Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 105b

Systems Group, Department of Computer Science, ETH Zurich

OS Support for Memory-to-Memory Transfers Using a DMA Engine

by

Sebastian Wicki

Supervised by

Kornilios Kourtis

July 2013 – January 2014

inf | Informatik
Computer Science

Abstract

Direct memory access (DMA) is commonly used to perform data movement between peripheral devices and main memory independently of the processor. This thesis describes how to use a system-wide DMA engine, such as the one found in the OMAP system-on-chip platform, to perform memory-to-memory transfers. A driver for this DMA engine is implemented for the Barrelfish operating system, a research operating system developed at ETH Zurich. The driver relies on Barrelfish's asynchronous message passing system for inter-process communication, as well as the capability system Barrelfish uses for memory management. The capability system allows user space applications to refer to physical memory regions in a secure and sound way, while the asynchronous remote procedure call interface of the driver allows applications to perform computations in parallel with the memory transfer which was offloaded to the device. As part of the evaluation, the raw performance of the hardware device is measured, as well as the overhead introduced by the software driver managing the device. The DMA engine is used in a bulk data transfer scenario, showing that it can outperform software-based memory-to-memory transfers for large (>512KB) payload sizes.

Contents

1 Introduction

Direct memory access (DMA) engines are traditionally being used for data transfers between devices and the system memory independently of the CPU. In most systems, each device which wants to perform transfers between the device and the system memory has its own DMA controller to enable this. In other systems, such as system-on-chip architectures, there is a central DMA engine which can be requested to perform transfers between memory and the device. Although these system-wide DMA engines are designed to offer DMA transfers between devices and memory, system DMA engines can sometimes also be used for memory-to-memory transfers.

The topic of this thesis is study support for DMA-assisted memory-to-memory transfers by implementing a driver for such a system DMA engine in the Barrelfish research operating system.

This document is divided into four parts: Section 2 gives some background about the Barrelfish operating system and its frameworks used in this thesis, as well as an overview over the OMAP4460 system-on-chip, the hardware platform used for this thesis. The design and implementation of the driver for the DMA engine is presented in Section 3, including a description of the interface implemented to support memory-to-memory transfers using the DMA engine. In Section 4 the performance of hardware device itself, as well as the implemented driver is evaluated. This section also includes the evaluation of the usefulness of the implementation in a bulk data transfer scenario. Finally, a survey over related work, the discussion of possible future work and the conclusion is found in Section 5.

1.1 Motivation

Using a DMA engine to perform memory-to-memory transfers allows offload work from the CPU to the DMA engine, therefore enabling parallelism and possibly power consumption advantages. However, these benefits are not free. As the DMA engine is an device external to the processor, there is not only a configuration overhead when setting up transfers, but also a communication overhead, as the DMA engine bypasses CPU caches and uses physical addresses instead of virtual addresses.

This suggests that offloading memory-to-memory transfers to a DMA engine might only pay off for larger transfers sizes, where the initialization overhead can be neglected, and performing the transfer on the CPU would not only hinder the processor in performing other computations, but also pollute the CPU cache with the transferred data.

Therefore, this thesis studies the capabilities of such a DMA engine in order to implement and evaluate a driver which can be used to offload large memory-to memory transfers to the device, while allowing applications to perform computations in parallel to the transfer.

2 Background

This section gives provides some background about the Barrelfish operating system and the frameworks it offers which were used in this thesis. In the second part, the hardware used in this thesis is introduced and the functionality offered by the OMAP SDMA engine is presented.

2.1 Barrelfish

Barrelfish is a research operating system from ETH Zurich, which was developed in collaboration with Microsoft Research. The goal of Barrelfish is to explore support for possibly heterogeneous multi-core and many-core systems, by viewing the hardware and the OS as a distributed system and treating it as such.

To achieve this, Barrelfish implements a so called multikernel, where each processor core independently runs its small kernel (called the CPU driver), with drivers and system services implemented separately in user space.

Communication between processes is realized using asynchronous message passing, which is implemented separately for communication on the same core and for communication between applications on different cores. [TN000]

2.1.1 Capabilities and Memory Management

In order to support distributed memory management, Barrelfish implements a capability system. Capabilities are unforgeable references to system resources, which are handed to applications which have the right to use them. Capabilities are also communicable, which means they can be passed to other applications if needed. In Barrelfish, capabilities can be used to represent regions in the physical memory address space, may they be actual memory or memory-mapped device registers.

Barrelfish uses a capability system where modification of capabilities are implemented as system calls to the CPU driver. Each capability in Barrelfish is typed and some capabilities can be retyped, as shown in Figure 1. For example a capability of the type `PhysAddr`, which is used to represent a physical address range, can be retyped into `RAM` to represent actual memory. A `RAM` capability can either be retyped into various kernel objects stored in this RAM region, or into a `Frame` capability, representing a a physical memory frame for use in user-level applications.

Capabilities used to represent regions of memory always have the size of a power of two in Barrelfish. Memory capabilities can be split into multiple capabilities representing smaller regions of memory.

Applications implement their own virtual memory management, by having the necessary capabilities for the page tables and the memory regions. An application can map its virtual addresses to physical memory by invoking a mapping operation on the according page table capability.

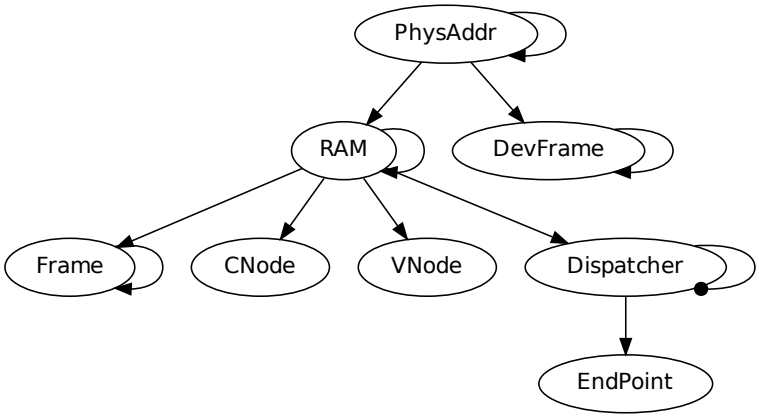

Figure 1: Barrelfish's capability inheritance tree. Taken from [TN013]

2.1.2 Dispatchers

Barrelfish has the concept of dispatches, which implement a form of scheduler activation, allowing the kernel to forward event processing to upcall handlers in user level applications. This technique is used to handle page faults in user space, but also to forward hardware interrupts from the CPU driver to user level drivers. Dispatchers are scheduled by the CPU driver and can be combined into a domain to group related dispatchers running on different cores.

2.1.3 Inter-dispatcher Communication

Barrelfish offers a channel based inter-dispatcher communication (IDC) system. There are multiple message transport implementations used for IDC, called interconnect drivers. Each interconnect driver is specialized for different requirements, such as intra-core message passing or message passing between different cores.

While there are multiple interconnect drivers which have to be used differently, there is an unified interface description language (IDL) called Flounder. Interfaces described in Flounder can be compiled to stubs for all the available interconnect drivers. The generated stubs might perform message fragmentation or other tasks needed to implement the Flounder interface in the selected interconnect driver.

Flounder has the notion of asynchronous remote procedure calls (RPC), which also allow capability passing to other dispatchers. [TN011]

2.1.4 Asynchronous I/O

Because the message passing system in Barrelfish is asynchronous, multiple callbacks are necessary for sending and receiving messages. This leads to so called "stack-ripped" code, meaning the control flow of the code is arguably harder to follow.

Therefore, Barrelfish provides an implementation of THC, a set of extensions to the C language designed to retain a sequential style of programming, while providing support for concurrent I/O operations without the use of multiple threads. [Har+11]

THC is integrated into the Barrelfish message passing interface, allowing applications to utilize parallelism when waiting for responses from system services.

2.1.5 Mackerel

Barrelfish also provides Mackerel, a domain specific language (DSL) designed to assist accessing devices registers in device drivers. Accessing individual registers usually involves many bit-level operations, which can be tedious and error-prone.

The Mackerel language can be used by describing the hardware registers from the data sheet in this DSL, with additional human readable descriptions for the registers and their values.

The Mackerel description files are compiled into C header files, containing inline functions for accessing the actual registers without the need for bit-level operations. Mackerel generated code also provides additional debugging functions for printing out the state of the registers in a human readable fashion.

2.2 OMAP4460 SoC and Pandaboard

The OMAP4460 is a system-on-chip (SoC) platform by Texas Instruments. It was designed for portable multimedia devices such as smartphones and tablets. It contains two ARM Cortex A9 cores as the application processor and two ARM Cortex M3 cores initially designed for controlling the imaging subsystem. There are various integrated modules, such as a graphics accelerator subsystem, digital signal and imaging processing modules, and systems for processor synchronization such as hardware spinlocks or a mailbox module.

The Pandaboard is a single-board computer, based on the OMAP4430/4460 platform. Additional to the modules integrated in the OMAP SoC, the Pandaboard also provides USB 2.0, Ethernet and WLAN/Bluetooth connectivity.

The Barrelfish OS was ported to run on the Pandaboard, with support for the heterogeneous Cortex A9 and Cortex M3 cores.

The OMAP4460 SoC internally has two main interconnects, called L3 and L4. Various subsystems, including the A9 and M3 cores, as well as the main memory are connected through the L3 interconnect. The L4 interconnect is mainly used for configuration and peripheral modules.

2.3 OMAP System DMA Module

The OMAP4460 SoC also contains a system direct memory access (SDMA) module, which offers high-performance direct access to the memory to other device modules in the system. The SDMA can be configured to transfer data between devices and memory, but also for data movement in the memory only.

The parameters, such as source and destination of a DMA transfer have to be pre-configured by software. The transfer itself can be triggered by hardware devices using a device-specific DMA request pin. Transfers can also be triggered by software by setting an enable bit in the SDMA configuration registers. Transfers triggered by hardware devices are also called synchronized transfers.

When a transfer is finished, or certain other conditions are met, an optional interrupt will be sent to the Cortex A9, Cortex M3 and the DSP cores.

2.3.1 Operation

The SDMA module has one read and one write port on the L3 interconnect. These ports are used for transferring data between memory and devices, or for memory-to-memory transfers. Configuration of the SDMA module is done through the configuration registers exposed on the configuration port on the L4 interconnect.

To support concurrent transfers, the SDMA module has four threads on the read port and two threads on the write port. Because the SDMA engine offers up to 32 logical channels which can be configured to run transfers concurrently, there is a logical channel scheduler which temporary assigns each of the 32 channels to one of port threads. If there are more active channels than threads on a port, the channels are queued according to their priority.

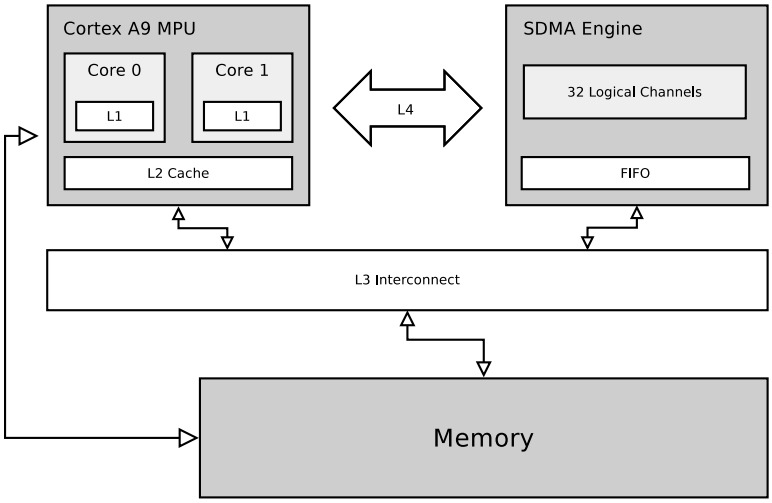

Figure 2: Schematic overview of the SDMA module

Buffering between the read and the write port is provided by a pool of FIFO queues, where each FIFO queue is associated with an active logical channel.

