HiPPAI: High Performance Portable Accelerator Interface for SoCs

Paul M. Stillwell Jr., Vineet Chadha, Omesh Tickoo, Steven Zhang, Ramesh Illikkal, Ravi Iyer, Don Newell
{paul.m.stillwell, vineet.chadha, omesh.tickoo, steven.zhang,
ramesh.g.illikkal, ravishankar.iyer, donald.newell} @intel.com

Abstract

Specialized hardware accelerators are enabling today's System on Chip (SoC) platforms to target various applications. In this paper we show that as these SoCs evolve in complexity and usage, the programming models for such platforms need to evolve beyond the traditional driver oriented architecture. Using a test set up that employs a programmable FPGA based accelerator to implement one of the critical computation functions of a Mobile Augmented Reality based workload, we describe the performance drawbacks that a conventional programming model brings to compute environments employing hardware accelerators. We show that these performance issues become more critical as the interface latencies continue to improve over time with better hardware integration and efficient interconnect technologies. Under these usage scenarios, we show with measurements that the software overheads enforced by the current programming model, like those associated with system calls, memory copy and memory address translations account for a major part of the performance overheads. We then propose a novel High Performance Portable Accelerator Interface (HiPPAI) for SoC platforms using hardware accelerators to reduce the software overheads mentioned above. In addition, we position the new programming interface to allow for function portability between software and hardware function accelerators to reduce the application development effort. Our proposed model relies on two major building blocks for performance improvement. A uniform virtual memory addressing model based on hardware IOMMU support and direct user mode access to accelerators. We demonstrate how these enhancements reduce the overheads of system calls and address translations at the user/kernel boundary in traditional software stacks and enable function portability.

1. Introduction

Increasing transistor density driven by Moore's law along with the limitations imposed by the frequency scaling and the need for single thread performance is forcing the processor architecture to integrate more and more specialized functionalities into the hardware. Increasing number of embedded platforms (SoCs) use specialized hardware accelerators or IP blocks to implement functionality in hardware [8, 9, 10, 11, 12]. Traditionally these accelerators were treated as devices in the platform by the software. In this paper, we analyze the suitability of the conventional programming models for such platforms. The programming models were based on a clear demarcation of functionality between hardware and software. Operating systems provided the isolation between high level user functions and low level hardware execution units/compute engines and platform devices. User applications typically rely on mechanisms like system calls to isolate the user space processes from directly controlling the hardware. This allowed the OS and drivers to be the single point of management for the shared HW resources. Such a model provides resource isolation and enables sharing at the cost of performance overheads in terms of system calls and resource indirection (page tables, interrupt vectors etc.) This model is well suited for IO devices and accelerators with execution times much higher than these interface overheads. As interconnect latencies shrink (especially with SoCs), the performance overheads introduced by the traditional programming models are becoming the major bottleneck in allowing fine grain acceleration opportunities [6, 7]. Reducing the overall interface overheads enables an efficient accelerator interface design capable of supporting a wide range of execution offloads.

In this paper we approach this problem from the performance and functionality points of view. We analyze the various performance hotspots in the conventional OS/driver model with respect to accelerator integration. The goal is to understand the major overheads of the current model and to motivate the need for a low overhead software interface for the low latency SoC accelerators. We show that the conventional programming methods impact the accelerator performance by introducing various overheads. We show that while the relative impact of these overheads on performance can be reduced by offloading more coarse grained functionality to the hardware accelerators, a change to the programming model is required to reduce these overheads in the general case. We propose a new software interface model based on virtual memory addressing that removes the performance bottlenecks identified along with the hardware support needed to enable it. Having a consistent programming model between the software and hardware execution of the same functionality provides greater flexibility in the software development and deployment in a wide range of hardware with varying degrees of acceleration support. We underline the need for application programmers to

