Optimierung eines Linux I/O Treibers für System Z

Implementieren und Testen von Mellanox BlueFlame

Praxisarbeit

im Fachgebiet Angewandte Informatik

vorgelegt von:	Filip Haase
Studienbereich:	Angewandte Informatik
Matrikelnummer:	3190170
Praxiseinsatz-Betreuer:	Alexander Schmidt

© 2012

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

Gegenstand dieser Praxisarbeit ist eine I/O Karte von Mellanox. Diese Karte hat ein Feature names BlueFlame welches die Performance von Netzwerktraffic verbessert, bzw die Latenzzeiten beim Senden und Empfangen der Daten senkt. Dieses Feature ist auf x-86 basierten Systemen bereits implementiert.

Erster Teil der Praxisarbeit ist die Erstellung einer Methode zur systematischen Messung der Latenzzeiten in Abhängigkeit der Datengrößen die Versendet werden sollen. Damit sollen Messungen auf x-86 basierten Systemen (mit und ohne "Blue Flame") durchgeführt werden. Die Ergebnisse sollen anschließend ausgewertet und grafisch dargestellt werden.

Zweiter Teil der Praxisarbeit ist die prototypische Implementierung des Features für die System Z, einer Großrechnerarchitektur von IBM. Für die Implementierung muss der entsprechende Linux Device Driver und die entsprechende Library erweitert werden. Anschließend sollen auch für die System Z die Latenzzeiten mit und ohne BlueFlame gemessen und grafisch ausgewertet werden.

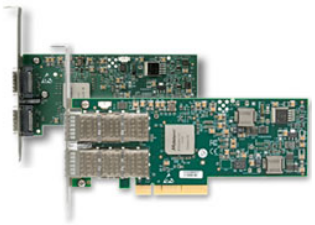
1.1. Ziel der Arbeit

Ziel dieses Berichts ist das am Ende eine prototypische Implementierung von BlueFlame für System Z steht. An dessen Performance soll anhand der gemessenen Daten bewertet werden, ob die Nutzung von BlueFlame auf System Z lohnenswert ist und welche weiteren Aufgaben und Verbesserungsmöglichkeiten es in dieser Hinsicht noch gibt.

1.2. Voraussetzungen zum Verständnis der Arbeit

Dieser Abschnitt dient als Einleitung in die benutzte Hard- und Software. Desweiteren sollen Grundkonzepte, die wichtig für das Verständnis der Praxisarbeit und der Testumgebung sind, erklärt werden.

1.2.1. Mellanox Connect X



Quelle: www.mellanox.com

Abbildung 1.1.: Mellanox Connect X

Die Connect X Familie der Firma Mellanox ist eine Reihe von Netzwerkkarten die speziell auf kleine Latenzzeiten und hohe Performance ausgelegt sind. Es gibt derzeit die Connect X in der Version 1 bis 3. Version 1 und 2 nutzen PCIe Gen 2 und mit Version 3 kann PCIe Gen 3 genutzt werden. Da System Z bisher "nur" PCIe Gen 2 beherscht, kann mit der Mellanox Connect X 3 kein Unterschied in den Latenzzeiten gemessen werden. Für die Messungen in dieser Praxisarbeit wurde die Mellanox Connect X 2 verwendet.

Die Karte kann Daten über Fiber-Chanel, Ethernet oder Infiniband Daten versenden.

1.2.1.1. BlueFlame

BlueFlame ist ein Mechanismus für die Mellanox Connect X Familie, der beim Senden von Daten für kleine Latenzzeiten sorgt. Der Gewinn wird durch ein schnelleres Schreiben der Daten an den Adapter erzielt. Anschließend werden die Daten vom Adapter mit und ohne BlueFlame identisch versendet. Abb.1.2.

zeigt vereinfacht welche Schritte zum Schreiben der Daten an den Adapter nötig sind.

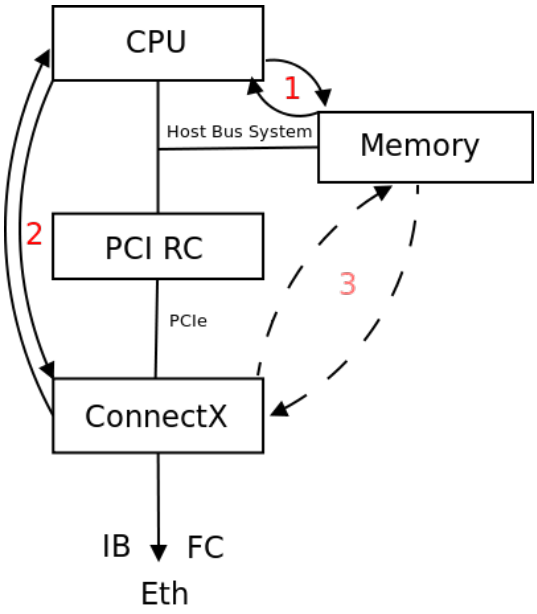


Abbildung 1.2.: Kommunikation zwischen CPU und Adapter

1. Das Schreiben der zu versendenden Daten von der CPU in den Speicher, dieser Schritt passiert sowohl bei Senden mit als auch ohne BlueFlame
2. Das Schreiben der zu versendenden Daten von der CPU an den Adapter.

   Ohne BlueFlame wird hier lediglich ein "DoorbellWritei" vorgenommen, was bedeutet dass ein minimales Paket in den Adapter geschrieben wird welches die zu versendenden Daten im Speicher identifizieren kann.

   Mit BlueFlame werden die kompletten zu versendenden Daten aus dem Speicher mit einem großen Write[1] in den Adapter geschrieben.
3. Dieser Schritt wird nur ohne BlueFlame benötigt. Um die Daten zum Adapter zu bekommen werden sie über die "Doorbell Write"-Daten identifiziert und dann per *DMA*[2] vom Adapter nachgeladen.

D.h. mit BlueFlame hat man einen Schritt weniger, dafür werden bei Schritt 2 größere Daten gesendet. Theoretisch ist dadurch eine geringere Latenz möglich.

[1] Siehe Kapitel 4.2.4.
[2] Direct Memory Access - Ein direkter Speicherzugriff ohne CPU

Dies ist allerdings Abhängig von der Performance vom Schreiben großer Daten an einen Adapter und der Performance eines DMAs.

1.2.2. System Z / X

System Z ist die aktuelle Großrechner Architektur der Firma IBM. Sie zeichnet sich vor allem durch hohe Parallelisierung aus. Das bedeutet das viele logische Partitionen(LPAR) und damit auch mehrere Betriebssystem Instanzen gleichzeitig ausgeführt werden können. Die LPARs teilen sich ein oder mehrere Prozessoren, weswegen höhere CPU-Belastung besonders kritisch gesehen werden muss. Außerdem zeichnet sie sich im Gegensatz zu Vorgängerarchitekturen durch eine 64-Bit Architektur aus. [3]

IBM System X ist dagegen die aktuelle x86-basierende Reihe der Server von IBM.

1.2.2.1. Testaufbau

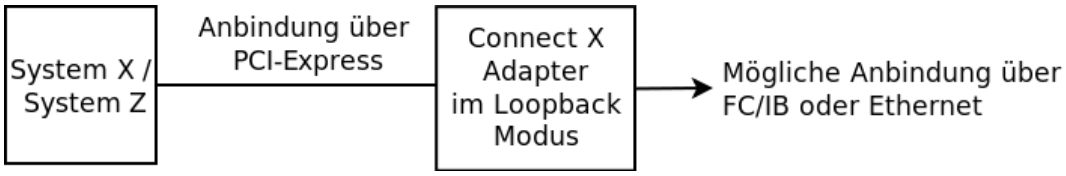


Abbildung 1.3.: Mellanox Connect X

PCIe "Peripheral Component Interconnect Express" ist ein Erweiterungsstandart zur Verbindung von Peripheriegeräten mit dem Chipsatz eines Hauptprozessors.[4]

Loopback Modus Im Loopback Modus bedeutet dass die Karte physisch keine Kabel angeschlossen hat. Sie sendet die Daten an sich selbst und stellt damit sowohl Sender als auch Empfänger dar.

[3] http://www-03.ibm.com/systems/z/hardware/
[4] http://de.wikipedia.org/wiki/PCI$_E$*xpress*

1.2.3. Linux Device Driver

Der Linux Kernel stellt eine Hardwareabstraktionsschicht d.h. eine von der Rechnerarchitektur unabhängige und einheitliche Schnittstelle(API) zwischen Software und Hardware dar. Dabei nehmen Device Driver eine besondere Rolle ein. Sie sind sogenannte "Black Boxen" die einer bestimmten Hardware zugeordnet werden können, und für diese Spezielle Hardware die Schnittstelle bereitstellen. Unter der Decke werden dieser Schnittstelle die entsprechenden Device-Operationen zugeordnet.

Das Device Driver Interface hat eine Architektur die es Möglich macht sie Modular zu entwickeln, dass sie zur Laufzeit hinzugefügt und entfernt werden können. Sie werden im Kernel als Module bezeichnet. Da der Kernel das "Herzstück" des Betriebssystems darstellt muss hier besonders auf Dinge wie Sicherheit und Nebenläufigkeiten geachtet werden. Um die Sicherheit garantieren zu können wird der Virtuelle Speicher des Betriebssystems in Kernel- und Userspace geteilt. Dabei wird von Device Drivern nur der Kernelspace, also speziell dem Kernel zugeordneter Speicher genutzt.[5]

Im Zuge dieser Praxisarbeit wird mit der Device Driver der Mellanox ConnectX Karte modifiziert.

