc()` functions for allocating data structures used for send/receive purposes. But this violates a major goal of the MPI standard: Architecture Independence.

Although it seems that Dolphins latest 64Bit PCI chip supports also SCI page sizes of 4kB [8], it solves the problem not completely. Although the alignment problem with these large 512kB blocks is no longer valid, there's still the protection problem to keep critical memory spaces away from wrong accesses.

III. New Features

Our major goal is to eliminate the problems described in the last section, especially to support protected user–level DMA within a SCI environment. But how to realize a real protected user–level DMA model on SCI?

The most often referenced projects in the area of reducing operating system overhead are the SHRIMP [16] and U–Net [20] projects. In these projects several ideas and concepts were shown allowing direct accesses to the communication hardware while keeping protection issues alive. Later the so called **Virtual Interface Architecture** [21] came out which was initially

promoted by Intel, Compaq, and Microsoft. The VIA spec. in its latest version (1.0 from Dec.1997) is more a suggestion than a standard and leaves a lot of implementation specific details open. Therefore it was a suitable point for us and we evaluated how we can integrate VIA features into a SCI architecture in real hardware (no emulation over SCI shared memory). Generally, it was and is not a major goal to implement the VIA as it is, but to port concepts into SCI.

Major other research on VIA is done at the University of California, Berkeley [4] and the Berkeley Lab as part of the NERSC [13] (two different projects!). However, these projects are more directed to real VIA without shared memory capabilities. The only known try to use VIA within a SCI environment was done by Dolphin [2]. But the problem here is that VIA is not really supported by their PCI–SCI bridges and therefore VIA on SCI is more an emulation in this case. The main disadvantage of VIA over a global SCI shared memory is CPU utilization. Remember that the CPU is completely busy during the transmission process. However, Dolphin showed in [2] that this emulation is relatively powerful in view of bandwidth. Therefore we are reliant with our project to get similar values with not so much CPU utilization by offering real protected user–level DMA.

VIA communication is completely based on explicit descriptor processing. Hence there is no way to achieve ultra–low latencies as it can be done in SCI by using simple memory references.

The two major communication methods in VIA are Send/Receive and Remote DMA (RDMA). While Send/Receive is a two–sided comunication requiring two descriptors (one at the sender, the other at the receiver), RDMA is one–sided and requires only one descriptor at the active node.

Send/Receive is intended for pure message passing. Although this mode is very powerful we don't want to discuss it here any further because it resides a bit far from SCI functionality. The RDMA feature in contrast is much closer to SCI and can be integrated very efficient into the SCI architecture.

In the following sections we describe how we want to integrate VIA RDMA into a SCI architecture. Much of this work is a result of a diploma thesis which dealt whith new concepts for a PCI–SCI bridge [18].

A. How is Protection achieved in VIA?

A VIA NIC has no direct view to local memory. A typical VIA hardware has an own local virtual address space which is mapped to the host physical address space via an address translation mechanism. This local address space is divided into pages which are typically of the same size as the host page size (e.g. 4kB on Intel architecture, 8kB on Alpha architecture). Every page has also assigned a so–called **Protection Tag**. Based on these protection tags the VIA hardware either allows the access or not.

Normally, every process involved with VIA communication indirectly owns an unique protection tag. The term '*indirectly*' means, that a process doesn't hand over its protection tag within a descriptor or so. Instead, with every virtual interface inside the VIA NIC one protection tag is associated. And because of the fact, that a process can communicate only with its registered virtual interfaces, it is unable to bypass the protection tag mechanism.

The differentiation of all the virtual interfaces is simply achieved by so–called **Doorbells**. Every doorbell represents one virtual interface with it's specific control registers. The size of a doorbell is equal to the page size of the host computer and so the handling which process may access which doorbell (or virtual interface) can be simply realized by the hosts virtual memory management system.

B. How works RDMA in VIA?

Figure 3 shows a small VIA scenario. In this example there are two hosts each with two processes involved in VIA communication. There's a virtual connection between processes A and C, and also between B and D. Each process has registered parts of its memory to the VIA hardware so that it can access this memory.

If, for instance, A wants to transfer data out of its registered memory segment (1) into the memory segment registered by C (2), then it must prepare an appropriate RDMA write descriptor specifying source and destination addresses (addresses within the VIA NICs local address spaces). After A has informed the VIA NIC about the new descriptor, the NIC can start transferring data. Based on the protection tag the NIC checks on–the–flow whether A specified right or wrong local addresses. With every data packet send out to the destination node an idendifier of the affected remote virtual interface (owned by C in our example) is send out too. Based on this information the NIC inside host 2 can also compare the protection tag of the accessed memory with the protection tag dedicated to the virtual interface owned by C.

So neither A is able to access wrong memory locations on its host (e.g. segment 3), nor it is able to access wrong locations at the remote host (e.g. segment 4).

This mechanism is a very powerful one and offers 100% of protection. However, it offers only a DMA transfer and no shared memory communication. Therefore it is by definition not so powerful for short transmission sizes.

C. How works RDMA in conventional SCI?

Although this should be well known within the SCI community, we also want to discuss a small example here for better understanding the next section.

Look at figure 4. Essentially, there is a similar scenario as shown in figure 3. But since we're on SCI now, process A is able to map segment 2 (owned by C) into its virtual memory. When A wants to transfer data into segment 2, then it can do this by using the shared memory. Here is absolutely no problem with protection issues, because protection is guaranteed by the hosts virtual memory management system. But as mentioned earlier in this paper, this type of communication becomes inefficient for larger transmission sizes and therefore A may decide to use the RDMA feature instead. Typically, this means that A must prepare a DMA descriptor. But neither today's PCI–SCI bridges offer a DMA engine only as a virtual multiple instance (locking mechanisms required for multiple users), nor these engines can guarantee any protection.

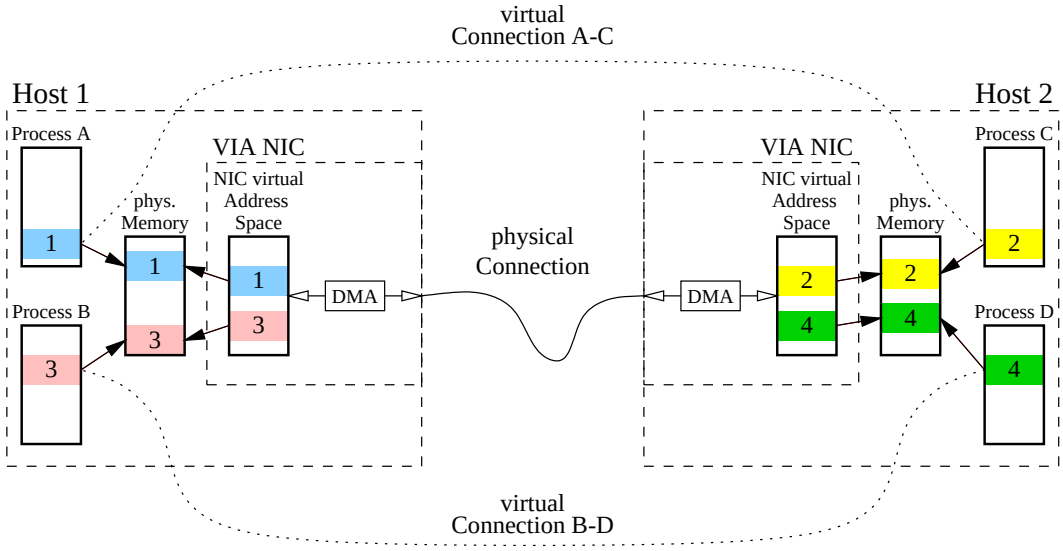

Fig. 3. VIA RDMA Example

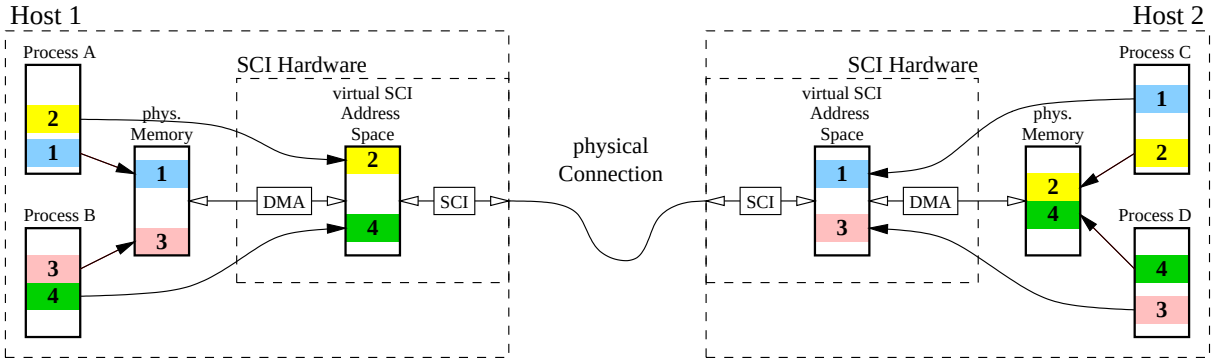

Fig. 4. SCI RDMA Example

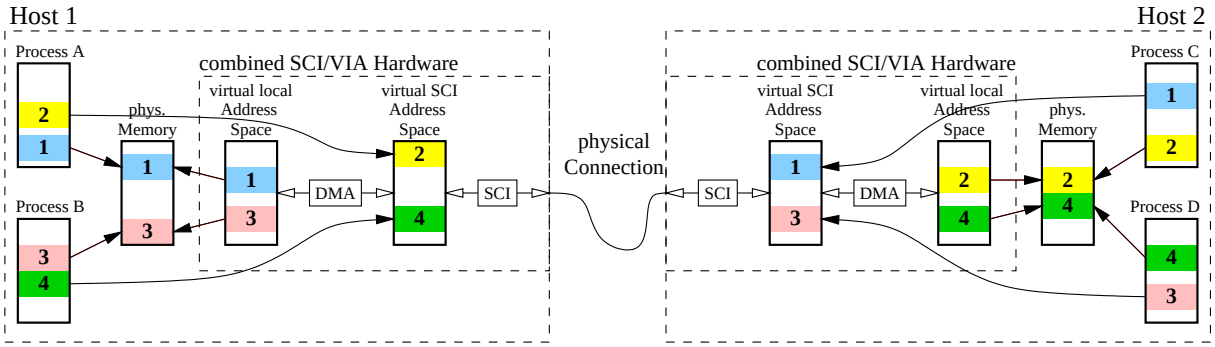

Fig. 5. Combined SCI/VIA RDMA Example

As it can be seen in figure 4, the DMA engine directly accesses local memory without any checks. The same is valid for accessing remote memory. Although there's a downstream address translation table, it's not concerned with any kind of protection.

So the only work–around here is to use the operating system kernel as a central controlling instance managing all DMA transfers and protection issues. Hence, user–level DMA is not possible (apart from proprietary closed systems where protection is not a concern).

D. *How to combine VIA and SCI RDMA?*

With our hardware we want to combine both the advances of low latency SCI shared memory as well as a better system throughput by using protected user–level DMA offered by VIA.

The practical combination is in principle very simple. We want to migrate the upstream address translation table as used in VIA (including the protection stuff, virtual interfaces, doorbells etc.) into the SCI architecture. In addition, the same protection mechanism as it is applied to local memory accesses is added to the downstream address translation.

Figure 5 shows the example scenario once more but inside the combined architecture. Accesses of the DMA engine are protected by protection tags at both sides (accesses to local and remote memory) and therefore a DMA transaction initiated by A can access only segments 1 and 2, for instance. Also, A has the choice whether to use shared memory or a DMA engine.

IV. Essential changes on current PCI–SCI Hardware

As a consequence of the things described in the last section we need a second address translation table for accessing local memory. In addition, appropriate mechanisms are required for handling all the VIA management stuff.

Although it may be possible to use the CERN PCI–SCI bridge as basic reconfigurable hardware to build up a trivial design, it can really only be a trivial design and it's difficult to get useful measurements later. Remember that this bridge is about 5 years in age and a lot changed in between. The most limiting factors are the FPGAs as well as the insufficient amount of local memory to store large translation tables.

In the remaining part of this paper we want to take a more detailed look inside our new hardware.

V. Architecture of our PCI–SCI Bridge

As mentioned earlier, the architecture is similar to the CERN [5] and Munich [1] ones. This means that there's

- an interface chip to the host PCI bus
- a FPGA for local (on board) PCI control (PCI FPGA)
- a dual ported static RAM for data storage
- a FPGA for SCI Link Chip control (SCI FPGA)
- a SCI Link Controller (Dolphin LC–2 in our case)
- an additional bank of static RAM (not in case of the CERN bridge)

Figure 6 shows the connections between all these components.

What is the intention of all these single circuits?

A. PCI–PCI Bridge

We use a special PCI bridge for interfacing with the host PCI bus — the 21554 [6] made by Digital (now maintained by Intel Corp.). This chip is not just a PCI–PCI bridge. It supports different address mappings for the primary (host) and secondary (local) PCI busses and hides the devices on the secondary bus from the primary bus. This is unlike to a real bridge which realizes a flat address space [15]. Thus, the PCI BIOS sees only one device (the 21554) and several address windows.

These address windows finally point to a location on the secondary PCI bus. With the ability of the 21554 to translate addresses from primary to secondary bus using a direct offset translation scheme, flexible addresses on the primary bus can be translated into hardwired addresses on the secondary bus. This significantly reduces address decoding complexity for the devices attached to the secondary PCI bus. In addition, it's not required to follow the PCI spec. at 100 percent. E.g. we don't need to implement a PCI Configuration Space inside the PCI FPGA. But the card as a whole stays still PCI spec. conformable.

These features allow a more efficient handling on the local PCI bus and leave FPGA space for more important things than a configuration space, for instance.

Of course, besides some disadvantages like increased costs and PCB complexity there's another problem: Transaction Latency. The 21554 adds some PCI cycles of additional latency when transactions are propagated through the chip. On the other hand the 21554 can help to increase peak bandwidth since it can accept transactions from both sides at the same time and hence doubles the bandwidth in this moment.

As a longer–term goal a bridge in between the data path should be removed. However this also eliminates the feature of a local PCI bus and reduces flexibility (in view of experimentations).

B. PCI FPGA

The PCI FPGA consists out of an ORCA 3T80 FPGA from Lucent Technologies [12]. This is a 3.3V 80kGate FPGA and there is an option to migrate to a 125kGate or 165kGate FPGA type of this series. The ORCA 3Txx series is an evolution of the ORCA 2Cxx series used on the CERN brigde. Since we collected a lot of experiences when working on the CERN card we decided to continue using the well known FPGA architecture and development tools.

The PCI FPGA is the most important unit on our card since it occupies key functions (relative to our architecture goals):

- management of transparent SCI memory accesses in a similar manner as in case of Dolphin's cards
- translation of addresses for outgoing and incoming transactions
- a powerful pipelined DMA engine to produce PCI bursts as large as possible
- integration of mechanisms to achieve Virtual Interface Architecture–like methods to invoke DMA transfers from user–level (at least Remote DMA, but also Send/Receive)

Generally, the PCI FPGA connects to the large 64Bit local PCI and DualPort Memory busses. A third 32Bit bus is used to interface with an SRAM bank. At the same time, this bus is used for primary handshaking and control exchange with the SCI FPGA. In addition, there are about 16 lines for other control purposes between both FPGAs.

C. SCI FPGA

Compared to the PCI FPGA, the SCI FPGA has relative simple jobs. The primary intention is handling of incoming and outgoing packet queues and interfacing with the SCI Link Controller.

The FPGA used here is from the same type as the PCI FPGA. However, later a different smaller one but with the same package

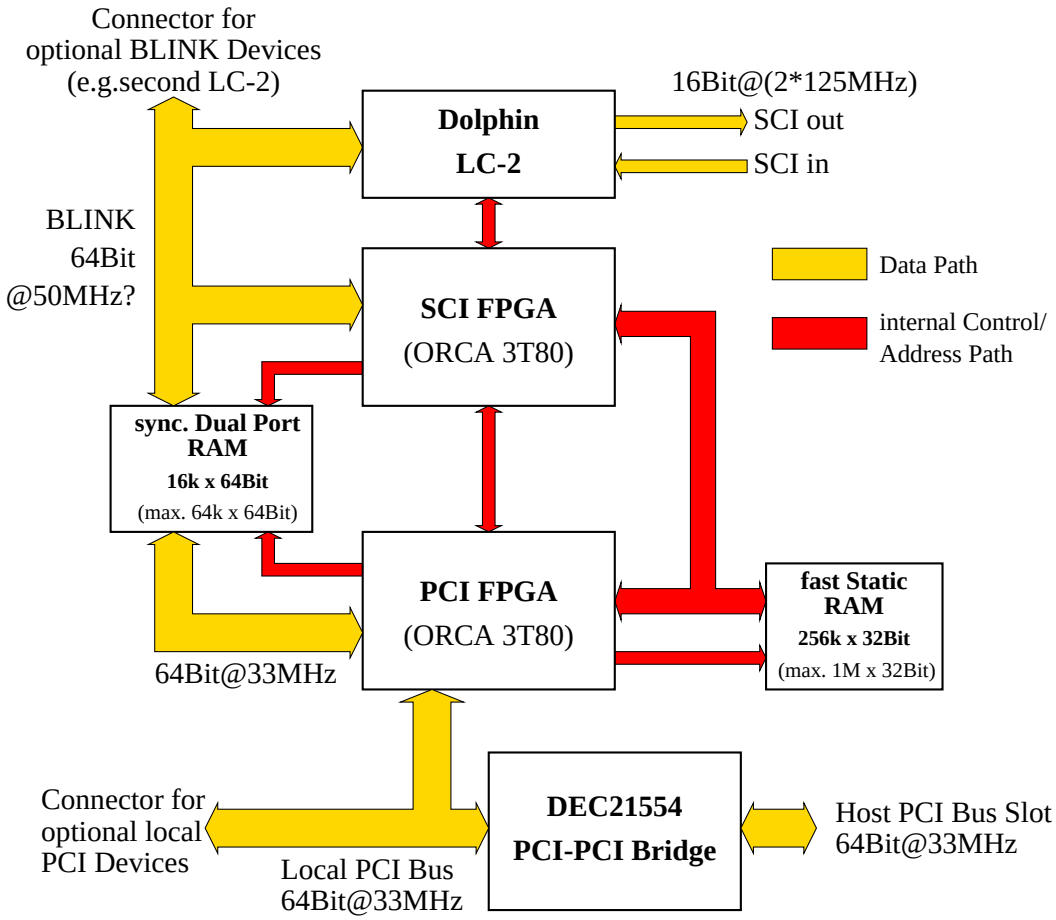

Fig. 6. Bridge Architecture

may be used to save costs.

D. Dual Ported Memory (DPM)

Likewise to CERN's and Munich's bridges, the primary intention of the DPM is for SCI packet storage. However, to simplify SCI packet handling we're planning to store only the data portions inside this memory. The remaining information (address and control) flows directly between the FPGAs. But this is implementation specific.

Physically, we use four synchronous DPM–chips (each 16Bits wide) which can go a maximum frequency of 50MHz. The synchronous feature simplifies especially the timing–critical write operations. The minimum DPM size of 128kB (or 16kWords @ 64Bit) is very much. However, we found no smaller chips on the market with similar functionality.

E. Static RAM

The extra bank of static RAM can take up 1–4MB of memory. It is organized in 2×512k×32Bit. Used RAM chips are either 256k×16 or 512k×16 (the later one may be not available on the market currently). Two 256k×16 gives 1MB and four 256k×16 gives 2MB (in case of the larger RAMs twice the amount). Used chips are common used asynchronous static RAMs in SOJ–package with revolutionary pinout. The access time (we use 17ns types) is fast enough to access one word within one cycle in case of 33MHz and slightly higher clocks.

The intentions for this static RAM in our design are:

- translation table storage for downstream address translation (for outgoing packets) and several SCI page attributes
  Note: This is well known from todays bridges.
- translation table storage for upstream address translation (for incoming packets) and several PCI page attributes
  Note: This is an absolutely new feature in the SCI architecture.
- Virtual Interface context memory storage
  Note: This is also new on SCI, but it's required for VIA functionality.

F. SCI Link Controller

Here's not a lot to say. We use Dolphin's LC–2 with a SCI link clock of max. 125MHz (adjustable via software). This results in a maximum duplex throughput of 1GByte/s over the SCI link. Dolphin's latest LC–3 is out now too, but unfortunately it is not pin–compatible with the LC–2 and therefore it can't be simply replaced. But when everything is working with the LC–2 it should be possible to change parts of the board layout to take up the LC–3.

G. Bus Clock Speeds

Normally, faster clocks on the local busses than on the host PCI bus doesn't increase sustained performance. Only latency

for short transmission sizes can be slightly reduced.

Therefore it is planned to drive the BLINK portion of the card (SCI FPGA etc.) with 50MHz asynchronous to the local PCI portion (PCI FPGA). But it's also possible to configure the board so that the BLINK portion uses the same frequency as the PCI portion.

Normally, the local PCI bus is clocked with the same frequency as the host PCI bus (33MHz). But here it's also possible to use an asynchronous separate clock. Tests will show if it's possible to overclock the local PCI bus somewhat (maybe 40MHz). But this mainly depends on the 21554 PCI–PCI bridge and the logic inside the PCI FPGA.

However, the simplest and maybe most stable configuration is to use a common clock derivated from the host PCI bus for all devices. Asynchronous clocks may tend to problems, especially on the interface between both FPGAs.

VI. Availability

At the time this paper was written (June 1999) we had a first prototype of the board ready and partly tested. So far, everything was looking ok and the board was running as planned. As soon as all of the remaining components are tested on this prototype a second prototype will follow (early July 1999). From there all attention is spent on FPGA development so that a first usable design should be available with beginning of the next millenium.

VII. Related Developments on VIA and SCI

Current tendencies of hardware development for both areas seem to go either strict VIA or strict SCI. In case of SCI Dolphin is still the one and only manufacturer for commercially available PCI–SCI hardware. Although Dolphin has put some efforts into an investigation of VIA in order to use it on their hardware, there are made no special hardware improvements for VIA support until now. Some of the results of Dolphin's work were presented in [2].

The number of available VIA hardware implementations increases more and more. Some hardware product developers are Giganet [11], Finisar [9], and Fujitsu [10]. But from now, it is difficult to evaluate these products from distance. They are relatively new and the informations given by the companies about the details of the implementation are very small.

Bandwidth parameters given for the so–called cLAN from Giganet looking very good (approx. 100MBytes/s for 32Bit/33MHz PCI bus). In fact, this is twice the amount which is achievable by Dolphin's DMA engine on the D310 PCI–SCI bridge (refer to section II of this paper). Latency of this cLAN hardware for short transmission sizes is comparable to Dolphin's DMA engine, but the SCI shared memory is even faster here (by a factor of approx. 3–4).

As mentioned earlier, more open research on VIA is done at the University of California, Berkeley [4] and the Berkeley Lab [13]. While the work at the University of California, Berkeley concentrates more on VIA hardware implementations based on Myrinet, the work at the Berkeley Lab is targeted to software development for Linux. Native hardware implementations seeming not to be planned there.

To class our work into all of these activities, we see us more besides Dolphin trying to port features of VIA into a SCI architecture. It is not our goal to implement the VIA as it is. Rather we are looking to develop a powerful mix of both SCI and VIA functionality.

VIII. Summary

In this paper we discussed and presented a relatively wide range of research issues. First, we showed some of our experience with comercially available PCI–SCI bridges from Dolphin. The main quintessence here was the fact, that a DMA mechanism has also significant advances over a SCI shared memory data transfers in view of CPU utilization. And this fact is not true only for very large transmission sizes in range of many kilobytes, but it becomes interesting for much smaller ones.

Following this, we explained our ideas how to migrate features of the Virtual Interface Architecture into a SCI architecture, and which consequences this causes.

We gave a rough overview about some important facts of our new reconfigurable PCI–SCI bridge. These facts shouldn't be seen by the reader only as a consequence of our new architectural ideas (protected user–level DMA etc.). It is also intended as a small roadmap for working groups which want to do some development on PCI–SCI hardware, but don't want to build up a new hardware from the scratch.

Finally, a short view was taken on other related hardware developments.