2.3.2 Programming Model

The internals of the SDMA module as described above are abstracted by 32 logical channels for the configuration. A channel describes all parameters of an individual transfer, most importantly the source and destination address and transfer size. A channel can either be configured to be hardware synchronized, meaning only a specific hardware device can trigger the transfer using its request pins, or the channel can be software synchronized, meaning the transfer can only be initiated by software.

Each one of the 32 logical channels is configured by writing to the device registers of the particular channel. Besides the channel specific configuration, there is also a set of global configuration registers which specify the channel-independent behavior of the device.

2.3.2.1 Device Initialization In contrast to other modules, the SDMA module does not need any special clock or power setup. The SDMA even can be used to restore the state of other modules without the need of the processor unit waking up [OMAP4460, Sec. 16.4.21].

The global configuration register allows adjusting the global budget and arbitration rate for high priority transfers, as well as the amount of bytes a channel can allocate in the FIFO pool. The default value of 16 bytes is not sufficient to support larger burst transfers, i.e. 64 bytes, which degrades the performance of large transfers significantly.

2.3.2.2 Channel Configuration Most configuration registers of the SDMA module are channel specific. There are two registers for each channel to set the physical start and destination address the transfer. There are some additional parameters which can be set for source and destination individually:

Burst Transfer Enable Instead of transferring each element individually, the SDMA module supports burst transfers, which will increase performance by transferring 16, 32 or 64 bytes at once.

Packed Port Access As the element size of 8, 16 or 32 bits might not match the native width of the source or destination port, the DMA engine can be configured to transfer multiple elements to optimize port access.

Endianism Conversion For both, the source and destination, it can be specified if the device uses big or little endian byte order. The SDMA engine also offers the functionality to convert the byte order during the transfer. However, the technical reference manual (TRM) states that the device is little-endian by construction, and therefore big endian mode should never be used [OMAP4460, Sec. 16.4.8].

There are also configuration values which influence the behavior of the whole channel transfer, independently of source and destination settings:

Transparent Copy and Constant Fill The SDMA module supports two graphic acceleration features. One is constant fill, which instead of copying elements from the source to the destination, fills the destination region with a constant value without accessing the source at all. The other feature is called transparent copy and works as follows: Each source element is matched against a constant value. If the source element is equal to the constant value, the element is not copied and the destination location is left untouched. If the source element is different from the constant value, the element is written to destination as in a normal transfer. Both of these features are mutually exclusive, only one mode can be used in a transfer. The constant value used in both modes is called the *color* value and is 24 bits wide, even for element sizes of 32 bits. The upper 8 bits are either ignored in transparent copy, or set to zero in constant fill mode.

Channel Linking Because only one pre-configured channel at a time can be triggered by a particular hardware device, channels can be chained together. If channel chaining is used and the first channel has finished its transfer, it will enable the next pre-configured channel to perform its transfer. Channels can also be chained into a continuous loop. Note that this is different from scatter-gather transfers as all channels have to be pre-configured using the configuration registers and are not available for other transfers.

Posted and Non-Posted Writes The SDMA module supports both, posted and non-posted transactions on the interconnect. When using posted writes, the device

does not wait for previously emitted writes to complete. Using non-posted writes, the SDMA module waits for each write to finish before the next write is issued. The module also supports a write mode where only the last write is non-posted, meaning the final write has to be completed before the whole transfer is considered complete.

Settings for Hardware Synchronized Transfers Each channel can be synchronized with a peripheral hardware device, meaning the channel has not to be enabled by software, but can be automatically triggered if a hardware device requests this. To achieve this, a hardware specific identifier can be assigned to a channel, meaning that this channel configuration is to be used when the hardware triggers a transfer. Hardware synchronized transfers also have some additional settings which were not considered in this thesis, as they are not applicable to memory-to-memory transfers. These settings include buffering, prefetching, or more flexible transfer sizes.

Addressing Modes The amount of data which should be copied during a transfer has to be specified using the notation of elements and frames. An element is the smallest unit of transfer, it can have the size of 8, 16 or 32 bits. A transfer frame consists of a fixed number of elements, and during a transfer multiple frames are transferred. Therefore, to specify the transfer size, one has to provide three values: The element size, the number of elements per frame and the number of frames.

The SDMA module supports four different addressing modes, these allow access in strides or even the rotation of images to 90, 180 or 270 degrees. The addressing mode can be specified for the source and destination separately. All addressing modes describe how the address is modified after each transfer to calculate the address of the next element.

Constant Addressing In this mode, the address of the transferred element is never changed. This mode is not supported for memory-to-memory transfers.

Post-Increment Addressing This is the default addressing mode. After an element was accessed, the address is incremented by the size of one element. This mode is used to transfer a contiguous block of memory.

Single-Index Addressing Instead of incrementing the address by the size of one element, the address of the next element is calculated by adding a specified *element index*, which can be negative. This allows stride access, or access in the reversed direction. Note that the element index is byte-based.

Double-Index Addressing Similar to the single-index addressing mode, in this mode the *element index* is added to the address after the transfer of an element. If the element however was the last element of a transfer frame, the *frame index* is used instead of the *element index*. This mode allows the rotation of images.

Note that because of hardware internals, when adding the *element index* and *frame index*, the address does not point to the first byte of the previously transferred element, but to the last byte. More formally, the following equations can be used to calculate the address of the n-th element, where as $A(1)$ is defined as the start address, ES the element size, and EI and FI the element and frame index.

Constant Addressing:

$$A(n) := A(n-1)$$

Post-Increment Addressing:

$$A(n) := A(n-1) + ES$$

Single-Index Addressing:

$$A(n) := A(n-1) + (ES-1) + EI$$

Double-Index Addressing:

$$A(n) := \begin{cases} A(n-1) + (ES-1) + FI & \text{if } n-1 \text{ was last element of frame} \\ A(n-1) + (ES-1) + EI & \text{otherwise} \end{cases}$$

Interrupt Generation The SDMA module has four interrupt lines which can be used to send interrupts to the processor units. For each of the four lines, it can be configured which channel can trigger an interrupt on that line. When an interrupt is triggered, the source channel of the interrupt is stored in the *interrupt line status register*, while the cause of the interrupt is stored in the *channel status register*.

If multiple channels at the same time cause an event which triggers an interrupt, only one interrupt is sent to the processor unit, so the interrupt handler must check for multiple interrupt sources. It is the software's responsibility to the read and clear the status flag for each channel individually. Regardless if interrupts are enabled for a certain interrupt line, the channel status flags are only set if they were enabled in the interrupt control register.

The following channel status flags are available, and can therefore be configured to be the source of an interrupt:

Half of Frame: The first half of the current frame has been transferred.

End of Frame: Indicates that a frame has been transferred.

Start of Last Frame: The transfer of the last frame has begun.

End of Block: Indicates the completion of a normal channel transfer.

End of Super Block: Indicates the completion of a scatter-gather transfer.

End of Packet: In hardware synchronized transfers, indicates the completion of a packet.

Misaligned Address Error: The element address was not aligned to the element size.

Transaction Error: A transaction error occurred on the interconnect.

Supervisor Error: Unprivileged access on a supervisor-reserved channel.

Drop Error: Indicates that a new DMA request was asserted before the pending request could be served.

Drained: Indicates that the write port became inactive.

3 Design and Implementation

This section outlines the basic design of the driver implemented for the OMAP SDMA engine, and presents the interface of this driver which can be used by other applications to offload memory-to-memory transfers to the DMA engine.

3.1 Basic Design

To enable applications to perform memory-to-memory transfers using the SDMA engine, a driver exposing a sufficient interface is necessary. The multikernel approach of Barrelfish suggests putting the driver in a separate user level application.

Barrelfish on ARM has support for user space drivers using the `driverkit` and `libbarrelfish` libraries. `driverkit` is simple library allowing access to memory mapped hardware devices from user level applications. Access control to memory-mapped hardware registers is implemented in Barrelfish's capability system by having `DevFrame` capabilities for the according address region. On ARM, the capability for the device frame is currently statically assigned to the user level application at boot time by Kaluga, Barrelish's device manager. The driver is spawned by Kaluga, with the necessary capability in its capability space. User level interrupts are realized using Barrelfish's dispatcher concept, which allows applications to register interrupt handlers by providing an IDC endpoint for receiving the interrupts. [TN019]

The functionality of the SDMA module is exposed to other applications through a Flounder interface. Sending asynchronous remote procedure calls using the THC language extension, applications can temporally overlap own computation with requested memory transfers.

3.2 Hardware Initialization and Configuration

The SDMA module exposes over 50 configuration parameters and status flags per channel, some of them interdependent, leading to over 700 individual registers which need to be managed by the device driver.

For the SDMA module, as for most of modules of the OMAP4460 SoC, there existed already existing auto-generated Mackerel files for accessing the device registers. The auto-generated code however did not make use of register arrays, which are heavily used for the channel-specific registers. The auto-generated code did also not include the descriptors used for scatter-gather transfers. Both of these were added manually to to the Mackerel file for the SDMA module.

While Mackerel abstracts the bit-level and addressing operations needed to access these registers, Mackerel cannot enforce constraints over the values of multiple registers, this is the responsibility of the device driver.

In order to be able to evaluate the usefulness of the implementation, it is necessary to measure the performance of the hardware without much software overhead. As the hardware performance evaluation would be executed in a kernel context, but the driver itself was to be run as an user level application, a unified way of accessing and validating the device registers was needed.

3.2.1 SDMA Module Mackerel Wrapper

To achieve this, an additional wrapper around the Mackerel generated macros was designed. Using this wrapper, the user programming the SDMA modules does not need to set the channel-specific registers individually. Instead, the channel configuration is represented by a C structure containing all parameters which can be set by the user. The user can then apply this configuration to a specific channel, which also includes some basic input validation for invalid configurations.

The channel has to be enabled in a separate function call, which sets the enable bit for the channel. This allows measuring the raw performance of the hardware, without the configuration overhead. The wrapper also registers a low-level interrupt handler which calls the necessary Mackerel macros to determine source and cause of the interrupt. The low-level interrupt handler then calls a user-specified callback, passing the error or success indication and the source channel as the function argument.

The following parameters can be set in the structure processed by the Mackerel wrapper. As the scope of this thesis is only to support memory-to-memory transfers, some parameters only needed for transfers from or to peripherals are omitted, but the wrapper could easily be extended to support these. Refer to Section 2.3.2.2 for more information about the semantics of these configuration parameters.

- Write and read priority of the channel.
- One of the following operation modes: Normal copy, transparent copy or constant fill.

- Write mode: Posted or non-posted writes.
- Transfer size, defined in terms of element size, element number and frame number.
- Flag to enable channel linking and if set, the channel number of the next channel in the chain.
- For both, source and destination separately:
  - Physical start address.
  - Addressing mode: Post-increment, single-index or double-index.
  - Element index and frame index for indexed addressing modes.
  - Flag to enable packed port access.
  - Burst mode size: No burst mode, 16 byte, 32 byte or 64 byte burst access.

The Mackerel wrapper also performs basic initialization of the hardware. As the device already uses sane default values for most global configuration registers, only the FIFO depth is changed to enable large burst transfer sizes.

3.3 Driver Interface

While the wrapper implementing the hardware configuration abstracts away the complexity of having individual device registers, the interface exposes still a lot of hardware specific limitations and requires domain-specific knowledge about the device's capabilities. Therefore, a higher-level interface is needed for applications which want to use the device.

3.3.1 Interface for Simple Transfers

The hardware offers fine-grade settings to improve performance, i.e. variation of element sizes, packed element access, posted transfers, various burst mode sizes. However, for memory-to-memory transfers, the performance evaluation of the hardware showed little impact of these parameters when enabling burst transfers. Therefore, there is not much gain in exposing all this fine-tuning parameters to the end-user. Instead the driver interface can be simplified, allowing the driver to use its own optimal settings for maximum performance, while the user only has to provide the most essential parameters for the transfers.