[5] Linux Device Driver - Chapter 0

2. Latenzmessungen

In diesem Abschnitt des Praxisberichtes geht es um die Latenzmessungen, die als Indikator für Perfomance von BlueFlame benutzt werden.

Hierzu wird zuerst rdma_lat vorgestellt, ein bereits bestehendes Programm zur Messung von durchschnittlichen Latenzzeiten. Anschließend wird die Implementierung eines Bash-Scripts vorgestellt, welches diese Messungen für verschiedene Payloads automatisiert. Zuletzt betrachten wir die Graphische Darstellung der gemessenen Daten.

2.1. rdma_lat

Das Commandline-Programm rdma_lat ist ein Teil der *OFED*[6]-Utilities , es dient dazu ein *RDMA*[7] Senden/Empfangen durchzuführen und die Latenzzeit vom Aufruf einer Send-Befehls im Programm, bis zum Ankommen der Nachricht auf der Empfängerseite zu messen.

2.1.1. Funktionsweise

rdma_lat kann als Server und als Client aufgerufen werden. Anschließend senden sich Server und Client abwechselnd Nachrichten zu und messen je Programm die Zeit von einem Senden und einem Empfangen. Dies wird in einer Schleife wiederholt und anschließend die Latenzen berechnet und ausgegeben. Bei der Latenzberechnung wird nun ein timestamp vor Senden/Empfangen und ein nachfolgender genommen, diese voneinander Subtrahiert und anschließend

[6] OpenFabrics Enterprise Distribution - Sammlung von Tools und Treibern der OpenFabrics Alliance, die RDMA fördern
[7] Remote Direct Memory Access - Der direkte Speicherzugriff ohne CPU von Außen

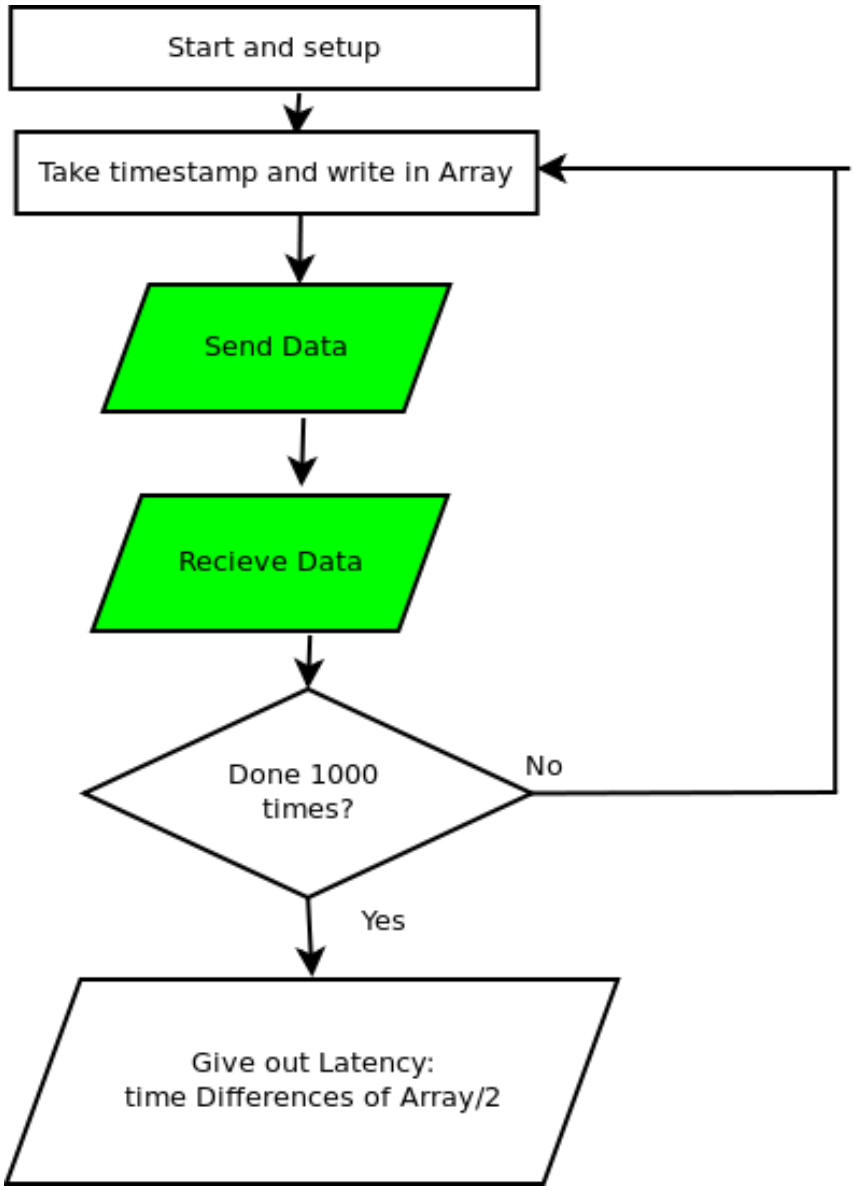


Abbildung 2.1.: Ablaufdiagramm von rdmaLat

die Differenz durch 2 geteilt um die Zeit eines Senden/Empfangen zu bekommen. Die Ausgabe von rdma_lat enthält nun 3 Latenzzeiten. Den besten, den schlechtesten und den typischen(gleichzusetzen mit durchschnittlichen) Wert. Für diese Praxisarbeit wird der typische Latenzwert benutzt um die Perfomance von BlueFlame zu bestimmen.

Die versendeten Daten beinhalten dabei einen Iterationzzähler, über den das Programm überprüft ob das Senden erfolgreich war. Damit wird sichergegangen das das Empfangene Packet auch wirklich das gesendete Packet ist.

2.1.2. Aufruf

Beim Aufruf können rdma_lat verschiedene Parameter mitgegeben werden. Hier eine Liste der für diesen Praxisbericht relevanten Parameter:

—**size** Mit diesem Parameter kann die Größe der zuversendenen Daten in Byte angegeben werden. Default-Value: 1 Byte

—**iter** Gibt an wie oft Nachrichten ausgetauscht werden sollen. Default-Value: 1000 Iterationen

<host> Verbinde zu Server <host>

Listing 2.1: Beispiel Aufruf von rdmaLat

```
1 # Beispiel fuer Server mit payload von 37 Byte und 1 Millionen Iterationen
2 rdma_lat --size=37 --iters=1000000
3
4 # Und Client
5 rdma_lat --size=37 --iters=1000000 127.0.0.1
```

2.2. Erstellung eines Bash-Scripts

rdma_lat gibt Latenzwerte für eine Payloadsize zurück, doch Ziel ist es Latenzwerte für verschiedene Payloadsizes zu bekommen. Zum Beispiel von 1 bis 1024 Byte in 10 Byte Schritten.

Um dies zu automatisieren sollte ein Bash-Script erstellt werden, welches Aufrufe mit den verschiedenen Payloadsizes macht, den Output parst und zur weiteren Verarbeitung in ein File schreibt.

2.2.1. Blackbox

Abbildung 2.2.: Blackbox Ansicht des Latenzmessung Bash-Scripts

Abbildung 2.2. zeigt die Blackbox-Ansicht des Scripts. Input sind die Maximale und Minimalen Payloadsizes und die Schrittgröße, also die Werte die bestimmen für welche Payloadsizes die Latenzwerte gemessen werden. Und außerdem Trennzeichen und Names des Outputfiles, also zur Bestimmung der Formatierung und der Ausgabe. Diese Werte werden über Parameter an das Script übergeben, da sie sich ständig unterscheiden(unterschiedliche Anforderungen bei verschiedenen Läufen in Hinblick auf Zeit, interessanter Ausschnitt,...).

Um möglichst schnell arbeiten zu können, sind alle Parameter optional und sinnvolle Default-Values definiert.

2.2.2. Code Aufbau

Listing 2.2: Pseudocode für das Bash-Script zum Latenzen messen

```
1 Setzen der DEFAULT Werten in Variablen MAX, MIN, SEPERATOR,
      OUTPUT_FILE, STEP_SIZE
2
3 SCHLEIFE mit ueber die eingegebenen Parameter{
4   WENN passender Parameter
5     Wert in passende Variable schreiben
6   SONST
```

```
7       AUSGABE Hilfe und ENDE
8   }
9
10  SCHLEIFE von MIN bis MAX mit STEP_SIZE{
11    AUFRUF von rdma_lat als Server
12    WARTEN
13    AUFRUF von rdma_lat als Client
14    WARTEN
15
16    PARSEN der return Werte und SCHREIBEN in Variablen lat_max,
          lat_min, lat_typ
17    SCHREIBEN in OUTPUT_FILE
18  }
```

In Listing 2.2. sieht man den Pseudocode des BashScripts.[8]

Besonders zu erwähnen sind die beiden Wartezeiten, die im Code gebraucht werden, damit Server und Client fehlerfreie Kommunikation betreiben können. Sonst könnte z.B. der Fall eintreten dass der Client probiert sich zu verbinden obwohl der Server noch nicht bereit ist.