Latest information about our project can be obtained from our website at

`http://www.tu-chemnitz.de/~mtr/VIA_SCI/`

IX. Acknowledgement

Last but not least we want to thank Dolphin Interconnect Solutions for their hints and cooperation in view of the Link Controller and high speed signal handling.

References

[1] Georg Acher: *Entwicklung eines SCI–Knotens zur Kopplung von PCI–basierten Arbeitsplatzrechnern mit Hilfe von VHDL.* Diploma Thesis, University of Technology Munich, October 1996.

[2] Torsten Amundsen and John Robinson: *High–performance cluster–computing with Dolphin's CluStar PCI adapter card.* In: Proceedings of SCI Europe '98, Pages 149–152, Bordeaux, 1998

[3] Matt Welsh, Roger Butenuth: *Bigphys Area Patch for Linux.*
`http://www.uni-paderborn.de/cs/heiss/linux/bigphysarea.html`

[4] Philip Buonadonna, Andrew Geweke: *An Implementation and Analysis of the Virtual Interface Architecture.* University of California at Berkeley, Computer Science Department, Berkeley, 1998. See also
`http://www.cs.berkeley.edu/~philipb/via/`

[5] Hans Müller, A. Bogaerts, C. Fernandes, L. McCulloch, P. Werner and Y. Ermoline: *PCI–SCI Bridge for high rate Data Aquisition Architectures at Large Hadron Collider.* PCI'95 Week, St.Clara, March 1995.

[6] Intel Corp.: *21554 PCI-to-PCI Bridge for Embedded Applications Hardware Reference Manual.*
http://developer.intel.com/design/bridge/dsc-21554.htm

[7] Dolphin Interconnect Solutions AS: *PCI–SCI Bridge Specification Rev. 4.01*. 1997.

[8] Dolphin Interconnect Solutions AS: *PSB-64/66, Features and Benefits.*
http://www.dolphinics.no/dolphin2/interconnect/chips/psb64/psb64-66.htm

[9] Finisar Homepage
http://www.finisar.com

[10] Fujitsu Synfinity CLUSTER Homepage
http://www.fjst.com/products/synfinitycluster/

[11] GigaNet Homepage
http://www.giganet.com

[12] Lucent Technologies: *ORCA Series 3 Field–Programmable Gate Arrays.* Datasheet, August 1998.
http://www.lucent.com/micro/fpga/3ctxx.html

[13] *M–VIA: A High Performance Modular VIA for Linux.* Project Homepage:
http://www.nersc.gov/research/FTG/via/

[14] Message Passing Interface Forum: *The MPI message–passing interface standard Rev. 2.0*, May 1998
http://www-unix.mcs.anl.gov/mpi/

[15] PCI Special Interest Group: *PCI to PCI Bridge Architecture Specification, Rev.1.0.*, April 1994
http://www.pcisig.com

[16] The SHRIMP Project: *Scalable High–performance Really Inexpensive Multi–Processor.*
http://www.cs.princeton.edu/shrimp/

[17] Mario Trams: *The new PCI–FPGA for the CERN PCI–SCI Bridge.* Study Work, Dept. of Computer Science, University of Technology Chemnitz, 1997. See also
http://www.tu-chemnitz.de/~mtr/puplications.html

[18] Mario Trams: *Design of a system–friendly PCI–SCI Bridge with an optimized User–Interface.* Diploma Thesis, Dept. of Computer Science, University of Technology Chemnitz, 1998. See also
http://www.tu-chemnitz.de/informatik/RA/themes/works.html

[19] Mario Trams, Wolfgang Rehm, and Friedrich Seifert: *An advanced PCI–SCI bridge with VIA support.* In: Proceedings of 2nd Cluster–Computing Workshop held in Karlsruhe, Pages 35–44, March 1999. See also:
http://www.tu-chemnitz.de/informatik/RA/CC99/

[20] The U-Net Project: *A User–Level Network Interface Architecture.*
http://www2.cs.cornell.edu/U-Net

[21] Intel, Compaq and Microsoft. *Virtual Interface Architecture Specification V1.0.*
Virtual Interface Architecture Homepage
http://www.viarch.org

Memory Management in a combined VIA/SCI Hardware

Mario Trams, Wolfgang Rehm, Daniel Balkanski and Stanislav Simeonov *
{mtr,rehm}@informatik.tu-chemnitz.de
DaniBalkanski@yahoo.com, stan@bfu.bg

Technische Universität Chemnitz
Fakultät für Informatik**
Straße der Nationen 62, 09111 Chemnitz, Germany

Abstract In this document we make a brief review of memory management and DMA considerations in case of common SCI hardware and the Virtual Interface Architecture. On this basis we expose our ideas for an improved memory management of a hardware combining the positive characteristics of both basic technologies in order to get one completely new design rather than simply adding one to the other. The described memory management concept provides the opportunity of a real zero–copy transfer for Send–Receive operations by keeping full flexibility and efficiency of a nodes' local memory management system. From the resulting hardware we expect a very good system throughput for message passing applications even if they are using a wide range of message sizes.

1 Motivation and Introduction

PCI–SCI bridges (Scalable Coherent Interface [12]) become a more and more preferable technological choice in the growing market of Cluster Computing based on non–proprietary hardware. Although absolute performance characteristics of this communication hardware increases more and more, it still has some disadvantages. Dolphin Interconnect Solutions AS (Norway) is the leading manufacturer of commercial SCI link chips as well as the only manufacturer of commercially available PCI–SCI bridges. These bridges offer very low latencies in range of some microseconds for their distributed shared memory and reach also relatively high bandwidths (more than 80MBytes/s). In our clusters we use Dolphins PCI–SCI bridges in junction with standard PC components [11]. MPI applications that we are running on our cluster can get a great acceleration from low latencies of the underlying SCI shared memory if it is used as communication medium for transferring messages. MPI implementations e.g. such as [7] show a bandwidth of about 35MByte/s for a message size of 1kByte which is quite a lot (refer also to figure 1 later).

The major problem of MPI implementations over shared memory is big CPU utilization on long message sizes due to copy operations. So the just referred good MPI

* Daniel Balkanski and Stanislav Simeonov are from the Burgas Free University, Bulgaria.

** The work presented in this paper is sponsored by the SMWK/SMWA Saxony ministries (AZ:7531.50-03-0380-98/6). It is also carried out in strong interaction with the project GRANT SFB393/B6 of the DFG (German National Science Foundation).

performance [7] is more an academic peak performance which is achieved with more or less total CPU consumption. A standard solution for this problem is to use a block–moving DMA engine for data transfers in background. Dolphins PCI–SCI bridges implement such a DMA engine. Unfortunately, this one can't be controlled directly from a user process without violating general protection issues. Therefore kernel calls are required here which in end effect increase the minimum achievable latency and require a lot of additional CPU cycles.

The Virtual Interface Architecture (VIA) Specification [16] defines mechanisms for moving the communication hardware closer to the application by migrating protection mechanisms into the hardware. In fact, VIA specifies nothing completely new since it can be seen as an evolution of U–Net [15]. But it is a first try to define a common industry–standard of a principle communication architecture for message passing — from hardware to software layers. Due to its DMA transfers and its reduced latency because of user–level hardware access, a VIA system will increase the general system throughput of a cluster computer compared to a cluster equipped with a conventional communication system with similar raw performance characteristics. But for very short transmission sizes a programmed IO over global distributed shared memory won't be reached by far in terms of latency and bandwidth. This is a natural fact because we can't compare a simple memory reference with DMA descriptor preparation and execution.

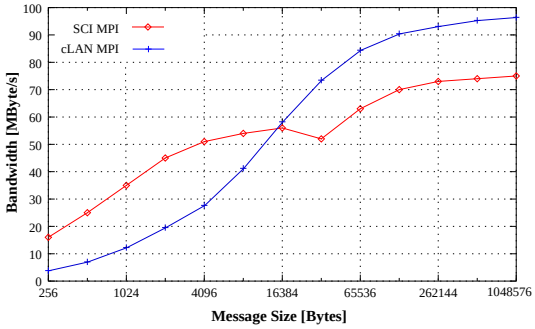

Figure1. Comparison of MPI Implementations for Dolphins PCI–SCI Bridges and GigaNets cLAN VIA Hardware

Figure 1 shows bandwidth curves of MPI implementations for both an SCI and a native VIA implementation (GigaNet cLAN). The hardware is in both cases based on the PCI bus and the machines where the measurements were taken are comparable. The concrete values are based on ping–pong measurements and where taken from [7] in case of SCI, and from [10] (Linux case) for the cLAN hardware.

As expected, the bandwidth in case of SCI is looking better in the range of smaller message sizes. For larger message sizes the cLAN implementation demonstrates higher bandwidth because of its advanced DMA engine. But not less important is the fact that a DMA engine gives the CPU more time for computations. Details of such CPU utilization considerations are outside the scope of this paper and are already discussed in [14] and [8].

As summarization of these motivating facts we can state that besides a powerful DMA engine controllable from user–level a distributed shared memory for programmed IO is an important feature which shouldn't be missed in a communication system.

2 What are the Memory Management Considerations?

First of all we want to make a short definition what belongs to memory management regarding this document.
This can be stated by the following aspects expressed in the form of questions:

1. How a process' memory area is made available to the Network Interface Controller (NIC) and in what way main memory is protected against wrong accesses?
2. At which point in the system a DMA engine is working and how are the transactions of this DMA engine validated?
3. In which way memory of a process on a remote node is made accessible for a local process?

Based on these questions we can classify the different communication system architectures in terms of advantages/disadvantages of their memory management. In the analysis that is presented in the following sections we'll reveal these advantages and disadvantages arisen from common PCI–SCI architecture and the VI Architecture.

3 PCI–SCI vs. VIA discussion and comparison

3.1 Question 1: How a process' memory area is made available to the NIC and in what way main memory is protected against wrong accesses?

Common PCI–SCI case: Current PCI–SCI bridges developed by Dolphin realize a quiet static memory management [4] to get access to main memory or rather PCI address space. To avoid unwanted accesses to sensitive locations, the PCI–SCI bridge is set up to allow accesses only to a dedicated memory window. Memory access requests caused by remote machines are only allowed if they fall within the specified window. This causes two big disadvantages:

- Continuous exported regions must also be continuous available inside the physical address space. Additionally, these regions must be aligned to the minimum exportable block size which is typically quite large (512kB for Dolphin's bridges).
- Exported Memory must reside within this window.

To handle these problems it is required to reserve main memory only for SCI purposes. This, in practice, 'wastes' a part of memory if it is not really exported later.

In consequence these disadvantages of common PCI–SCI bridge architecture make their use with MPI applications very difficult. Especially in view of zero–copy transfer operations. Because data transfers can be processed using the reserved memory region only, it would require that MPI applications use special `malloc()` functions for allocating data structures used for send/receive purposes later. But this violates a major goal of the MPI standard: Architecture Independence.

VIA case: The VI Architecture specifies a much better view the NIC has on main memory. Instead of a flat one–to–one representation of the physical memory space it implements a more flexible lookup–table address translation. Comparing this mechanism with the PCI–SCI pendant the following advantages become visible.

- Continuous regions seen by the VIA hardware are not required to be also continuous inside the host physical address space.
- Accesses to sensitive address ranges are prevented by just not including them into the translation table.
- The NIC can get access to **every** physical memory page, even if this may not be possible for all physical pages at once (when the translation table has less entries than the number of physical pages).

The translation table is not only for address translation purposes, but also for protection of memory. To achieve this a so–called *Protection Tag* is included for each translation and protection table entry. This tag is checked prior to each access to main memory to qualify the access. For more information about this see later in section 3.2.

Conclusions regarding question 1: It is clear, that the VIA approach offers much more flexibility. Using this local memory access strategy in a PCI–SCI bridge design will eliminate all of the problems seen in current designs.
Of course, the drawback is the more complicated hardware and the additional cycles to translate the address.

3.2 Question 2: At which point in the system a DMA engine is working and how are the transactions of this DMA engine validated?

Common PCI–SCI case: The DMA engine accesses local memory in the same way as already discussed in section 3.1. Therefore it inherits also all disadvantages when dealing with physical addresses on the PCI–SCI bridge.

For accesses to global SCI memory a more flexible translation table is used. This *Downstream Translation Table* realizes a virtual view onto global SCI memory — similar as the view of a VIA NIC onto local memory. Every page of the virtual SCI memory can be mapped to a page of the global SCI memory.

Regarding validation, the DMA engine can't distinguish between regions owned by different processes (neither local nor remote). Therefore the hardware can't make a check of access rights on–the–flow. Rather it is required that the DMA descriptor containing the information about the block to copy is assured to be right. In other words the operating system kernel has to prepare or at least to check any DMA descriptor to be posted to the NIC. This requires OS calls that we want to remove at all cost.

VIA case: A VIA NIC implements mechanisms to execute a DMA descriptor from user–level while assuring protection among multiple processes using the same VIA hardware. An user process can own one or more interfaces of the VIA hardware (so–called *Virtual Interfaces*). In other words, a virtual interface is a virtual representation

of a virtual unique communication hardware. The connection between the virtual interfaces and the VIA hardware is made by *Doorbells* that represent a virtual interface with its specific control registers. An user–level process can insert a new DMA descriptor into a job queue of the VIA hardware by writing an appropriate value into a doorbell assigned to this process. The size of a doorbell is equal to the page size of the host computer and so the handling which process may access which doorbell (or virtual interface) can be simply realized by the hosts' virtual memory management system. Protection during DMA transfers is achieved by usage of *Protection Tags*. These tags are used by the DMA engine to check if the access of the current processed virtual interface to a memory page is right. The protection tag of the accessed memory page is compared with the protection tag assigned to the virtual interface of the process that provided this DMA descriptor. Only if both tags are equal, the access is legal and can be performed. A more detailed description of this mechanism is outside the scope of this document (refer to [13] and [16]).

Conclusions regarding question 2: The location of the DMA engine is in both cases principally the same. The difference is that in case of VIA a real lookup–table based address translation is performed between the DMA engine and PCI memory. That is, the VIA DMA operates on a virtual local address space, while the PCI–SCI DMA operates directly with local physical addresses.
The answer for the access protection is simple: The common PCI–SCI DMA engine supports no protection in hardware and must trust on right DMA descriptors. The VIA hardware supports full protection in hardware where the DMA engine is only one part of the whole protection mechanism.

3.3 Question 3: In which way memory of a process on a remote node is made accessible for a local process?

Common PCI–SCI case: Making remote memory accessible is a key function in a SCI system, of course. Each PCI–SCI bridge offers a special PCI memory window which is practically the virtual SCI memory seen by the card. So the same SCI memory the DMA engine may access can be also accessed via memory references (also called programmed IO here). The procedure of making globally available SCI memory accessible for the local host is also referred as *importing global memory into local address space*.

On the other side, every PCI–SCI bridge can open a window to local address space and make it accessible for remote SCI nodes. The mechanism of this window is already described in section 3.1 regarding question 1. The procedure of making local memory globally accessible is also called *exporting local memory into global SCI space*.

Protection is totally guaranteed when dealing with imported and exported memory in point of view of memory references. Only if a process has got a valid mapping of a remote process' memory page it is able to access this memory.

VIA case: The VI Architecture offers principally no mechanism to access remote memory as it is realized in a distributed shared memory communication system such as SCI. But there is an indirect way by using a so–called Remote DMA (or RDMA) mechanism.

This method is very similar to DMA transfers as they are used in common PCI–SCI bridges. A process that wants to transfer data between its local memory and memory of a remote process specifies a RDMA descriptor. This contains an address for the local VIA virtual address space and an address for the remote nodes' local VIA virtual address space.

Conclusions regarding question 3: While a PCI–SCI architecture allows processes to really share their memory globally across a system, this is not possible with a VIA hardware. Of course, VIA was never designed for realizing distributed shared memory.

4 A new PCI–SCI Architecture with VIA Approaches

In our design we want to combine the advances of an ultra–low latency SCI Shared Memory with a VIA–like advanced memory management and protected user–level DMA. This combination will make our SCI hardware more suitable for our message passing oriented parallel applications requiring short as well as long transmission sizes.

4.1 Advanced Memory Management

In order to eliminate the discussed above restrictions with continuous and aligned exported memory regions that must reside in a special window, our PCI–SCI architecture will implement two address translation tables — for both local and remote memory accesses. In contrast, common PCI–SCI bridges use only one translation table for accesses to remote memory. This new and more flexible memory management combined with reduced minimal page size of distributed shared memory leads to a much better usage of the main memory of the host system.

In fact, our targeted amount of imported SCI memory is 1GB with a page granularity of 16kB. With a larger downstream address translation table this page size may be reduced further to match exactly the page size used in the host systems (such as 4kB for x86 CPUs).

In case of the granularity of memory to be exported in SCI terminology or to be made available for VIA operations there's no question: It must be equal to the host system page size. In other words, 4kB since the primary target system is a x86 one. 128MB is the planned maximum window size here.

4.2 Operation of Distributed Shared Memory from a memory–related point of view

Figure 2 gives an overall example of exporting/importing memory regions. The example illustrates the address translations performed when the importing process accesses memory exported by a process on the remote node.

The exporting process exports some of its previously allocated memory by registering it within its local PCI–SCI hardware. Registering memory is done on a by–page basis. Remember that in case of a common PCI–SCI system it would be required that

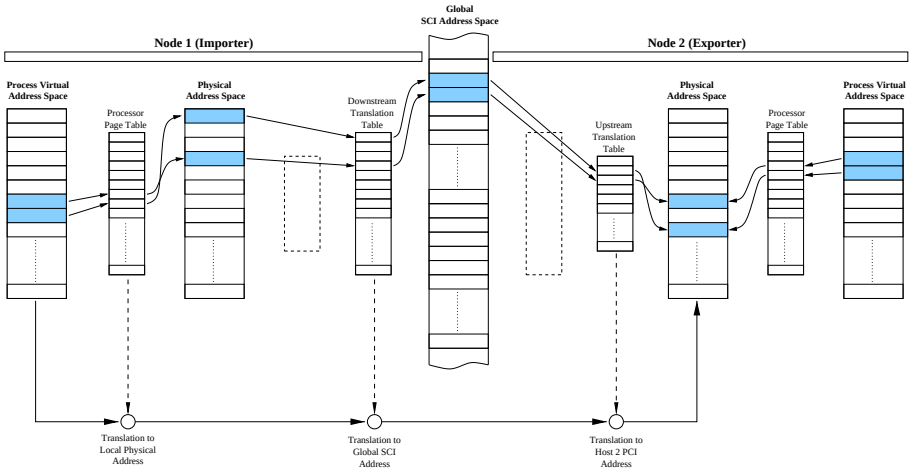

Figure2. Address Translations between exporting and importing Processes for programmed IO

this exported memory is physically located inside this special memory area reserved for SCI purposes. But here we can take the advantage of the virtual view onto local memory similar to this in VI Architecture.

Once the upstream address translation table entries are adjusted, the exported memory can be accessed from remote machines since it became part of the global SCI memory. To access this memory, the remote machine must import it first. The major step to do here is to set up entries inside its downstream address translation table so that they point to the region inside the global SCI memory that belongs to the exporter. From now, the only remaining task is to map the physical PCI pages that correspond to the prepared downstream translation entries into the virtual address space of the importing process.

When the importing process accesses the imported area, the transaction is forwarded through the PCI–SCI system and addresses are translated three times. At first the host MMU translates the address from the process' virtual address space into physical address space (or rather PCI space). Then the PCI–SCI bridge takes up the transaction and translates the address into the global SCI address space by usage of the downstream translation table. The downstream address translation includes generation of the remote node id and address offset inside the remote nodes' virtual local PCI address space. When the remote node receives the transaction, it translates the address to the correct local physical (or rather PCI) address by using the upstream address translation table.

4.3 Operation of Protected User–Level Remote DMA from a memory–related point of view

Figure 3 shows the principle work of the DMA engine of our PCI–SCI bridge design. This figure shows principally the same address spaces and translation tables as shown by figure 2. Only the process' virtual address spaces and the corresponding translation into physical address spaces are skipped to not overload the figure.

The DMA engine inside the bridge is surrounded by two address translation tables, or more correct said by two address **translation and protection** tables. On the active node (that is, where the DMA engine is executing DMA descriptors — node 1 here) both translation tables are involved. However, on the remote node there has practically nothing changed compared to the programmed IO case. Hence the remote node doesn't make any difference between transactions whether they were generated by the DMA engine or not.

Both translation tables of one PCI–SCI bridge incorporate protection tags as described in section 3.2. But while this is used in VIA for accesses to local memory, here it is also used for accesses to remote SCI memory. Together with VIA mechanisms for descriptor notification and execution the DMA engine is unable to access wrong memory pages — whether local (exported) nor remote (imported) ones. Note that a check for right protection tags is really made only for the DMA engine and only on the active node (node 1 in figure 3). In all other cases the same translation and protection tables are used, but the protection tags inside are ignored.

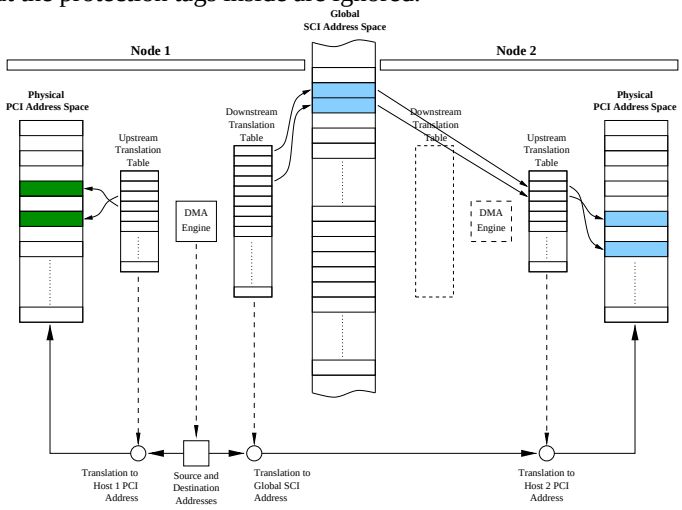

Figure3. Address Translations performed during RDMA Transfers

4.4 A free choice of using either Programmed I/O or User–Level Remote DMA

This kind of a global memory management allows applications or more exactly communication libraries to decide on–the–fly depending on data size in which way it should be transferred. In case of a short message a PIO transfer may be used, and in case of

a longer message a RDMA transfer may be suitable. The corresponding remote node is not concerned in this decision since it doesn't see any differences. This keeps the protocol overhead very low.

And finally we want to remember the VIA case. Although we already have the opportunity of a relatively low–latency protected user–level remote DMA mechanism without the memory handling problems as in case of common PCI–SCI, there's nothing like a PIO mechanism for realizing a distributed shared memory. Hence the advantages of an ultra–low latency PIO transfer are not available here.

5 Influence on MPI Libraries

To show the advantages of the presented advanced memory management we want to take a look at the so–called *Rendezvous Protocol* that is commonly used for Send–Receive operations.

Figure 4 illustrates the principle of the Rendezvous protocol used in common MPI implementations [7] based on Dolphins PCI–SCI bridges. One big problem in this

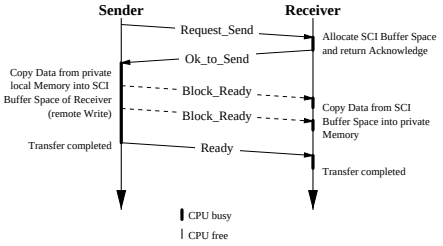

Figure4. Typical Rendezvous–Protocol in common PCI–SCI Implementations

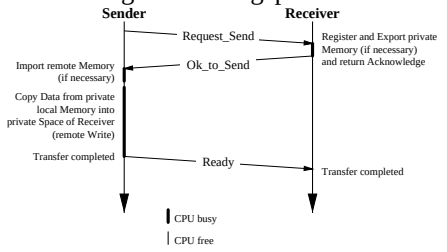

Figure5. Improved Rendezvous–Protocol based on advanced PCI–SCI Memory Management

model is the copy operation that takes place on the receivers' side to take data out of the SCI buffer. Although the principally increasing latency can be hidden due to the overlapping mechanism a lot of CPU cycles are burned there.