978-1-4244-4921-7/09/$25.00 ©2009 IEEE

have the flexibility to uniformly develop their programs for both hardware assisted and non-assisted environments. This flexibility enables portability of applications across platforms with hardware function accelerators and the ones without the hardware assist. In the later case, various software modules/libraries can provide the required functionality. Keeping the actual function implementation in hardware or software transparent to the application designer improves the development and debugging time while enabling portability. Our proposed programming model enables portability using standardized function interfaces and data transfer paths between the application and the function execution unit (in software or hardware accelerator). We enumerate the hardware and software support needed to enable such a function portability capability. We combine these functional and performance requirements to propose a system architecture framework called **Hi**gh **P**erformance **P**ortable **A**ccelerator **I**nterfacing for SoC platforms (HiPPAI).

Our prototype implementation uses a modified Linux/driver stack with FPGA based accelerator. The FPGA is programmed to emulate a complex compute function (Fast Hessian Detect) of an open source implementation of a Mobile Augmented Reality (MAR) based workload [13]. Using this set-up we profile the traditional device driver model to identify performance bottlenecks. Our experiments show that the user-kernel interface and the kernel-hardware interfaces introduce both set-up and data path inefficiencies in case of SoCs with hardware accelerators. Our measurements show that these inefficiencies increase the total function execution time by an order of magnitude. We have implemented our proposed programming model on the prototype set-up and our measurements show that various pieces of the model address the performance bottlenecks and portability issues present with traditional software stacks.

The rest of the paper is organized as follows. Section 2 details the motivation for a new accelerator interface – we present a detailed analysis of the overheads associated with today's device model based approach. In Section 3 we introduce a new SoC interface which avoids most of the unnecessary overheads. We describe how different methods can be adopted to remove different overhead components. In section 4 we describe the MAR workload used for our prototype studies. In section 5, we describe our evaluation platform which comprises the hardware and software components and changes we made to build the prototype environment. We also describe our analysis methodology in this section. Section 6 contains the data we collected and the results of our analysis. We demonstrate the performance benefits of the new interface. We conclude with future work and summary in Section 7.

2. Background and Motivation

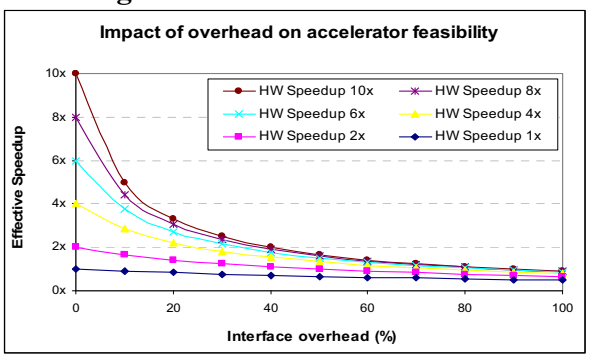


Figure 1: Impact of communication overhead on the effective speedup of HW accelerators.

For many application specific computations, use of dedicated hardware designs can achieve better performance and power-efficiency than software running on a general purpose CPU core. But specialized hardware implementations also have their limitations. Hardware development incurs relatively higher costs compared to traditional software development. It has limited application domain and high upgrade cost. While software running on general purpose processors is easy to develop and upgrade, it may not achieve the performance and power requirement guidelines needed for an SoC. A hybrid computation model where most of the application runs on general purpose processors and only the essential portions of the application are offloaded to special purpose hardware is a natural solution to the problem. Identification of the functions for offload and the implementation of these in the hardware in a performance and power efficient way is key to the success of this hybrid model. The first step in this process is to run software applications under study on a general purpose CPU core to identify the hotspots.

Figure 1 demonstrates the impact of interface overhead on the effective speedup of an accelerator. The x-axis shows the overhead in terms of the execution time of the function being accelerated. The y-axis shows the effective speedup for various absolute accelerator speed ups.