Bei der Verarbeitung der "return-Werte" wird lediglich die Ausgabe des Clients geparst. Der Unterschied zwischen Client und Server kann hierbei vernachlässigt werden, da beide die Werte gleich gemessen und beberechnet werden.

2.3. Graphische Darstellung der Ergebnisse

In diesem Teil der Praxisarbeit wird kurz vorgestellt wie die Daten graphisch dargestellt werden, um schnelle Auswertung und Vergleiche von Messwerten zu ermöglichen.

GnuPlot ist ein Command-Line Grafik-Programm für Linux, OS/2, MS Windows, OSX, VMS und viele andere Plattformen. Es ist ursprünglich für Studenten und Wissenschaftler geschaffen worden, um Ihnen die Möglichkeit

[8] Der dazugehörige Code ist im Anhang

zu geben Mathematische Funktionen und Daten interaktiv visualisieren zu können. [9]

Aufgrund der Tatsache das GnuPlot Command-Line und damit auch Script orientiert ist, hat GnuPlot den Vorzug zu den 'Office-Pendants' zur graphischen Auswertung erhalten. Mit Scripten in denen nur die notwendigen Beschriftungen und Dateinamen geändert werden müssen, konnte so die Entwicklungs- und Arbeitszeit für die Graphische Darstellung sehr gering gehalten werden.

Die benötigten Diagrame für eine graphische Darstellung zeigen stets die typischen Latenzwerte von rdma_lat in Abhängigkeit von verschiedenen Payloads. Dazu werden die Payloads auf die x-Achse, und die Latenzzeiten auf die y-Achse gelegt. Der Datensatz wird als Linie dargestellt.

[9] http://www.gnuplot.info

3. System X

Thema dieses Teils des Praxisberichts sind die Latenzmessungen auf System X. Im ersten Abschnitt geht es um den ersten Kontakt mit dem relevanten Code, in diesem Fall der System-Library "libmlx4", über welche BlueFlame ausgeschaltet werden kann. Anschließend geht es um den Vergleich und die Auswertung der Messwerte mit und ohne BlueFlame.

3.1. Messungen auf System X - Implementierung

In diesem Abschnitt wird der bestehende Treiber Codes im Userspace(System-Libraries) genauer betrachtet. Daraufhin soll im Code die Nutzung von BlueFlame ausgeschaltet werden, um Vergleichs-Messungen mit und ohne BlueFlame zu ermöglichen.

3.1.1. System-Libraries

Da BlueFlame auf x86 bereits implementiert ist, muss sich zuerst mit dem Code der *System-Library*[10] auseinander gesetzt werden. Bei der Kommunikation mit Mellanox ConnectX Devices spielen hier 2 System-Libraries eine Rolle:

libmlx4 ist ein Userspace Treiber für Mellanox ConnectX InfiniBand HCAs. Es ist ein "plug-in" Modul für libibverbs das Programmen erlaubt die Mellanox Hardware direkt aus dem Userspace zu benutzen.[11]

[10] Programmbibliothek
[11] http://www.openfabrics.org/downloads/libmlx4/

libibverbs ist eine Library die es Userspace Prozessen erlaubt 'RDMA verbs' wie in *IAS*[12] und *RDMA Protocol Verbs Specification*[13] zu benutzen. Dies beinhaltet direkten Hardware Zugriff vom Userspace auf den IB Adapter. Damit diese Library sinnvoll benutzbar ist, sollte auch noch ein Device spezifisches Modul installiert sein.[14]

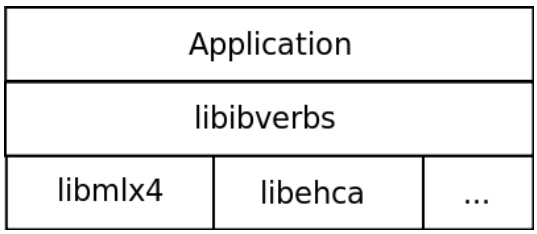


Abbildung 3.1.: Zusammenspiel Libraries

libibverbs ist also Allgemein für Kommunikation mit Prozessen zuständig, und enthält die Verbs(Funktionen), die für die Kommunikation vom Userspace aus benutzt werden können. Wogegen libmlx4 von libibverbs aufgerufen wird, Device-spezifische(z.B. BlueFlame) Einzelheiten enthält und als Bindeglied zwischen libibverbs und Kernel benutzt wird.

3.1.2. libmlx4

Der Device-spezifische Code und damit auch jeglicher BlueFlame Code befindet sich in libmlx4. Davon beinhalten 2 Files BlueFlame spezifischen Code:

mlx4.c Dieses File enthält wichtige Zuweisungen, wie die der Operationen von libibverbs-Funktionen auf libmlx4-Funktionen oder die Vendor-/Device-ID Tabelle über die die genauen Mellanox Devices identifiziert werden können. Außerdem wird hier die Funktion mlx4_alloc_context(struct ib_dev, int) definiert, in welcher die BlueFlame page allokiert wird.

Listing 3.1: Code-Ausschnitt mlx4.c

[12] Infiniband Architecture Specification - Beschreibt die Kommunikation zwischen CPUs und Devices
[13] Beschreibt abstrakt das RDMA Interface
[14] http://rpmfind.net/linux/rpm2html/search.php?query=libibverbs

```
static struct ibv_context *mlx4_alloc_context(struct ibv_device *ibdev,
    int cmd_fd)
{
  struct mlx4_context        *context;
  struct mlx4_alloc_ucontext_resp resp;
   [...]
  if (ibv_cmd_get_context(&context->ibv_ctx, &cmd, sizeof cmd,
        &resp.ibv_resp, sizeof resp))
    goto err_free;
   [...]
  if (resp.bf_reg_size) {
    context->bf_page = mmap(NULL, to_mdev(ibdev)->page_size,
          PROT_WRITE, MAP_SHARED, cmd_fd,
          to_mdev(ibdev)->page_size);
    if (context->bf_page == MAP_FAILED) {
      fprintf(stderr, PFX "Warning: BlueFlame available, "
        "but failed to mmap() BlueFlame page.\n");
        context->bf_page   = NULL;
        context->bf_buf_size = 0;
    } else {
      context->bf_buf_size = resp.bf_reg_size / 2;
      context->bf_offset   = 0;
      pthread_spin_init(&context->bf_lock,
          PTHREAD_PROCESS_PRIVATE);
    }
  } else {
    context->bf_page   = NULL;
    context->bf_buf_size = 0;
  }
   [...]
}
```

In diesem Code-Ausschnitt von mlx4_alloc_context(...) wird der struct mlx4_alloc_ucontext_res über die ibv_cmd_get_context(...) Funktion definiert. Dieser struct enthält

die Integer Variable bf_reg_size, welche die Registergröße der BlueFlame-Register[15] enthält.(Für unsere ConnectX 1-3 Adapter immer 512 Byte)

Wenn die Register größer 0 ist wird anschließend die BF-Page gemappt, und im Context ein Zeiger auf sie gehalten(context->bf_page). Bei Problemen gibt es eine Fehlermeldung und die BlueFlame Variablen werden auf 0 gesetzt, sonst wird die Buffersize auf ein halbes Register gesetzt, der Offset auf 0 und ein BlueFlame Lock initialisiert, der dazu dient Nebenläufigkeitsprobleme zu verhinden.

Im Falle das als Registergröße 0 zurückgegeben wird, wird ebenfalls bf_page und bf_buf_size auf NULL bzw. 0 gesetzt. Das wird also immer gemacht wenn BlueFlame nicht benutzt werden kann. Dadurch liegt die Vermutung nahe das damit die Benutzung von BlueFalme ausgeschaltet werden kann.

qp.c enhält Definition von verschiedenen Funktionen rund um das Queue-Pair(qp), also um Senden und Empfangen von Daten. Die wichtige Funktion für BlueFlame ist die mlx4_post_send(...) Funktion, welche den Code zum schreiben der Daten auf das Device enthält.

Listing 3.2: Code-Ausschnitt qp.c

```
int mlx4_post_send(struct ibv_qp *ibqp, struct ibv_send_wr *wr,
        struct ibv_send_wr **bad_wr)
{
   [...]
  if  ([...]  && size < ctx->bf_buf_size / 16 ) {
     [...]
    mlx4_bf_copy(ctx->bf_page + ctx->bf_offset, (unsigned long *) ctrl
        , align(size * 16, 64));
    ctx->bf_offset ^= ctx->bf_buf_size;
     [...]
  } else if (nreq) {
     [... Senden ohne BF, mit Doorbell...]
  }
```

[15] Siehe zum Aufbau Kapitel 3.3.

```
    [...]
}
```

Hier können wir sehen dass über ein if-/else Statement über die Benutzung von BlueFlame entschieden wird. Hierbei wird überprüft ob die Größe der zu versendenden Daten kleiner als die bf_buf_size ist, also ob die Daten "klein genug" sind um mit BlueFlame versendet zu werden. Die Variable size wird zuvor in dieser Funktion definiert, und enhält die Größe der Daten in ***Octowords!***[16]. Da bf_buf_size dagegen in Byte angegeben wird, wird diese durch 16 geteilt.