With our proposed memory management there's a chance to remove this copy operation on the receivers' side. The basic operation of the Rendezvous protocol can be implemented as described in figure 5. Here the sender informs the receiver as usual. Before the receiver sends back an acknowledge it checks if the data structure the data is to be written to is already exported to the sender. If not, the memory region that includes the data structure is registered within the receivers' PCI–SCI bridge and exported to the sender. The sender itself must also import this memory region if this was not already done before. After this the sender copies data from private memory of the sending process directly into private memory of the receiving process. As further optimization the sender may decide to use the DMA engine to copy data without further CPU intervention. This decision will be typically based on the message size.

6 State of the project (November 1999)

We developed our own FPGA–based PCI–SCI card and have prototypes of this card already running. At the moment they only offer a so–called *Manual Packet Mode* for now that is intended for sideband communication besides the regular programmed IO and DMA transfers.

The card itself is a 64Bit/33MHz PCI Rev.2.1 one [8]. As SCI link controller we are using Dolphins LC–2 for now, and we are looking to migrate to the LC–3 as soon as it is available. The reprogrammable FPGA design leads to a flexible reconfigurable hardware and offers also the opportunity for experiments.

Linux low–level drivers for Alpha and x86 platforms and several configuration/test programs were developed. In addition our research group is working on an appropriate higher–level Linux driver for our card [5, 6]. This offers a software–interface (advanced Virtual Interface Provider Library) that combines SCI and VIA features such as importing/exporting memory regions, VI connection management etc. Also it emulates parts of the hardware so that it is possible to run other software on top of it although the real hardware is not available. As an example, a parallelized MPI–version of the popular raytracer *POVRAY* is already running over this emulation. This program uses an MPI–2 library for our combined SCI/VIA hardware. This library is also under development at our department [3].

For more details and latest news refer to our project homepage at
`http://www.tu-chemnitz.de/~mtr/VIA_SCI/`

7 Other Works on SCI and VIA

Dolphin already presented some performance measurements in [1] for their VIA implementation which is a emulation over SCI shared memory. Although the presented VIA performance is looking very good, it's achieved by the cost of too big CPU utilization again.

The number of vendors of native VIA hardware is growing more and more. One of these companies is GigaNet [17] where performance values are already available. GigaNet gives on their web pages latencies of 8μs for short transmission sizes. Dolphin gives a latency for PIO operations (remote memory access) of 2.3μs. This demonstrates the relatively big performance advantage a distributed shared memory offers here.

University of California, Berkeley [2] and the Berkeley Lab [9] are doing more open research also in direction of improving the VIA specification. The work at the University of California, Berkeley is concentrated more on VIA hardware implementations based on Myrinet. In contrast, the work at the Berkeley Lab is targeted mainly to software development for Linux.

8 Conclusions and Outlook

The combined PCI–SCI/VIA system is not just a simple result of adding two different things. Rather it is a real integration of both in one design. More concrete it is an integration of concepts defined by the VIA specification into a common PCI–SCI architecture

since major PCI–SCI characteristics are kept. The result is a hardware design with completely new qualitative characteristics. It combines the most powerful features of SCI and VIA in order to get highly efficient messaging mechanisms and high throughput over a broad range of message lengths.

The advantage that MPI libraries can take from a more flexible memory management was illustrated for the case of a Rendezvous Send–Receive for MPI. The final proof in practice is still pending due to lack of a hardware with all implemented features.

References

1. Torsten Amundsen and John Robinson: *High–performance cluster–computing with Dolphin's CluStar PCI adapter card.* In: Proceedings of SCI Europe '98, Pages 149–152, Bordeaux, 1998
2. Philip Buonadonna, Andrew Geweke: *An Implementation and Analysis of the Virtual Interface Architecture.* University of California at Berkeley, Dept.of Computer Science, Berkeley, 1998. `www.cs.berkeley.edu/~philipb/via/`
3. *A new MPI–2–Standard MPI Implementation with support for the VIA.* `www.tu-chemnitz.de/informatik/RA/projects/chempi-html/`
4. Dolphin Interconnect Solutions AS: *PCI–SCI Bridge Spec. Rev. 4.01*. 1997.
5. Friedrich Seifert: *Design and Implementation of System Software for Transparent Mode Communication over SCI.*, Student Work, Dept. of Computer Science, University of Technology Chemnitz, 1999. See also: `www.tu-chemnitz.de/~sfri/publications.html`
6. Friedrich Seifert: *Development of System Software to integrate the Virtual Interface Architecture (VIA) into Linux Operating System Kernel for optimized Message Passing.* Diploma Thesis, TU–Chemnitz, Sept. 1999. See also: `www.tu-chemnitz.de/informatik/RA/themes/works.html`
7. Joachim Worringen and Thomas Bemmerl: *MPICH for SCI–connected Clusters.* In: Proceedings of SCI–Europe'99, Toulouse, Sept. 1999, Pages 3–11. See also: `wwwbode.in.tum.de/events/sci-europe99/`
8. Mario Trams and Wolfgang Rehm: *A new generic and reconfigurable PCI–SCI bridge.* In: Proceedings of SCI–Europe'99, Toulouse, Sept. 1999, Pages 113–120. See also: `wwwbode.in.tum.de/events/sci-europe99/`
9. *M–VIA: A High Performance Modular VIA for Linux.* Project Homepage: `http://www.nersc.gov/research/FTG/via/`
10. MPI Software Technology, Inc. *Performance of MPI/Pro for cLAN on Linux and Windows.* `www.mpi-softtech.com/performance/perf-win-lin.html`
11. *The Open Scalable Cluster ARchitecture (OSCAR) Project.* TU Chemnitz. `www.tu-chemnitz.de/informatik/RA/projects/oscar_html/`
12. *IEEE Standard for Scalable Coherent Interface (SCI).* IEEE Std. 1596-1992. SCI Homepage: `www.SCIzzL.com`
13. Mario Trams: *Design of a system–friendly PCI–SCI Bridge with an optimized User–Interface.* Diploma Thesis, TU-Chemnitz, 1998. See also: `www.tu-chemnitz.de/informatik/RA/themes/works.html`

14. Mario Trams, Wolfgang Rehm, and Friedrich Seifert: *An advanced PCI–SCI bridge with VIA support.* In: Proceedings of 2nd Cluster–Computing Workshop, Karlsruhe, 1999, Pages 35–44. See also: `www.tu-chemnitz.de/informatik/RA/CC99/`
15. The U-Net Project: *A User–Level Network Interface Architecture.* `www2.cs.cornell.edu/U-Net`
16. Intel, Compaq and Microsoft. *Virtual Interface Architecture Specification V1.0.*, VIA Homepage: `www.viarch.org`
17. GigaNet Homepage: `www.giganet.com`

An optimized MPI library for VIA/SCI cards

Sven Schindler, Wolfgang Rehm, Carsten Dinkelmann
Technische Universität Chemnitz
Fakultät für Informatik
Straße der Nationen 62, 09107 Chemnitz
({svsc,rehm,din}@informatik.tu-chemnitz.de)

Abstract

Rapid developments in computer architecture and in networking technology have driven the construction of clusters of cluster. Now cluster computers are an inexpensive alternative to parallel computers.

High bandwidth low latency communication networks have become more and more important. In conjunction with the Scalable Coherent Interface (SCI) the Virtual Interface Architecture (VIA) offers excellent opportunities for high speed networks. For this reason our research group is developing such a VIA/SCI network card.

This paper describes the demands and the design of a new MPI library for heterogenous communication networks. The implementation of this MPI library is demonstrated on an example VIA/SCI card.

Keywords: MPI, VIA, SCI, shared memory, communication protocol

1. The VIA/SCI card

1.1. Motivation

Our research group has been investigating cluster architectures for some time. Connection networks are a very important component of clusters of workstations and SCI (Scalable Coherent Interface [1]), an example of such a connection network, has become well accepted for clusters computing. The intention of the design of SCI was to provide a fast network for distributed shared memory applications, but the high bandwidth and the low latency allows the use in applications based on message passing, too. Our primary focus are applications based on message passing because the other members of our research project using MPI for their applications [1]. For this reasons our research group is developing an SCI card especially for message passing purposes[2][3].

The simplest way to implement an MPI library on SCI is to emulate message passing on shared memory, but this implies that the CPU actively participates on the data transfer, e.g. for copying the data into the SCI memory. The DMA engine offers an alternative solution for this problem. After their initialization it copies the data to the SCI card without additionally usage of the CPU. But the use of the DMA engine leads to another problem. To avoid access to memory which is not part of the process it is necessary to involve the operating system kernel for starting DMA transfers, because only the kernel is able to perform this security check. With the VIA (Virtual Interface Architecture, discussed in next section) a mechanism for protected user–level DMA without a kernel call is provided.

For this reason our card supports SCI shared memory as well as VIA and, thus, allows very high performance for message passing.

1.2. What is VIA ?

The idea of removing the the kernel from the critical path of communication operations is described in projects like U–net[4] or SHRIMP[5] already. Out of these projects the industry–driven standard VIA[6] was created, initially by Intel, Compaq and Microsoft. This standard specifies an interface between the high–speed network cards and the system. Later Intel has published some enhancements and some suggestions for the implementation[7].

The structure of the VIA is shown in Figure 1. For the creation of a new MPI library the two important parts of this structure are the VI User Agent and the VI Kernel Agent. The User Agent provides the user–level part of the interface to the network(card) whereas the Kernel Agent is a kernel–level device driver that performs operations that require kernel calls (e.g. memory registration). Intel called their example User Agent ”VIPL” (Virtual Interface Provider Li-

[1]This work is part of the GRANT SFB 393/B6 of the DFG (German national Science Foundation)

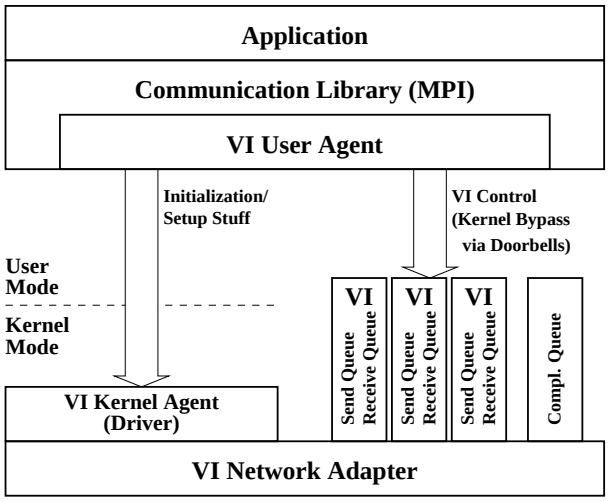

Figure 1. Structure of the Virtual Interface Architecture

brary[2]).

The VI Architecture is based on so–called VI's (Virtual Interface), a kind of a bidirectional communication channels comparable with sockets. Communication between two processes will be established by connecting their local VI's. Two principles exist for the connection of two VI's, a client–server based one and a peer–to–peer based one.

VIA makes the two communication methods Send/Receive and RDMA (Remote Direct Memory Access) available. Both methods are based on descriptor[3] processing so there is now way to archive ultra low latencies.

The Send/Receive mechanism is well suitable for message passing. Each Send/Receive requires two descriptors one on the senderside and the other on the receiverside. A receive descriptor with a data buffer of sufficient size has to be posted before the sender's data arrives. This requirement leads to an increased synchronization effort for the message passing software.

The RDMA mode is a one sided operation. Hence, it requires a descriptor only at the active node. There are two types of RDMA operations, RDMA Reads and RDMA Writes. The initiator of the RDMA operation specifies the source as well as the destination of the operation.

The VIA requires to identify memory used for data transfer. For this identification the memory has to be registered before the transfer. The registration of the memory works with so–called protection tags[4]. As already noted the memory registration (as well as the deregistration) is realized by courtesy of the VI Kernel Agent and consequently requires a kernel call. Once registered, the memory can be used without any additionally kernel call as often as needed. Because the registration of the memory is independent from the communication operation the kernel call for the security check is removed from the critical path. In order to get high performance it is profitably to use registered buffer again like in the MPI persistent communication. Because the amount of memory for registration is limited it is important to deregister memory not required any longer.

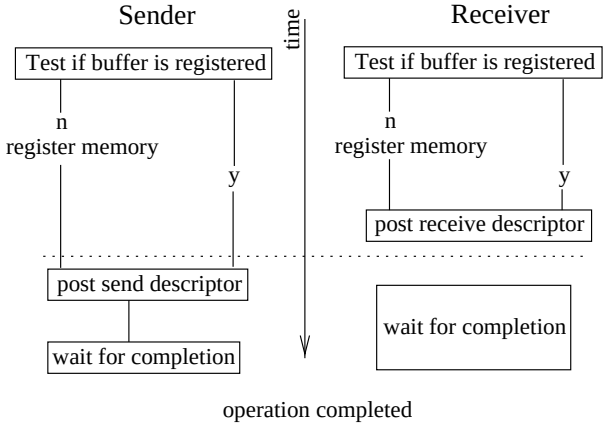

Figure 2. Simple VIA Send/Receive

Figure 2 describes a simple blocking Send/Receive in the VIA environment.

2. An brief overview of related MPI libraries

In this section some important members of the abundance of MPI libraries will be presented in relation to the VIA/SCI card and their advantages and disadvantages will be discussed.

2.1. MPICH

MPICH (Message Passing Interface CHameleon)[8] is a result of a cooperation of the Argonne National Laboratory and the Mississippi State University. It is one of the oldest MPI libraries with support of many architectures and communication networks and today is the most common implementation of the MPI-1 Standard[9]. The simplified structure of MPICH is shown in Figure 3.

The main advantage of MPICH is the layered design. MPICH divides the MPI–1 functionality into a hardware independent and a hardware dependent part (layer). Due

[2]This term was introduced in [7].

[3]A data structure that describes a data transfer request.

[4]Usually, a process uses a unique protection tag which is created after opening the VIA environment. This protection tag is bounded to each VI and used for all registration purposes.

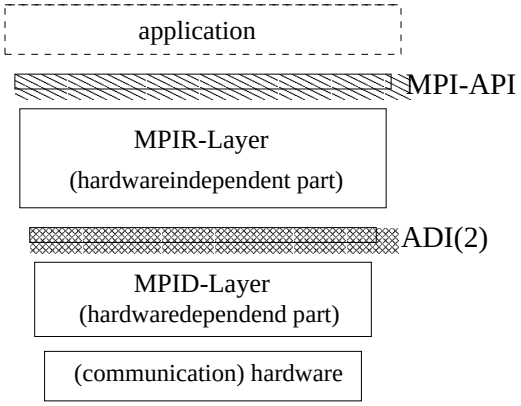

Figure 3. Simplified structure of MPICH

to this structure it is possible to generate an MPI implementation for a new (or not yet supported) communication network with small effort because the effort to implement a hardware dependent part is much smaller than the implementation of the full MPI functionality.

MPICH has some performance disadvantages which results from its design. The chances for optimization are limited because some important features (e.g. memory management) are implemented in the hardware independent part which should not be changed by the creation of a new hardware dependent part. Hence some problems result for the VIA registration and deregistration which should be a part of the memory management. MPICH has some problems to support multiple devices as well.

The design of MPICH is not thread safe and so it is difficult to overlap computation and communication. Therefore the communication progresses only when an MPI function is called. As MPICH uses polling for synchronization it wastes CPU time.

2.2. MPI-Pro

MPI/Pro[10], a product of the MPI Software Technology Inc., was the first MPI library for the Virtual Interface Architecture.

The main advantage of MPI/Pro is the threaded design that leads to a high degree of overlapping computation and communication. The optimized persistent operations allow to reuse registered buffers and to reduce the registration overhead in this way. For large messages a zero-copy protocol is implemented based on the VIA RDMA operations. Hence MPI/Pro achieves a very high bandwidth[5].

The biggest problem for the VIA/SCI card is that MPI/Pro supports no protocol for SCI shared memory, so that ultra–low latencies for short messages and synchronization are not reachable. Because MPI/Pro is a commercial products the code is not available and changes and optimizations of the code are impossible.

2.3. Other related projects

ScaMPI (or Scali MPI[11]) by Scali is a commercial MPI implementation for SCI connected clusters of workstations. ScaMPI is a thread safe, high–performance library but there is no VIA support available.

MVIA[12], a project of the National Energy Research Center Scientific Computing Center at Lawrence Berkeley National Laboratory, is a VIA implementation for Packet Engines Gigabit Ethernet cards respectively Fast Ethernet Cards with the Tulip Chips under Linux. Currently the Functional Conformance[7] of the VIPL is implemented so that this project is a good environment for developing a new VIA-MPI library. It is planned to make an MPI for MVIA available with release 2.0[6].

The University of California, Berkeley develops an VIA implementation for Myrinet Cards, the so-called Berkeley VIA[13].

3. Requirements for the new MPI library

As shown in the last section no MPI implementation offers optimal conditions for the VIA/SCI card. But there are many good approaches and concepts which shall influence the design and the implementation of the new library.

3.1. General requirements

The first requirement is the design of an internal interface which separates the hardware dependent functionality in a special layer (so–called device) similar to MPICH. In contrast to MPICH it is necessary to provide an excellent support for multiple devices because clusters of workstation often use heterogenous networks, e.g. our research cluster OSCAR[14]. The optimal multiple device support involves a special design of the hardware independent part. The management of the network information and the decision which devices are necessary for an MPI application requires a global instance, a so–called daemon. Additionally, the daemon decides where to start the MPI application on the strength of the demand of the application and so realizes a static load–balancing. Some approaches for multiple devices can be found in [15].

An other general requirement is to support the threaded programming model which implies that the MPI library has

[5] up to over 100 MB/sec on 400 Mhz Xeon Pentium II system with GigaNet GNN1000 network

[6] Indeed the current release is 1.0 so that the MPI should take a while.

to be thread–safe. Many modern applications[7] use this technique for fine-grain parallel processing.

An enhancement of the last requirement is the decision to use a threaded design for the new MPI library. Thereby computation and communication can take place simultaneously and communication progresses without MPI calls, too. To get this features it is necessary to implement the network connection in the devices as an independent thread.

The last general requirement is to put the MPI–2[16] functionality into the design because features like the dynamic process management are requested by the applications. It is not our first intention to implement the complete MPI–2 functionality but it is better to design the full standard than to change the design later.

3.2. Special requirements for VIA/SCI

The necessity of registration and deregistration memory for VIA requires a VIA memory management of the data buffers. There are 3 kinds of data buffers:

Data memory of the MPI library This memory is allocated and deallocated by the MPI library and not by the application. Usually, such memory is used for buffered Send/Receive or for noncontiguous data types that have to be packed before communication.

User memory for persistent comm. Memory for persistent operation has to be registered once and thereafter the memory will be used time and again[8].

Normal user memory These buffers are allocated by user for a communication operation and may be used only once in the most cases.

When the free registration memory runs off the VIA memory management has to decide which memory should be chosen for deregistration. The best solution is to select the memory with the smallest probability for reuse and so normal user memory will be chosen (if such memory is registered already).

Another problem is the partition of the registration memory between different MPI applications on a node. A fair solution requires the use of the global instance daemon. The daemon should deliver a initial registration buffer size which could be enlarged or reduced depending on start and finish of other MPI applications. For the exchange of such information a permanent connection to the daemon is required.

As mentioned above the VIA/SCI cards supports SCI shared memory as well as VIA. So it is very useful to support the following protocols for different message sizes:

- A shared memory protocol that will be used for short messages.
- A protocol that uses the VIA Send/Receive mechanism. This protocol for medium size messages should require a small amount of synchronization only in exchange to an additional copy operation (on receiver-side).
- A protocol that uses the VIA RDMA mechanism. It will be require more synchronization effort then the previous one. But this protocol will not require the additional copy operation and allow zero–copy for large messages.

4. The design of CHEMPI

4.1. Raw structure

Deducing from the requirements described in the last sections we now present the design of CHEMPI. Its raw structure is illustrated in Figure 4.

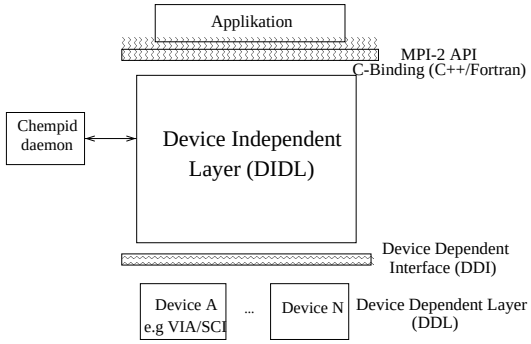

Figure 4. Raw structure of CHEMPI

CHEMPI is divided into two main parts the so-called device independent layer (DIDL) which is discussed in 4.2 and the so-called device dependent layer or device (an example of a device is introduced in section 5). A short description of the interface between the two layers can be found in section 4.3.

As mentioned in 3.1 the CHEMPI daemon (short chempid) distributes MPI applications and information about the network. Additionally the chempid provides an administration network which can be used to set up the connections in the device dependent layer.

4.2. Device independent layer

The functionality of the device independent layer is subdivided into the following modules:

[7] e.g. physical applications of our SFB 393

[8] Strictly speaking, a reusable or persistent request is created and start address of the buffer, size and data type is saved in the request. This request can be used until it is freed explicitly.

- collective operations — a mapping of the collective operations, like Barrier or Broadcast to point–to–point communication
- communicators and groups — handling of groups, contexts and communicators
- connection handling — management of the network, decides which devices have to be used
- daemon binding — the connection to the chempid
- datatypes — provides predefined and user defined datatypes
- environment — everything for the management of startup and shutdown and useful routines for administration
- error handling — central handling and logging of errors and warnings
- memory management — management of system communication buffers
- onesided communication — mapping of the MPI–2 onesided communication to point–to–point communication and system messages
- parallel IO — a mapping to point–to–point communication
- process management — management of the MPI tasks and threads
- point–to–point communication — sending and receiving of messages in different modes
- system messages — handling of system internal messages[9]
- topologies — creation and handling of virtual topologies

4.3. Device dependent interface

Writing a device for CHEMPI should be an easy task, so only a small part of the device dependent interface (DDI) is required (so so–called core functions). The major part of the DDI consists of optional functions which could be implemented to increase the performance. Part of the core functionality are:

- initialization and shutdown
- basic point–to–point communication (synchronous/normal and blocking/nonblocking)
- sending and receiving system messages

And the optional functionality has the following parts:

- collective operations
- onesided communication (recommended for shared memory networks like SCI)
- other point–to–point communication modes
- memory management

5. The VIA/SCI Device

5.1. Initialization

First of all the initialization has to bring up the VIA environment (e.g. opening of the Network Interface Controller and creation of a protection tag). Thereafter the main tasks are the creation and the connection of VI‘s respectively the export and import of the shared memory between the MPI tasks.

The VIA client/server model is used for connecting the VI’s. The different protocols for data transfer (explained in the next sections) require several VI’s so that two VI’s are connected between each couple of MPI tasks during the initialization.

For the creation of the shared memory the VIPL extensions described in [17] are used. Since remote read is an expensive operation the synchronization takes place using shared memory between pairs of tasks instead of global shared memory pages on a master node.

For all VI’s and the shared memory of a MPI task a unique protection tag is used.

An example of connections established after the initialization for four nodes is shown in Figure 5.

5.2. Community of the communication protocols

The basis of all communication protocols used in the VIA/SCI device is the SCI global shared memory.

Every message is represented by a so called message info struct. This struct consists of:

- tag
- communicator id
- message id
- protocol type
- protocol specific data

[9] System messages are used to provide a routing of messages inside an MPI application. A detailed description can be found in [15].