The arguably most simple interface for memory-to-memory transfers for programmers is the `memcpy` function of the C language: The user provides the virtual address of the source and destination memory, as well as the number of bytes to copy.

As the SDMA module deals with physical addresses and the driver runs in its own virtual address space anyway, the application requesting a memory-to-memory transfer needs to provide physical addresses to the driver.

The capability system of Barrelfish simplifies this significantly, as client applications already have a representation for the physical memory frames they have access to. `Frame` capabilities in Barrelfish also include the size of the region represented by the capability. Therefore, sending a source and destination frame capability to the SDMA driver not only provides all necessary information to perform the memory-to-memory transfer, it also proves that the application has access right to the physical location represented by the capability.

As the OMAP SDMA module also provides the functionality of filling a specified region with a constant value, the driver interface also offers an equivalent to the `memset` function of C. This leads to the following two functions which offer client applications simple memory-to-memory transfers using the SDMA module:

```
// Copies the content of the 'source' frame capability
// into the 'dest' frame. The outcome of the operation
// is indicated in the 'err' return value
rpc mem_copy(in cap dest, in cap source, out errval err);

// Fills the content of the 'dest' frame capability
// with the byte value specified by 'color'.
// The outcome is indicated in the 'err' return value.
rpc mem_fill(in cap dest, in uint8 color, out errval err);
```

Listing 1: Interface for simple memory-to-memory transfers

3.3.2 Interface for Two-Dimensional Transfers

This simple interface however is not sufficient to support more complex requests, for example transfers which only want to parts of a frame, or requests which would like to make use of the SDMA special features such as transparent copies or two-dimensional addressing modes for image rotation.

Therefore, a more sophisticated interface is needed to support more complex requests. As the hardware already requires the transfer size to be two-dimensional, the representation of the memory region as a two-dimensional image is adopted in the driver interface.

To define the size of the memory region as a two-dimensional graphical image, the user has to provide three values: The size of an individual pixel, the width of the image and the height of the image in number of pixels. The pixel size, or bit depth, can be set to 8, 16 or 32 bits. The height and width of the image is specified in number of pixels per axis, in `x_count` and `y_count` respectively. All these values are grouped together in the `count_2d` struct.

To support image rotations or other forms of stride access, the `addr_2d` structure has to be provided for the source and the destination separately. As with the simple memory transfers, a capability representing the memory frame has to be specified. In contrast to the simple interface, not the whole memory frame will be necessarily accessed in the transfer, but of course all accessed memory locations

have to lie in between the frame boundaries.

For each dimension, the user has to provide a start offset and a modifier. The two-dimensional start offset is the first pixel accessed by the transfer, it is represented by `x_start` and `y_start` field.

After the first pixel has been transferred, the value of `x_modify` is added (or subtracted) to the memory address, providing the address of the next pixel. For each subsequent pixel transferred, the value of `x_modify` is used to calculate the address of the next pixel. This step is repeated for the number of pixels on the x axis. After `x_count` pixels have been transferred this way, the value of `y_modify` is applied to calculate the address of the next pixel instead of `x_modify`. This allows the transfer to continue on a different row. For the next subsequent `x_count` accesses, the value of `x_modify` is used again.

```
// Bits per pixel
typedef enum {
    DATA_TYPE_8BIT,
    DATA_TYPE_16BIT,
    DATA_TYPE_32BIT
} data_type;

// Size of the two-dimensional image
typedef struct {
    data_type pixel_size;

    uint32 y_count;
    uint32 x_count;
} count_2d;

// Values needed for address calculation
typedef struct {
    cap cap;

    uint32 x_start;
    uint32 y_start;

    int32 x_modify;
    int32 y_modify;
} addr_2d;
```

Listing 2: Types needed to declare the parameters of a two-dimensional transfer

This model and terminology of representing two-dimensional transfers is based on the one presented by Katz and Gentile [KG05], p. 80-87. Note that this is slightly different addressing scheme than the one offered by the OMAP SDMA engine: The SDMA addressing scheme is byte-based, and the addresses point to last byte of the previously transferred element.

A more formal description of this addressing scheme is provided by the following pseudo-code, note however that the actual computations are carried out by the hardware:

```
pixel_t base_addr; // base address of the accessed frame
pixel_t addr = base_addr + x_start + y_ count * y_start;

for (y = 1; y <= y_count; y++) {
    for (x = 1; x <= x_count; x++) {
        access( *addr );

        if (x < x_count) {
            // within the frame
            addr += x_modify;
        } else {
            // at the end of a frame
            addr += y_modify;
        }
    }
}
```

Listing 3: Loop representation of the two-dimensional addressing scheme

Using this addressing scheme, memory-to-memory transfers can be requested through the `mem_copy_2d` and `mem_fill_2d` functions. The `mem_copy_2d` also includes a `color` parameter, which will be used for the transparent copy it is enabled using the `transparent` boolean parameter.

As with the simple interface, the `mem_fill_2d` function is used to fill parts of a memory region with a constant value.

For both, the transparent copy and the constant fill, the `color` parameter is a 24 bit integer because of hardware limitations. If a pixel size of 32 bits is used, the upper 8 bits are always zero for the constant fill mode, and are ignored in transparent copy mode.

```
    rpc mem_copy_ 2d(in addr_2d dest, in addr_2d source,
                     in count_2d count, in bool transparent,
                     in uint24 color, out errval err);

    rpc mem_fill_ 2d(in addr_2d dest, in uint24 color,
                     in count_2d count, out errval err);
```

Listing 4: Interface for two-dimensional memory-to-memory transfers

3.4 Interface Implementation

The implementation of the above interface is mostly straight-forward when using the Mackerel wrapper. First, the driver has to get the physical base address and the size of the source and destination capability. This is done through the frame identity system call, which returns the physical base address of the frame, as well as the size of the frame as the binary exponent.

Because the SDMA module expects the transfer size to be specified in the two-dimensional format with number of elements and number of frames, the driver splits the the memory region size for the simple transfers into multiple transfer frames.

For the two-dimensional interface, the values of `x_modify` and `y_modify` have to be converted for the SDMA module into the transfer element index and transfer frame index:

$$ElementIndex := (\texttt{x\_modify} - 1) * PixelSize + 1$$
$$FrameIndex := (\texttt{y\_modify} - 1) * PixelSize + 1$$

For the two-dimensional addressing scheme, the driver also has to ensure that every access that will be performed during the transfers lies in the boundary of the given frame capability.

The driver has to assign a SDMA channel to each request. In the current implementation an error is returned if all 32 channels are currently occupied. An alternative would be to implement a waiting list for pending requests.

In order to be able to serve multiple concurrent clients, the implementation makes use the THC language extension. As soon as a channel has initiated the transfer, the interface service can handle the next client, as the hardware performs the transfer in parallel. When the transfer on a channel is completed, an interrupt is raised and the interrupt handler uses a channel assigned THC semaphore to signal completion of the request to the waiting task.

The current implementation is completely single-threaded, all clients and interrupts are handled in a single event loop.

3.4.1 Not Implemented Hardware Features

It should be noted that although the two-dimensional interface has quite some complexity, there are still various features of the SDMA engine which are not available through this interface. Most notably, high-priority requests, channel chaining and scatter-gather transfers.

The technical reference manual of the SDMA module [OMAP4460, Sec. 16.4.10] implies that high-priority transfers are intended for latency-critical hardware synchronized transfers, and therefore are not appropriate for memory-to- memory transfers. Assuming potentially malicious client applications, supporting high-priority requests would also raise the need for some kind of access control, to ensure that only trusted processes are allowed to issue latency-critical high-priority transfers.

The channel chaining feature is also mostly intended for hardware synchronized transfers, as only one channel can be synchronized with a particular hardware device. As the current driver interface does not expose the concept of DMA channels to the client applications, channel chaining support would demand a way to refer to individual requests, in order to be able to chain them together.

While the previous two features were intentionally omitted for software-synchronized transfers, scatter-gather transfers on the other hand would be quite useful for memory-to- memory transfers, as a region of virtual memory does not necessary have to be contiguous in physical memory. Therefore, an opportunity for future work could be support for scatter-gather transfers. However, Flounder currently does not support variable length arrays. As scatter-gather transfers impose the need of specifying multiples memory regions at once, this might complicate the driver interface a bit.

One possibility would be to have multiple RPC calls to request a scatter-gather transfer, specifying one memory region each call. Another possibility would be to use fixed-size arrays for the the list of memory regions to transfer, this however would impose a limit of how many regions a scatter-gather transfer could access. A third possibility, at least for simple transfers, would be to send the capability for a `CNode` containing all the frames which should be transferred.

4 Evaluation

In this section, three different performance evaluations are presented: First, the raw performance of the device in direct use. Second the performance of the SDMA module when used by one or multiple applications through the device driver. And third the performance is studies when device is used in a realistic scenario, such as bulk transfers.

All timing measurements were taken using the ARM Cortex A9 Global Timer [A9TRM, p. 61-71], which is a 64-bit incrementing counter, therefore allowing longer measurements without overflowing. Rudimentary support for the A9 Global Timer was added to the Barrelfish CPU driver specifically for this thesis. Access to the timer is currently exposed to user level applications through a system call, as the timer is memory mapped in the ARM the private memory region. As the data sheets did not provide any insight on the actual frequency of the timer, the Cortex A9 Global Timer was measured. During these benchmarks, the frequency of the timer was 166.4 MHz.

4.1 Performance of the Device

To be able to evaluate the overhead of managing the device in an user space driver, it is necessary to measure the performance when using the device as directly as possible. Having the raw performance measurements of the device also helped simplifying the driver interface, as not all parameters are necessary for memory-to-memory transfers to perform well.

4.1.1 Setting

To be as direct as possible, with as little overhead as necessary, the performance evaluation of the hardware was executed in the kernel.

Two of the design decisions of the Barrelfish CPU driver are that the kernel does not allocate any memory, and that the kernel is executed in a single, non-preemptable thread. For the performance evaluation however, the benchmark running in the kernel would have to allocate some memory to run the transfers and it would need to be able to handle hardware interrupts.

Therefore, the kernel benchmark is executed after early kernel initialization, but does never return control back to the kernel, because it violates the assumptions of the Barrelfish CPU driver by enabling interrupts and allocating memory for itself. This method however has the advantage of still being able to use the basic C standard library (i.e. serial output) and the paging functions implemented by the Barrelfish kernel.

The memory allocation is implemented in the same fashion as the kernel does (as an exception) allocate some memory for the first user level process running on the system. On the Pandaboard, all virtual addresses in the kernel are mapped one-to-one to the same physical addresses. Therefore it is sufficient to just mark parts

of the memory as reserved, and pass the virtual address, which is equivalent to the physical address, to the SDMA module for transfers.

As the interrupt dispatcher in Barrelfish is not designed to return to a kernel thread, the SDMA benchmark has to implement its own interrupt handler. This is done using GCC specific function attributes [GCC, Sec. 6.30] which generate the necessary ARMv7 machine code for a C function to be used as an interrupt handler. This way, the SDMA benchmark code only has to overwrite the address of the Barrelfish interrupt handler with the address of the custom interrupt handler in the interrupt vector table.

For programming the SDMA module, the kernel benchmark uses the same Mackerel wrapper as the user level device driver, since this wrapper was written to be used in both, kernel and user space.

4.1.2 Results

Transfer of different sizes for different channel configurations were executed and measured multiple times, in order to get an understanding of the performance characteristics of the device.

A first observation was the impact of the accessed element size on performance. The SDMA module allows the programmer to choose different element sizes as the smallest unit of transfer. Figure 3 shows that the element size has a significant impact on throughput when performing normal memory-to-memory transfers. However, when packed access is enabled, the SDMA modules accesses the memory in its native port width, transferring the optimal amount of elements each access. This improves throughput significantly even all element sizes and thus makes the choice of the logical element size insignificant.