Damit sehen wir dass die Vermutung richtig war, BlueFlame über die bf_buf_size ausschalten zu können. Wenn wir diese(wie in den Fehlerfällen) in mlx4.c auf 0 setzen, wird der Code zum Versenden über BlueFlame nie ausgeführt, sondern immer das Senden ohne BlueFlame.

3.1.3. Code Veränderung

Um die Code Veränderung möglichst gering zuhalten, kann nun der Code in mlx4.c so abgeändert werden, damit dieser "denkt"dass die Karte kein BlueFlame kann. Dazu muss vor der Abfrage ob die bf_reg_size lediglich vor der Abfrage ob sie 0 ist, auf 0 gesetzt werden. Dadurch läuft der Code immer in das else-Statement in welchem bf_page gleich NULL und bf_buf_size gleich 0 gesetzt werden. Daraufhin wird bei mlx4_post_send(...) der Code zum Versenden über BlueFlame nie ausgeführt.

3.2. Messungen auf System X - Ergebnisse

Mit den im vorherigen Kapitel beschriebenen Veränderungen am Code, können 2 Versionen der libmlx4 gebaut werden(mit und ohne BlueFlame). In Abbildung 3.2 und der Tabelle 3.1 können wir die gemessenen Werte genauer betrachten. Dabei ist zu beachten dass die Zeiten, die jeweiligen Durchschnitts Zeiten sind. "Zeit mit BF" sind dabei die Messungen mit Original-Library und

[16] **Octowords!**

"Zeit ohne BF" die modifizierte Library, bei der BlueFlame nichtmehr genutzt wird.

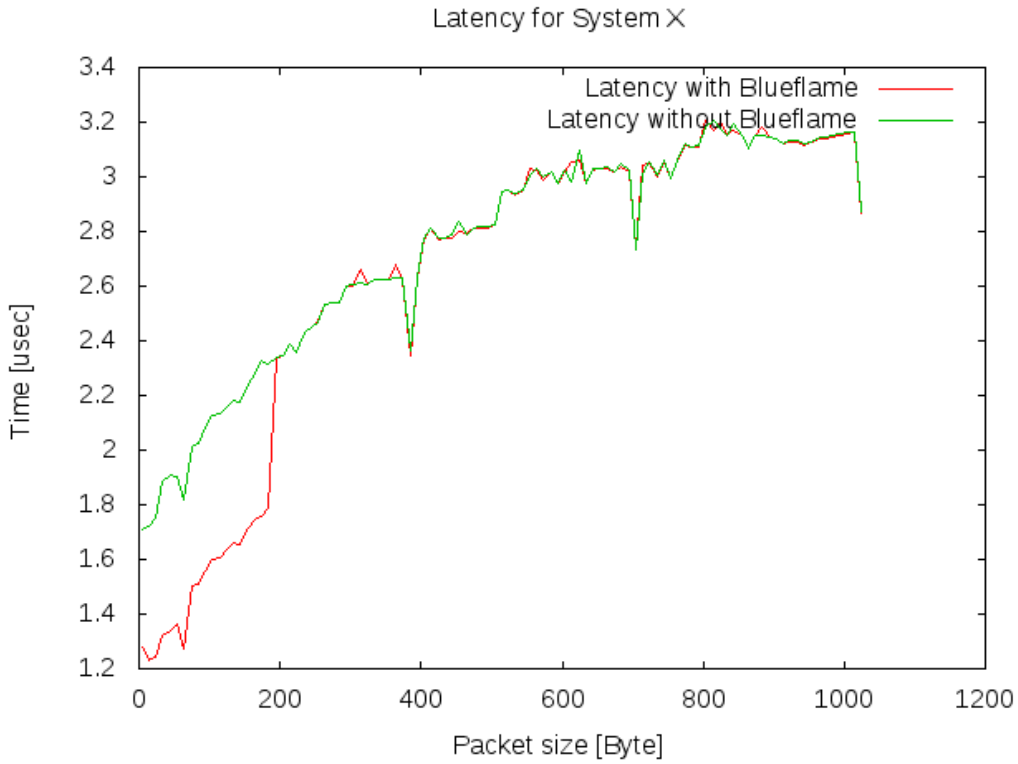


Abbildung 3.2.: System X - mit/ohne BlueFlame

Payloadsize[Byte]	Zeit mit BF[usec]	Zeit ohne BF[usec]	Verbesserung
1-192	1.51	2.06	26.78%
192 - 1024	2.92	2.92	0.00%

Tabelle 3.1.: System X - mit/ohne BlueFlame

3.2.1. Interpretation

Anhand des Diagramms und der Tabelle kann erkannt werden dass bis 192 Byte Payload die Daten mit BlueFlame gesendet werden. Ebenfalls können wir daran erkennen dass BlueFlame uns einen klaren Geschwindigkeitsvorteil bringt. Dieser ist im Durchschnitt 26.78% bzw. 0.55 usec.

3.3. Messungen auf System X - Verbesserung

Wenn Daten via BlueFlame versendet werden von der Library noch einige Daten hinzugefügt, die für die Karte bestimmt sind(sozusagen ein "Header" für die Karte). Diese Daten beinhalten die *QPN*[17], die Größe des Pakets, den *Opcode*[18], einige Flags... Daraus folgt:

$$DatenKarte = Payload +'' DataHeader'' \tag{3.1}$$

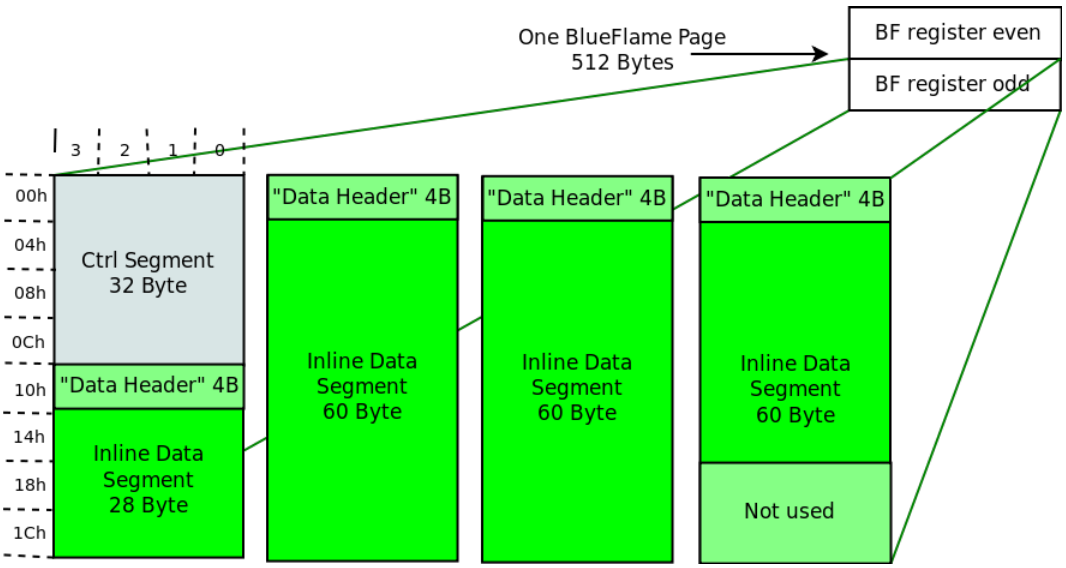


Abbildung 3.3.: Aufbau BlueFlame Paket

In Abb. 3.3 können wir sehen wie sich ein BlueFlame Paket zusammensetzt. Ein Paket kann höchstens die Größe des dafür vorgesehenen Registers annehmen. Das ist die Hälfte einer BlueFlame Page, und damit 256 Byte. Die Größe wird auf je 64 Byte angepasst, deshalb die Zeichnung des Pakets in 4 64 Byte Blöcken. Ein Beispiel: Bei Paketen einer Größe von 40 Byte wird der Karte trotzdem ein 64 Byte Paket übermittelt, dessen letzten 24 Bytes dann nur Nullen enthalten.

Jedes Paket enthält einen Ctrl-Segment(gleichzusetzen mit "DataHeader") im ersten 64 Byte Block, welches 32 Byte groß ist. Dazu kommt in jedem 64

[17] Queue Pair Number - ist i.d.R. direkt einem Prozess auf Sender- und Empfängerseite zugeordnet
[18] Ein Code der aussagt welche Operation ausgeführt werden soll

Byte Block ein eigenes Inline Data Segment, welches wieder ein 4 Byte Header hat(Größe und Flags) und dazu die eigentlichen Daten enthält. Daraus folgt für das größtmögliche Paket:

$$"DataHeader" = 32(Ctrl-Segement)+4*4(DataHeaders) = 48Byte \quad (3.2)$$

$$MaxDatenKarte = 192(gemessen) + 48 = 240Byte \quad (3.3)$$

Die (zum Teil aus Messwerten) berechnete maximale Datenmenge, die in das BlueFlame Register geschrieben werden kann, ist also 240 Byte. Allerdings ist das BlueFlame Register 256 Byte groß, d.h. wir haben hier 16 ungenutzte Byte.