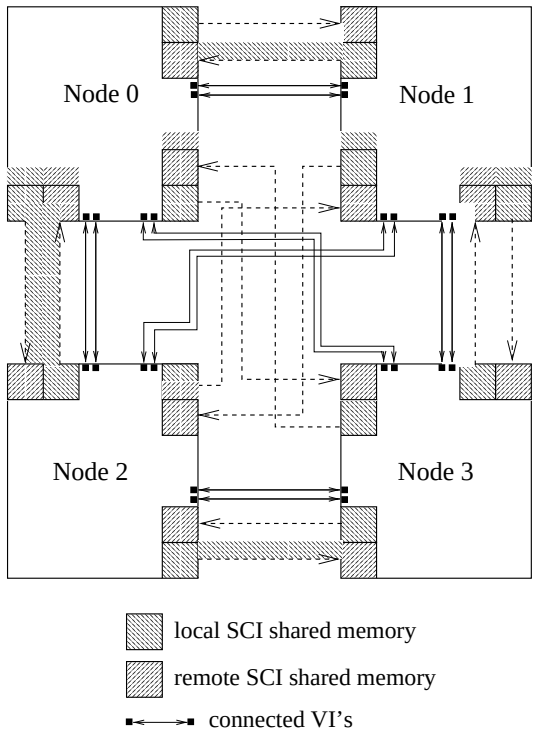

Figure 5. VIA/SCI device after initialization

This message info struct is located in the shared memory at receiver side and is filled by the sender because the receiver will need this information to determine how to get the data. Tag and communicator id are MPI specific data, message id is an internal number and manages the message ordering. The protocol type signals the receiver which protocol is used for data transfer. The protocol specific data is explained in the next sections.

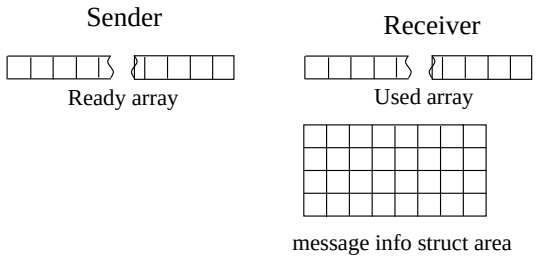

Figure 6. Structures for synchronization

Figure 6 shows the structures used for a simple synchronization. As all structures are located in the shared memory area, both sender and receiver are able to write and read the structures[10]. Each message info struct has an entry in the used array and in the ready array. The management of the message info struct area is done by a module on senderside.

Following the rough plot of a communication operation on senderside:

1. get a free message info struct from the management
2. fill in the message info struct and set the message info struct active in the used array
3. transfer the data with the specified protocol
4. wait for the completion with the ready flag in the ready array

And on receiverside:

1. search the first matching entry in the message info struct area of the sender[11]
2. if not found wait for a matching entry
3. transfer the message in the receiver buffer
4. signalize the sender that the operation is completed via the ready flag

Depending on the transfer protocol some steps will be changed.

In the case of receive from MPI_ANY_SOURCE all message info struct areas must be searched in step 1 respectively step 2 instead of the message info struct of the sender only.

5.3. Shared memory protocol

The shared memory protocol uses a dedicated shared memory buffer for data transfer. For this reasons the message has to be copied into the shared memory on senderside (concurrently this is the copy operation to the receiver) and from shared memory into the user buffer on the receiverside.

The management of the shared memory for the data transfer is done by a shared memory module (as part of the VIA/SCI Device) on senderside. The terms *allocate* and *free* have a special meaning and should not be confused with the system memory management. Allocate means that the caller gets a piece of shared memory from the shared memory module and free means that the caller give back the memory to the shared memory module. The protocols of the previous section get the following enhancements:

In step 2 on senderside: Before setting the message info struct active a buffer is allocated in the shared memory and the user buffer is copied into this buffer. The protocol specific data contains the location of this buffer as an offset to the start of the shared memory and the

[10] Since a remote read is an expensive operation in the SCI environment only remote writes and local reads are used.

[11] This area is local on the receiver node.

size of the buffer. Now the active flag in the used array signalizes the receiver that the data transfer is finished already.

In step 4 on senderside: When the sender detects that the readyflag is set it frees the memory buffer allocated in step 2.

In step 3 on receiverside: When the receiver detects a matching entry in the message info struct area it has to copy the data from the shared memory into the user buffer and set the ready flag.

5.4. One Copy VIA protocol

As mentioned above the VIA Send/Recv mechanism requires that the receive descriptor will be posted before the send descriptor. For this reason the receiver has to post a sufficient number of receive descriptors with fixed buffer size M. Each descriptor has a unique number. The receiver writes the number of the descriptor posted lastly in the shared memory of the sender. Hence the sender could determine the minimal number of posted but unused descriptors. Likewise the sender writes the number of the lastly posted send descriptor in the shared memory of the receiver so that the receiver can post new receive descriptors if the number of posted but unused descriptors reaches a minimal value.

The following enhancements are necessary for the one copy VIA protocol:

In step 2 on senderside: Before any data transfer will be done the sender has to check if the memory is already registered in the VIA memory management and register it if not. Thereafter the VIA memory will be set used so that the VIA memory will not be deregistered during the data transfer. Than the sender splits the message in N chunks with size M[12] and tests if there are enough free descriptors posted by receiver. If not it sets a flag in the receiver shared memory to indicate that more receive descriptors should be posted and waits until the number of posted descriptors is >= N. The sender fills the protocol specific data with the number of the first descriptor X and the number of the last descriptor Y. Afterwards the sender posts every chunk in an individual descriptor and sets the used flag.

In step 4 on senderside: When the sender detects the ready flag it sets the memory unused in the VIA memory management so that the memory could be deregistered if more VIA memory will be needed.

In step 3 on receiverside: When the receiver find a matching entry for his request in the message info struct area it completes the receive descriptors X to Y. Thereafter the receiver copies the data from the VIA Buffer into the user buffer and sets the ready flag.

The one copy VIA protocols is described graphically in figure 7.

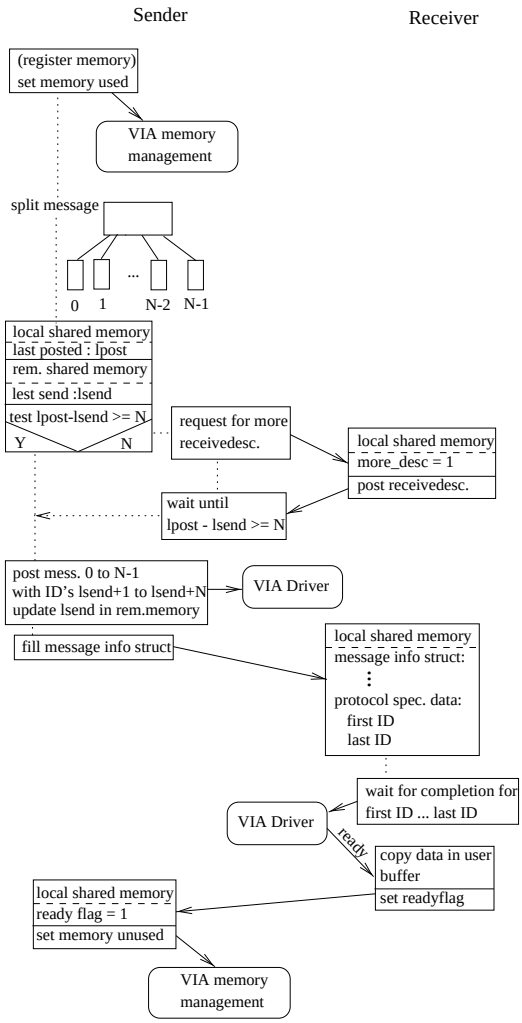

Figure 7. One copy protocol

5.5. Zero Copy VIA protocol

The term Zero Copy means that the data transfer requires no additional copy operations neither on senderside nor on receiverside. The user buffers on senderside as well as on receiverside have to be registered for this reason. The condition to post the receive descriptor before posting the send descriptor demands a higher synchronization effort in comparison with the one copy protocol in the last section.

For the purpose of synchronization a new structure, the so called posted_ring (a ring of message ID's) is introduced.

[12] certainly the last chunk could be smaller

The posted_ring is necessary to justify the descriptor postings on senderside and receiverside. The receiver controls this justification that implies that the structure is located in the sender shared memory and written by the receiver.

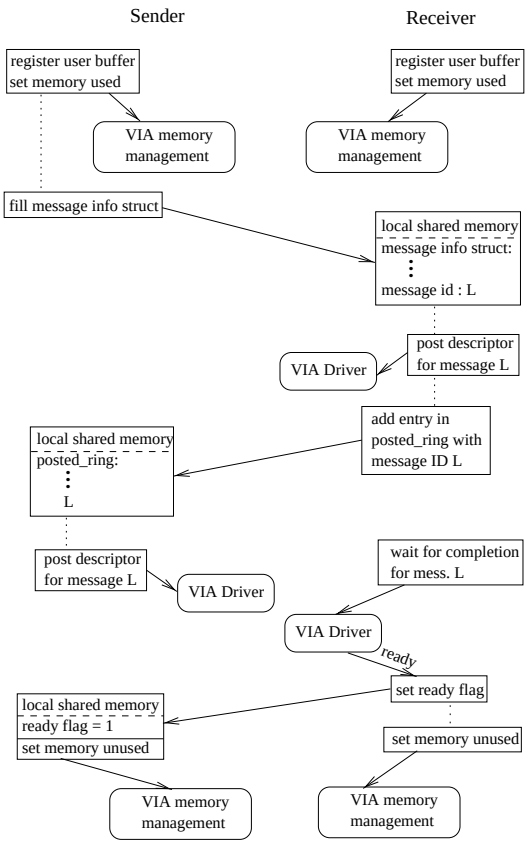

Figure 8. Zero copy protocol

The following enhancements are added to the protocol in section 5.2:

In step 2 on senderside: First the sender has to register (if necessary) the user buffer and to set this memory used (like in the one copy VIA protocol). Thereafter the sender fills out the message info struct[13] and sets the structure active. Here setting the message info struct active has the meaning that the message is ready for send (not that the message is already sent as in the other transfer protocols).

In step 3 on senderside: When the sender detects a new entry in the posted_ring it posts the descriptor for the message belonging to the message ID of the new entry.

In step 4 on senderside: After detecting the readyflag the user buffer is set unused in the VIA memory management.

[13]This protocol does not need protocol specific data.

In step 1 respectively 2 on receiverside: When the receiver finds a matching entry it has to register the user buffer and set the registered memory used (see step 2 on senderside). Thereafter it builds a receive descriptor for the user buffer and posts this descriptor. Afterwards the receiver inserts an entry with the message ID belonging to the posted receive descriptor in the posted_ring to signalize the sender to post the matching send descriptor.

In step 3 on receiverside: After completing the receive the user buffer has to be set unused in the VIA memory management.

A graphical description of the zero copy VIA protocol is depicted in Figure 8.

6. Future work

In this paper we have presented some of the main concepts of CHEMPI, but especially the field of the MPI–2 functionality lacks the implementation. So the next step in the device independent layer is the implementation of this functionality. But since this paper is dedicated to the VIA/SCI device the next steps for the device should be represented here.

The next task is the evaluation of the protocols with the real hardware[14] and the determination of the switching points for the different protocols.

Another important part of the future work is the implementation of collective operations because VIA as well as SCI offer excellent features for the implementation of, e.g., a barrier (SCI) or a broadcast (VIA).

References

[1] IEEE Computer Society. IEEE Standard for Scalable Coherent Interface (SCI) IEEE Std 1596-1992, August 1993.

[2] Mario Trams. Design of a system–friendly PCI–SCI Bridge with an optimized User–Interface. Diploma Thesis, Dept. of Computer Science, University of Technology Chemnitz, 1998.

[3] Mario Trams and Wolfgang Rehm. A new generic and reconfigurable PCI-SCI bridge. To be published in proceedings of SCI Europe'99, Toulouse, September 1999.

[4] 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*

[14]The development of the VIA/SCI card is still not finished.

System Principles, Copper Mountain, Colorado, December 1995.

[5] The SHRIMP Project. Scalable High–performance Really Inexpensive Multi–Processor. http://www.cs.princeton.edu/shrimp.

[6] Intel, Compaq, and Microsoft. Virtual Interface Architecture Specification V1.0. http://www.viarch.org.

[7] Intel Cooperation. Virtual Interface Architecture (VI) Implementation Guide, May 1998.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High–Performance, Portable Implementation of the MPI message–passing standard. Argonne National Laboratory and Mississippi State University, 1996.

[9] Message Passing Interface Forum. MPI: A message–passing interface standard Vers. 1.1. http://www.mcs.anl.gov/mpi/standard.html, June 1995.

[10] Rossen Dimitrov and Anthony Skjellum. An Effizient MPI Implementation for Virtual Interface (VI) Architecture–Enabled Cluster Computing. In *Proceedings of the Third MPI Developers Conference*, March 1999.

[11] http://www.scali.com/.

[12] http://www.nersc.gov/research/FTG/via/.

[13] Philip Buonadonna, Andrew Geweke, and David Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of SC'98*, November 1998. http://www.cs.berkeley.edu/ philipb/via/sc98/paper/.

[14] http://www.tu-chemnitz.de/informatik/RA/oscar/ oscar.html.

[15] Sven Schindler and Wolfgang Rehm. Multiple devices under MPICH. In *Proceedings of the workshops ARCS'99*, October 1999.

[16] The MPI Forum. The MPI message-passing interface standard Vers. 2.0, May 1998.

[17] Friedrich Seifert. Development of system software to integrate the virtual interface architecture (via) into the linux operating system kernel for optimized message passing. Diploma Thesis, Dept. of Computer Science, University of Technology Chemnitz, 1999.

A NEW ARCHITECTURAL CONCEPT FOR HIGHLY EFFICIENT MESSAGE-PASSING ON PCI–SCI NETWORK INTERFACES

Wolfgang Rehm[✠], Mario Trams[✠], Daniel Balkanski[✠], Stanislav Simeonov[✠✠]
{rehm, mtr, danib}@informatik.tu-chemnitz.de, stan@bfu.bg

Abstract

SCI has many advantages compared with other modern high-speed interconnections that makes it very attractive for building cost-effective clusters from standard computer systems and PCI–SCI bridges. In this paper we present some ideas for new architecture of such communication hardware, which will combine the benefits of SCI distributed shared memory with high system throughput and low latencies of protected user level DMA of the Virtual Interface Architecture (VIA). The final goal of our research is to develop PCI–SCI Bridge with completely new design that will be very suitable for building inexpensive clusters optimized for message passing applications.

Keywords*: Message passing, Cluster Computing, Protected User–Level DMA, Distributed Shared memory, Scalable Coherent Interface (SCI), Virtual Interface Architecture (VIA), Network Interface.*

I. Motivation and Introduction

Scaleable Coherent Interface (SCI) [1] is modern communication technology with very high bandwidth, extremely low latency, scalable architecture, and support of distributed multiprocessing that allows building of large systems out of many inexpensive building blocks. It is also truly open standard, fully standardized from ANSI, IEEE, and ISO and is a stable standard since 1991.

SCI acts like a modern equivalent of processor/memory/IO bus and Local Area network, which uses a single address space to specify data as well as its source and destination when being transported and in this way implements distributed shared memory with cache coherency. For this, when two tasks share data using SCI, the data remains stored in ordinary variables, with ordinary memory addresses, at all times. Thus processor instructions like Load and Store suffice to address the data for doing computation with it. Load and Store instructions are highly optimized in all processors, and underlying SCI transport mechanism is transparent to the user, performing all the network protocol effectively as a fraction of one instruction. It is a great advantage compared with other modern high-speed interconnections (ATM, HIPPI, etc.), where data is moved as streams. In this kind of stream communications the data must be copied from user home variables to buffers. After call of library routine it's further transferred from the buffers to an I/O interface for transporting as a byte stream. On receiving end, when the data arrives and fills buffers, filled buffers are handed to the operating system (with an interrupt to get its attention). After that the operating system hands the buffers to waiting user task, the user task parses the buffers to find the data and finally copies the data into variables for use in computation again. So its it not surprising that communication latencies of these stream-based channels or networks are typically about 1000 microseconds, because of the many instructions required at each end involved in transfer. For comparison, in case of SCI because data is moved from one computational context to another, without unlabeling the data enroute, typical latencies are under microsecond.

All mentioned above advantages of SCI makes it a preferred communication technology for cluster computing and it is also the choice we made for building our OSCAR [12] cluster. It's is built from standard PC systems connected together with Dolphin's PCI–SCI bridges.

Applications that we running on this system are based on message passing (MPI) [3]. These applications takes big acceleration because of low latencies of underlying distributed shared

✠ From University of Technologies Chemitz, Germany

✠✠ From Burgas Free Univesity, Bulgaria

memory witch is used like communication media for message passing. Major Problem of these MPI applications is that MPI implementations over such hardware produce high CPU utilization while sending long messages.

Standard solution for this problem is usage of DMA instead CPU to perform data transfers in background without CPU utilization. The Dolphin bridges that we use offer such DMA engine, but it is not controllable directly from user tasks without violating the system protection. So every DMA transfer require a kernel call (to prove the process access rights) which introduce unacceptable latencies, killing in this way one of the main advantages of SCI technology.

One of the more promising approaches for removing the operating system kernel from the critical communication paths is the Virtual Interface Architecture (VIA) [2] introduced in December 1997 by Intel, Microsoft and Compaq. It defines relatively high-level, hardware independent mechanisms for access to the network interface from user level ensuring in the same time protection between multiple processes. This is achieved by providing each consumer process with one or more protected, directly accessible interfaces to the network hardware, called Virtual Interface (VI).

VIA also is not an ideal solution. It is targeted mainly to provide mechanisms for high-speed message passing and is not so effective for very short messages. This is because we can't compare the simple memory reference in case of PIO over shared memory with preparation and execution of DMA descriptors in case of VIA. So the goal of our research is not only to implement a combination of these two technologies but also to find ways for improving the VIA using the shared memory concepts of SCI. As end result we want to have communication hardware with low latencies and high throughput for short as well for long message sizes and therefore suitable for building clusters for wide range of message passing applications.

II. Memory Management and Remote DMA mechanism in conventional PCI–SCI bridges.

Before start discussion about proposed by us modified model for protected user level RDMA, we will make brief analysis of the remote DMA model, typically used in PCI–SCI bridges.

In the conventional PCI–SCI bridge designs SCI distributed shared memory is used like a communication media by mapping parts of remote memory from processes in their address space. All these designs until now have the disadvantage that the whole memory protection is based on the standard virtual to physical address translation at CPU level. Thus, memory protection works only for the remote memory accesses by CPU but not for the DMA transfers. For security reasons, the processes can not build and activate DMA descriptors because there is no way to qualify addresses specified by the process. There is another approach called Restricted User Level DMA in which descriptor contents are checked by the kernel only on initial preparation and after this it can be executed from user level whenever and how often is needed. But if DMA descriptors are rarely reused this becomes ineffective.

Besides of miss of user level DMA current bridge designs suffer also from inflexible static memory management. They offer a one-to-one mapping of whole exported memory (which in this case acts like communication memory) in only one memory window. This causes two significant disadvantages:

- Continuos exported regions must also be continuous and aligned to the minimum exportable block size inside the physical address space.
- Exported memory must reside within this window.

It's not possible in this short writing to go in details but we must mention that these two problems make implementation of MPI applications on top of such hardware very difficult and resource wasting.

III. Memory Management and User Level Remote DMA mechanism by VIA

The central point in the VIA are Virtual Interfaces or shortly VI. They allow direct access of VIA NIC to the processes' memory. Applications see four components of VI: Send and Receive Queue and Send and Receive Doorbell. The queues hold descriptors that describe a data transfer request. Doorbells represent a fast mechanism for a process to notify a VIA NIC that work has been placed on his Work Queue. The Doorbell mechanism is protected by operating system – only the operating system is able to establish Doorbell and the VIA NIC is able to identify the owner of VI by the use of Doorbell. In case of native VIA NIC, Doorbells are practically memory mapped hardware registers. Only the two processes associated with pair connected VI's are allowed to exchange memory contents and a VI can be unconnected or connected to one and only one other VI.

VIA offer quite flexible memory management. For communication can be used any part of virtual memory but before use it must be registered to given Virtual Interface.

There are two types of data transfer facilities provided by the VI Architecture. These data transfer models are traditional Send/Receive messaging model and the Remote Direct Memory access (RDMA) model. The RDMA transfer model is more suitable to be implemented in our PCI–SCI bridge design because of the SCI distributed shared memory used like communication media.

In RDMA operations the user process or consumer in terms of VIA has to specify the source and the destination of data transfer which must reside within registered memory regions in local and remote memory. This is achieved by posting a descriptor to Send/Receive Queue of a VI. Memory protection is based on Memory Protection Tags that is assigned to VIs on creation and to Memory Regions on registration to given VI. The VIA NIC only allows a memory access if the Memory Protection Tag of the VI where the descriptor has been posted and the Memory Regions involved are identical. Access that violates this rule result in a memory protection error and no data is transferred. We must mention here that in addition to the protection tag, the VIA specification also defines Read/Write Enable attributes for Remote DMA transfer mode.

IV. New PCI–SCI bridge design with improved Memory Management and Protected User Level DMA

Because the analysis we made [8] showed that usage of efficient and low latencies DMA is very important for overall system performance we implemented in our design a VIA like protected user level DMA with similar Memory Management. We must slightly extend the mechanisms specified by VIA to become suitable for PCI–SCI architecture because DMA engine must check access rights not only to exported (local), but also to imported (remote) memory. We achieve this by adding a Protection Tag to both downstream and upstream address translation and protection tables witches are used for translations from virtual SCI to Global SCI and from virtual PCI (inside the bridge) to physical PCI (inside the host) address spaces respectively.

Introduction of upstream address translation table also gives additional flexibility to the Memory Management scheme and eliminates the mentioned above problems with continuous and aligned to minimal exportable block size exported memory regions. This combined with reduced minimal block size of shared memory leads to more effective usage of host system memory.

Fig. 1 illustrates the principle work of the Protected User Level DMA in our design. It shows flow of address translations of the addresses generated by the DMA Engine while data transfer is performed between two nodes. Every node has implemented two address translation tables because now DMA works between the virtual PCI and the virtual SCI address spaces. In contrast in conventional architectures DMA works between the physical PCI and the virtual SCI address spaces. Node 1, which is initiator of the transfer, activates its DMA Engine. The generated addresses for addressing the local (exported) memory are translated from virtual PCI to physical

PCI addresses. This is so called upstream address translation. In the same time addresses that address the remote (imported) memory are translated from virtual SCI to the Global SCI address space, which is called downstream address translation. On the remote Node 2, the addresses are translated finally to the physical addresses inside the PCI address space by the upstream translation table of its PCI–SCI Bridge. Here no differentiation between transactions initiated by DMA and PIO is made.

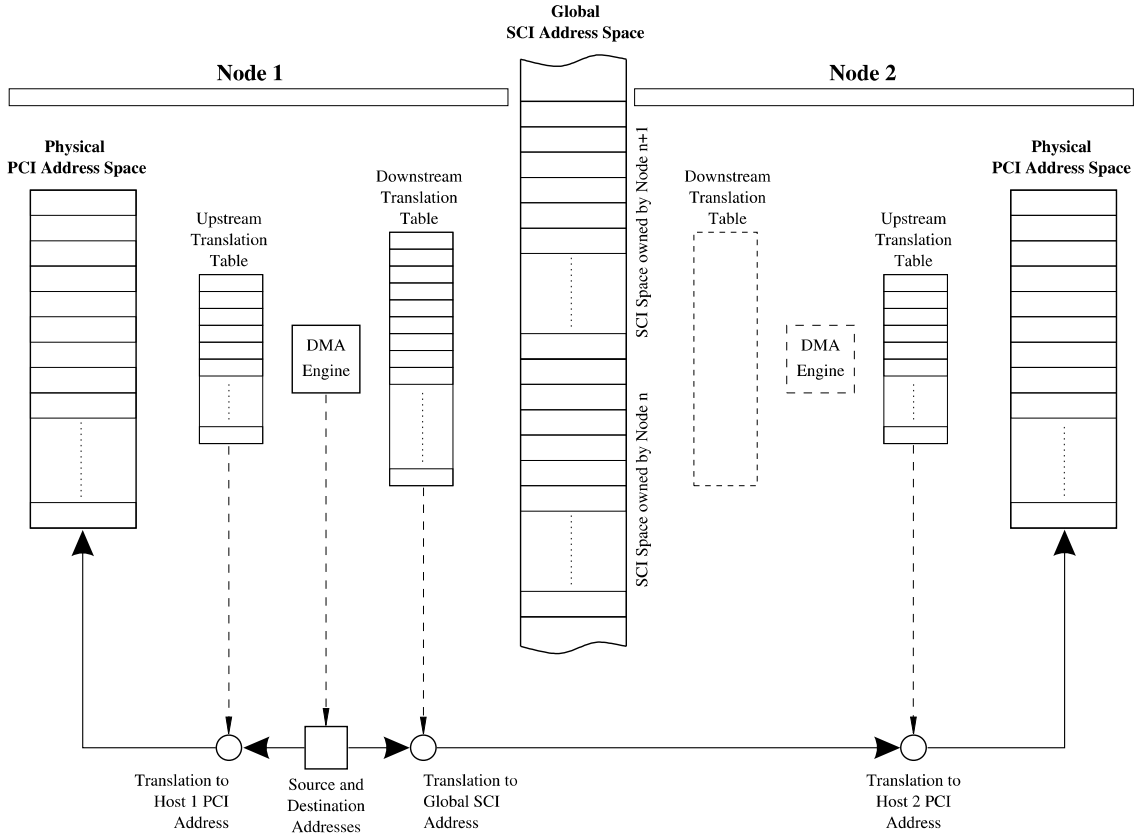

Fig. 1. Address translations during DMA transfer in the improved PCI–SCI bridge design.