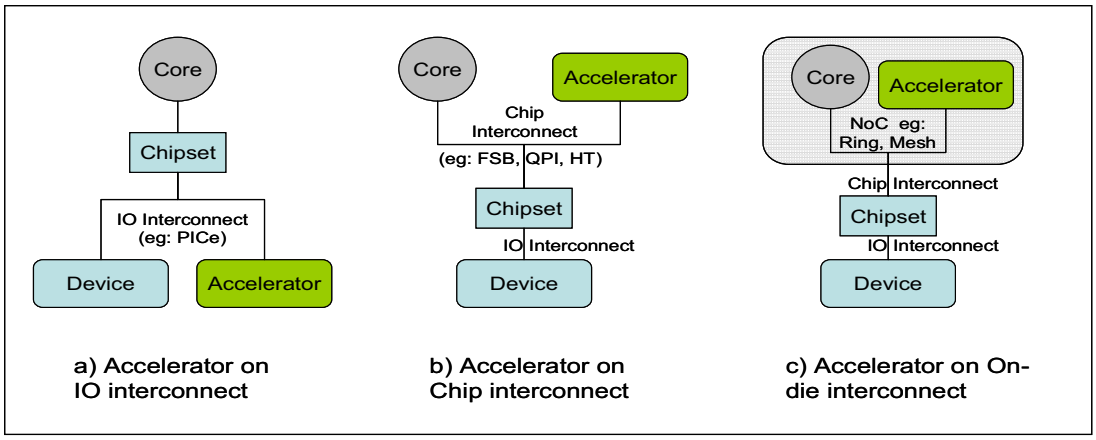


Figure 2: Accelerator interfacing options

For example, the effective speedup goes down from 10x with no communication overhead to a 2x effective speed with a communication overhead of 40% of the execution time. With large offloads, the effect of communication cost can be kept minimal, but as the granularity of the offloaded function decreases, the impact of communication overhead becomes a deciding factor in the effective speedup. On the other hand, as the granularity of the offloaded function goes up, it becomes highly specialized. Fine grain accelerators are more amenable to reuse by multiple applications. Design complexity and cost can be kept low with small offload designs. The interface cost needs to be kept to a minimum for fine grain offload to be performance effective. We outline the two main contributors of latency in an accelerator environment below.

2.1. **Hardware Interconnect Latency**

IO interconnect and bus technologies have come far from their early inception days. We have seen steady decline in the IO latencies from bus technologies that span through ISA, PCI, PCIe, SATA, USB2 and alike. Other technologies in the pipeline promise to make the IO latency a very insignificant part of the total function execution time in the future. Figure 2 shows the various accelerator interfacing options available today.

Until recently, the only interface available for accelerator integration was through standard IO buses discussed above. Recently with Intel's Quick Assist [1, 17] and AMD's Torrenza [2], it has become possible to connect an accelerator using chip-to-chip interconnects like FSB, QPI and Hyper Transport. This drastically reduces the accelerator interconnect latency. For comparison – round trip latency across a PCIe Gen 2 connected accelerator can be around 1 microsecond, while the latency on FSB could be one magnitude lower – under 100ns. Moving to system on chip (SoC) architectures where the core and the accelerators are integrated into single chip, the latencies can be another magnitude lower (~10ns). These reduced interconnect latencies enable smaller offloads to be more effective.

2.2. Software and Interfacing overhead

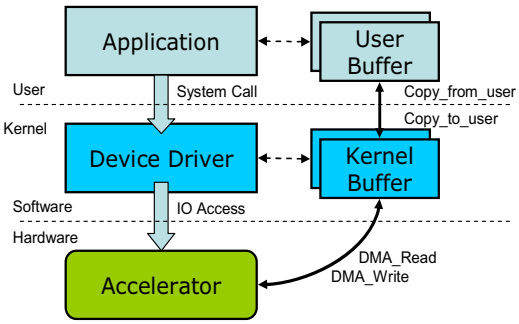


Figure 3: Classic Device Driver Model

As the IO interconnects get faster, the performance bottlenecks are shifting from hardware to software. Once the function to be offloaded is encountered, there needs to be a sequence of operations done in software and hardware before the actual function is executed in hardware and the results are made available to the application. We now look at the major components in this overhead based on the classic device driver model used today. As shown in Figure 3, a classic device driver works in Kernel mode directly accessing and managing the accelerator hardware. Due to the inherent nature of kernel mode device drivers, there are several overheads associated with this model.