Verantwortlich dafür ist der Code aus qp.c den wir in Listing 3.2 gesehen haben:

Listing 3.3: Code-Ausschnitt qp.c

```
if(   [...]   && size < ctx->bf_buf_size/16){
  //Sending via BlueFlame
}else ...
```

Die Variable size ist die Größe der gesamten Daten in Octowords(16 Byte). Das bedeutet im eigentlich möglichen Maximalfall(241-256 Bytes) wäre size = 16, da 16 Ocotwords 256 Bytes sind. Die bf_buf_size ist immer 256 wenn BlueFlame benutzt werden kann. Im maximal Fall vergleicht diese Zeile also ob 16 kleiner 16 ist, evaluiert das zu false, springt in den else Zweig und führt Blueflame nicht aus.

Als Lösung dieses Bugs kann ein einfaches <= anstatt < verwendet werden, damit Blueflame bis zur maximalen Größe verwendet wird.[19]

[19]Es wurde eine Mail an die zuständigen Entwickler von Mellanox geschickt

4. System Z

Thema dieses Abschnitts des Praxisberichts sind die Latenzmessungen auf System Z. Im ersten Abschnitt geht es um die Problematik die es in diesem Zusammenhang auf der System Z gibt. Anschließend ist Implementierung in User- und Kernelspace das Thema. Außerdem sollen Ergebnisse und Debugging vorgestellt werden.

4.1. Problemstellung

4.1.1. Kommunikation

Für x-86 Architekturen gibt es eine bereits bestehende Implementierung. Die Frage dieses Abschnitts ist welche Anpassungen an dieser Implementierung vorgenommen werden müssen und wieso diese Nötig sind.

Eine System Z kann durch *LPAR*[20]S verwendet werden, welche eigene (virtuelle) Server/Computer darstellen. Diese LPARs können sich auch ein Stück Hardware, wie z.B. eine I/O Karte teilen. Zugriff aus mehreren Betriebssystemen auf einen Adapter ist allerdings ein möglicher Punkt für Nebenläufigkeitsprobleme. Damit diese Nebenläufigkeitsprobleme für einen Adapter vermieden werden können, wird die Kommunikation von Userspace mit Hardware normalerweise über einen Treiber im Kernelspace geführt.

Auf x86-Architekturen gibt es zusätzlich zur Kommunikation über den Treiber noch einen "direkten" Weg. Über den mmap(...) Systemcall kann aus der Userspace Anwendung ein Bereich auf der Hardware allokiert werden. Dieser Bereich wird anschließend in den Adressebereich des Prozess gemappt. Das bedeutet dass der Prozess eine für ihn sichtbare und beschreibbare Adresse bekommt, über

[20] Eine logische Partition

die im Hintergrund direkt auf die entsprechende Hardware geschrieben wird. Daraufhin kann die Userspace Anwendung quasi direkt (über den gemappten Bereich) an die Hardware schreiben.

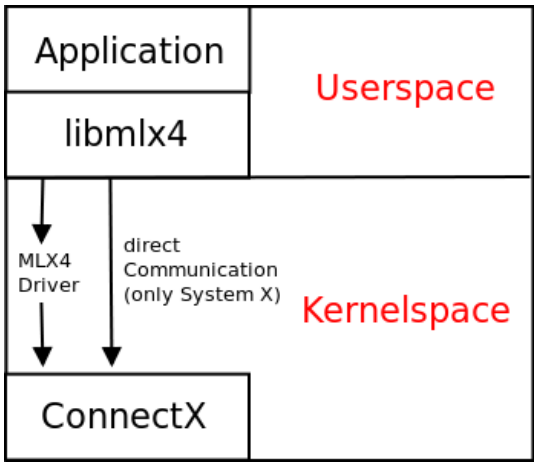


Abbildung 4.1.: Kommunikation von Userspace zu Hardware

Dieser Kommunikationsweg ist auf SystemZ, auf Grund hohen Virtualisierung nicht erlaubt, um Nebenläufigkeitsprobleme zu vermeiden. Deswegen muss der Code der eine über mmap(...) enthält für System Z abgeändert werden und Teile von User- in Kernelspace verlagert werden.

4.1.2. Write Combining/Store Block

Ein zweiter, für uns relevanter, Unterschied ist das schreiben großer Datenmengen. Mit BlueFlame sollen größere Datenmengen(max. 256 Byte) zur Hardware geschrieben werden. Dies sollte möglichst in einem Schreib-Zyklus geschehen um unnötigen Datentransfer zu vermeiden und minimale Latenzen zu erreichen.

Dafür gibt es auf System X einen Mechanismus namens Write Combining(WC) und auf System Z Store Block(SB). Da diese Mechanismen unterschiedliche Funktionen benötigen muss der Code von WC auf SB abgeändert werden.

4.2. Implementierung

4.2.1. Konzept

Das Konzept um BlueFlame auf System Z zu implementieren kann in einzelne Teilschritte zusammengefasst werden.

- Übergabe der Daten in Kernelspace
- Allokieren von BlueFlame page in Kernel
- Schreiben der Daten an Adapter via Store Block

4.2.2. Übergabe der Daten in Kernelspace

Die Daten(Ctrl-Segement, InlineData und Offset) müssen vom Kernel aus an den Adapter geschrieben werden. Das bedeutet dass sie zuvor bereits in den Kernelspace geladen werden müssen. Vereinfacht bedeutet es dass Daten von einer Funktion an eine andere übergeben werden müssen, was keine schwere Aufgabe darstellt. Allerdings können Userspace Pointer nicht direkt im Kernel dereferenziert werden[21]. Dafür gibt es folgende Gründe:

- Abhängig von Architektur und Konfiguration könnte der Pointer nicht valide sein, wenn im Kernel darauf zugegriffen wird. Ein Grund dafür könnte ein fehlendes Mapping sein oder ein Zeigen des Pointers auf andere, vlt. zufällige Adresse zeigt.
- Userspace Memory ist in "pages" aufgebaut, was bei direktem dereferenzieren aus dem Kernel zu einem "page fault" und somit zu einem Absturz des Prozesses führen könnte.
- Bei direkter Derefenzierung gibt es keine Kontrolle über die Daten die geladen werden, da Anwendungen Fehler haben können oder böswillig sein könnten. In diesem Fall würde ein direkter Zugriff eine "Tür" zum Kernel öffnen.

[21] Linux Device Drivers Seite 63 - Chapter 03

Deshalb werden Daten über ein spezielles Kernel-Macro copy_from_user(...) geladen. Dieses Macro kümmert sich um die eben genannten Aspekte. Im IB Device Driver Code gibt es ein spezielles Macro ib_copy_from_udata(...) welches diese Aufgabe übernimmt.

Ein zweiter Punkt weswegen die Übergabe der Daten keinen trivialen Teil der Implementierung darstellt ist die Tatsache dass für Kommunikation zwischen Kernel- und Userspace ein Interface bereitgestellt werden muss. Aus zeitlichen Gründen wurde im Zuge dieser Praxisarbeit darauf verzichtet ein neues Interface zu implementieren, sondern ein bereits bestehendes Interface benutzt um Pointer Größe und Offset in den Kernelspace zu geben.

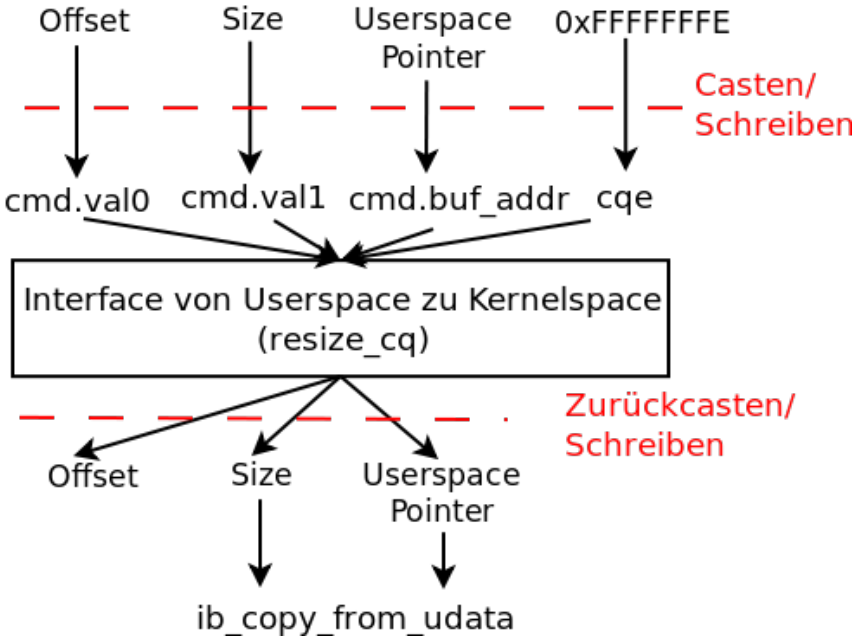


Abbildung 4.2.: Interface zwischen Kernel- und Userspace

Abb. 4.2. zeigt das benutzt Interface. Die Werte wurden in nicht benutzte Variablen des "command struct" gecastet und geschrieben. Zusätzlich wird in die Integer Variable cqe ein sonst nicht benutzter Wert (0xFFFFFFFE) geschrieben. Dieser Wert kann über ein if-Statement im Kernelspace abgefangen werden, und anschließend die Variablen "zurückgecastet" und die Daten über ib_copy_from_udata in den Kernelspace geladen werden.