Upstream and downstream Translation Tables of the bridges except translated bases contain also the VIA Protection Tags and for this are called also Protection and Translation Tables. The Protection Tags are set by system software when process creates and registers memory regions to its Virtual Interfaces. The DMA engine is capable to initiate transfers only between local and remote regions that belong to the same context that DMA is currently running (and therefore have identical Protection Tags). The Initiator PCI–SCI Bridge doesn't perform Protection Tags check of PIO transfers, because in these cases protection is guaranteed by the memory management system of the host CPU. The Remote node doesn't perform Protection Tag checking neither in case of DMA nor in case of PIO transfers because in both cases this is already done in the Initiator node.

V. Conclusion

In conclusion we can say that this improved Protected User–Level DMA combined with flexible Memory Management helps to increase significantly the system throughput and to closes the latency gap between data transfers by PIO and by DMA. Even more the applications or more exactly communication libraries have flexibility depending on data size and application to decide in which way message should be transferred. Short messages may be transferred by PIO while long may be transferred by DMA. Of course communication oriented applications can use PIO for all or

switch to DMA for much longer messages in this way reducing the latencies and increasing the bandwidth in the cost of more high CPU usage.

To prove in practice our new architecture concept we currently work on creation of a hardware prototype. For more information from architectural aspect about our project and details of hardware realization of the prototype refer to [7], [8]. Currently (Sep, 1999) we have implemented some basic functionality and we continue working on the firmware and low-level device drivers improvement. We plan initially to support two attractive platforms for building cost effective clusters — Intel and Alpha PC. So additional parts of our research group involved in this project work on appropriate High-Level device drivers for integrating the PCI–SCI bridge in memory management system of Linux [9], [10] and also on adoption of MPI–2.0 — high-level communication library [11].

References

[1] IEEE: *Standard for Scalable Coherent Interface (SCI)* IEEE Std. 1596-1992.
SCI Homepage: `http://www.SCIzzL.com`

[2] Intel, Compaq and Microsoft: *Virtual Interface Architecture Specification V1.0.* See also Virtual Interface Architecture Homepage `http://www.viarch.org`

[3] Message Passing Interface Forum: *The MPI message--passing interface standard Rev. 2.0*, May 1998 `http://www-unix.mcs.anl.gov/mpi/`

[4] Dolphin Interconnect Solutions AS: *PCI–SCI Bridge Specification Rev. 4.01*. 1997

[5] Dolphin Interconnect Solutions AS: *PSB-64/66, Features and Benefits.* `http://www.dolphinics.no`

[6] Philip Buonadonna, Andrew Geweke: *An Implementation and Analysis of the Virtual Interface Architecture.* University of California at Berkeley, Computer Science Department, Berkeley, 1998. See also `http://www.cs.berkeley.edu/~philipb/via/`

[7] Mario Trams, Wolfgang Rehm, and Friedrich Seifert: *An advanced PCI--SCI bridge with VIA support.* In: Proceedings of 2nd Cluster-Computing Workshop held in Karlsruhe, Pages 35-44, March 1999. See also: `http://www.tu-chemnitz.de/informatik/RA/CC99`

[8] Mario Trams and Wolfgang Rehm: *A new generic and reconfigurable PCI--SCI bridge.* Proceedings of SCI Europe'99. Toulouse, September 1999.

[9] Friedrich Seifert: *Design and Implementation of System Software for Transparent Mode Communication over SCI,* Student Work, Dept. of Computer Science, University of Technology Chemnitz, 1999. See also:
`http://www.tu-chemnitz.de/~sfri/publications.html`

[10] Friedrich Seifert: *Development of System Software to integrate the Virtual Interface Architecture (VIA) into Linux Operating System Kernel for optimized Message Passing.* Diploma Thesis, Dept. of Computer Science, University of Technology Chemnitz, Informatik. See also: `http://www.tu-chemnitz.de/informatik/RA/themes/works.html`

[11] Sven Schindler and Wolfgang Rehm: *Multiple devices under MPICH*. Proceedings of the PASA'99.

[12] Chair of Computer Architecture Homepage: *OSCAR Project – VIASCI,* University of Technology Chemnitz. `http://www.tu-chemnitz.de/informatik/RA`

Comparing MPI Performance of SCI and VIA

Friedrich Seifert, Daniel Balkanski, Wolfgang Rehm
{sfr,danib,rehm}@informatik.tu-chemnitz.de

Technische Universität Chemnitz
Fakultät für Informatik
Straße der Nationen 62, 09111 Chemnitz

Abstract— **Both the Scalable Coherent Interface (SCI) and the Virtual Interface Architecture (VIA) aim at providing effective cluster communication. While the former is a standardized subnet technology the latter is a generic architecture which can be applied to a variety of physical medias. Both approaches provide user level communication, but they achieve it on different ways and thus, have different characteristics that are independent of the actual implementation. In this paper we report and compare the raw network speed of an SCI and a VIA implementation as delivered by MPI and show how it affects application performance by means of the NAS Parallel Benchmark Suite.**

Keywords—**NetPIPE, NAS Parallel Benchmarks, SCI, VIA, MPI**

I. Introduction

High performing interconnect systems are of key importance for effective cluster computing. Several solutions have been introduced, one of which is the Scalable Coherent Interface (SCI), a technology to build distributed shared memory as well as message passing systems on top of it. Its outstanding characteristic, however, is the shared memory. SCI allows nodes to share parts of their physical address spaces. Hence, in combination with an appropriate paging system processes are able to share their virtual address spaces. Communication can thus take place by the processors issuing simple load and store instructions. That means there is no need for additional protocol layers to transfer data from one node to another, not to mention operating system kernel calls. This gives SCI a clear advantage in terms of latency. Dolphin ICS states 2.3μs for their PCI–SCI–bridges (type D310), which is the version we used for our tests.

The Virtual Interface Architecture [2] also aims at reducing latencies, however, at a higher level. It defines a generic system architecture independently from the physical layer. The main objective of VIA is to move the operating system out of the time critical communication path. For this a VIA NIC provides applications direct access to the network through so called *Virtual Interfaces* or *VI*s. VIs are connected to each other in a point-to-point fashion. The VI Architecture is based on descriptor processing. A VI comprises two *work queues*, one for send descriptors and one for receive descriptors, and a pair of appendant *doorbells*. In order to start a data transfer the sender prepares a descriptor that contains the virtual address of the data to be sent, puts it on the send work queue and rings the doorbell. This is referred to as *posting* a descriptor. Thereupon the NIC starts processing this descriptor, i.e. it reads the data from the user buffer via DMA and sends it through the network to NIC hosting the peer VI. By this time the receiver must have posted a receive descriptor pointing to the destination buffer. When the message arrives the NIC takes the next descriptor from the head of the specified VI's receive queue and writes the data, again by DMA, into memory right into the user buffer. The work queues reside in user memory and the doorbells are mapped into user address space as well, thus there is no kernel call needed to start a data transfer. The completion of a descriptor can be checked either by polling its status or by blocking the process by means of a kernel trap. VIA also provides a remote DMA (*RDMA*) mechanism where the sender specifies both the local and the remote address.

Another characteristic of VIA is that all memory which is to be used to hold descriptors or data buffers must be *registered* in advance. That means that all involved memory pages are locked into physical memory and the addresses are stored in the NIC's *Translation and Protection Table* (TPT).

There are several VIA implementations around. While Berkeley-VIA [9], Compaq's ServerNet I and M-VIA [10] emulate the VIA functionality in software[3], Giganet's cLAN [11], ServerNet II [12], Finisar's FC-VI host bus adapter [13] and Fujitsu's Synfinity CLUSTER are native implementations.

It is important to mention that a combination of both technologies, SCI and VIA, is also possible. In February 1998 Dolphin announced a VIA implementation based on their PCI–SCI bridge [3]. Although they did not publish any technical details of their system, it is very likely that it is a case of software-emulated VIA, as they said no modifications to the existing hardware were needed [4]. Unfortunately, there have been no more public announcements about progress in this direction, neither has this software been made available for testing. In contrast to Dolphin's pure software solution our research group is working on a native, i.e. hardware supported, extended VIA implementation based on SCI [5], [6], [7], [8]. However, until now the project is in an too early stage to be suitable for comprehensive measurements.

In this paper we want to investigate the performance differences between these technologies and how they affect the run time of parallel applications by means of one specific implementation of each, Dolphin's SCI hardware and Giganet's VIA hardware. For comparison to conventional networking technology we also included FastEthernet into our tests. Since all tests are based on MPI [16] the results give an overall assessment

The work presented in this paper is sponsored by the SMWK/SMWA Saxony ministries (AZ:7531.50-03-0380-98/6). It is also carried out in strong interaction with the project GRANT SFB393/B6 of the DFG (German National Science Foundation). We also thank Giganet Inc. and MPI Software Technology Inc. for providing us with their cLAN VIA hardware and MPI/Pro software. Further thanks are extended to Scali Computer AS for providing us with their Scali Software Platform.

[3] M-VIA 2.0 is intended to support native implementations as well.

of the communication hardware and software, which is what an application programmer experiences.

II. The test bed

A. Hardware

Out test environment consisted of nine Pentium III 450 MHz machines. Half the machines were equipped with 512 MB, the others with 384 MB. All of them were connected by switched FastEthernet (using a 3com SuperStack switch) and by SCI in a ring topology based on Dolphin's D310 PCI–SCI–bridges. Using our SCI switch was impossible because it is not supported by Scali's Software (see below). Moreover, eight machines had a cLAN 1000 VIA adapter from Giganet Inc. connected through an eight-port cLAN5000 switch.

B. Software

All machines were running RedHat Linux 6.0 resp. 6.1. For the SCI test we used Scali's SSP version 2.0 [14]. It has been designed for 2D-torus topologies, which rings are a subset of. MP-MPICH from Aachen[15] would have been an alternative MPI implementation but at the time of the tests it was still in an early stage and didn't run stable on our cluster. Besides, it showed performance problems on Linux platforms.

The measurements on Giganet and FastEthernet were done using MPI/Pro by MPI Software Technology Inc., which is the only MPI with support for Giganet's cLAN hardware at the moment.

The NAS Parallel Benchmarks were compiled using the Fujitsu C++ Express Compiler version 1.0 and Fujitsu Fortran 95 Express Compiler version 1.0.

III. Measuring message passing performance using NetPIPE

To measure the pure MPI performance we chose the NetPIPE (Network Protocol Independent Performance Evaluator) benchmark [18]. It was designed as a tool to evaluate different types of networks and protocols and gives answers to the following questions:

- How long does it take to transfer a data block of a given size to its destination?
- What's the maximal bandwidth?
- Which network and protocol transfer a block of given size the fastest?
- What overhead is incurred by different protocols on the same network?

As we examine only one protocol on all networks, which is MPI, the last question is of less interest for us, and we concentrate on throughput and latency.

A. NetPIPE's measuring method

The heart of NetPIPE is a ping-pong loop. A message of a given size is sent out. As soon as the peer receives it, it sends a message of equal size back to the requester. This is repeated several times for each size and the smallest round trip time is recorded. The amount of data is increased with each pass. The number of repetitions depends on it. It is calculated in a way that the transfer lasts for a fixed time (0.5 seconds by default). The program produces a file containing transfer time, throughput, block size and transfer time variance for each message size.

B. Throughput

Figure 1 shows the MPI send/receive throughput of Giganet, SCI and FastEthernet (in Mbit/sec) depending on the message size. There is, as expected, a clear difference between FastEthernet and the other systems.

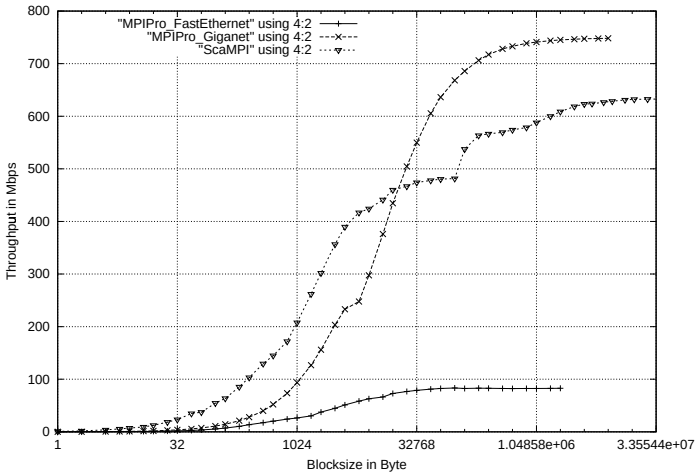

Fig. 1. NetPIPE throughput chart

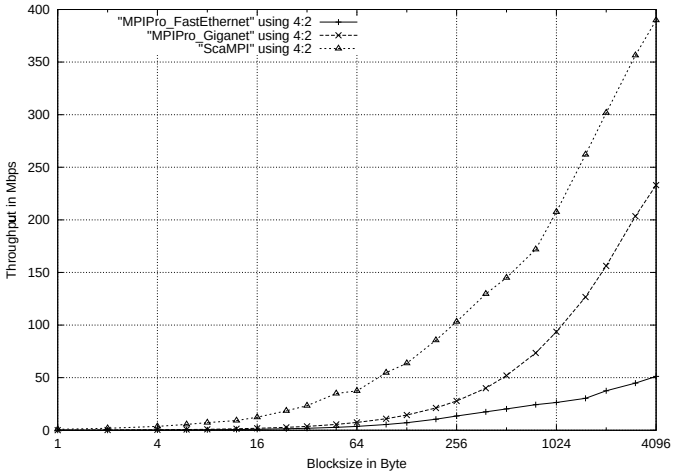

Fig. 2. Small message detail of throughput chart

For large messages MPI/Pro on Giganet delivers the highest performance at 748 Mbit/s or 93.5 MB/s. The kink at 4 KB is caused by switching from eager to long protocol. The former sends messages immediately using VIA send/receive mode which are then buffered as unexpected messages at the receiver. There is an additional copy needed from the MPI-internal buffer to the user buffer. The long protocol is a rendezvous protocol. The sender waits until the receive operation is started. The receiver registers the destination buffer with the VI NIC and sends the destination address to the sender which then transfers the data by RDMA. If the protocol switch point, which is at 4 KB by default, is moved to 8 KB the curve becomes smoother, i.e.

the throughput between 4 and 8 KB is improved a bit. Above that message size the copy operation of the eager protocol is more expensive than the additional message and the memory registration of the long protocol.

ScaMPI's peak performance is about 20 percent lower at 608 Mbit/s (76 MB/s). There is one major kink in the curve at 128 KB. It results from switching from eager protocol to transport protocol, by analogy to MPI/Pro. The default eager buffer size of ScaMPI is just 128 KB. There is one more protocol, the *inline* mode, which is used for messages up to 560 bytes, but this switch point cannot be seen in the curve. This suggests that the ScaMPI developers chose the optimal value in order to get an even performance curve.

MPI/Pro on TCP is able to exploit about 83 percent of FastEthernet's wire speed, i.e. 10.3 MB/s.

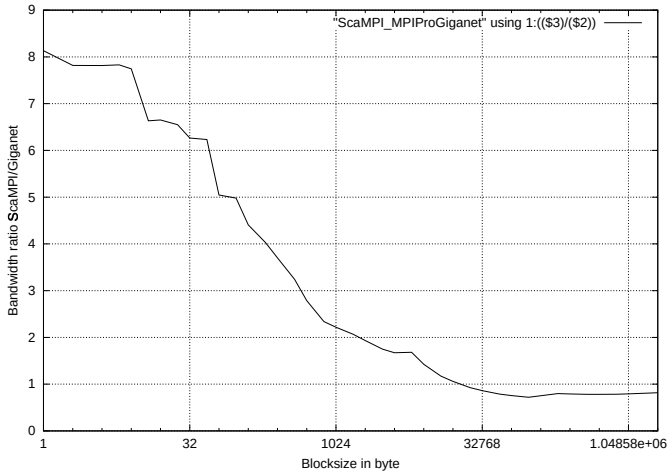

Fig. 3. Bandwidth ratio of SCI and VIA

For small messages the relation between the networks is different. While for large messages Giganet is the fastest for messages up to 16 KB SCI is clearly ahead. As it can be seen from figure 3 SCI is up to eight times faster than VIA. An explanation for this will be given in the following section.

C. Latency

Besides the bandwidth chart NetPIPE offers a so called *saturation* chart. It shows what amount of data can be transfered within a given time. The minimal transfer time represents the latency of the network and the software layers. The following table summarizes the times measured:

Network	Latency
FastEthernet	125 μsec
SCI	8 μsec
VIA	65 μsec

SCI shows the best latencies by far due to its shared memory concept. A simple store operation by the CPU is needed to transfer a data item. The PCI-SCI-bridge transparently translates it into an SCI request and sends it to the target node. In contrast to the value given by Scali (see section I) the lowest raw latencies we could measure on SCI shared memory were about 5 μsec, i.e. MPI imposes an overhead of 3 μsec for very small messages.

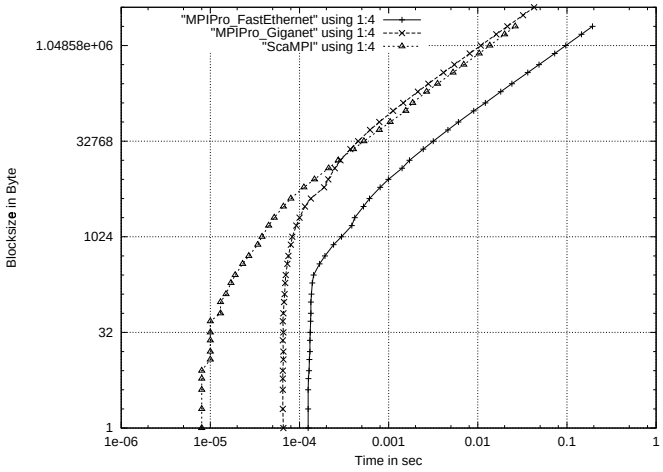

Fig. 4. NetPIPE saturation

The reason for the higher latencies of Giganet can be found in the VI Architecture itself. As described above, VIA uses descriptor based communication. A descriptor must be prepared and posted to the NIC. Then the hardware starts reading the descriptor from main memory by means of DMA. After retrieving the data address it must perform another DMA cycle in order to get the actual data. VIA also provides a mechanism to save the second DMA action: a descriptor may contain up to four bytes of *immediate data*, which is transfered from the send descriptor directly to the receive descriptor. However, this cannot reduce latency essentially since, on the one hand the amount of immediate data is rather small and, on the other hand, the DMA transfer for reading the descriptor remains. Thus, SCI shared memory will always have better short message performance than any VIA implementation on a comparable physical media. With SCI the data itself is written to the NIC, not a one or even two step reference to the data.

But that does not really explain the eight times higher latencies of VIA. The hardware latency of Giganet is as low as 7 μsec after all. Another reason is the synchronization needed between the sender and the receiver. The VI Architecture requires that a receive descriptor be posted before the peer starts the send operation. Otherwise the message would be dropped and the even connection would be broken if the VI connection was established in reliable mode. That's why an MPI implementation for VIA must unconditionally avoid such situations by synchronizing sender and receiver. These synchronizaton messages increase communication startup times.

A third reason for the higher latencies of VIA is the way MPI/Pro checks completion of send and receive descriptors. The version we tested uses waiting mode where the communicating thread is blocked until a descriptor has been completed. Reawakening a process is, of course, more expensive than polling on a local memory location as it it done in ScaMPI. Future versions of MPI/Pro will also be able to use polling mode. A prototype implementation has already shown latencies below 20 μsec [19].

D. Summary

The NetPIPE benchmark revealed clear performance differences between the high speed networks examined. SCI's extremely low startup times provide superior bandwidth for messages up to 16 KB. For larger messages Giganet is faster, but not significantly. In the following section we will examine how these differences impacts particular parallel applications.

IV. NAS Parallel Benchmark

Latency and bandwidth figures are often used for marketing purposes to convince potential customers of the capabilities of a certain network technology. However, the performance gain for parallel applications cannot be derived directly from them. It depends extremely on the communication pattern of the algorithms used. Of course, a high speed network is useless if the tasks exchange only a few bytes every now and then. But even for intensely communicating programs the influence of the network depends on the size of the messages. We have chosen the NAS Parallel Benchmark (NPB) suite version 2.3, which consists of five parallel kernels and three simulated applications from the field of Computational Fluid Dynamics (CFD). They have been designed by NASA Ames Research Center. The benchmarks, which are derived from computational fluid dynamics codes, have gained wide acceptance as a standard indicator of supercomputer performance [20]. Of course, the NAS benchmark suite is limited to specific numerical problems and hence is not representative for other kinds of applications, such as databases or datamining. However, since our group is involved in a Collaborative Research Center "Numerical Simulation on Massively Parallel Computers", we've found this benchmark suitable. We will give the results of all these tests at the end of this document (see figure 7), and we will discuss one benchmark from each group in more detail in this section.

In order to find explanations for the results gained we analyzed the communication of the programs by means of the MPI profiling interface. We wrote wrappers for each MPI data transfer function used by the benchmarks and counted their calls. That was done separately for each power of two of the message size. The findings were quite instructive. So, the EP test, for instance, is really inappropriate for evaluating cluster interconnects as its entire communication consists of three allreduce operations of four to eight byte and one allreduce of 64 to 128 byte, independently on the problem class and number of processes.

A. The IS benchmark

The IS (Integer Sort) benchmark is one of the parallel kernels of the NPB suite. It performs a sorting operation that is important in particle method codes. In such kind of applications particles are assigned to cells and may drift out. They are reassigned to the appropriate cells by means of the sorting operation [21].

A.1 The results

Figure 5 shows the performance for the IS test on FastEthernet, Giganet (both with MPI/Pro) and SCI with ScaMPI. All NPB tests can be run for several sizes: W (Workstation), A, B and C. We were not able to run class C as it required too much memory. IS runs at a power-of-two number of processes, i.e. 2, 4 and 8. We also added the numbers for the serial version to see the benefit of parallelization. The charts show the total million operations per second rate (Mop/s) rate over number of processes.

For all cases MPI/Pro on Giganet is the winner. However, the distance to ScaMPI shrinks as the problem size increases, so that for class B ScaMPI and gets very close to Giganet. There is a performance improvement of factor 2.5 in comparison to FastEthernet. For the latter the two process version is even slower than the serial implementation, and the speedup for eight processes is not even two. It is even worse for the smaller classes. The workstation test with eight nodes is not faster than the serial version which suggests that the communication overhead eats up all the acceleration of the computation by the additional processors.

A.2 The communication pattern

Size	Count	Message Type
4	1	send
4	1	reduce
8	1	reduce
8192	11	allreduce
4	11	alltoall
16777216	11	alltoallv

TABLE I
COMMUNICATION STATISTICS FOR IS CLASS B USING 8 TASKS

As table I shows the IS algorithm mainly relies on *allreduce*, *alltoall* and *alltoallv*, which is a vectorized variant of the all-to-all operation. *allreduce* performs an certain operation on a set of data distributed across all processes whereat all processes get the result. In an all-to-all operation, as the name suggests, each task sends a piece of data to all others, i.e., one such operation comprises $n(n-1)$ individual messages with n as the number of tasks.