Following this observation, the consequence is to enable burst transfer mode, where the DMA engine uses burst transactions on the interconnect to access multiple elements at once. For normal sequential access, the hardware requires that packed access is also enabled. As shown in Figure 4, using burst transfers increases throughput up to a factor of 9, leading to a peak performance of 915 MiB/s, where as the peak performance without burst transfer is 117 MiB/s. Note that in order to be able to conduct burst transfers of size 32 and 64 bytes, the maximum FIFO depth of the device has to be increased. This is done by the SDMA Mackerel wrapper in both the kernel space benchmark as well as in the user space driver.

The device also offers to accelerate graphical operations such as constant fill, transparent copy and orthogonal image rotation. The throughput for a normal non-burst, non- packed constant fill operation is shown in Figure 5. As with normal transfers, the throughput increases when writing larger values, but becomes insignificant when using burst transfers. The other mode of operation is transparent copy, where device only copies values which are different from the value of color register. Interestingly, there is no performance difference if a larger amount of values match the color value. The performance of a transparent copy transfer is virtually the same as for a normal copy one, see Figure 6

Both, constant fill and transparent copy mode can also be used in burst mode, leading in both cases to a huge performance gain, although burst transparent copy has the same characteristics as a normal burst mode copy. Constant fill in burst mode achieves the maximum average throughput of all measurements, 1285 MiB/s, as shown in Figure 7.

Using the double-index addressing mode presented in Section 2.3.2.2, the SDMA module can also be used to perform orthogonal rotations. Burst transfer is compatible with the double-index addressing mode used for rotation. Figures 9 and 10 show the throughput when treating the memory region as a rectangular image and rotating it by 90, 180 and 270 degrees. For an transfer size of 2^b bytes, the width and height of the image is set to $2^{\lceil \frac{b}{2} \rceil}$ bytes and $2^{\lfloor \frac{b}{2} \rfloor}$ bytes, respectively.

However, as seen in 10, while burst mode does increase throughput for rotations, the speedup by using burst transaction is not nearly as large as with post-increment addressing mode. The performance of burst transfers with rotated address generation on the destination port is somewhat comparable to normal copy operation where only the source port is accessed in burst mode. Also, there is virtually no performance difference when rotating the image 90° or 270°.

The impact on performance of using different access settings for the source and destination port is illustrated in Figure 8. Enabling burst access or packed access only on the write or the read port shows that optimized source access does improve performance even for slow writes, but fast writes cannot be performed if read access is slow.

The device supports up to logical 32 transfers in parallel. Although the device itself can only perform up to four transactions on the interconnect concurrently, thus has to schedule access to the read and write ports. Figure 11 shows the throughput per channel when requesting multiple transfers at once. Not surprisingly, the throughput per channel decreases, as the device has to schedule other channels concurrently. However, the aggregated throughput displayed in Figure 12 shows that the total throughput of the device eventually increases when using concurrent transfers. The exception are two concurrent transfers, where the device actually performs worse than when issuing a single transfer. Peak throughput can be reached when issuing four concurrent transfers, which matches the internal mode of operation of the device.

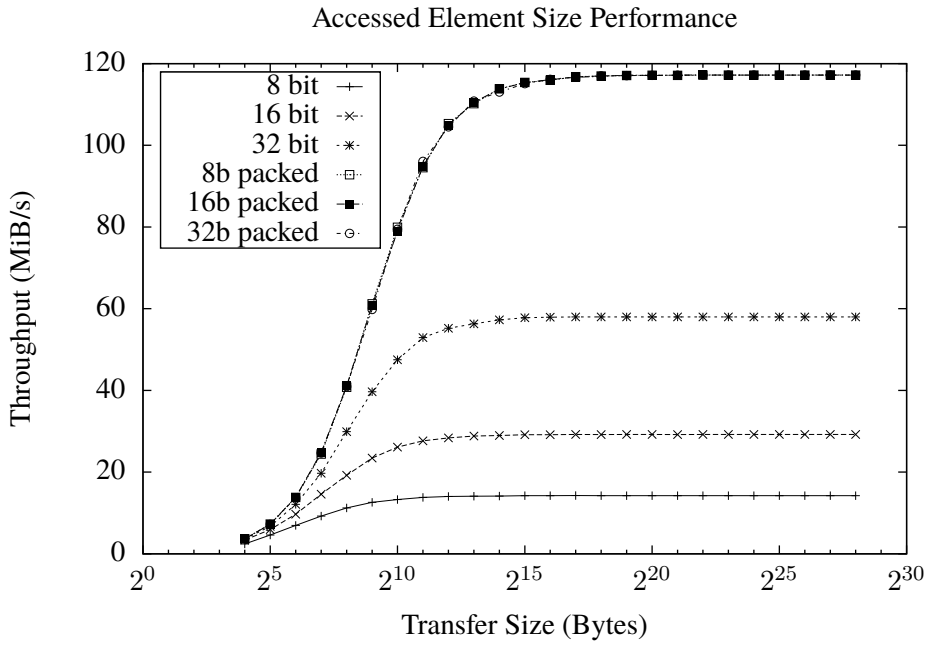

Figure 3: Impact of the element size (8, 16 or 32 bits) on the throughput for different transfer sizes. It can be seen that since the optimal amount of elements are transferred in packed access, the chosen element size becomes irrelevant when using packed access.

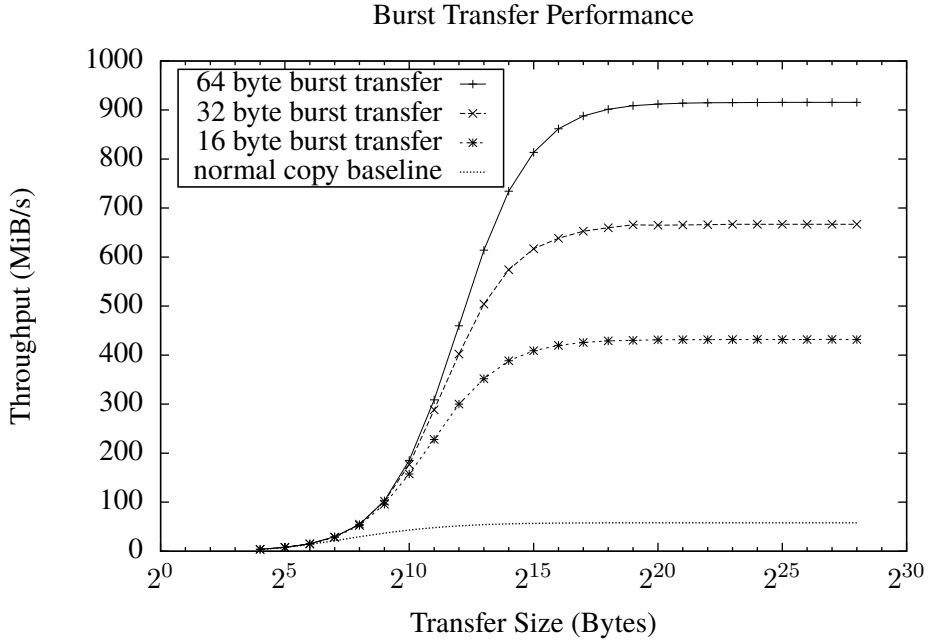

Figure 4: Performance evaluation of different burst transfer sizes. As shown, burst transfers can improve throughput significantly.

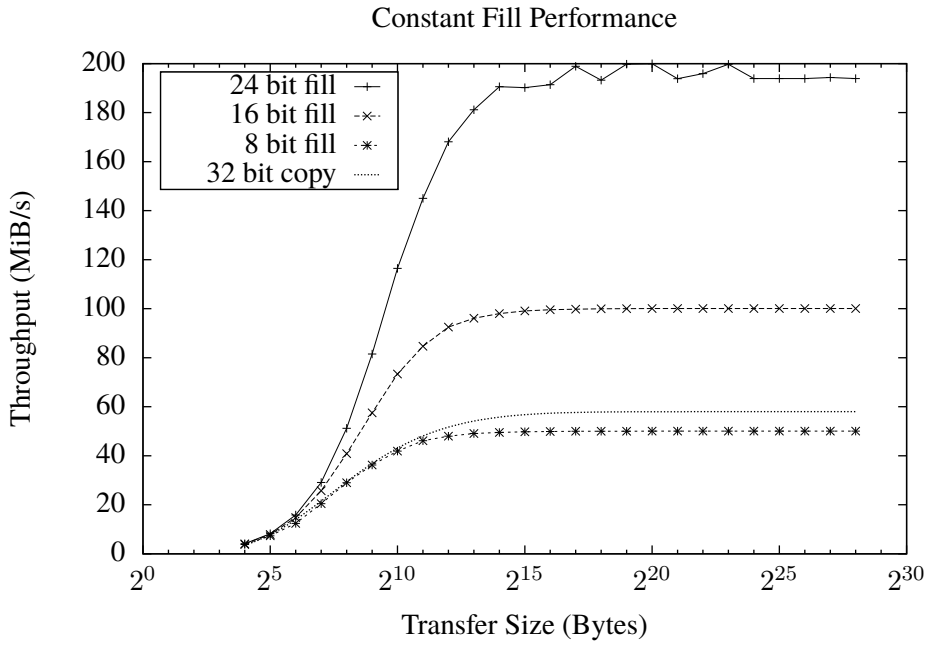

Figure 5: Performance of the constant fill graphical acceleration feature, using color values of size 8, 16 and 24 bits. The spikes when using a 24 bit fill value where consistent in all measurements.

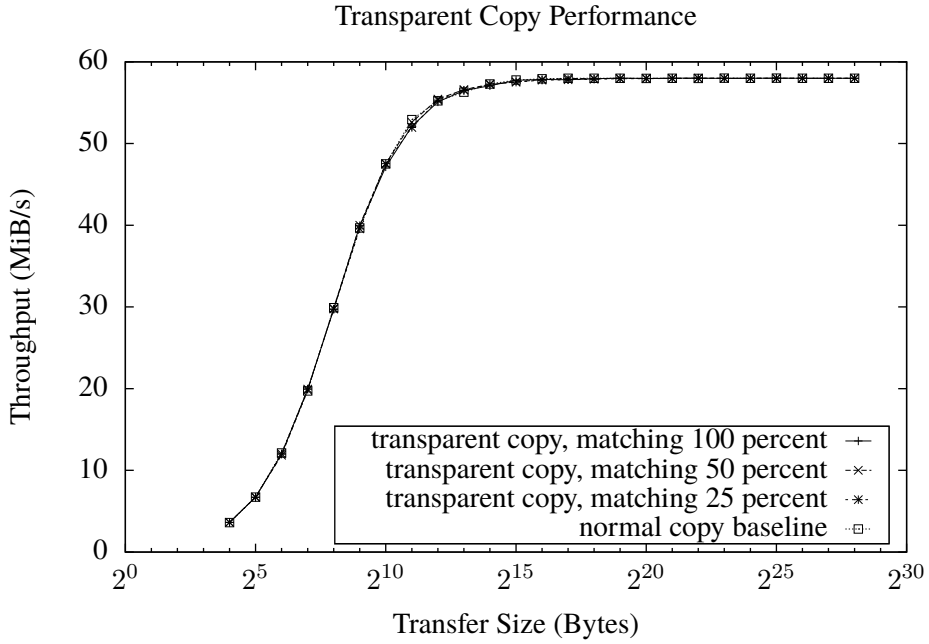

Figure 6: Performance of the transparent copy graphical acceleration feature, using an element size of 32 bits with a 24 bit comparison value. Note that the percentage of matching elements has virtually no impact on performance.