4.2.3. Allokieren der BlueFlame page in Kernel

Damit in das passende BlueFlame Register geschrieben werden kann, muss im Kernel bereits die BlueFlame page richtig allokiert und für den späteren Gebrauch referenziert sein.

Dies geschieht über den Context. Der Context ist ein struct dass relevante Daten für das Device hält und kann je einem Prozess(der dass Device nutzt) zugeordnet werden. Jeder Context enthält eine User Adress Region(UAR), die Teil des PCI Address Bereichs ist und für spezielle Daten(z.B. von *HCA*[22] und CPU) bereitgehalten wird, und damit einer BlueFlame page ähnelt. Bei jedem allokieren eines Contexts wird die Funktion mlx4_uar_alloc aufgerufen, in der dann der Code für das Allokieren der BlueFlame page ausgeführt werden kann.

Listing 4.1: Code-Ausschnitt pd.c

```
int  mlx4_uar_alloc(struct mlx4_dev *dev, struct mlx4_uar *uar)
{
    [...]
  uar->bf_map = ioremap(pci_resource_start(dev->pdev,2) + (dev->
      caps.num_uars << PAGE_SHIFT), PAGE_SIZE);
  uar->bf_data = kmalloc(256, GFP_ATOMIC);
    [...]
}

void mlx4_uar_free(struct mlx4_dev *dev, struct mlx4_uar *uar)
{
    [...]
  iounmap(uar->bf_map);
  kfree(uar->bf_data);
}
```

In Listing 4.1. sehen wir einen Teil des angepassten Codes. Das struct mlx4_uar wurde um 2 void* namens bf_map(Zeiger auf BlueFlame page) und bf_data

[22]Host Channel Adapter

erweitert. bf_data dient als *Buffer*[23] um die Daten aus dem Userspace zu laden.

Um die BlueFlame page zu mappen wird die Funktion ioremap(...) verwendet. Diese bekommt als ersten Parameter eine Wunschadresse zum Mappen der Daten und als zweiten Parameter die Größe des Bereichs der Gemappt werden soll. Für die "Wunschadresse" gilt: Der PCI-Speicherbereich pro device ist in 2 *BAR*[24]s Aufgeteilt. UARs und BF pages liegen auf dem 2. Bar. Zuerst kommen die UARs anschließend die BF pages. Deshalb wird mit pci_resource_start(dev,2) die Anfangsadresse unseres 2. Bars genommen und darauf noch die Anzahl der UARs(geschiftet um PAGE_SHIFT um es in eine Adresse umzuwandeln) addiert. Damit erhält man den Anfang der BF page Adressen.

bf_data kann über kmalloc(...) allokiert werden, was mit malloc(...) zu vergleichen ist, aber spezielle Aspekte für den Kernel beachtet.

Bei Beendigung des Prozesses müssen die allokieren Bereiche wieder freigegeben werden. Dies passiert über iounmap(...) und kfree(...)

4.2.4. Schreiben der Daten an Adapter via Store Block

Nachdem Daten in den Kernelspace geladen sind und die BF page für den Prozess allokiert ist, brauchen die Daten nurnoch an die Adresse geschrieben werden. Dies ist die Code-Ärmste Teil der Implementierung, da hierfür die Funktion zpci_memcpy_toio(...) benutzt werden, welche im Hintergrund Store Block Operation(en) ausführt. In dieser Funktion wird Store Block auch in mehrere Operationen zerteilt, falls das 128 Byte Maximum überschritten wird.

4.3. Erste Ergebnisse

Mit den zuvor beschriebenen Implementierungen war zu Beobachten dass Daten richtig gesendet wurden. D.h. das als Testcase benutzte Programm rd-

[23] Ein Bereich der zur kurzfristigen Datenspeicherung genutzt wird.
[24] Base Address Register - bei PCI die einzelnen Speicherbereiche

ma_lat sendete und empfing die Pakete richtig und beendete sich ohne Fehlermeldung. Folgende Ergebnisse ergaben sich aus den darauf folgenden Messungen:

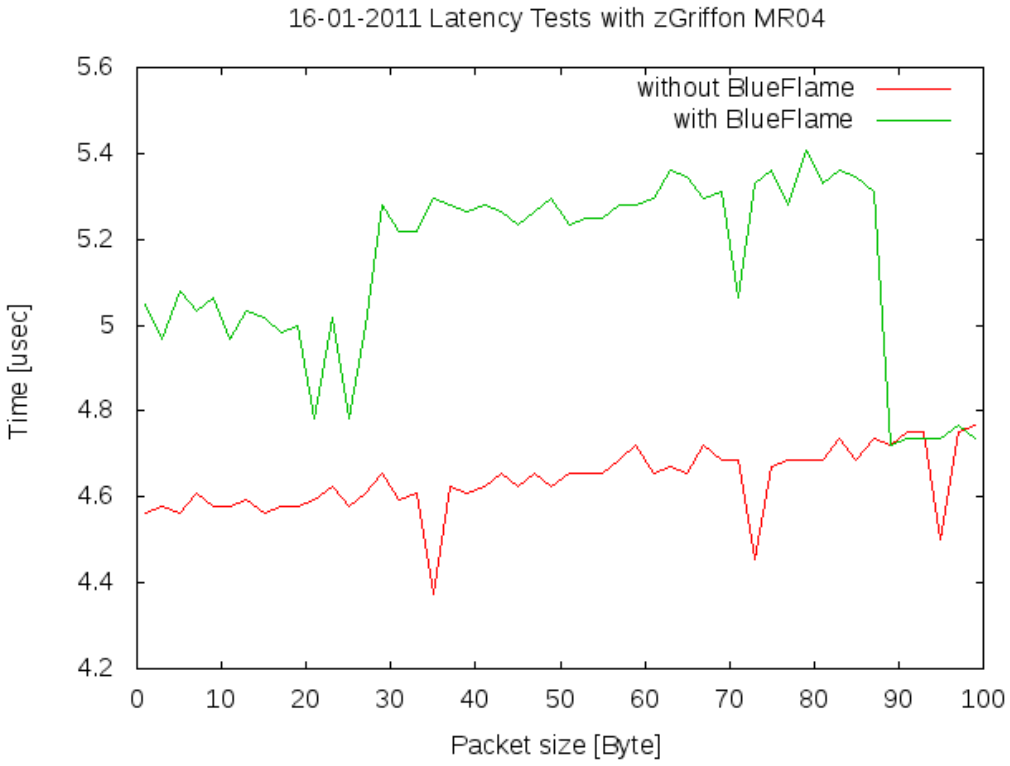


Abbildung 4.3.: Erste Ergebnisse auf System Z

Bei Abb. 4.2. ist zu beachten das BlueFlame bereits bei 88 Byte Payload ausschält. Dieser Wert wurde manuell gesetzt, da 128 Byte die maximale Größe einer Store Block Operation sind. Daraus folgt bei 32 Byte Ctrl Segment und 2 Inline Data Segments mit je 4 Byte Header, noch 88 Byte maximale Payload.

4.3.1. Interpretation

An Abbildung 4.3. ist klar zu erkennen dass das Senden mit der ersten Implementierung von BlueFlame langsamer ist, als bei Senden der Daten ohne BlueFlame. Zu unterscheiden sind 2 Bereiche. zwischen einer payload von 1-28 Byte wird eine 64 Byte Store Block Operation verwendet. Das Senden der Daten ist in diesem Bereich ca. 0.5 usec langsamer geworden. Von 29 - 88 Byte

wird eine 128 Store Block Operation verwendet. Hier sind ca. 0.7 usec höhere Latenzen zu sehen.

4.4. Debugging

Da die Nutzung von BlueFlame theoretisch eine Verringerung der Latenzen bedeuten sollte, gilt es herauszufinden weshalb die Latenzen bei der Nutzung von BlueFlame auf System Z größer geworden ist:

- Auf System X wird Write Combining zum schreiben der Daten auf den Adapter verwendet. Es gibt keine genauen Werte ob Store Block für System Z eine ähnlich gute Perfomance bietet. Deshalb liegt es nahe die Performance von Store Block zu überprüfen. Dazu können die Zeiten einer Store Block Operation ähnlich der bisherigen Latenzmessungen vermessen werden, um Rückschlüsse ziehen zu können.
- Um mögliche Verständnisfehler oder Fehler im Code ausschließen zu können, kann mit einem PCI-Analyzer der genaue Datentransfer des PCI-Adapters aufgezeichnet werden. Diese Lösung bedeutet allerdings größeren Zeitaufwand und zusätzliche Hardware(den PCI-Analyzer).

4.4.1. Store Block Perfomance

Zur Vermessung der Store Block Operation kann der selbe Mechanismus wie für die Messung der Gesamt-Latenzen verwendet werden. Das bedeutet dass bei jedem Senden die Zeit einer Store Block Operation gemessen wird. Anschließend kann der Durchschnitt der 1000 Aufrufe eines rdma_lat Laufs berechnet werden. Dieser Wert ist die Durchschnittszeit von 1000 Store Block Operationen für eine Payload size. Dies kann dann gleich wie bisherige Latenzwerte mit dem bash-Script verarbeitet und anschließend graphisch ausgewertet werden.

