Scalable Fabric Interfaces

Sean Hefty
Intel Corporation

OFI software will be backward compatible
Develop an extensible, open source framework and interfaces aligned with ULP and application needs for high-performance fabric services.
Enable..

**Minimal footprint**
Reduced cache and memory footprint

**High performance**
Optimized software path to hardware
- Independent of hardware interface, version, features

**App-centric**
Analyze application needs
- Implement them in a coherent, concise, high-performance manner

**Extensible**
More agile development
- Time-boxed, iterative development
- Application focused APIs
- Adaptable
How can an API affect application scalability?

I’m glad I asked.

Minimal footprint
struct rdma_route {
    struct rdma_addr addr;
    struct ibv_sa_path_rec *path_rec;
    ...
};

struct rdma_cm_id {...};

rdma_create_id()
rdma_resolve_addr()
rdma_resolve_route()
rdma_connect()
Scalable Communication

- *Application* driven communication models
- Reliable unconnected transfers
  - Abstract hardware features
    - SRQ, XRC, dynamic connections
- Optimize addressing
  - Resolve multiple resolution requests at once
  - Compact address data storage
    - Compressed address ranges, path data
- Support multiple resolution mechanisms
  - Optimized for different topologies and fabric sizes
SFI - Address Vectors

Store addresses/host names
- Insert range of addresses with single call

Share between processes

Reference entries by handle or index
- Handle may be encoded fabric address
Reference vector for group communication

Enable provider optimization techniques
- Greatly reduce storage requirements

Example only

<table>
<thead>
<tr>
<th>Start Range</th>
<th>End Range</th>
<th>Base LID</th>
<th>SL</th>
</tr>
</thead>
<tbody>
<tr>
<td>host10</td>
<td>host1000</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>host1001</td>
<td>host4999</td>
<td>2000</td>
<td>2</td>
</tr>
</tbody>
</table>
Can API changes unlock higher performance?

Just a guess, but is the answer “yes”?
Application Send

Significant SW overhead

```c
struct ibv_sge {
    uint64_t addr;
    uint32_t length;
    uint32_t lkey;
};
```

```c
struct ibv_send_wr {
    uint64_t wr_id;
    struct ibv_send_wr *next;
    struct ibv_sge *sg_list;
    int num_sge;
    enum ibv_wr_opcode opcode;
    int send_flags;
    uint32_t imm_data;
    ...
};
```

Application request

- <buffer, length, context>
- 3 x 8 = 24 bytes of data needed
- SGE + WR = 88 bytes allocated

- Requests may be linked - next must be set to NULL
- Must link to separate SGL and initialize count
- App must set and provider must switch on opcode
- Must clear flags
- 28 additional bytes initialized

Significant SW overhead

March 30 – April 2, 2014 #OFADevWorkshop
Provider Send

For each work request
  Check for available queue space
  Check SGL size
  Check valid opcode
  Check flags x 2
  Check specific opcode
  Switch on QP type
    Switch on opcode
  Check flags
    For each SGE
      Check size
      Loop over length
  Check flags
  Check
  Check for last request
Other checks x 3

Most often 1 (overlap operations)
Often 1 or 2 (fixed in source)
Artifact of API
QP type usually fixed in source
Flags may be fixed or app may have taken branches

19+ branches including loops
100+ lines of C code
50-60 lines of code to HW
Scalable Transfer Interfaces

- *Application* optimized code paths based on usage model
- Optimize call(s) for single work request
  - Single data buffer or 2-entry SGL
  - Still support more complex WR lists/SGL
- Per endpoint send/receive operations
  - Separate RMA function calls
- Pre-configure data transfer flags
  - Known before post request
  - Select software path through provider
SFI – Send Message

Allocate WR
Allocate SGE
Format SGE – 3 writes
Format WR – 6 writes

50-60 lines of C-code

Reduce setup cost
- Tighter data

Direct call – 3 writes

25-30 lines of C-code

optimized send call

Checks – 2 branches

Loop 1

Checks – 9 branches

Loop 2

Check

Loop 3

Checks – 3 branches

Checks – 3 branches

Loop 3

Checks – 3 branches

Eliminate loops and branches
- Remaining branches predictable

Selective optimization paths to HW
- Manual function expansion
struct ibv_wc {
  uint64_t   wr_id;  // Application accessed field
  enum ibv_wc_status status;
  enum ibv_wc_opcode opcode;
  uint32_t   vendor_err;
  uint32_t   byte_len;  // Provider must fill out all fields, even those ignored by the app
  uint32_t   imm_data;
  uint32_t   qp_num;
  uint32_t   src_qp;
  int        wc_flags;
  uint16_t   pkey_index;
  uint16_t   slid;
  uint8_t    sl;
  uint8_t    dlid_path_bits;
};
Scalable Completion Interfaces

- **Application** optimized code paths based on usage model
- Use compact data structures
  - Only needed data exchanged across interface
  - Limited to fields required by application
  - Separate addressing from completion data
  - Report errors ‘out of band’
- Per CQ operations
  - Support multiple wait objects
  - Allow provider to optimize event signaling
SFI – Events

Generic completion

App selects completion structure

Op context

optimized CQ

+1 write, +0 branches

Support provider updating counters

read CQ

Send: +4-6 writes, +2 branches

Recv: +10-13 writes, +4 branches
Is there anything else behind this proposal?

I have two more puzzle pieces.

App-centric
Application Interface Mismatch

Instructions retired in MPI_Isend

Look up connection, check memory registration, formatting requests, etc.

MVAPICH2-2.0rc1 (latest) code is used with default configuration options (CH3:mrail)
All userspace instructions are counted for full execution of MPI_Isend
Memory copies and locks are also included in the component that uses them

MVAPICH2 lib compile flags: `-O3 -DNDEBUG -ipo`
App compile flags: `-O3 -DNDEBUG -ipo -finline-limit=2097152 -no-inline-factor -inline-max-per-routine=10000000
-inplace-max-per-compile=10000000 -Bstatic -Impich -Bdynamic -lropa -lml -libverbs -libumad -libmad -lrndmacm -lrt -lpthread`
Application-Centric Interfaces

Reducing instruction count requires a better application impedance match

- Collect application requirements
- Identify common, fast path usage models
  - Too many use cases to optimize them all
- Build primitives around fabric services
  - Not device specific interface
Application-Centric Interfaces

- **Myth**: app-centric interfaces imply more overhead
  - Poor implementations result in poor performance
  - Difficult to use APIs are likely to result in poor implementations
  - Provider knows best method for accessing their HW
  - These are still low-level interfaces (C), just not device interfaces (assembly)
Application Configured Interfaces

App specifies comm model
Communication type
Capabilities
Data transfer flags
Endpoint
Provider directs app to best API sets

sm. msg
inline send
send
lg. msg
write
read
RMA
RMA Ops
NIC
Message Queue Ops

March 30 – April 2, 2014 #OFADevWorkshop
What’s the purple piece representing again?

Extensible
Extensible Framework

Focus on longer-lived interfaces – software leading hardware

- Take growth into consideration
- Reduce effort to incorporate new application features
  - Addition of new interfaces, structures, or fields
  - Modification of existing functions
- Allow time to design new interfaces correctly
  - Support prototyping interfaces prior to integration
Future Extensions

• Design framework and APIs with anticipated capabilities
  – Stage delivering features
• Documentation defines supported usage models
• Use static inline calls to simplify application interactions with objects
  – Convert object-oriented model to procedural model
Claim

These concepts are necessary, not revolutionary
  – Communication addressing, optimized data transfers, app-centric interfaces, future looking

Want a solution where the pieces fit tightly together
Thank you!