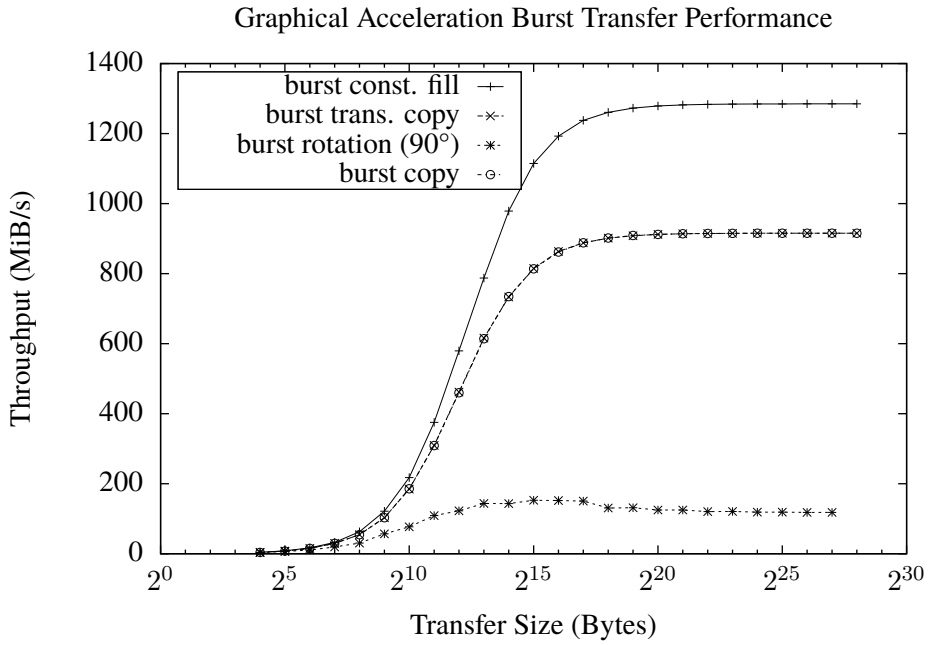

Figure 7: Performance evaluation of using burst transfer mode for the graphical acceleration features such as constant fill, transparent copy and image rotation, compared to normal burst copy. Note that there is no performance difference between normal copy and transparent copy in burst mode.

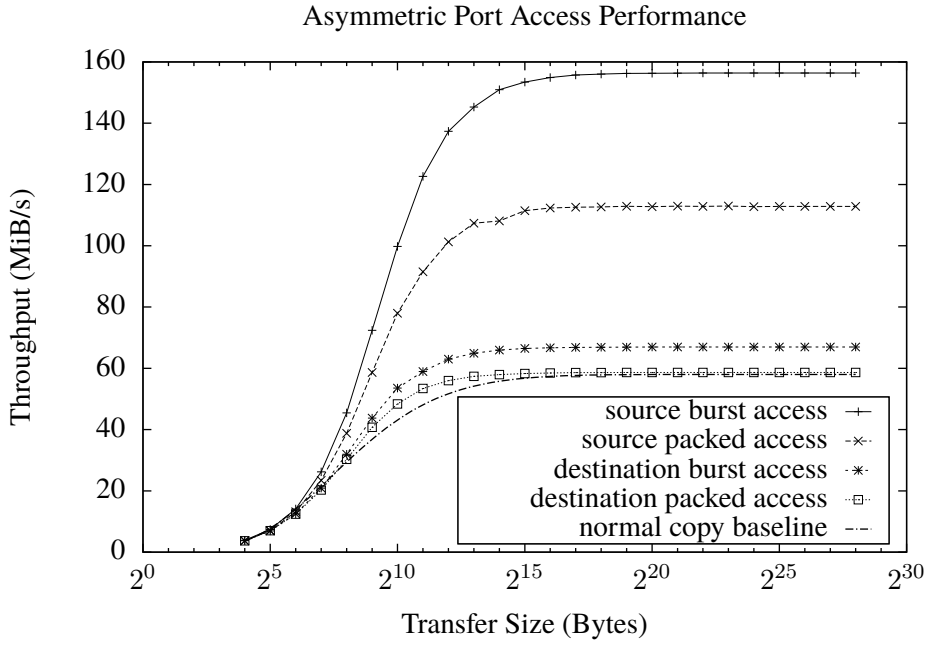

Figure 8: Impact of asymmetric port access on performance. Note that packed or burst access only on the destination port does not show the same improvements as burst/packed access on the source port. The element size was set to 32 bit, burst transfer size was set to 64 bytes.

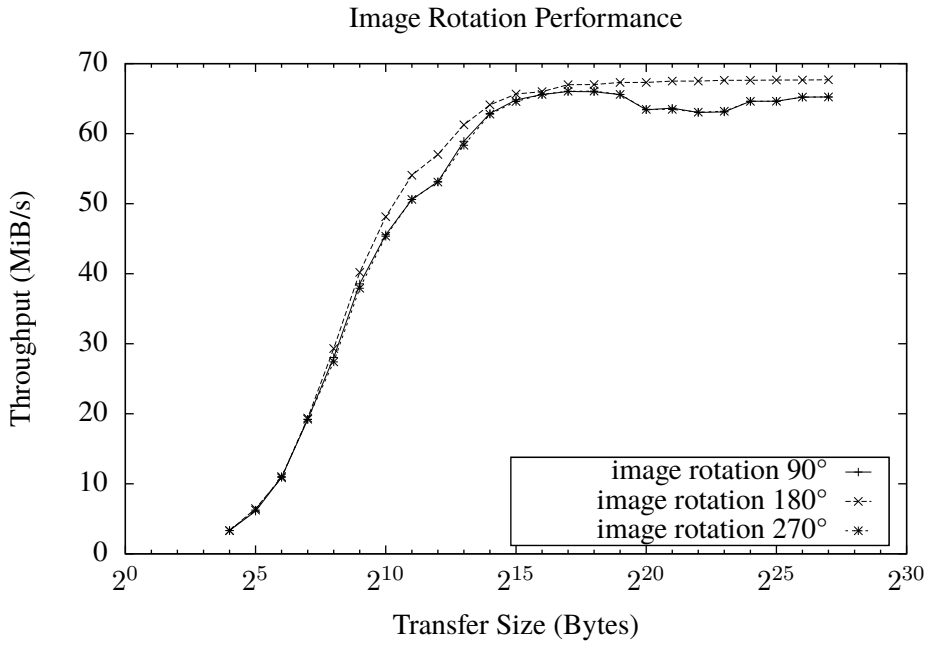

Figure 9: Performance of image rotation by using the double-index addressing mode on the destination. Element and frame size was chosen to represent a nearly quadratic image.

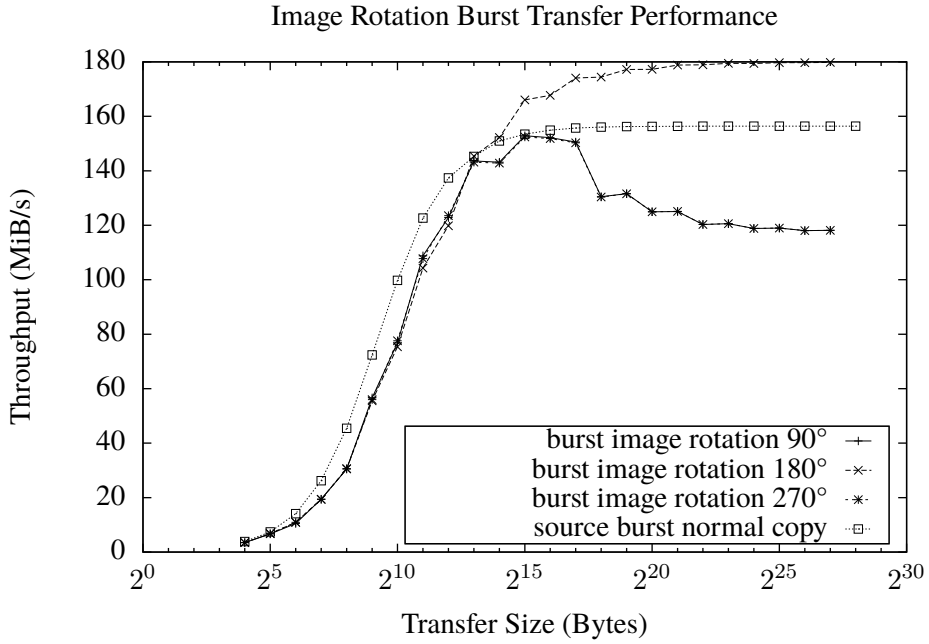

Figure 10: Performance of image rotation by using the double-index addressing mode on the destination with burst mode enabled for both source and destination, compared to a normal copy transfer with burst mode on the source port only.

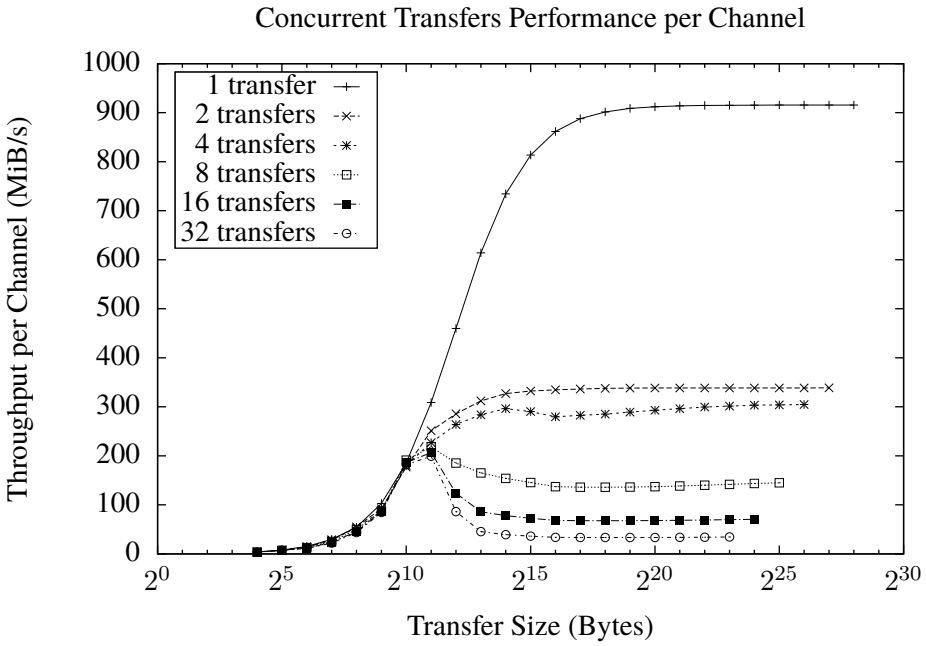

Figure 11: Throughput *per channel* when simultaneously running up to 32 transfers in parallel.

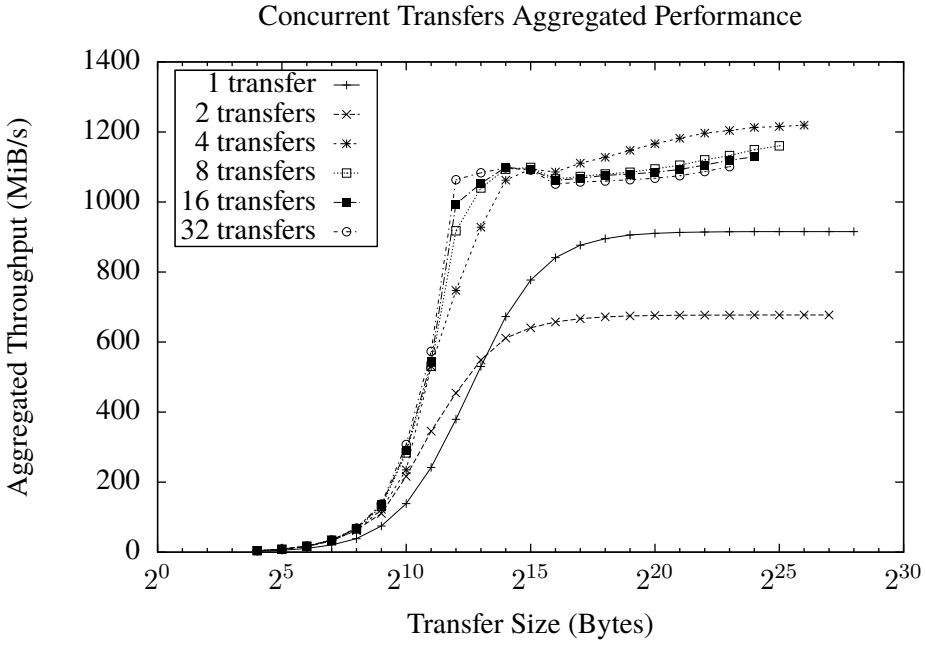

Figure 12: Aggregated throughput when simultaneously running up to 32 burst transfers in parallel.