In Abb. 4.4. kann man die Store Block Performance erkennen mit ca. 0.5-0.7 usec entspricht sie genau den Werten um welche das Senden mit BlueFlame langsamer geworden ist. Deshalb liegt die Vermutung nahe dass nicht wie zu erwarten ist gesendet wird, was nur mit dem PCI-Analyzer zu erkennen ist.

Abbildung 4.4.: Store Block Performance

4.4.2. PCI-Analyzer

Den PCI-Analyzer kann man sich als Stück Hardware vorstellen dass physikalisch an einen PCI-Slot (oder in unserem Fall einer extra Karte mit PCI-Slot) angeschlossen werden kann. Durch diese Leitungen(je eine Leitung für aus- und eingehenden Datenverkehr) kann alles was an Datentransfer an diesem PCI-Slot stattfindet aufnehmen und über an einen angeschlossenen Computer weitergeben. Dieser kann mit passender Software die gesammelten Daten auswerten und sie für Menschlich verständlich anzeigen.

Abb.4.5. zeigt einen Ausschnitt des PCI-Analyzers bei der Messung mit rdma_lat für eine , der gewählte Ausschnitt beinhaltet nur die Spalten die für diese Praxisarbeit relevant sind und ist nur ein Sende Zyklus der 1000 von rdma_lat.

Timestamp Ein Timestamp bei ausführen der Operation.

Timestamp	102/1 : Type	104/1 : Type	Length	
9,324 us	Memory Write		0 08	Data=00 00 00 48 00 00 00 00 00 00 00 00 00
11,128 us		Memory Write	0 01	Data=00 00 49 00
11,684 us	Memory Read		0 04	
12,356 us		Completion with ...	0 04	Data=00 00 00 08 00 00 00 03 00 00 00 0C 00
12,836 us	Memory Read		0 7C	
13,468 us		Completion with ...	0 3C	Data=00 00 00 00 80 00 A0 01 00 00 1A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13,600 us		Completion with ...	0 40	Data=FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13,892 us	Memory Write		0 01	Data=00 90 00 00
14,084 us	Memory Write		0 08	Data=00 00 00 49 00 00 00 00 00 00 00 00 16

Abbildung 4.5.: PCI Analyzer Output(1)

102/1 oder 104/1 : Type Die Namen der auszuführenden Operation, jeweils in der Spalte des Ports auf dem sie ausgeführt wird.

Length Die Länge der Daten in Hexadezimal und 4 Byte Wörtern. Bsp: 10h = 16d = 64 Byte

letzte Spalte Die genauen Daten(abgeschnitten)

Abb. 4.5. zeigt dass am Anfang ein 64 Byte Store Block gemacht wird(der die Daten in die BlueFlame page schreibt). Danach sollten der Adapter die Daten selbstständig verschicken und erst auf der Empfängerseite wieder Datentransfer zu beobachten sein(PCI-Protokoll). Allerdings können Memory Reads(gleichzusetzen mit DMA) und ihre Completion erkannt werden, welche im Falle einer Benutzung von BlueFlame nicht vorkommen sollten.

Das Fazit dieser Kommunikationsaufzeichnung ist dass die Daten per Store-Block an den Adapter geschrieben werden, dann allerdings die Daten trotzdem per DMA nachgeladen und ohne die Nutzung BlueFlame versendet werden. Eine mögliche Erklärung hierfür ist ein ”Fallback” des Adapters der mit

einer gewissen Intelligenz einen Fehler in den Daten erkennt, deswegen nicht per BlueFlame sendet, die QPN aus den gesendeten Daten erkennt und über diese dann (wie beim normalen Senden) die Daten über DMA nachlädt und versendet. Dieses Verhalten erklärt auch dass die Latenzzeiten relativ genau um die Zeit der Store Block Operation zugenommen haben. Da normal gesendet wird + eine Store Block Anweisung.

4.4.2.1. Lösung

Da aus den Daten zu schließen war dass BlueFlame nicht benutzt wird, musste nach dem Problem gesucht werden. Einen Hinweis gaben die Adressen der BlueFlame page auf dem PCI-Bus, die der PCI-Analyzer angezeigt hat. Hier war zu erkennen dass trotz Offset stets zwei aufeinander folgende Send-Operationen die selbe Adresse benutzten.

Später stellte sich heraus dass bei der Allokierung der BlueFlame page ein Fehler war.

Listing 4.2: Code-Änderung pd.c

```
int mlx4_uar_alloc(struct mlx4_dev *dev, struct mlx4_uar *uar)
{
   [...]
  // old Code:
  // uar->bf_map = ioremap(pci_resource_start(dev->pdev,2) + (dev
      ->caps.num_uars << PAGE_SHIFT), PAGE_SIZE);
  // new Code:
  uar->bf_map = ioremap(pci_resource_start(dev->pdev,2) + (dev->
      caps.num_uars << PAGE_SHIFT)+ (uar->index <<
      PAGE_SHIFT), PAGE_SIZE);
   [...]
}
```

Listing 4.2. zeigt die kleine Codemodifizierung die vorgenommen werden musste. In der ersten Version des Codes wurde als erster Parameter für io_remap(...) eine Adresse übergeben, die auf den Anfang der BlueFlame page(s) zeigt.

Es gibt jedoch verschiedene BlueFlame pages und es muss die BlueFlame page mit dem gleichen Index wie die dazugehörige UAR allokiert werden. Deshalb wird der Index der UAR geshiftet und addiert. Damit wird die BlueFlame page mit dem selben Index wie die dazugehörige UAR allokiert.

Ob dabei wirklich mehrere BlueFlame pages und Register als Hardware auf dem Adapter sind, oder diese nur virtuell in den Adressraum des Registers gemappt sind, ist dabei nicht zu erkennen, für den Zweck dieses Praxisberichts aber auch irrelevant.

4.4.2.2. Neue PCI-Analyzer Aufzeichnung

Timestamp	102/1 : Type	104/1 : Type	Length	
9,364 us		Memory Write	0 10	Data=00 25 A7 08 00 00 48 03 00 80 00 00 01 A8 00 00 00 00 00 00
9,716 us	Memory Write		0 01	Data=00 A8 00 00
9,908 us	Memory Write		0 08	Data=00 00 00 48 20 00 00 00 00

Abbildung 4.6.: PCI Analyzer Output(2)

Abb.4.6. zeigt die Ausgabe des PCI-Analyzers, nach den beschriebenen Code-Änderungen. Auf Sender Seite nur wird 1 Memory Write und keine Memory Reads gemacht. Das bedeutet dass die Daten per BlueFlame an den Adapter geschrieben werden.

4.5. Ergebnisse auf System Z

Payload[Byte]	SB OPS	t ohne BF[usec]	t mit BF[usec]	Veränderung
1-28	64B	4.6	2.6	43%
29 - 88	128B	4.7	2.9	38%
89 - 148	64B + 128B	4.9	3.6	26%
149 - 208	128B + 128B	5.1	4.0	21%

Tabelle 4.1.: System Z - mit/ohne BlueFlame

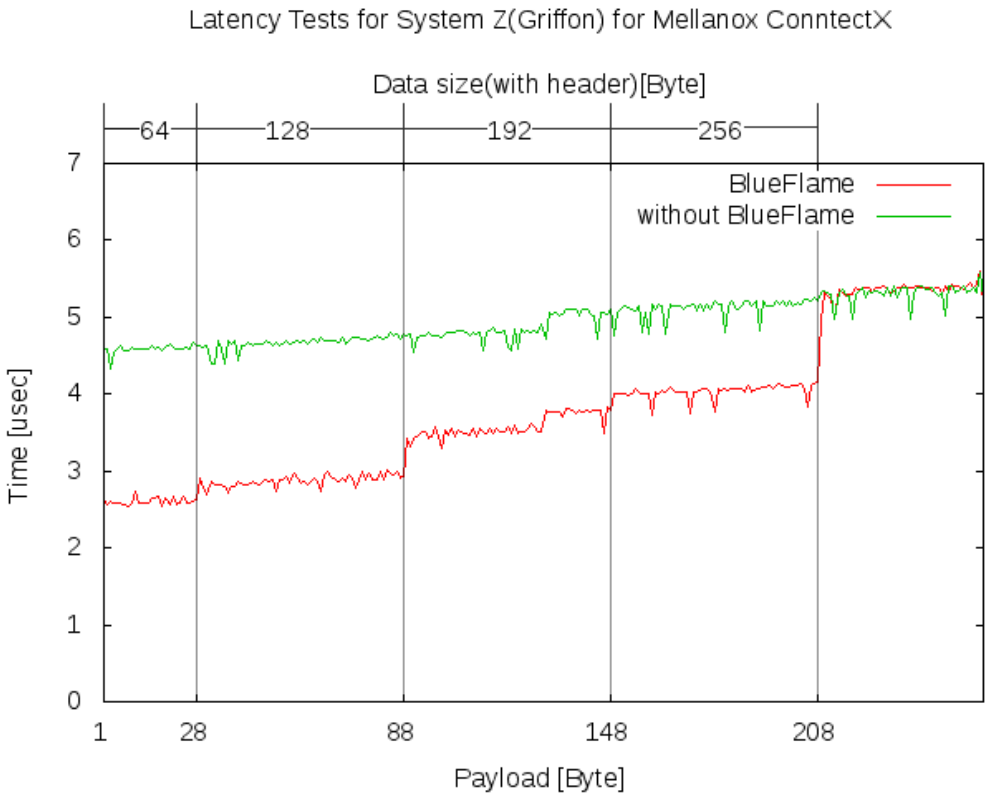


Abbildung 4.7.: Finale Ergebnisse auf System Z

4.5.1. Interpretation

In Abb. 4.7. und Tabelle 4.1. sind die Ergebnisse für die Latenzmessungen auf System Z dargestellt. Die Ergebnisse sind zur besseren Analyse in 4(5) Abschnitte eingeteilt, abhängig von den StoreBlock Operationen die in diesem Bereich genutzt werden. So können die Daten besser ausgewertet werden.