The only parameter that changes with the number of processes and the problem class is the size of the *alltoallv* messages. It increases with the problem size, but it is the smaller the more processes are involved since each processor gets a smaller piece of the data as shown in table II.

ntasks:	2	4	8
Class W	2 MB	1 MB	0.5 MB
Class A	16 MB	8 MB	4 MB
Class B	64 MB	32 MB	16 MB

TABLE II
MESSAGE SIZES FOR *alltoallv*

The huge all-to-all messages are most probably the reason for the dramatic differences between FastEthernet and the two high speed networks. For the B class with eight processes a total amount of 896 MB must be transferred at each all-to-all exchange. In our opinion the *allreduce* and *alltoall* operations are of nearly no importance compared with those voluminous ones.

The difference between SCI and VIA are difficult to explain by the measurements given in section III. The performance rise of ScaMPI from class W to A at eight processes could be attributed to the higher bandwidth of about 8.5 percent for 4 MB messages as against 512 KB messages. However, this does not hold true for four processes although the transfer rate for 8 MB is higher than for 1 MB. Hence there must be other nonobvious factors influencing overall performance.

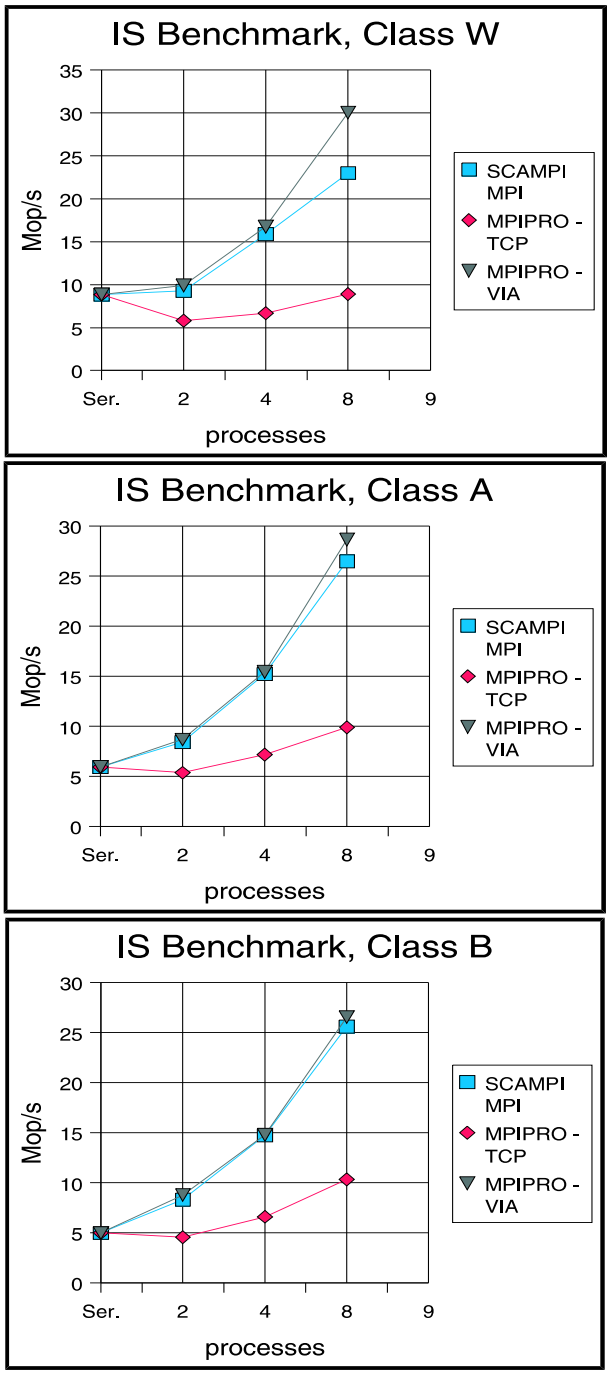

Fig. 5. IS results

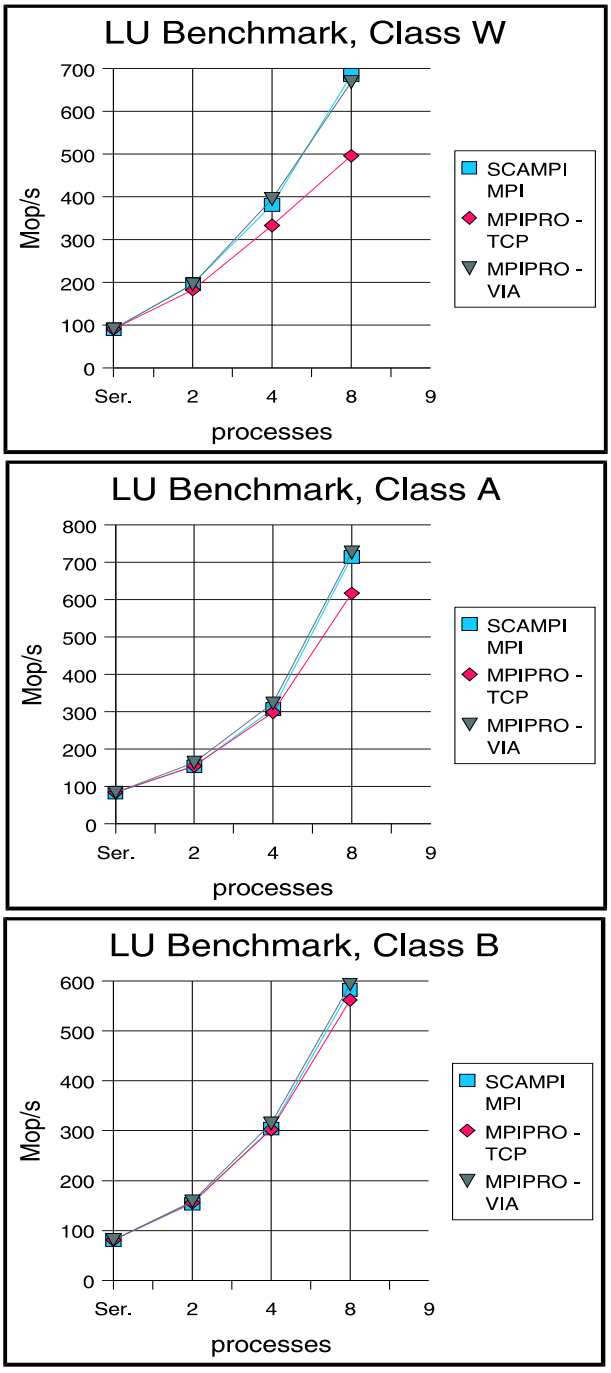

Fig. 6. LU results

B. The LU benchmark

Unlike the name would suggest the LU benchmark doesn't perform an LU factorization but instead employs a SSOR(symmetric successive over-relaxation) numerical scheme to solve a regular-sparse, 5x5 lower and upper triangular system. This problem represents the computations associated with a newer class of implicit CFD algorithms, typified at NASA Ames by the code INS3D-LU [21]. LU is one of the simulated application benchmarks.

B.1 The results

Figure 6 shows the results. Like IS it runs on power-of-two numbers of processes.

All in all there is nearly no difference between the high speed interconnects. For Workstation class on eight processes SCI

is slightly faster than Giganet whereas the things are opposite in the other cases. Further, the advantage of SCI and Giganet over traditional FastEthernet becomes the smaller the bigger the problem size is. For class B there is no significant difference between them any more. This behavior can explained if one considers communication pattern of the algorithm.

B.2 The communication pattern

Size	Count	Message Type
512	9300	send/recv
1024	9300	send/recv
32768	302	send/recv
65536	302	send/recv
4	3	bcast
8	5	bcast
64	1	bcast
8	4	allreduce
64	4	allreduce
	1	barrier

Size	Count	Message Type
1024	25000	send/recv
2048	25000	send/recv
262144	252	send/recv
524288	252	send/recv
4	3	bcast
8	5	bcast
64	1	bcast
8	4	allreduce
64	4	allreduce
	1	barrier

TABLE III
COMMUNICATION STATISTICS FOR LU CLASS W (TOP) AND CLASS B USING 8 TASKS

As can be seen from table IV-B.2 the communication of LU is dominated by send/receive operations. The few *allreduce* and *broadcast* operations can be neglected. For smaller process groups the message sizes grow but the total amount of data transferred remains relatively the same. When the problem size is increased the message sizes as well as the number of messages rise.

Table IV gives the total execution times for ScaMPI in seconds.

ntasks:	2	4	8
Class W	92.0	47.4	26.4
Class A	769	388	167
Class B	3250	1630	856

TABLE IV
TOTAL EXECUTION TIMES FOR LU ON SCAMPI (IN SEC)

The times for the maximum number of tasks are in relation $1 : 6.3 : 32$ for W, A and B. The number of messages, however, rises far slower. The relation is $1 : 1.6 : 2.6$. This explains why FastEthernet achieves nearly the same overall Mop/s rate as SCI and VIA, since runtime is mainly determined by computation in this case.

LU's communication comprises small messages where SCI is faster as well as larger messages where VIA is faster. The total amount of data sent in large messages (above 32 KB) is approximately twice this sent in small messages (below 2 KB). But the bandwidth for the small messages is only half of the large message bandwidth. Thus none of the high speed technologies can take pure advantage.

C. The other benchmarks

The results of all NPB tests are given in figure 7. The numbers shown represent the total Mop/s rate. The BT and SP tests run on a square number of processors while the others need a power of 2 number of processors. This limitation and the fact that we only had an eight port Giganet switch explain the gaps in the table. Although we did not analyze the results of the other tests in the suite [21], we will give a brief explanation of them for the reader's convenience:

Conjugate Gradient (CG) Benchmark In this benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication and employs sparse matrix-vector multiplication.

The Embarrassingly Parallel (EP) Benchmark In this kernel benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, which are generated according to a particular scheme that is well-suited for parallel computation. This problem is typical of many Monte Carlo applications. Since it requires almost no communication, in some sense this benchmark provides an estimate of the upper achievable limits for floating-point performance on a particular system.

3-D FFT PDE (FT) Benchmark In this benchmark a 3-D partial differential equation is solved using FFTs. This kernel performs the essence of many spectral methods. It is a good test of long-distance communication performance.

Multigrid (MG) Benchmark The MG benchmark is a simplified multigrid kernel, which solves a 3-D Poisson PDE. This problem is simplified in the sense that it has constant rather than variable coefficients as in a more realistic application. This code is a good test of both short and long distance highly structured communication.

SP Simulated CFD Application (SP) Benchmark This simulated CFD application is called the scalar pentadiagonal (SP) benchmark. In this benchmark, multiple independent systems of nondiagonally dominant, scalar pentadiagonal equations are solved.

BT Simulated CFD Application (BT) Benchmark This simulated CFD application is called the block tridiagonal (BT) benchmark. In this benchmark, multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5x5 block size are solved. SP and BT are representative of computations associated with the implicit operators of CFD codes such as ARC3D at NASA Ames. SP and BT are similar in many respects, but there is a fundamental difference with respect to the communication to computation ratio.

NPB results for other systems can be found at the NAS Parallel Benchmark home page [21] and also at [22].

V. CONCLUSIONS

Summarizing we must say that both high speed interconnect technologies show significant improvements over traditional FastEthernet connections in terms of raw MPI bandwidth and latency. Comparing the fast networks among themselves ScaMPI is clearly ahead with the bandwidth from 2 to 8 times higher for messages less than 1 KB. It also shows the lowest latency for very small messages. MPI/Pro on Giganet has got its advantages above 16 KB messages.

As the results from the NAS Parallel Benchmark tests show improving overall application performance depends on many factors. The resulting application performance cannot directly be derived from the speed of the cluster interconnection. Only intensely communicating applications can benefit from a faster network. Although ScaMPI has a clear advantage in the small message range the NAS tests do not take advantage of it. Even more VIA shows better results in most of the cases. This is probably because traditional applications tend to use large messages. So in order to decide which network to chose one should know the communication behavior of the applications intended to be run on the cluster. A not less important criteria of course is the price of the entire system.

REFERENCES

[1] Dolphin Interconnect Solutions AS: *PCI SCI-64 Adaptor Card, Features and Benefits*, http://www.dolphinics.com/dics_pci-sci64.htm.

[2] *The Virtual Interface Architecture Specification. Version 1.0*, Compaq, Intel and Microsoft Corporations, Dec 16, 1997, Available at http://www.viarch.org.

[3] *Dolphin Interconnect Server Cluster Test shows optimal Virtual Interface Architecture (VIA) Performance*, Dolphin Interconnect Solutions Inc., Press Release, San Jose, California, February 1998.

[4] *Dolphin Interconnect Unveils VI Architecture Roadmap*, Dolphin Interconnect Solutions Inc., Press Release, Santa Clara, California, June 1998.

[5] M. Trams, *Design of a system-friendly PCI-SCI bridge with an optimized user-interface*, Diploma Thesis, Chair of Computer Architecture, Chemnitz University of Technology, 1998.

[6] F. Seifert, *Development of system software to integrate the Virtual Interface Architecture (VIA) into the Linux operating system kernel for optimized message passing*, Diploma Thesis, Chair of Computer Architecture, Chemnitz University of Technology, October 1999.

[7] M. Trams, W. Rehm, D. Balkanski, S. Simeonov, *Memory Management in a combined VIA/SCI Hardware*, In proceedings to PC-NOW 2000, International Workshop on Personal Computer based Networks of Workstations held in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2000), May 1-5 2000, Cancun/Mexico.

[8] Sven Schindler, Wolfgang Rehm, Carsten Dinkelmann, *An optimized MPI library for VIA/SCI cards*, In: Proceedings of the Asia-Pacific International Symposium on Cluster Computing (APSCC'2000) held in conjunction with the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPCAsia2000), May 14-17, 2000, Beijing, China.

[9] P. Buonadonna, A. Geweke, and D. Culler, *An Implementation and Analysis of the Virtual Interface Architecture*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1998.

[10] *M-VIA: A High Performance Modular VIA for Linux*, http://www.nersc.gov/research/FTG/via.

[11] *cLAN for Linux*, http://www.giganet.com/products/indexlinux.htm.

[12] *Compaq's ServerNet II*, http://www.servernet.com.

[13] *New Fibre Channel Virtual Interface Architecture Host Bus Adapter for Windows NT Cluster Computing from Finisar Corporation*, Finisar Corporation, Press Release, Mountain View, California, November 1998.

[14] *SCALI - Scalable Linux Systems*, http://www.scali.com.

[15] textslMP-MPICH: Multi-Platform MPICH, http://www.lfbs.rwth-aachen.de/users/joachim/MP-MPICH.

[16] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, 1994, http://www.mpi-forum.org/docs.

[17] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, 1997, http://www.mpi-forum.org/docs/mpi-20.ps.

[18] Quinn O. Snell, Armin R. Mikler and John L. Gustafson, *NetPIPE: A Network Protocol Independent Performance Evaluator*, Ames Laboratory/Scalable Computing Lab, Ames, Iowa 50011, USA.

[19] Rossen Dimitrov and Anthony Skjellum, *Impact of Latency on Applications Performance*, MPIDC 2000.

[20] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow, *The NAS Parallel Benchmarks 2.0*, Report NAS-95-020, December, 1995.

[21] Subhash Saini and David H. Bailey, *NAS Parallel Benchmark(Version 1.0) Results 11-96*, Report NAS-96-18, November, 1996, http://www.nas.nasa.gov/Software/NPB.

[22] *NAS Serial Benchmark Performance at NERSC*, http://www.nersc.gov/research/FTG/pcp/performance.html.

ScaMPI	W						A						B					
	Ser.	1	2	4	8	9	Ser.	1	2	4	8	9	Ser.	1	2	4	8	9
BT	69.48	64.22		249.62		539.10	66.49			243.30		539.37				248.00		537.21
CG	32.56	31.11	50.18	88.98	151.36		32.11		50.74	94.72	195.88		22.13		53.03	88.74	199.48	
EP	1.35	1.26	2.52	4.84	9.24		1.34		2.52	5.14	10.28		1.35		2.54	5.04	10.40	
FT	48.18	38.72	63.39	119.38	216.72		50.26		62.36	118.72	248.12					125.86	238.24	
IS	8.85	5.66	9.28	15.90	23.00		5.94		8.45	15.22	26.48		5.00		8.35	14.74	25.60	
LU	90.30	82.71	196.25	381.50	685.68		84.09		154.82	307.70	714.72		81.14		153.45	304.38	583.04	
MG	63.16	49.93	94.43	157.72	304.36		52.91		86.23	156.62	349.92		56.72		92.36	166.58	375.44	
SP	55.90	50.96		206.22		441.36	55.11			190.22		400.05	53.27			183.82		421.43

MPI/PRO TCP	W						A						B					
	Ser.	1	2	4	8	9	Ser.	1	2	4	8	9	Ser.	1	2	4	8	9
BT	69.48	65.12		225.02		422.15	66.49			238.12		494.46				248.58		512.96
CG	32.56	27.27	49.12	55.12	80.44		32.11		55.23	81.04	110.68		22.13		45.42	72.36	148.56	
EP	1.35	1.31	2.55	5.08	10.20		1.34		2.54	5.10	10.24		1.35		2.56	5.10	10.24	
FT	48.18	47.15	60.75	93.06	150.64		50.26		68.65	113.22	187.72					117.04	201.36	
IS	8.85	7.10	5.78	6.68	8.92		5.94		5.38	7.18	9.92		5.00		5.38	6.62	10.36	
LU	90.30	88.44	182.69	333.40	495.88		84.09		155.99	299.12	617.12		81.14		155.54	302.50	562.56	
MG	63.16	57.69	81.60	109.86	172.36		52.91		93.22	160.94	335.68		56.72		99.63	173.80	360.56	
SP	55.90	53.07		173.86		309.65	55.11			177.78		298.80	53.27			181.46		357.71

MPI/PRO VIA	W						A						B					
	Ser.	1	2	4	8	9	Ser.	1	2	4	8	9	Ser.	1	2	4	8	9
BT	69.48			261.40			66.49			253.14						251.14		
CG	32.56		58.28	98.68	152.80		32.11		60.23	108.64	172.56		22.13		47.09	89.52	204.40	
EP	1.35		2.56	5.10	10.20		1.34		2.56	5.12	10.20		1.35		2.56	5.12	10.16	
FT	48.18		74.27	139.42	265.64		50.26		81.26	153.88	299.08						302.04	
IS	8.85		9.90	16.78	30.04		5.94		8.70	15.46	28.64		5.00		8.60	14.82	26.56	
LU	90.30		194.87	395.44	669.28		84.09		164.20	322.96	726.76		81.14		158.72	314.86	593.48	
MG	63.16		106.21	167.12	284.96		52.91		99.93	181.26	403.64		56.72		106.54	192.10	430.16	
SP	55.90			217.22			55.11			201.76			53.27			190.18		

Fig. 7. Overview of NAS Parallel Benchmark results. The numbers represent the total Mop/s rate. The "Ser." columns contain the results of the purely serial versions of the tests, while the columns labeled "1" are for result from running the parallel program on one processor.

Design Choices and first Results of our VIA–capable PCI–SCI Bridge

Mario Trams, Ralph Schlosser, and Wolfgang Rehm
Technische Universität Chemnitz
{mtr,sral,rehm}@informatik.tu-chemnitz.de

Abstract

Both the Scalable Coherent Interface (SCI) [7] and the Virtual Interface Architecture (VIA) [8] aim at providing effective cluster communication systems. In previous publications we showed that SCI and VIA can be efficiently combined to form a new communication architecture (e.g. [1, 2]).

In this paper we show that despite of having adopted an flexible relatively slow FPGA–based design an appropriate architectural solution may lead to a PCI–SCI hardware of which the final performance can keep race along with commercial pendants. We also describe concrete design choices we made for our hardware and present first performance measurements that demonstrate the strength of it.

1 Introduction

Although SCI is intended for distributed shared memory, a lot of groups [4, 6] including our working group [3] use SCI as low–level layer for message passing. This results in very low messaging latencies.

The key points that our design shall realize in addition to features offered by current commercially available PCI–SCI hardware are:

- Protected User–Level DMA for large block transfers to unload the CPU whenever possible.
- An improved memory management to increase flexibility for exporting local memory to remote nodes and thus making real zero–copy possible.

This paper is intended to provide some concrete information about the internal operation of the hardware, especially the internals of the FPGAs used in our design where the majority of the know–how is concentrated. Also we are able to present some measurements. Although these measurements don't show the final performance, we can demonstrate that our design can keep pace along with other PCI–SCI hardware.

2 Bridge Internals

As described in [1] the whole custom logic is housed in two FPGAs. Other components are a Dual–Ported Memory for SCI packet storage and a SRAM for other data such as translation tables.

2.1 The SCI FPGA

The job of the SCI FPGA is concentrated on dealing with SCI packets and talking with the SCI Link Controller. This means that there are mainly implemented several queues for incoming and outgoing SCI packets. A very raw overview about the things contained in the SCI FPGA is shown by figure 1.

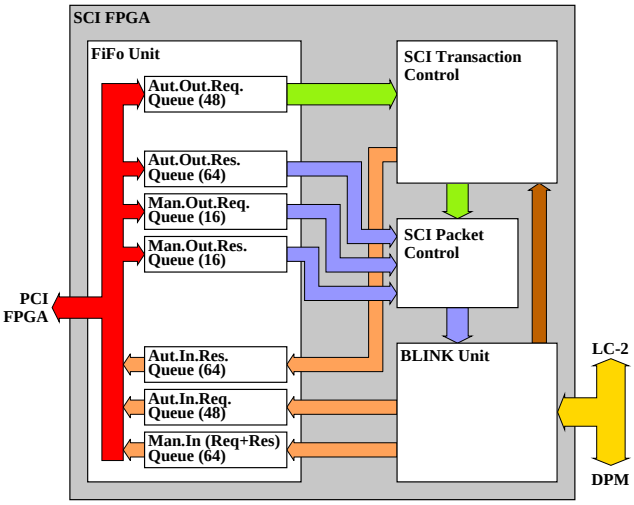

Figure 1: Simplified internal Structure of SCI FPGA

As suggested by the queues contained in the FiFo Unit, there are two basic types of SCI transactions: Manual and Automatic ones.

2.2 The PCI FPGA

This one is indended to implement key features of the whole bridge. The most important functions are translation of PCI into SCI transactions and reverse, Downstream Address Translation for outgoing transactions, Upstream Address Translation that is required to ”virtualize” the exportable memory for exporting any arbitrary memory page rather than a fixed memory portion, and a Protected User–Level DMA Engine including Virtual Interface Architecture functionality (Doorbells, Work Queues, ...) to offer an handy mechanism for user processes to use block–moving DMA instead of the processor for data transmission.

While the first two points are known from current commercially available PCI–SCI hardware [5], the remaining ones are dedicated only to our hardware solution. Figure 2 gives a view of the PCI FPGA internals.

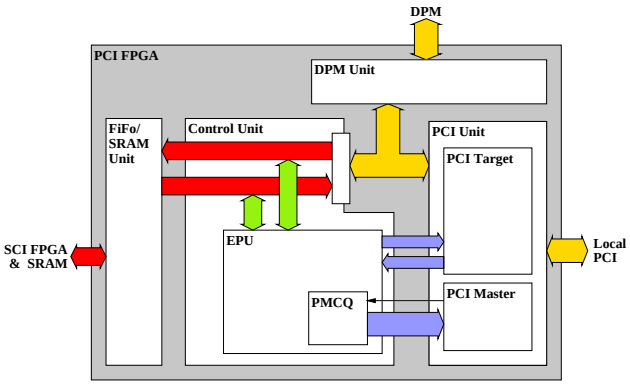

Figure 2: Simplified internal Structure of PCI FPGA

An important unit here is that one labeled with ”EPU”. This stands for *Embedded Processing Unit* and is working like a small processor. The EPU is intended to deal with relatively complex hardware operations.

3 Remote Write Latency

Although the FPGAs are not fully implemented yet (Sept. 2000) and only the manual packet mode is ready, we are able to present some values for the expected latency on remote memory accesses. The measurements were made on a 533MHz Alpha 21164 (EV56) system based on the Samsung UX board with 64Bit PCI bus and a 450MHz Pentium III 440BX system. We used the ability of the hardware to send out a pre-prepared SCI packet manually, that is interpreted as an automatic packet at the receiver that generates the PCI transaction automatically.