4.2 Driver Performance

In this section, the performance of the driver is evaluated, first assuming that only a single application has access to the device, then when multiple applications are concurrently using the device.

4.2.1 Setting

The evaluate the overhead introduced by the driver managing the SDMA module, the execution time of memory-to-memory transfers using the SDMA driver was measured in a user level application.

The benchmark application uses the Flounder interface exposed by the driver, using the THC remote procedure stubs and measures execution using the Cortex A9 Global Timer. Because the Cortex A9 Global Timer has to be accessed using a system call to the CPU driver, the overhead of the syscall itself measured and subtracted from subsequent calls.

To test performance when multiple applications have access to the device, the benchmark program spawns several worker processes, all running in their own dispatcher as it would be in a real-world scenario. In order to assume the worst case scenario that all applications access the device at the same time, Barrelfish's distributed synchronization barriers (`dist/barrier.h`) are used to issue transfer requests in all worker processes at the same time. However, because multi-core support on the Pandaboard is not yet mature, all applications were running on the same core.

To synchronize all processes, a master process is responsible to spawn all worker processes and has to manage the use of the synchronization barriers. The master process also performs the benchmark where only a single application has access to the device. Each process measures its transfer time individually. In order to determine the overhead of the timer syscall accurately, the synchronization barriers are also used to ensure that the overhead is measured consecutively.

In contrast to the previous section, only burst transfers were issued, as the driver configures the channels to use burst transactions by default. Note that the transfer sizes in this section are smaller, as the user space benchmark was not allowed to allocate arbitrarily large physical frames, such as frames smaller than 4 KiB, or larger than 32 MiB. The number of concurrent clients in this benchmark was limited to six, as each client had to allocate 128 MiB of memory to perform the actual transfers on and the Pandaboard only has 1 GiB of memory available.

4.2.2 Results

Figure 13 shows the latency of a transfer when using the user level driver, compared to the latency of a normal burst transfer when using the hardware directly from the previous section. It shows that the driver has an overhead of roughly 1 millisecond to initiate the transfer, although the overhead is a bit higher for the two-dimensional interface as the remote procedure call carries more arguments. However, it can be seen that the latency converges with the latency of the kernel benchmark with increasing transfer size. The same effect can be observed in Figure 14 for the throughput. As the transfer size was limited to 32 MiB, the maximum throughput of the kernel benchmark was never reached.

As the driver is able to serve multiple clients concurrently, the latency and throughput for up to six clients was also measured and is displayed in Figures 15 and 16. The driver shows some of the same patterns when issuing concurrent transfers as the hardware does, for example maximum throughput with four concurrent requests and minimum throughput with two concurrent requests.

As the motivation for this benchmark was to evaluate the overhead of the driver, only normal sequential access transfers were executed. Constant fill and transparent copy were also measured, but are not pictured here, because they all behaved similarly when compared to the results of the kernel benchmark.

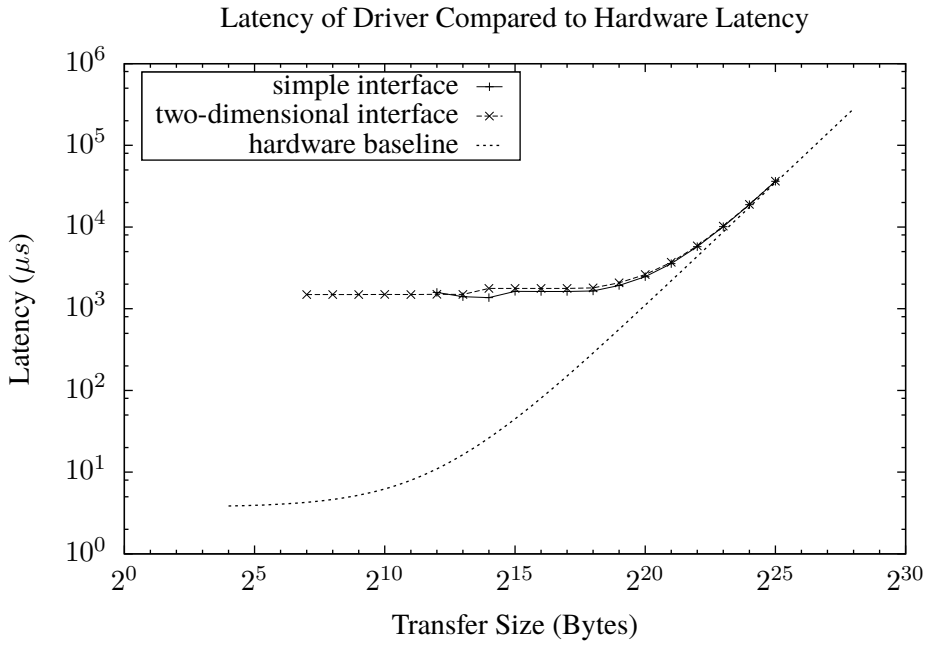

Figure 13: The latency of the two interfaces for different transfer sizes compared to the latency of the direct use of the device.

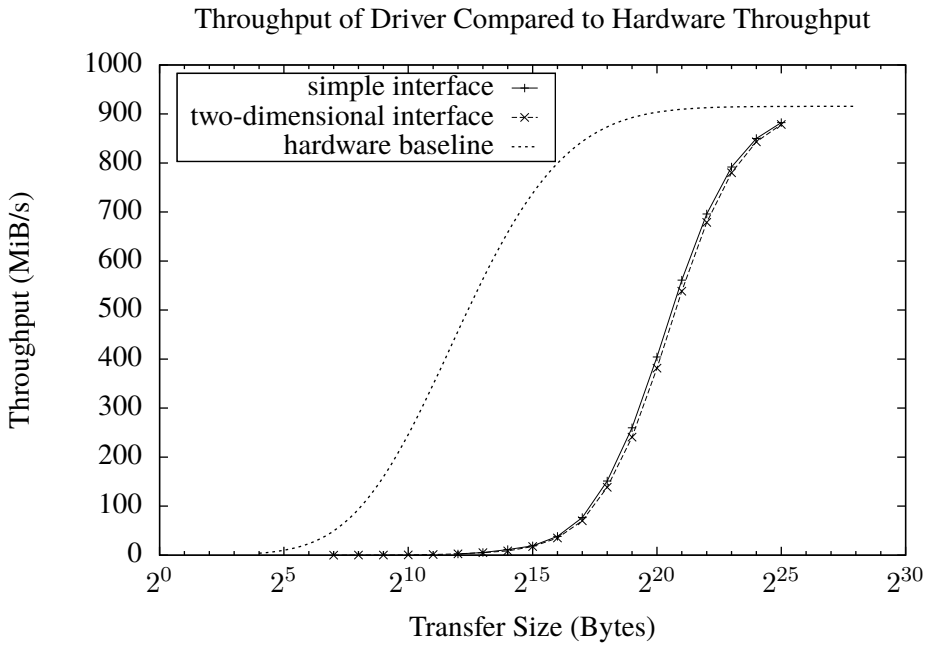

Figure 14: The throughput of the two interfaces compared to the throughput when using the device directly.

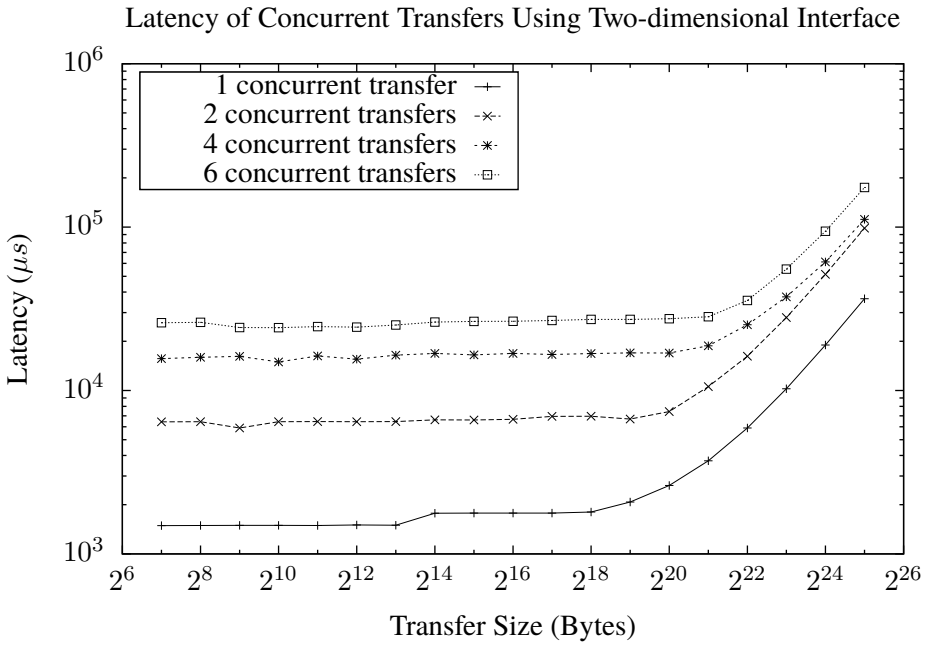

Figure 15: Latency per client when issuing concurrent transfers. The duration of individual transfer increases with the number of concurrent transfers

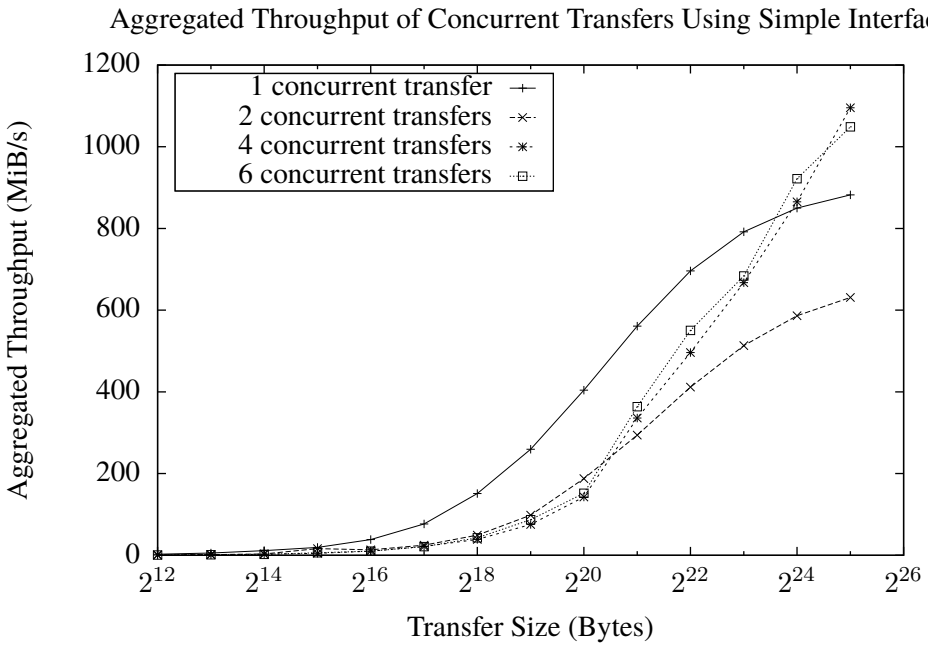

Figure 16: Aggregated throughput when issuing concurrent transfers.