Es ist zu erkennen dass BlueFlame die Latenzzeiten auf System Z deutlich verringern kann . Der Gewinn wird mit größeren StoreBlock Operationen immer kleiner (von 43 % Gewinn auf 21%) da die StoreBlock Operationen einen größeren Overhead haben als die DMA Zugriffe(Implementierung von StoreBlock im *Millicode*[25]. Ab 208 Byte wird BlueFlame nicht länger genutzt und die Latenzzeiten sind quasi identisch.

[25]Eine Firmware der IBM System Z

5. Fazit und kritische Bewertung

Für viele Anwendungen auf System Z sind aber 256 Byte Daten wichtig, weswegen es wünschenswert wäre BlueFlame auch für Größere Pakete einsetzen zu können. Hierfü müssen jedoch 2 Dinge geschehen:

- Es werden Adapter mit einer größeren BlueFlame Page gebraucht, da diese das Maximum für die Nutzung von BlueFlame vorgibt.
- Die Performance von StoreBlock auf System Z muss besser werden, da aufgrund des Performance Unterschiedes zwischen DMA und StoreBlock der Gewinn für größere Pakete immer kleiner wird.

DMA-Zugriffe laufen über den Adapter, wogegen StoreBlock Operationen von der CPU initialisiert werden. Deswegen ist zu erwarten das die Nutzung von BlueFlame eine größere CPU-Last zur Folge hat. Die CPU-Last ist auf System Z kritisch, da die CPU von verschiedenen LPARs geteilt wird. Deshalb sollte zusätzlich zu dieser Arbeit in der Zukunft die Belastung der CPU mit und ohne BlueFlame ausgemessen werden, damit zwischen höhere CPU-Last und geringerer Latenzzeiten abgewogen werden kann.

Zu guter Letzt muss die Wirtschaftlichkeit dieser Praxisarbeit betrachtet werden. Für IBM ist mit der Arbeit noch kein direkter Business Case verbunden, und ein zukünftiger Einsatz von BlueFlame auf System Z ist damit ungewiss. Allerdings ist mit dieser Arbeit die Tür für eine Nutzung von BlueFlame und mehr Latenz-kritischen Anwendungen auf System Z geöffnet worden, indem genaue Daten über die Latenzzeiten vorliegen.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meinen Praxisbericht mit dem Thema

> *Optimierung eines Linux I/O Treibers für System Z Implementieren und Testen von Mellanox BlueFlame*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Die Ergebnisse der Arbeit stehen ausschließlich dem auf dem Deckblatt angeführten Unternehmen zur Verfügung (**Arbeit mit Sperrvermerk**).

Stuttgart, den 21. Februar 2012

Filip Haase

Abkürzungsverzeichnis

DMA Direct Memory Access - Ein direkter Speicherzugriff ohne CPU

OFED OpenFabrics Enterprise Distribution - Sammlung von Tools und Treibern der OpenFabrics Alliance, die RDMA fördern

System-Library Programmbibliothek

IAS Infiniband Architecture Specification - Beschreibt die Kommunikation zwischen CPUs und Devices

RDMA Remote Direct Memory Access - Der direkte Speicherzugriff ohne CPU von Außen

RDMA Protocol Verbs Specification Beschreibt abstrakt das RDMA Interface

Octoword Ein 16 Byte Word

QPN Queue Pair Number - ist i.d.R. direkt einem Prozess auf Sender- und Empfängerseite zugeordnet

Opcode Ein Code der aussagt welche Operation ausgeführt werden soll

LPAR Eine logische Partition

HCA Host Channel Adapter

Buffer Ein Bereich der zur kurzfristigen Datenspeicherung genutzt wird.

BAR Base Address Register - bei PCI die einzelnen Speicherbereiche

Millicode Eine Firmware der IBM System Z

A. Anhang

Listing A.1: Anhang: Das Bash-Script zum Latenzen messen

```
#!/bin/bash
# copyright 2011 Filip Haase
# implementation for running latency tests with rdma_lat
# with different  packetsizes
# and write the output formatted in a  file

# Function for show help
func_show_help(){
  echo -e " This script  runs rdma_lat with different packet  sizes  and
      writes a  well  formatted output"
  echo -e " Call:${0##/*} [OPTION]... [FILE]"
  echo
  echo -e "\t--min=<min_size>, \tsets the minimal packet size to test
      with in Bytes"
  echo -e "\t\t\t\tdefault: 4"
  echo -e "\t--max=<max_size>, \tsets the maximal packet size to test
      with in Bytes"
  echo -e "\t\t\t\tdefault: 1024"
  echo -e "\t-s=<step_size>, \tsets the step size for the packet sizes  in
      Bytes"
  echo -e "\t\t\t\tdefault: 10"
  echo -e "\t-c, \tmakes the Output comma seperated instead of space
      seperated"
  echo -e "\t-h, \tshows help"
  echo
  echo -e "Without a specified File the 'latencyResult.dat' will  be used as
       output  file "
```

```
  echo
}

# This function tests if the given parameter $1 is a parameter is a number
    or not
# if it's not a number the script terminates with an error message
# if it's a number it should happen nothing and the script continues
func_test_number(){
  if (![[ $1 =~ ^-?[0-9]+$ ]])
  then
    echo "Please enter only valid Parameter"
    echo "$1 is not a number"
    echo -e "\t'${0##/*} -h' for more information"
    exit
  fi
}

# Set the default values
MAX_SIZE=1024
MIN_SIZE=4
STEP=10
OUTPUT_FILE=latencyResult.dat
SEPERATOR=' '

# Go through the arguments given
# and parse them, if they are in the right format they are taken
for ARG in "$@"
do
  case $ARG in
    -max=*)
      func_test_number ${ARG:5}
      MAX_SIZE=${ARG:5};;
    -min=*)
      func_test_number ${ARG:5}
      MIN_SIZE=${ARG:5};;
```

```
    -s=*)
      func_test_number ${ARG:3}
      STEP=${ARG:3};;
    -h)
      func_show_help
      exit ;;
    -c)
      SEPERATOR=',';;
    -*)
      echo "Unknown Option: ${ARG}"
      echo "\t'${0##/*} -h' for more information"
      exit ;;
    *)
      OUTPUT_FILE=$ARG;;
  esac
done



# Give out the taken values
echo "Starting latency test with packet size from ${MIN_SIZE} to ${
    MAX_SIZE} in ${STEP} Byte Steps"
echo "Output goes in ${OUTPUT_FILE}"

# Start with writing some Information about what it is, and what the
    Columns mean
# at the top of the Output file and clear it
echo "# Results for max-packet size ${MAX_SIZE}, min-packet size ${
    MIN_SIZE} and with step-size ${STEP}" > ${OUTPUT_FILE}
echo "# PacketSize, LatTypical, LatBest, LatWorst" >> ${
    OUTPUT_FILE}

# Main loop of this script, each Iteration gets the latency values
# for one packet size, and writes it with the write format in the
    $OUTPUT_FILE
```

```
for ((p_size=${MIN_SIZE}; p_size <= ${MAX_SIZE} ; p_size=p_size +
    ${STEP}))
do
  # Start Server in background and delegate the Output to /dev/null
  rdma_lat --size=${p_size} > /dev/null 2>&1 &

  # Sleep a little time, that the server is guaranteed started, when the
      client starts
  sleep 0.4

  # Let the client connect and get the answer String
  # thereby surpress warnings by sending them to /dev/null
  clientResult=`rdma_lat localhost --size=${p_size} 2> /dev/null `

  # Now seperate the Values for Latency typical, beste and worst
  latTypical=`(echo "${clientResult}")|grep 'Latency typical'`
  latTypical=${latTypical##*:}
  latBest=`(echo "${clientResult}")|grep 'Latency best'`
  latBest=${latBest##*:}
  latWorst=`(echo "${clientResult}")|grep 'Latency worst'`
  latWorst=${latWorst##*:}

  sleep 0.2

  # Write the Values in the ouput file, and directly cut off the 'usec'
      String with % *
  (echo "${p_size}${SEPERATOR}${latTypical% *}${SEPERATOR}${
      latBest% *}${SEPERATOR}${latWorst% *}") >> ${
      OUTPUT_FILE}

  if [ -n "$latTypical" ] && [ -n "$latBest" ] && [ -n "$latWorst" ]
  then
    echo "Packet size: ${p_size} successfully tested and written"
  else
    echo "#Warning: An empty String appeared at packet site: ${p_size}"
```

```
    fi
done
```