For a single remote write operation we could measure a hardware latency of **2.04μs** (PCI–to–PCI). In case of the software latency (CPU–to–CPU) we measured about **2.78μs** on the Pentium system and **2.91μs** on the Alpha. As comparison, Dolphin’s hardware achieves a software latency of **2.3μs** on the same Pentium system.

However, our final hardware latency will increase slightly for real remote write operations and should be around **2.5μs**.

4 Conclusions

We gave a very rough description of both FPGAs we use in our design. The latency values we presented show that our proof–of–concept for new techniques applied to SCI is not very far behind fast commercial ASIC implementations.

Latest information can be obtained from our web page: `www.tu-chemnitz.de/~mtr/VIA_SCI/`

References

[1] M.TRAMS, W.REHM: *A new generic and reconfigurable PCI–SCI bridge.* In: Proceedings of SCI–Europe’99 held on 2nd/3rd of September 1999 in Toulouse, Pages 113–120. See also: `wwwbode.in.tum.de/events/sci-europe99/`

[2] M.TRAMS, W.REHM, D.BALKANSKI, S.SIMEONOV: *Memory Management in a combined VIA/SCI Hardware.* In: Proceedings of PC–NOW 2000, Intl. Workshop on Personal Computer based Networks of Workstations held on 5th of May 2000 in Cancun, Mexico. Springer LNCS, ISBN 3-540-67442-X, Pages 4-15.

[3] S.SCHINDLER, W.REHM, C.DINKELMANN. *An optimized MPI library for VIA/SCI cards.* In: Proceedings of the Asia-Pacific International Symposium on Cluster Computing (APSCC’2000)May 14-17, 2000, Beijing, China.

[4] J.WORRINGEN: *SCI–MPICH: The Second Generation* In: Proceedings of SCI–Europe’2000, Munich, Aug. 2000, Pages 11–20.

[5] Dolphin Interconnect Solutions AS: `www.dolphinics.no`

[6] Scali AS — Scalable Linux Solutions: `www.scali.com`

[7] *IEEE Standard for Scalable Coherent Interface (SCI).* IEEE Std. 1596-1992. `www.SCIzzL.com`

[8] Intel, Compaq and Microsoft. *Virtual Interface Architecture Specification V1.0.* `www.viarch.org`

Proposing a Mechanism for Reliably Locking VIA Communication Memory in Linux*

Friedrich Seifert and Wolfgang Rehm
Chemnitz University of Technology
Department of Computer Science
Chair of Computer Architecture
Straße der Nationen 62, 09111 Chemnitz, Germany.
{sfr,rehm}@informatik.tu-chemnitz.de

Abstract

The Virtual Interface Architecture (VIA) [4] is an industry standard specifying how user processes can access networking hardware directly in a protected manner. One characteristic of the VI Architecture is that it requires that all memory used for communication be locked down into physical memory. Above that, the VIA specification explicitly allows memory regions to be registered several times. However, all freely available VIA implementations for Linux either do not reliably lock the memory or they do not allow multiple registrations. In this paper we propose a mechanism for reliably locking VIA communication memory in Linux that meets all requirements. It is based on a recently introduced kernel mechanism, called kiobufs. Although the proposed locking mechanism has been developed for a VIA implementation it can be utilized for any type of user level communication.

1 Introduction

The Virtual Interface Architecture (VIA in short) aims at providing user level DMA (UDMA) for communication hardware, i.e. it allows the NIC to read and write data directly from and to parts of the user address space, thus enabling zero–copy protocols. While U-Net/MM [12], one of the predecessors of VIA, allows communication memory to be swapped out by maintaining a Translation Lookaside Buffer (TLB) on the NIC, which is kept consistent with the kernel page tables, the VI Architecture requires that memory that is to be accessed by the NIC be pinned down permanently. This approach saves the expensive page–in operations during communication. More information about the evolution of the Virtual Interface Architecture can be found in [8, Chapter 2].

In VIA terminology communication memory must be *registered* with the NIC. On the occasion all pages belonging to that memory area must be paged in in case they are not present. Then they must be locked and, finally, the physical addresses are stored in the so called *Translation and Protection Table (TPT)* on the NIC. When such a *memory region* is not needed any more for communication it should be deregistered, whereupon the pages are released and made available for swapping again. It is important to note that the VIA specification explicitly allows a certain memory area to be registered several times. This complicates the memory locking mechanism as we will see later on. Multiple registrations can occur when communication buffers are registered dynamically. This is necessary to implement zero-copy protocols. The networking hard must transfer the data directly from and to the user buffers, the addresses of which are given to the communication library, e.g. MPI. Since any arbitrary user space address can be used, MPI cannot predict it. Neither is it possible to register the whole user space in advance due to resource limitation. Hence, the buffers must be registered on the fly. This is actually a contradiction to the aim of the VI Architecture, namely to remove operating system calls from the communication path but it is the only way to achieve zero-copy. Furthermore, the bad effects can be remedied by "caching" registered regions, i.e. by keeping them registered as long as possible. However, there are still situations where an area must be registered several times, e.g. when it is registered with different protection tags or attributes.

There are several implementations of VIA for Linux around, such as Berkeley-VIA [3], M-VIA [1] and Giganet's cLAN [2]. Another VIA project is being con-

*The work presented in this paper is sponsored by the SMWK/SMWA Saxony ministries (AZ:7531.50-03-0380-98/6). It is also carried out in strong interaction with the project GRANT SFB393/B6 of the DFG (German National Science Foundation).

ducted by our research group. Our aim is to provide superior message passing performance in terms of latency, bandwidth and CPU usage by combining the Virtual Interface Architecture and the Scalable Coherent Interface (SCI) [9, 8, 10, 6]. Of course, we have also been faced with the task of locking memory. We took a different approach than the other VIA groups, however, our first implementation still did not meet all requirements.

In the following we will discuss the challenges in locking memory and how they can be mastered. We will start with a description of the Linux swapping mechanism in section 2. After that we will point out the weak spots of the different implementations. In the fourth section we will propose a solution that is based on a recent kernel mechanism, called kiobufs, and evaluate its performance.

2 The Linux swapping mechanism

In this section we will take a look at how Linux swaps memory pages and where are the points of departure for locking them down.

2.1 Administration of physical pages

The Linux kernel keeps a so called `mem_map_t` data structure for each physical page in the system. This structure contains pointers to establish lists for several purposes e.g. the free list or the page cache, a reference counter and a flag field. If the reference counter is zero the page is free, otherwise the counter denotes the number of users of the page. One of the flags, `PG_locked`, indicates if the page is currently locked for I/O. It is set whenever the page is involved in a disk I/O operation, i.e. when an memory mapped file is read. Another flag, `PG_reserved` is set for pages that are not available to the system at all. They are not even counted to the total amount of available memory.

The array of these `mem_map_t` structures is called *page map*.

2.2 Discarding pages

Whenever a memory page is needed, for instance to execute a copy-on-write operation, the kernel function `get_free_pages()` (mm/page_alloc.c) is called. First it tries to allocate a page from the list of unused pages. If this is impossible, i.e. all physical memory is used for process memory, page cache, buffer cache or the kernel itself, it tries to free some pages by calling `try_to_free_pages()` which again calls `do_try_to_free_pages()` (mm/vmscan.c). Now the kernel applies several strategies in turn in order to free some pages.

The first units to be shrunk are the buffer cache and the page cache. The function `shrink_mmap()` (mm/filemap.c) applies a so called ”clock algorithm” to go through the page map in order to find pages that can be discarded. Pages with the `PG_locked` bit set are left untouched. Also pages with a reference counter other than one are skipped. Although `shrink_mmap()` is a place where memory pages are freed it does not touch user pages of a process. Thus it is not interesting when we search for ways to lock communication buffers.

After trying to swap out some shared memory the kernel starts the actual swapping by calling `swap_out()` (mm/vmscan.c). It selects a process from the task list and passes it to `swap_out_process()`. This function goes through the process' list of virtual memory areas (also referred to as VM area, VMAs) and tries to swap one of them out through `swap_out_vma()`. VMAs with the `VM_LOCKED` bit set (see below) are skipped. If the VMA is not marked locked, the kernel goes through the page tables to find the physical pages. Now it writes the page to swap space if necessary and calls `__free_page()`. The latter function decrements the reference counter and adds the page to the free list if the counter has reached zero. Like in `shrink_mmap()`, all pages with the `PG_locked` bit set won't be touched. The same holds true for reserved pages.

Summarizing this analysis of the Linux memory management we can state that there are on principal two ways to lock memory:

- VMA-based: by setting the `VM_LOCKED` bit in the VMA
- page-based: by setting the `PG_locked` bit in the page map.

3 Current VIA implementations

There are several implementations of the VI Architecture available. One of the first was Berkeley-VIA [3] which is based on Myrinet. Another one is M–VIA [1], which aims at providing a modular implementation that can be used in conjunction with a variety of hardware ranging from dumb FastEthernet cards to native VIA hardware. Giganet Inc. has developed a native VIA implementation called cLAN. All of them support Linux and provide open source drivers, which enabled us to investigate their software. In this section we examine how those VIA implementors have tried to solve the task of locking memory. We also describe our first approach and its short comings.

3.1 The page–based approach

All of the implementations mentioned above manipulate the page map in order to prevent registered pages from being swapped out. Berkeley–VIA and M–VIA simply increment the reference counter of the pages. More recent versions of the Giganet driver set the `PG_locked` resp. the `PG_reserverd` bit in addition to that. However, even this cannot be regarded a clean solution since they do not check if the page is possibly already locked by the kernel. On deregistration the counter is decremented again and, in case of Giganet, the `PG_locked` flag is reset regardless of the counter state.

Although the manipulation of only the reference counter would make the implementation of multiple registrations very simple, that only is not a solution to the locking problem. We have conducted some experiments that show that pages are swapped out even when their reference counters are bigger than one. Following is the experiment in detail:

1. The *locktest* program allocates some memory and fills it with data. After that one can be sure that each virtual page is mapped to a distinct physical page.
2. We simulate the registration by incrementing the reference counters and storing the physical addresses.
3. Now we start another *allocator* process that allocates[1] as much memory as possible forcing a large amount of pages to be swapped out.
4. *locktest* writes again to each page of the memory block.
5. The kernel agent writes a certain value to the first page of the block using the physical address obtained during the registration. In this way we simulate a DMA operation of the NIC.
6. The physical addresses of all pages are derived from the page tables again and compared to those acquired during the registration.
7. The memory block is deregistered by decrementing the reference counters.
8. The contents of the first page is printed.

If this mechanism ensures memory locking one should assume that the physical pages are the same before and after the *allocator* process ran, and the first page should contain the value written by the kernel agent. However, in most cases we observed a different behavior: all physical addresses had changed and the first page still contained its original value. This indicated clearly that the pages had been swapped out.

This behavior can be explained when we take a look at the swapping mechanism again.

When the *allocator* process runs and writes to the allocated memory it causes a large number of major page faults[2] that are dealt with by the kernel by allocating physical pages through `get_free_page()`. As described in section 2.2 the kernel will start discarding pages currently owned by other processes when all free pages are used up. On this occasion it happens that the *locktest* process is chosen by the `swap_out()` function. As none of the VMAs is locked by means of the `VM_LOCKED` flag the registered area becomes subject to swapping sometime. When the swap function looks at the individual pages it finds them neither locked nor reserved. So it allocates a new swap page, writes the contents of the memory page to it, stores the swap address in the page table and marks the entry not–present. Then it calls `free_page()` for the old page on the assumption that the page is made available to other processes, in our case to the *allocator* process. Since, however, the reference counter was increased during registration the page is not really released. It is not associated with the virtual page just swapped out any more but it is still in use.

When we come to step 4 in the experiment the *locktest* process will cause a not–present page fault. The memory subsystem extracts the the swap file index from the page table entry and starts reading the page back from disk. A *new* page is allocated for this. Note, that it cannot be one of the pages formerly mapped to the registered region since the kernel still regards them used. The contents of the page before it was swapped out is written to the new page and the page table entry is redirected to this page. If the kernel agent now writes directly to the physical pages of the registered region, that it obtained during the registration (step 5), the changes will not be visible to the *locktest* process anymore. In a real VIA environment that means that the page table of the NIC, i.e. the TPT, is not consistent with the system's page tables any longer. Consequently, the NIC will use wrong memory addresses for its DMA operations. Communication fails, the system stability, however, is not affected by this lapse since the original physical pages have not been freed yet and, thus, will not be used for other purposes.

The experiment has proven that altering only the reference counter of the registered pages does not prevent the pages from being swapped out. Although simply setting the `PG_locked` bit or the `PG_reserved` bit would ensure that the page is not touched it must be considered a very risky and unclean solution. Moreover a special mech-

[1] Due to the demand paging mechanism it is necessary to write to the allocated pages in order to cause a COW and really consume physical memory.

[2] I.e. there is no valid page table entry.

anism is needed to determine when to reset the bit.

3.2 The VMA–based approach

In this section we will describe how memory can be locked down based on virtual addresses.

The API of Unix provides a so called `mlock` system call. It allows a process to disable paging for a specified range of its address space. This is exactly what is needed for memory registration, and that is why we took this approach for our first implementation of the Kernel Agent, but there are some severe restrictions.

First, only super-user processes are allowed to use `mlock`. This makes it at least impossible to use it at user level, i.e. to call `mlock` already in the VI User Agent. As the Kernel Agent, which is a device driver and hence part of the operating system kernel, has got higher privileges, it is able to carry out that task. But it requires some changes in the kernel. There are two possibilities to circumvent the user-id checking:

Rewriting `do_mlock()` Upon the `mlock` system call the kernel invokes `sys_mlock` which again calls `do_mlock`. The latter function finally checks the current process' effective used-id and returns an error if it is not root. There is a so called User-DMA patch [5] which moves the check from to `sys_mlock`. Thereby it is possible for the driver to call `do_mlock` directly.

Changing the process' capabilities The privileges of a process are controlled by *capabilities*, and only root processes have got the `CAP_IPC_LOCK` capability for locking memory. As the capabilities can be changed by the kernel, the Kernel Agent's registration function can grant that capability to the current process by means of `cap_raise()`, then call `do_mlock` and reclaim the capability again by `cap_lower()`.

Either variant requires that `do_mlock` be added to the kernel symbol table since it is not visible to drivers in the standard kernel.

`do_mlock` sets the `VM_LOCKED` flag of all VMAs corresponding to the given virtual address range. The original VMAs are split up if necessary. As described in section 2.2 these VMAs will be left untouched by the swapping functions.

Another major drawback of this approach is that mlock calls do not nest, i.e. a single unlock operation annuls multiple lock operation on the same address. Consequently, the driver must keep track of which address ranges are registered how often in order to allow for multiple registrations of the same address. It must unlock the memory only upon the last deregistration.

Summarizing we must say that all examined VIA implementations do not provide a reliable mechanism for locking registered memory. Either they do not ensure that the pages are locked, or they use risky techniques, or they do not allow for multiple registrations.

4 A reliable locking mechanism based on kiobufs

4.1 Conformance with Standard Kernels

Before we get to explaining our solution we want to discuss the issue of conformance to the main stream kernels briefly.

As Linux is an open source project everyone can contribute and modify it at will to support new features. This fact has made Linux very popular in research environments. Everything can be changed and every idea can be implemented immediately. However, if for instance a VIA implementation is supposed to be released to the "real world" as a building block for clusters it should work with the main stream kernels without applying additional patches. One typical user of a cluster is an engineer who wants to do computational fluid dynamics instead of modifying the kernel. If we want to meet this requirement we must restrict ourselves to use the mechanisms the standard kernel provides or will do in the future.

A further question is if such a VIA driver will be incorporated into the official kernel sometime. Although this is not a strict requirement, as the driver can be supplied directly with the hardware, it increases its acceptance. Besides, the driver will be maintained and adopted to new versions by the kernel developers. However, there are some very strict rules a driver must observe to be accepted by Linus Torvalds. One of them is that it must not touch the page tables[3], not even read them. This means that not even the VMA-based approach can be used to create an adequate solution. It only ensures that the pages are locked. Their physical addresses must still be derived directly from the page tables.

This challenged us to work out a solution based on a new kernel mechanism, called *kiobufs*.

4.2 RAW I/O and kiobufs

The RAW I/O mechanism was introduced to the Linux kernel by Stephen C. Tweedie of RedHat in order to ac-

[3] On 18 Aug 1999 in the thread "Re: [bigmem-patch] 4GB with Linux on IA32" in the linux-kernel mailing list Linus Torvalds wrote: "I will NOT allow anything that walks page tables. That's pretty much completely out of the question. The rawio stuff gives access to the page tables, and no device will be accepted that does more than that. Page table walking has always been a complete disaster, and it's much simpler to just set up the array of physical addresses separately (ie either at mmap time or through the factilities that rawio does offer)."

celerate SCSI disk accesses[4]. Traditional implementations first read data from disk to kernel buffers and then copy it to the user buffer. While this allows the kernel to cache frequently used data, it consumes additional time and bus bandwidth. The RAW I/O extension can be used to transfer data directly from disk to user buffers without intermediate copies. Since the buffer pages must be locked into physical memory during the transfer Tweedie invented the so called *kiobufs*. They are an integral part of the 2.3.x development kernels, and hence, of the next stable version 2.4.x as well. But there is also a patch for 2.2.x available.

A kiobuf describes the set of physical pages of a part of a process' user address space, where all pages are locked down. An array of kiobufs is called *kiovec*. After its creation a single kiobuf has got space to hold pages for 64KB of memory. Afterwards it can be expanded to span an arbitrary number of pages. When a kiobuf is mapped to a part of the user address space all corresponding pages are faulted in, their reference counter is incremented, the `PG_locked` bit is set and the physical addresses are stored in the kiobuf structure. Although it resembles quite closely what Giganet's latest driver does, it is by far safer because it checks for odd situations.

A drawback of kiobufs as for our application is that the same physical page can be held in at most one kiobuf at any time. This means that we need an additional layer on top of the kiobufs that keeps track of which page is already locked, i.e. it is already held in a kiobuf.

4.3 The Locked Memory Manager – LMM

Since kiobufs represent a page-based approach for locking memory a per-page lock counter seems obvious. But the difficulty is that the physical addresses are needed before the kiobuf is created. This, however, objects the condition to not traverse the page tables. Hence, the lock count tracking must be based on virtual addresses, and therewith separately for each process. The question is now if different virtual addresses could refer to the same physical page, because the mechanism would fail in this case. For that purpose we have to distinguish two cases:

Virtual addresses within a process The copy-on-write mechanism ensures that each virtual page is mapped to a distinct physical page. Thus, we have to make sure that a COW has happened for each page to be registered.

Virtual addresses of different processes Except for SystemV shared memory or shared file mappings the address spaces of different processes are isolated from each other by definition. I.e. the memory management will never assign the same physical page to more than one process at a time. As for shared mappings, only one process could register such an area. The second process attempting to register its mapping of that area would get an error. A general solution to this problem is subject to further work.

With the aforesaid exception we can assume that the mapping

$$(VirtualPages \times Processes) \rightarrow PhysicalPages$$

is injective. I.e. with $VpP_i := (VirtualPage_i, Process_i)$ and $Pp_k := PhysicalPage_k$, if

$$(VpP_i) \mapsto Pp_k$$

there will not exist another pair

$$(VpP_j) \mapsto Pp_k$$

In other words, a physical page is referenced by at most one pair VpP_i and, thus, we can track the lock count of the physical page Pp_k by means of VpP_i.

4.3.1 Locked Memory Areas – LMAs

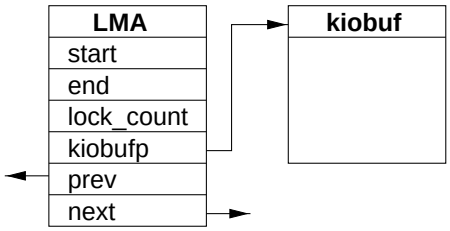

Figure 1. Structure of an LMA

This leads us to the introduction of *locked memory area*, in short *LMA*s. An LMA describes a contiguous range of virtual addresses with the same lock count. Figure 1 shows the structure of an LMA. It spans the address interval $[start, end)$. There is one kiobuf per LMA, that describes the corresponding physical pages. LMAs are kept in per-process lists[5] in ascending order of their start addresses.

4.3.2 The LMM Interface

The LMAs are managed by the *Locked Memory Manager (LMM)*. Its interface comprises the following three functions (in C syntax):

[4]The original patch from Tweedie has been enhanced further by SGI. The original version as well as SGI's extensions and some documentation can found at [11].

[5]This is realized by means of arrays that have one entry for every process that is using the Kernel Agent. More information about how it is done can be found in [7, Section 2.3.4] and [8, Section 4.2.1].

`lmm_lock_area(start, end)` After successful return the virtual address range $[start, end)$ is locked in physical memory. It can be called multiple times for the same address range. Ranges of subsequent calls may overlap.

`lmm_unlock_area(start, end)` Releases the specified address range. Only intervals that `lmm_lock_area` has been called on before may be given[6]. The corresponding pages may still stay locked, if the memory area has been locked multiple times. Upon the last unlock operation on an certain area the pages are made available for swapping again.

`lmm_get_pages(start, end, *pages)` Returns the physical page addresses of a locked memory area.

4.3.3 The LMM Implementation

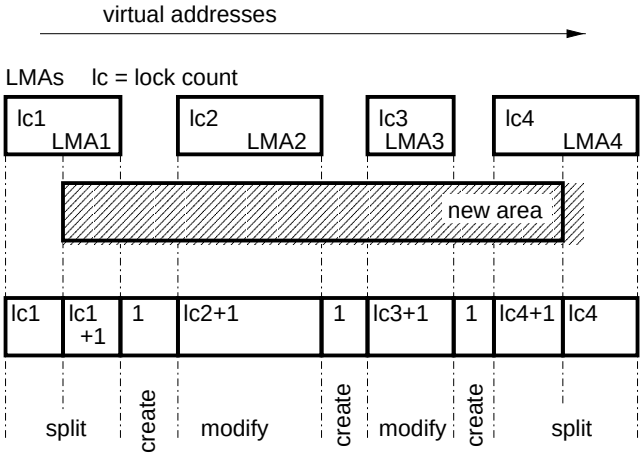

Figure 2. Locking a new area

The LMM maintains an LMA for every part of the virtual address space that is locked down. The lock count of each LMA specifies how many lock operations have been performed on this area.

lmm_lock_area(start, end) When a new area is to be locked the LMM scans the current process' list of LMAs and changes them appropriately. Figure 2 shows an example situation. Let's assume four LMAs already exist with the lock counts lc_1 to lc_4. Now the "new area" is to be locked down in the course of a registration operation. As an LMA describes a contiguous area of the same lock count some LMAs may have to be split up (LMA_1 and LMA_4), new

[6]This approach is sufficient since the LMM is intended to be used by the registration and deregistration functions of the Kernel Agent, and memory regions are only deregistered as a whole.

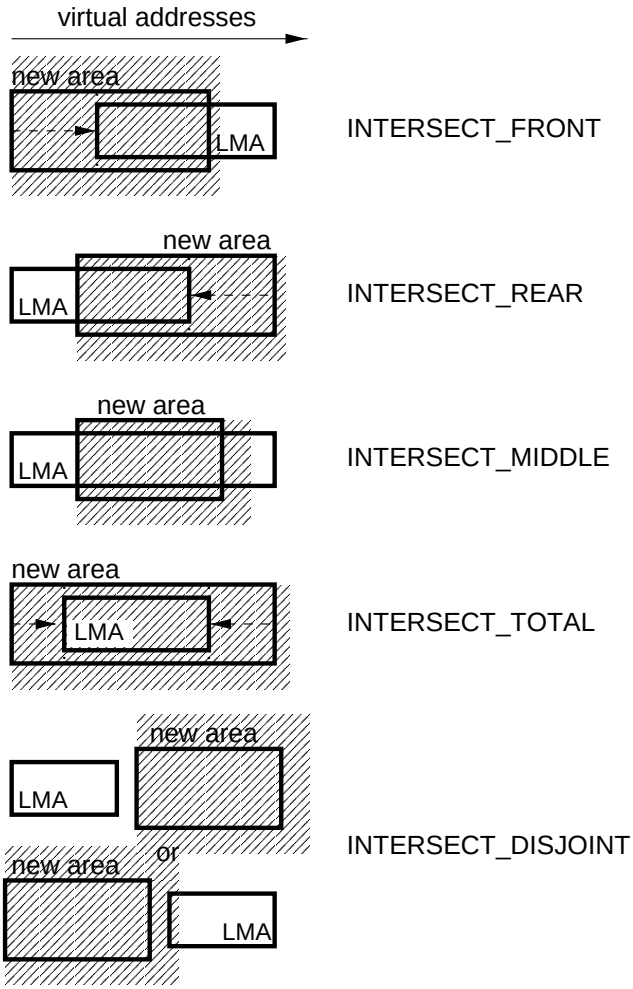

Figure 3. Possible types of intersection of new area and LMA

LMAs have to be created or the lock count of some LMAs must be incremented (LMA_2 and LMA_3).