4.3 Bulk Transfer Scenario

The evaluations shown above have shown that there is some overhead when using the SDMA driver to offload memory-to-memory transfers. Therefore, this section is concerned with the usefulness of the implementation.

4.3.1 Setting

For this evaluation, a simple bulk transfer benchmark was written for the purpose of comparing the performance of software-based memory-to-memory transfers with memory-to-memory transfers offloaded to the hardware device. The benchmark implements a small set of functions which can be used to issue bulk data transfers.

The interface allows a client to specify a source and destination memory region, as well as the amount of bytes to transfer from the source to the destination frame. The purpose of this interface is to support shared memory bulk transfers, where there is a set of frames known to both the producer and the consumer, and the producer can send its data to the consumer by writing it into one of shared frames.

There are three implementations of this interface:

Frames Mapped Each Request In this implementation, the client specifies a source and destination frame capability with each transfer request. This means that the frames have to be mapped into the virtual address space of the producer for each request before the data can be copied.

Pre-Mapped Frames To avoid mapping the frames each time, this implementation has a setup phase, where the client specifies a set of frames which are mapped in the server during this initialization phase. During the request phase, the client only has to specify a numerical identifier for the frame, allowing the producer to start the copy phase immediately.

DMA-Assisted Using the SDMA driver, this bulk transfer implementation can offload its requests to the DMA-engine, thus can avoid mapping the frames in the server completely. This implementation also has a setup phase where the a connection to the SDMA driver is initiated.

For the software-based implementations, the frames are mapped as cacheable memory regions, thus giving the advantage of using the processor caches to improve performance for smaller transfer sizes. In this benchmark, all code is running in a single application which acts as both, the server and the client.

The DMA-assisted implementation uses the `mem_copy_2d` function of the SDMA driver, in order to be able to transfer only parts of a frame. Note that the flexible addressing modes are not used, the access pattern is sequential.

Two sets of scenarios were tested for all implementations: In the first scenario, the same frame is used for all requests, thus allowing the processor to keep the data in cache. In the second scenario, the frames were alternated, forcing the implementations to copy a different memory region each time.

As the SDMA driver can perform multiple requests in paralellel, the SDMA implementation was used in the second scenario in two different ways: In one simulated client, the client waits for completion of the previous request as it does for the software-based implementations. In the second client, the code uses the `async` construct of THC to issue a subsequent request even before the previous one has finished. The amount of concurrent requests is limited to eight, due the number of available frames.

4.3.2 Results

Figures 17 and 19 show the throughput and latency, respectively. The clients were reusing the same frame for all requests with different transfer sizes. The label *mapped* refers to the implementation where the frames are mapped each request. This implementation shows the worst performance, due the overhead introduced by the mapping. In the *premapped* graph, which illustrates the performance of the implementation which pre-maps frames, the caching effects are clearly visible for transfer sizes between 4-32 KiB. In those transfer sizes, it performs much better than the DMA-assisted implementation labeled as *sdma*. However, the SDMA engine is known to work well with large transfer sizes, thus showing that starting with a payload size of 512 KiB, the DMA-assisted implementation shows its benefits.

When the frames are alternated, the cache effects are not visible anymore in the implementation which pre-maps frames, and transfer rates are not as good as in the previous experiment. Figures 18 and 20 illustrate this. As there are now eight frames, the SDMA implementation can be used to cascade up to eight requests, labeled in the graph as *sdma async*. This slightly increases performance of the DMA-assisted bulk transfers.

In both cases, the implementation which has to map the frames each request only reaches nearly the throughput of the pre-mapped solution for very large transfer sizes.

In general, it can be concluded that using the DMA engine for larger bulk transfers is useful. However, it should be noted that these transfers are not cache-coherent, therefore an additional overhead might be introduced when caches have to be flushed in order to access the data.

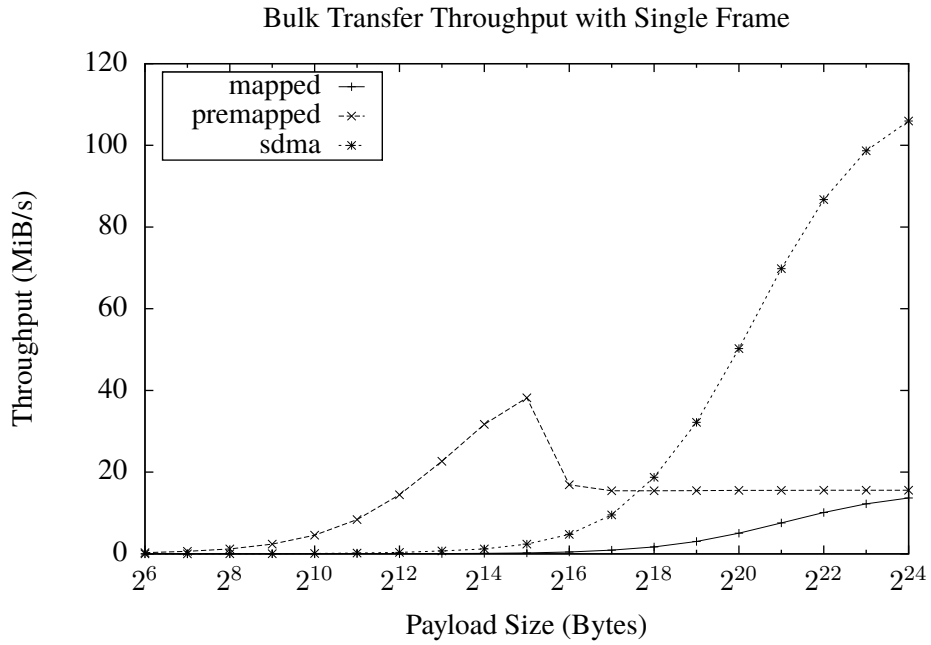

Figure 17: Throughput of bulk transfers using different payload sizes. The same memory region was used in all requests in this experiment.

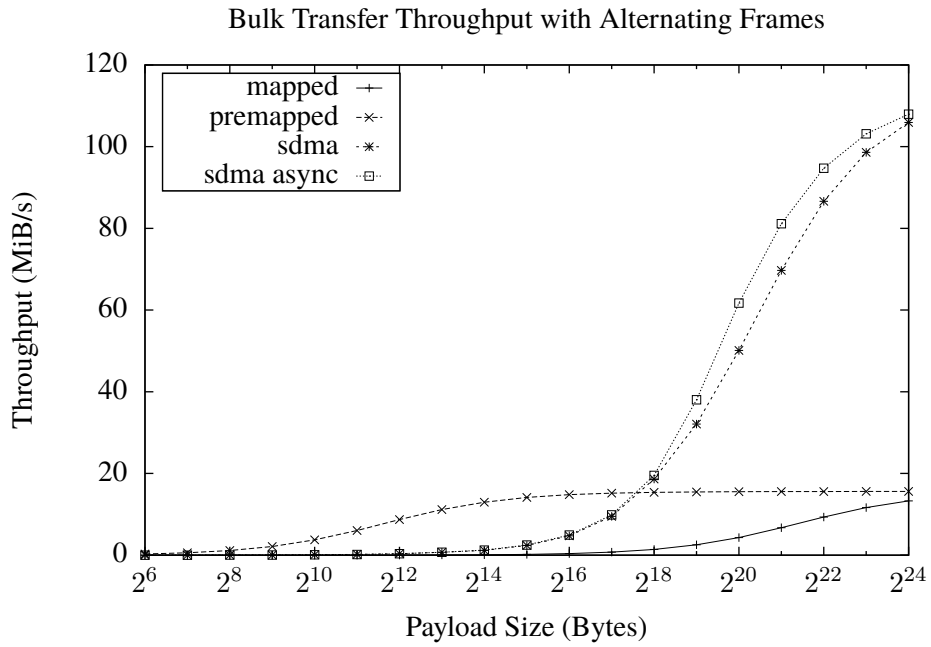

Figure 18: Throughput of bulk transfers using different payload sizes. The memory region used in the request was alternated between eight different frames.

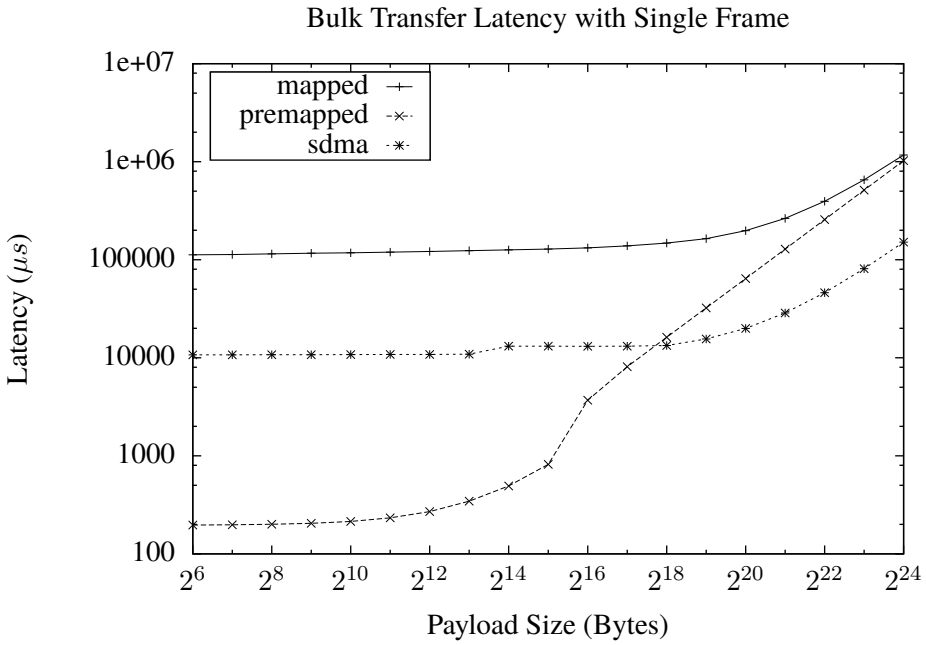

Figure 19: Latency of bulk transfers using different payload sizes with only one frame used for all requests.

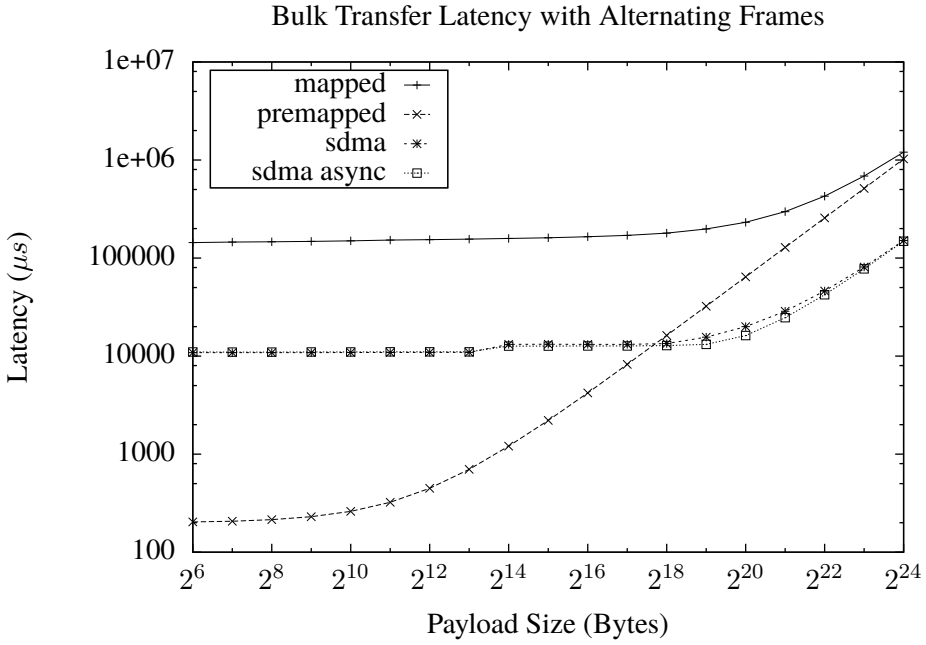

Figure 20: Latency of bulk transfers using different payload sizes with multiple frames used in the requests.

5 Discussion

This section contains the discussion of unresolved issues, a survey of related work, a selection of possible future work and the conclusion.

5.1 Unresolved Issues

Besides some features which were not implemented, which are featured in Section 5.3, there is also an unresolved issue regarding cache coherency.

The current implementation of the driver interface does not perform cache coherent transfers, as memory transfers done by the SDMA module bypass the CPU cache. It is the applications responsibility to map the transfered frames as non-cacheable, in order to ensure a consistent view of the data.

An alternative would be to flush the cache entries associated with the transferred frame before the transfer for the source frame, or rather after the transfer for the destination frame. It is a trade-off to choose between those two alternatives, as both have a different impact on performance. The functionality to flush only parts of the cache is currently not implemented in the Pandaboard port of Barrelfish.

5.2 Related Work

Offloading memory-to-memory transfers to separate copy engines has been suggested before. Zhao et al. [Zha+05] have found such engines could be useful for bulk data movement, but have also noted that classical DMA engines do have significant overhead in communication between the DMA engine and the CPU.

5.2.1 I/OAT Support in Linux

Part of Intels I/O Acceleration Technology [I/OAT] is a DMA engine embedded into modern Intel Xeon processors, primarily designed to improve performance of network traffic, by offloading memory-to-memory transfers to this DMA engine.

Support for I/OAT DMA engine was implemented for the Linux kernel in 2005 by Grover and Leech [GL05] by implementing an asynchronous `memcpy` interface for the use in kernel drivers.

An initial evaluation of using the I/OAT framework for different network traffic offloading was performed by Vaidyanathan and Panda [VP07]. In this evaluation, the use of I/OAT showed reduced CPU usage and improved performance especially for concurrent memory transfers.

5.2.1.1 Asynchronous `memcpy` in user space

As the I/OAT DMA engine only was used to increase network throughput of kernel drivers, in a later papers Vaidyanathan et al. evaluate the usefulness for other workloads, by enabling user space applications access to the I/OAT engine. They argue that offloading memory-to-memory copies to the DMA engine might not only increase throughput, as a CPU has to perform memory copies as a series of smaller sized load and stores instructions,

but that using a separate DMA engine also avoids polluting CPU caches. Using a DMA engine can also enable parallelism, as the CPU can overlap computation with the memory transfers.

In a first iteration, they implemented an asynchronous memory copy interface similar to `memcpy` which allows user space applications to offload memory transfers to the DMA engine. As the interface works with virtual addresses, the implementation has to translate these into physical addresses for the DMA engine. The kernel module must also pin the according pages to memory in order to to avoid swapping during the transfer. To perform memory-to-memory transfers between different processes, they also implemented a file-descriptor based interface for inter-process communication. [Vai+07a]

In a third paper [Vai+07b], they present an approach of increasing the overlap with computation even more by offloading the initialization phase to a different CPU core on the system, therefore achieving up to 100% overlap.

Another interesting approach presented in the third paper is support for the I/OAT DMA engine in an application-transparent matter. Implemented as a drop-in replacement for `memcpy`, they offload memory transfers initiated via `memcpy` to the DMA engine, but return immediately to the application before the actual transfer is finished. To avoid that the application reads data which might has not yet been copied, the according memory pages are read and write protected, therefore any access before the end of the transfer results in a page fault. After an offloaded memory transfer has finished, the page protection is lifted and the application is free to access the memory again. If an application does not immediately try access the memory transferred by a `memcpy` call, it gains some computation overlap. Using this copy mechanism in the *gzip* program, Vaidyanathan et al. were able to show up to 10% performance benefit.

5.2.1.2 KNEM A different approach of using the I/OAT DMA engine in applications is presented by Buntinas et al. [Bun+09]. Designed for improving performance of large intra-node messages in the MPI framework, they implemented a Linux kernel module called Kernel Nemesis (KNEM), which exposes access to the DMA engine through a pseudo-character device.

Because it is designed to be platform-independent, KNEM does also offer other methods of kernel-assisted message transfers. Therefore, depending on the message size, KNEM might not actually use the DMA engine to perform the transfer, because the overhead of initializing the hardware would be to high for small messages. In their paper, the authors propose a formula to calculate this minimum message size based on the CPU cache size on the Xeon architecture.

In contrast to the OMAP SDMA engine, the I/OAT DMA engine does not use interrupts to notify the processor about the completion of the transfer. To avoid polling in a kernel thread, the KNEM implementation can also schedule a single-byte DMA transfer following the actual DMA transfer, which writes a status flag in memory to indicate completion. This allows the user application to poll for com-

pletion in its own memory, which already is the done by design in many message passing systems. [Bun+09; GM12]

5.3 Future Work

There are several opportunities for future work. This section outlines a few of them.

Support for Scatter-Gather Transfers One characteristic of the OMAP SDMA module was not implemented as part of this thesis, as there was not enough time left: Support for scatter-gather transfers. Currently, applications which want to use the SDMA engine to offload memory-to-memory transfers have to use a contiguous region of physical memory, represented by a `Frame` capability. While this is suitable for shared memory systems, a memory region in a virtual address space can consist of multiple scattered physical frames. Thus, an application which would like to execute a memory-to-memory transfer in its virtual address space could make use of the scatter-gather feature by specifying a list of frames to transfer. This could reduce the overhead compared to when one request per frame is issued.

Application Transparent Interfaces The current implementation of the driver offers an interface which asynchronously mimics the `memcpy` and `memset` functions of the C language. However, an application has to explicitly make use of the driver's functions when it wants to offload memory-to-memory copies to the DMA engine. Barrelfish's self-paging system would be very suitable for a DMA-assisted, application transparent implementation of the actual `memcpy` or `memset` functions, similar to the one presented in [Vai+07b]. Such an application transparent implementation for offloading memory copies in the virtual address space would possibly benefit from scatter-gather transfer support.

Hardware Synchronized Transfers The scope of this thesis was to support memory-to-memory transfers. The OMAP SDMA module however can also be used to support transfers initiated by hardware devices, which copy data from the device to memory or vice-versa. The current driver could be extended to support such transfers with a reasonable amount of work, as Barrelfish's capability system treats memory-mapped regions similar to regions of physical memory. Some review of the current driver however would be necessary, as the Mackerel wrapper makes some assumptions only valid for software-synchronized memory-to-memory transfers.

Deeper Integration into Barrelfish Barrelfish's inter-dispatcher message passing system can be extended to support different methods for transporting messages. As others have shown, using a DMA engine for large message transfers has been successfully implemented, therefore it could be evaluated if a message passing backend based on the OMAP SDMA engine would beneficial.

This thesis has successfully evaluated the usefulness of the SDMA module in a possible bulk transfer system, but it was not integrated into the existing bulk transfer interface already available in Barrelfish, this could also be a subject of future work.

Lastly, there are other places in an operating system were asynchronous memory operations are useful, such as zeroing out newly allocated frame capabilities. This is currently done in the CPU driver in Barrelfish, therefore additional work would be necessary to use the SDMA engine for this task, as the driver is running in user space.

5.4 Conclusion

This thesis has presented a service implementation which performs memory-to-memory transfers using the OMAP SDMA module in the Barrelfish research operating system. This was done by implementing a user space driver which offers an interface for asynchronous memory transfers between physical memory frames. The driver also offers interfaces for filling a frame with a constant value and supports more flexible addressing schemes, which can be used for orthogonal image rotation or stride access.

The frameworks provided by the Barrelfish OS simplified this task significantly. Configuring the hardware using its memory-mapped configuration registers was eased by the Mackerel DSL, eliminating the need for tedious and error-prone bit-level operations. The capability system of Barrelfish used for memory management enables client applications to easily refer to physical memory regions, thus eliminating the need of translating virtual addresses to physical addresses for the DMA engine. Using capabilities to refer to physical frames also means that the driver does not need to validate ownership of the memory region, as capabilities are unforgeable. Lastly, the asynchronous nature of Barrelfish's inter-dispatcher communication system and the integration of the THC language extension enables client applications to easily exploit the parallelism enabled by offloading transfers to an external DMA engine.

Efforts were also made to explore the performance characteristics of the various features offered by the SDMA hardware module. The observations made helped settling on a simple, but useful interface for client applications. The performance of the resulting driver was also evaluated and compared to the direct use of the device, showing the overhead introduced by managing the device in a user space driver. The usefulness of the device was evaluated by using it in a simulated simple bulk data transfer framework. It was shown that such a system can benefit from offloading larger payloads (>512 KB) to the SDMA engine, instead of performing the transfers in software.

References

[TN000] Team Barrelfish. *Barrelfish Architecture Overview, Barrelfish Technical Note 000*. Tech. rep. ETH Zurich, 2013. URL: http://www.barrelfish.org/TN-000-Overview.pdf.

[TN013] Akhilesh Singhania, Ihor Kuz, and Mark Nevill. *Capability Management in Barrelfish, Barrelfish Technical Note 013*. Tech. rep. ETH Zurich, 2013. URL: http://www.barrelfish.org/TN-013-CapabilityManagement.pdf.

[TN011] Andrew Baumann. *Inter-dispatcher communication in Barrelfish, Barrelfish Technical Note 011*. Tech. rep. ETH Zurich, 2011. URL: http://www.barrelfish.org/TN-011-IDC.pdf.

[Har+11] Tim Harris et al. "AC: composable asynchronous IO for native languages". In: *ACM SIGPLAN Notices* 46.10 (2011), pp. 903–920.

[OMAP4460] *OMAP4460 Silicon Revision 1.x Multimedia Device Technical Reference Manual*. Version AA. Texas Instruments. July 2013.

[TN019] Team Barrelfish. *Device Drivers in Barrelfish, Barrelfish Technical Note 019*. Tech. rep. ETH Zurich, 2013. URL: http://www.barrelfish.org/TN-019-DeviceDriver.pdf.

[KG05] David J. Katz and Rick Gentile. *Embedded Media Processing*. Newnes, Sept. 2005. ISBN: 9780750679121.

[A9TRM] *Cortex-A9 MPCore Technical Reference Manual*. Issue I. ARM. June 2012.

[GCC] *GNU GCC Manual*. Free Software Foundation, 2010. URL: http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/.

[Zha+05] Li Zhao et al. "Hardware support for bulk data movement in server platforms". In: *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*. IEEE. 2005, pp. 53–60.

[I/OAT] *White Paper: Accelerating High-Speed Networking with Intel I/O Acceleration Technology*. Tech. rep. 2006. URL: http://download.intel.com/support/network/sb/98856.pdf.

[GL05] Andrew Grover and Christopher Leech. "Accelerating Network Receive Processing". In: *Linux Symposium*. 2005, p. 281.

[VP07] Karthikeyan Vaidyanathan and Dhabaleswar K Panda. "Benefits of I/O acceleration technology (I/OAT) in clusters". In: *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*. IEEE. 2007, pp. 220–229.

[Vai+07a] Karthikeyan Vaidyanathan et al. “Designing efficient asynchronous memory operations using hardware copy engine: A case study with I/OAT”. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE. 2007, pp. 1–8.

[Vai+07b] Karthikeyan Vaidyanathan et al. “Efficient asynchronous memory copy operations on multi-core systems and I/OAT”. In: *Cluster Computing, 2007 IEEE International Conference on*. IEEE. 2007, pp. 159–168.

[Bun+09] Darius Buntinas et al. “Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis”. In: *Parallel Processing, 2009. ICPP’09. International Conference on*. IEEE. 2009, pp. 462–469.

[GM12] Brice Goglin and Stéphanie Moreaud. “KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework”. In: *Journal of Parallel and Distributed Computing* (2012).

List of Figures