What has to be done depends on the intersection of the new area with the existing LMAs. There are five types of intersections, as illustrated in figure 3. (In the following we assume the new area is bounded by `start` and `end`, and the LMA currently looked at is bounded by `lma->start` and `lma->end`.)

INTERSECT_FRONT The new area overlaps the left (low address) part of the LMA. It may start before the LMA. The conditions are:

$$\texttt{start} \leq \texttt{lma->start}\ \ and$$
$$\texttt{lma->start} < \texttt{end} < \texttt{lma->end}$$

A new LMA is created for the $[$`start`, `lma->start`$)$

interval and the LMA is split at `end`. The lock count of the left part is incremented. The remaining new area starts at `end`, i.e. we have reached the end of the new area.

INTERSECT_REAR The new area overlaps the right (high address) part of the LMA. It may end behind the LMA. The conditions are:

$$\texttt{lma->start} < \texttt{start} < \texttt{lma->end} \ \textit{and} \ \texttt{lma->end} \leq \texttt{end}$$

The LMA is split at `start` and the lock count of the right part is incremented. No new LMA is created for the new area on the right, instead the beginning of the remaining new area is set to `lma->end`.

INTERSECT_MIDDLE The new area lies completely within the LMA, but it does not span the whole LMA. The exact conditions are:

$$\texttt{lma->start} < \texttt{start} < \texttt{lma->end} \ \textit{and} \ \texttt{lma->start} < \texttt{end} < \texttt{lma->end}$$

The LMA is split into three pieces at `start` and `end`. The lock count of the middle part is incremented. The remaining new area starts at `end`, i.e. we have reached the end of the new area.

INTERSECT_TOTAL The new area covers the whole LMA whereat it may start before and/or end behind the LMA. The conditions for this case are:

$$\texttt{start} \leq \texttt{lma->start} \ \textit{and} \ \texttt{lma->end} \leq \texttt{end}$$

A new LMA is created for the $[\texttt{start}, \texttt{lma->start})$ interval and the lock count of the old LMA is incremented. In analogy to the INTERSECT_REAR case no LMA is created for the area beyond the old LMA, and the beginning of the remaining new area is set to `lma->end`.

INTERSECT_DISJOINT The new area and the LMA have got no common pages in this case. The conditions are:

$$\texttt{lma->end} \leq \texttt{start} \ \textit{or} \ \texttt{end} \leq \texttt{lma->start}$$

Simply a new LMA is created for the entire new area $[\texttt{start}, \texttt{end})$.

When an LMA is created a kiobuf is allocated and expanded to the required size. Then the given virtual address range is mapped to the kiobuf by means of the kernel function `map_user_kiobuf` (mm/memory.c). Splitting an LMA is a more complex operation as it requires that parts of the kiobuf of the original LMA be moved to another kiobuf.

The `lmm_lock_area` function seeks for the first LMA ending above `start`. Then it determines the type of intersection and takes the proper actions as described above. After finishing an LMA the procedure is repeated with the next LMA and the remaining new area until its end is reached. All newly created LMAs are inserted in the list.

lmm_unlock_area(start, end) Unlocking memory is less complex than locking. The reason is that, with the proposed locking algorithm and the restrictions for `start` and `end`, the given interval always spans entire LMAs. Further, there are LMAs for all parts of the interval.

Hence, the LMM only needs to traverse the LMA list from `start` to `end` and decrement the lock count of each LMA. When it drops to zero the LMA is destroyed. I.e. the kiobuf is unmapped by means of `unmap_kiobuf()` (mm/memory.c). On this occasion the physical pages are unlocked again and can be swapped out from now on. Finally, the LMA is removed from the list and freed.

lmm_get_pages(start, end, *pages) This function simply walks the LMA list and collects the physical page addresses from the kiobufs. An error is returned if a hole in the LMA list should be detected.

4.4 Performance Evaluation

The intention of the Virtual Interface Architecture is to remove the operating system from the time critical communication path. This is possible if a fixed set of buffers is used for communication. They need to be registered only once during application startup, hence the registration costs have nearly no influence on application performance.

Things are different if zero copy protocols are to be build on top of VIA. In this case the buffers may change with every transfer. Since it is impossible to register the entire address space due to resource limitations, the buffers must be registered on the fly, i.e. within the send or receive function. This turns the times for registering memory into an essential parameter. Table 1 shows the times for registering and deregistering a memory region. The measurements were conducted on a 450 MHz Pentium III machine. The registration time depends nearly linearly of the number of pages, where about 2.7 GB can registered per second. The deregistration rate is even higher at approximately 10 GB/s. Although the current implementation does not yet store the

Table 1. Times for memory registration/deregistration (in μs)

Size (KB)	register	deregister
4	6.7	3.5
8	8.0	3.8
16	10.7	4.4
32	16.4	5.8
64	27.8	8.7
128	49.7	14.2
256	92.8	25.9
512	185	47.5
1024	366	103
2048	734	203
4096	1493	381
8192	2987	817
16384	5856	1666

physical addresses on the NIC but in main memory instead, the rates will be only slightly worse in the final version since a write access to the NIC's memory costs less than $0.1\mu s$, and only one write operation is needed for each page.

Furthermore, any communication library, such as MPI, or application on top of VIA should endeavour to reuse registered memory as much as possible whether by means of MPI's persistant communication or by using a caching mechanism for registered memory.

5 Conclusions and Outlook

The mechanism proposed ensures on the one hand that a given range of the user address space is reliably locked into physical memory. On the other hand it allows virtual pages to be registered multiple times, as required by the VIA specification. Moreover, it applies only those mechanisms that are provided by the main stream kernels, resp. will be provided in the near future, and it meets the conditions for a device driver to be accepted for the official kernel.

Although the proposed locking mechanism has been developed for a VIA implementation it can be utilized for any type of user level communication.

In the current implementation the LMAs of a process are stored in linear lists. An optimization could be achieved by using AVL trees. Besides, adjacent LMAs could be merged if they have the same lock count. Further, it should be examined how the mechanism can be extended to handle shared mappings properly, i.e. to allow every process to register a shared area.

References

[1] M-VIA: A high performance modular VIA for Linux. http://www.nersc.gov/research/FTG/via.

[2] cLAN for Linux. http://www.giganet.com/products/indexlinux.htm.

[3] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. Technical report, Department of Electrical Engineering and Computer Science, University of California, May 1998.

[4] Compaq, Intel and Microsoft Corporations. *The Virtual Interface Architecture Specification Version 1.0*, December 1997. Available at http://www.viarch.org.

[5] R. Kaiser. User level DMA patch for Linux. ftp://ftp.sysgo.de/pub/Linux.

[6] S. Schindler, W. Rehm, and C. Dinkelmann. An optimized MPI library for VIA/SCI cards. In *Proceedings of the Asia-Pacific International Symposium on Cluster Computing (APSCC'2000) held in conjunction with the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPCAsia2000)*, Beijing, China, May 14-17 2000.

[7] F. Seifert. Design and implementation of system software for transparent mode communication over SCI. Study thesis, Chair of Computer Architecture, Chemnitz University of Technology, February 1999.

[8] F. Seifert. Development of system software to integrate the virtual interface architecture (VIA) into the linux operating system kernel for optimized message passing. Diploma thesis, Chair of Computer Architecture, Chemnitz University of Technology, October 1999.

[9] M. Trams. Design of a system-friendly PCI-SCI bridge with an optimized user-interface. Diploma thesis, Chair of Computer Architecture, Chemnitz University of Technology, 1998.

[10] M. Trams, W. Rehm, D. Balkanski, and S. Simeonov. Memory management in a combined VIA/SCI hardware. In *Proceedings to PC-NOW 2000, International Workshop on Personal Computer based Networks of Workstations held in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, May 1-5 2000.

[11] S. C. Tweedie et al. Raw I/O enhancements. http://oss.sgi.com/projects/rawio.

[12] M. Welsh, A. Basu, and T. v. Eicken. Incorporating memory management into user-level network interfaces. Technical report, Department of Computer Science, Cornell University, Ithaca, 1997.

References

[RS99] Wolfgang Rehm und Sven Schindler. *Multiple Devices Under MPICH* , PASA'99 5. Workshop Parallele Systeme und Algorithmen, 7. Oktober 1999, Jena, im Rahmen der ARCS'99 — 15. GI/ITG-Fachtagung Architektur von Rechensystemen.

[TR99] Mario Trams und Wolfgang Rehm: *A New Generic and Reconfigurable PCI-SCI Bridge*, Proceedings of the SCI-Europe'99, 2.–3. September 1999, Toulouse/Frankreich, im Rahmen der Euro-PAR'99.

[TRBS00] Mario Trams, Wolfgang Rehm, Daniel Balkanski und Stanislav Simeonov: *Memory Management in a combined VIA/SCI Hardware*, Proceedings of the PC-NOW 2000, International Workshop on Personal Computer based Networks of Workstations im Rahmen des International Parallel and Distributed Processing Symposium (IPDPS 2000), 1.–5. Mai 2000, Cancun/Mexico. Springer LNCS Series, ISSN 0302-9743, ISBN 3-540-67442-X, Seiten 4–15.

[SRD00] Sven Schindler, Wolfgang Rehm, Carsten Dinkelmann. *An optimized MPI library for VIA/SCI cards*, In: Proceedings of the Asia-Pacific International Symposium on Cluster Computing (APSCC'2000) held in conjunction with the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPCAsia2000), May 14-17, 2000, Beijing, China. Volume II, pp. 895-903.

[RTBS00] Wolfgang Rehm, Mario Trams, Daniel Balkanski und Stanislav Simeonov. *A New Architectural Concept for Highly Efficient Message-Passing on PCI–SCI Network Interfaces*, 14th International Conference "System for Automation of Engineering and Research"SAER'2000, 18.–20. September 2000, St. Konstantin (Varna), Bulgarien.

[SBR00] Friedrich Seifert, Daniel Balkanski, Wolfgang Rehm. *Comparing MPI Performance of SCI and VIA*, In proceedings of SCI-Europe 2000 held in conjunction with Euro-Par 2000, Munich, Germany, August/September 2000.

[TSR00C] Mario Trams, Ralph Schlosser und Wolfgang Rehm: *Design Choices and First Results of Our VIA-Capable PCI-SCI Bridge*, Proceedings of the CLUSTER2000 — IEEE International Conference on Cluster Computing, 28.Nov.–1.Dez. 2000, Chemnitz, ISBN 0-7695-0896-0, Seiten 349–350.

[SR00] Friedrich Seifert und Wolfgang Rehm. *Proposing a Mechanism for Reliably Locking VIA Communication Memory in Linux* , Proceedings of the CLUSTER2000 — IEEE International Conference on Cluster Computing, 28.Nov.–1.Dez. 2000, Chemnitz, ISBN 0-7695-0896-0, Seiten 225–232.

Other titles in the SFB393 series:

99-01 P. Kunkel, V. Mehrmann, W. Rath. Analysis and numerical solution of control problems in descriptor form. January 1999.

99-02 A. Meyer. Hierarchical preconditioners for higher order elements and applications in computational mechanics. January 1999.

99-03 T. Apel. Anisotropic finite elements: local estimates and applications (Habilitationsschrift). January 1999.

99-04 C. Villagonzalo, R. A. Römer, M. Schreiber. Thermoelectric transport properties in disordered systems near the Anderson transition. February 1999.

99-05 D. Michael. Notizen zu einer geometrisch motivierten Plastizitätstheorie. Februar 1999.

99-06 T. Apel, U. Reichel. SPC-PM Po 3D V 3.3, User's Manual. February 1999.

99-07 F. Tröltzsch, A. Unger. Fast solution of optimal control problems in the selective cooling of steel. March 1999.

99-08 W. Rehm, T. Ungerer (Eds.). Ausgewählte Beiträge zum 2. Workshop Cluster-Computing 25./26. März 1999, Universität Karlsruhe. März 1999.

99-09 M. Arav, D. Hershkowitz, V. Mehrmann, H. Schneider. The recursive inverse eigenvalue problem. March 1999.

99-10 T. Apel, S. Nicaise, J. Schöberl. Crouzeix-Raviart type finite elements on anisotropic meshes. May 1999.

99-11 M. Jung. Einige Klassen iterativer Auflösungsverfahren (Habilitationsschrift). Mai 1999.

99-12 V. Mehrmann, H. Xu. Numerical methods in control, from pole assignment via linear quadratic to H_∞ control. June 1999.

99-13 K. Bernert, A. Eppler. Two-stage testing of advanced dynamic subgrid-scale models for Large-Eddy Simulation on parallel computers. June 1999.

99-14 R. A. Römer, M. E. Raikh. The Aharonov-Bohm effect for an exciton. June 1999.

99-15 P. Benner, R. Byers, V. Mehrmann, H. Xu. Numerical computation of deflating subspaces of embedded Hamiltonian pencils. June 1999.

99-16 S. V. Nepomnyaschikh. Domain decomposition for isotropic and anisotropic elliptic problems. July 1999.

99-17 T. Stykel. On a criterion for asymptotic stability of differential-algebraic equations. August 1999.

99-18 U. Grimm, R. A. Römer, M. Schreiber, J. X. Zhong. Universal level-spacing statistics in quasiperiodic tight-binding models. August 1999.

99-19 R. A. Römer, M. Leadbeater, M. Schreiber. Numerical results for two interacting particles in a random environment. August 1999.

99-20 C. Villagonzalo, R. A. Römer, M. Schreiber. Transport Properties near the Anderson Transition. August 1999.

99-21 P. Cain, R. A. Römer, M. Schreiber. Phase diagram of the three-dimensional Anderson model of localization with random hopping. August 1999.

99-22 M. Bollhöfer, V. Mehrmann. A new approach to algebraic multilevel methods based on sparse approximate inverses. August 1999.

99-23 D. S. Watkins. Infinite eigenvalues and the QZ algorithm. September 1999.

99-24 V. Uski, R. A. Römer, B. Mehlig, M. Schreiber. Incipient localization in the Anderson model. August 1999.

99-25 A. Meyer. Projected PCGM for handling hanging in adaptive finite element procedures. September 1999.

99-26 F. Milde, R. A. Römer, M. Schreiber. Energy-level statistics at the metal-insulator transition in anisotropic system. September 1999.

99-27 F. Milde, R. A. Römer, M. Schreiber, V. Uski. Critical properties of the metal-insulator transition in anisotropic systems. October 1999.

99-28 M. Theß. Parallel multilevel preconditioners for thin shell problems. November 1999.

99-29 P. Biswas, P. Cain, R. A. Römer, M. Schreiber. Off-diagonal disorder in the Anderson model of localization. November 1999.

99-30 C. Mehl. Anti-triangular and anti-m-Hessenberg forms for Hermitian matrices and pencils. November 1999.

99-31 A. Barinka, T. Barsch, S. Dahlke, M. Konik. Some remarks for quadrature formulas for refinable functions and wavelets. November 1999.

99-32 H. Harbrecht, C. Perez, R. Schneider. Biorthogonal wavelet approximation for the coupling of FEM-BEM. November 1999.

99-33 C. Perez, R. Schneider. Wavelet Galerkin methods for boundary integral equations and the coupling with finite element methods. November 1999.

99-34 W. Dahmen, A. Kunoth, R. Schneider. Wavelet least squares methods for boundary value problems. November 1999.

99-35 S. I. Solovev. Convergence of the modified subspace iteration method for nonlinear eigenvalue problems. November 1999.

99-36 B. Heinrich, B. Nkemzi. The Fourier-finite-element method for the Lamé equations in axisymmetric domains. December 1999.

99-37 T. Apel, F. Milde, U. Reichel. SPC-PM Po 3D v 4.0 - Programmers Manual II. December 1999.

99-38 B. Nkemzi. Singularities in elasticity and their treatment with Fourier series. December 1999.

99-39 T. Penzl. Eigenvalue decay bounds for solutions of Lyapunov equations: The symmetric case. December 1999.

99-40 T. Penzl. Algorithms for model reduction of large dynamical systems. December 1999.

00-01 G. Kunert. Anisotropic mesh construction and error estimation in the finite element method. January 2000.

00-02 V. Mehrmann, D. Watkins. Structure-preserving methods for computing eigenpairs of large sparse skew-Hamiltonian/Hamiltonian pencils. January 2000.

00-03 X. W. Guan, U. Grimm, R. A. Römer, M. Schreiber. Integrable impurities for an open fermion chain. January 2000.

00-04 R. A. Römer, M. Schreiber, T. Vojta. Disorder and two-particle interaction in low-dimensional quantum systems. January 2000.

00-05 P. Benner, R. Byers, V. Mehrmann, H. Xu. A unified deflating subspace approach for classes of polynomial and rational matrix equations. January 2000.

00-06 M. Jung, S. Nicaise, J. Tabka. Some multilevel methods on graded meshes. February 2000.

00-07 H. Harbrecht, F. Paiva, C. Perez, R. Schneider. Multiscale Preconditioning for the Coupling of FEM-BEM. February 2000.

00-08 P. Kunkel, V. Mehrmann. Analysis of over- and underdetermined nonlinear differential-algebraic systems with application to nonlinear control problems. February 2000.

00-09 U.-J. Görke, A. Bucher, R. Kreißig, D. Michael. Ein Beitrag zur Lösung von Anfangs-Randwert-Problemen einschließlich der Materialmodellierung bei finiten elastisch-plastischen Verzerrungen mit Hilfe der FEM. März 2000.

00-10 M. J. Martins, X.-W. Guan. Integrability of the D_n^2 vertex models with open boundary. March 2000.

00-11 T. Apel, S. Nicaise, J. Schöberl. A non-conforming finite element method with anisotropic mesh grading for the Stokes problem in domains with edges. March 2000.

00-12 B. Lins, P. Meade, C. Mehl, L. Rodman. Normal Matrices and Polar Decompositions in Indefinite Inner Products. March 2000.

00-13 C. Bourgeois. Two boundary element methods for the clamped plate. March 2000.

00-14 C. Bourgeois, R. Schneider. Biorthogonal wavelets for the direct integral formulation of the heat equation. March 2000.

00-15 A. Rathsfeld, R. Schneider. On a quadrature algorithm for the piecewise linear collocation applied to boundary integral equations. March 2000.

00-16 S. Meinel. Untersuchungen zu Druckiterationsverfahren für dichteveränderliche Strömungen mit niedriger Machzahl. März 2000.

00-17 M. Konstantinov, V. Mehrmann, P. Petkov. On Fractional Exponents in Perturbed Matrix Spectra of Defective Matrices. April 2000.

00-18 J. Xue. On the blockwise perturbation of nearly uncoupled Markov chains. April 2000.

00-19 N. Arada, J.-P. Raymond, F. Tröltzsch. On an Augmented Lagrangian SQP Method for a Class of Optimal Control Problems in Banach Spaces. April 2000.

00-20 H. Harbrecht, R. Schneider. Wavelet Galerkin Schemes for 2D-BEM. April 2000.

00-21 V. Uski, B. Mehlig, R. A. Römer, M. Schreiber. An exact-diagonalization study of rare events in disordered conductors. April 2000.

00-22 V. Uski, B. Mehlig, R. A. Römer, M. Schreiber. Numerical study of eigenvector statistics for random banded matrices. May 2000.

00-23 R. A. Römer, M. Raikh. Aharonov-Bohm oscillations in the exciton luminescence from a semiconductor nanoring. May 2000.

00-24 R. A. Römer, P. Ziesche. Hellmann-Feynman theorem and fluctuation-correlation analysis of i the Calogero-Sutherland model. May 2000.

00-25 S. Beuchler. A preconditioner for solving the inner problem of the p-version of the FEM. May 2000.

00-26 C. Villagonzalo, R.A. Römer, M. Schreiber, A. MacKinnon. Behavior of the thermopower in amorphous materials at the metal-insulator transition. June 2000.

00-27 C. Mehl, V. Mehrmann, H. Xu. Canonical forms for doubly structured matrices and pencils. June 2000. S. I. Solov'ev. Preconditioned gradient iterative methods for nonlinear eigenvalue problems. June 2000.

00-29 A. Eilmes, R. A. Römer, M. Schreiber. Exponents of the localization lengths in the bipartite Anderson model with off-diagonal disorder. June 2000.

00-30 T. Grund, A. Rösch. Optimal control of a linear elliptic equation with a supremum-norm functional. July 2000.

00-31 M. Bollhöfer. A Robust ILU Based on Monitoring the Growth of the Inverse Factors. July 2000.

00-32 N. Arada, E. Casas, F. Tröltzsch. Error estimates for a semilinear elliptic control problem. July 2000.

00-33 T. Penzl. LYAPACK Users Guide. August 2000.

00-34 B. Heinrich, K. Pietsch. Nitsche type mortaring for some elliptic problem with corner singularities. September 2000.

00-35 P. Benner, R. Byers, H. Faßbender, V. Mehrmann, D. Watkins. Cholesky-like Factorizations of Skew-Symmetric Matrices. September 2000.

00-36 C. Villagonzalo, R. A. Römer, M. Schreiber, A. MacKinnon. Critical Behavior of the Thermoelectric Transport Properties in Amorphous Systems near the Metal-Insulator Transition. September 2000.

00-37 F. Milde, R. A. Römer, M. Schreiber. Metal-insulator transition in anisotropic systems. October 2000.

00-38 T. Stykel. Generalized Lyapunov Equations for Descriptor Systems: Stability and Inertia Theorems. October 2000.

00-39 G. Kunert. Robust a posteriori error estimation for a singularly perturbed reaction-diffusion equation on anisotropic tetrahedral meshes. November 2000.

01-01 G. Kunert. Robust local problem error estimation for a singularly perturbed problem on anisotropic finite element meshes. January 2001.

01-02 G. Kunert. A note on the energy norm for a singularly perturbed model problem. January 2001.

01-03 U.-J. Görke, A. Bucher, R. Kreißig. Ein Beitrag zur Materialparameteridentifikation bei finiten elastisch-plastischen Verzerrungen durch Analyse inhomogener Verschiebungsfelder mit Hilfe der FEM. Februar 2001.

01-04 R. A. Römer. Percolation, Renormalization and the Quantum-Hall Transition. February 2001.

01-05 A. Eilmes, R. A. Römer, C. Schuster, M. Schreiber. Two and more interacting particles at a metal-insulator transition. February 2001.

01-06 D. Michael. Kontinuumstheoretische Grundlagen und algorithmische Behandlung von ausgewählten Problemen der assoziierten Fließtheorie. März 2001.

01-07 S. Beuchler. A preconditioner for solving the inner problem of the p-version of the FEM, Part II - algebraic multi-grid proof. March 2001.

01-08 S. Beuchler, A. Meyer. SPC-PM3AdH v 1.0 - Programmer's Manual. March 2001.

01-09 D. Michael, M. Springmann. Zur numerischen Simulation des Versagens duktiler metallischer Werkstoffe (Algorithmische Behandlung und Vergleichsrechnungen). März 2001.

01-10 B. Heinrich, S. Nicaise. Nitsche mortar finite element method for transmission problems with singularities. March 2001.

01-11 T. Apel, S. Grosman, P. K. Jimack, A. Meyer. A New Methodology for Anisotropic Mesh Refinement Based Upon Error Gradients. March 2001.

The complete list of current and former preprints is available via
`http://www.tu-chemnitz.de/sfb393/preprints.html`.