- *System Call Overhead*

To protect kernel data security, the OS doesn't allow the user application to directly access the kernel code & data. So when a user application wants to access a device, it must go through the system call interface which is much more expensive than a normal function call. For hardware devices that implement a

course-grain function, this may be okay, but for devices that implement fine-grained functionality the transition from user space to kernel space is too costly. In the results section we present our measurements for the most frequently invoked system calls.

- *Data Copying Overhead*

Hardware accelerators cannot directly access user buffer pages in memory since user buffer pages can be swapped out to disk without notice. Devices and accelerators are not capable of handling page faults so the device drivers use kernel buffers which are pinned down and guaranteed to never be swapped out. The device driver must then copy data from the user buffer into its kernel buffer during a ring transition in order to perform any useful work.

If the user application needs to exchange a large amount of data with the accelerator, there is also another option. Instead of creating a kernel buffer, the device driver can lock user buffer pages into memory and unlock them when the accelerator finishes accessing the page. In this case, the device driver still needs to pay lock/unlock user page overhead instead of data copying overhead.

- *Address Translation Overhead*

A traditional operating system typically employs a multi-tiered approach to address translation. For example under Linux, a resource (memory/IO) virtual address from a user application gets translated to a Kernel (OS) virtual address before a hierarchical page walk returns a physical address to be used for the transaction. While this approach reduces the need for multiple data copies between the user and the OS context, the address translations themselves come with a non-negligible performance penalty. Experimental data for these latencies is presented in our results sections ahead.

- *Other OS overheads*

To the OS software, an accelerator device can appear as a standard I/O device residing on a system IO bus. In this case the IO command, configuration, read and write latencies can prove to be a significant performance bottleneck if the software is not designed to handle such configurations optimally. Traditional operating system kernels like Linux, Windows, etc assume high IO latencies and as a result the user applications have to wait between the issue of a transaction to an IO device and getting its completion. These wait times can add to the function execution times and in turn, reduce performance. With reduced IO latencies, the software needs to be reconfigured to avoid long wait states to take advantage of speedy IO busses.

For classical devices like hard disk or human input devices, the device operation delay dominates the total IO delay. The device driver overheads mentioned above only accounts for a very tiny portion of the total IO delay. And these device driver overheads will not have any significant effects on classic device IO performance. But for hardware accelerator devices, the device operation latency can be as small as a few hundred cycles. These device driver overheads can be a dominant component affecting hardware accelerator IO performance.

To avoid the device driver overheads discussed above, developers have used a direct device driver model in the past. The basic idea behind this model is to pre-allocate a large fixed-size physical memory as device memory and used by the accelerator in a dedicated fashion. Both accelerator IO resources and device memory are memory-mapped into user application address space. The device driver can directly manage the hardware accelerator in user mode and directly manipulate the data in device memory without data copying overhead.

However this model has limitations that restrict its adoption on systems involving function hardware acceleration. The direct-mapping device driver model can reduce some of the overhead inherent in the classic device driver model, but it still has serious limitations. Each accelerator must pre-allocate a large fixed-size physical memory block and as such that memory cannot be reused for other purposes. The memory must also be physically contiguous so that the hardware device can access it. Due to memory fragmentation, it is difficult for system software to find large physically contiguous memory during runtime. So the common practice is to pre-allocate this large physical memory when the accelerator is initialized. For an SoC platform with limited total physical memory capacity and many hardware accelerators, this can cause serious memory pressure. Also if dynamic user buffers are to be used with accelerators, this model will have the data copy overhead.

In both these driver models we face issues when sharing of complex data structures is involved. Because the accelerator cannot access data through a user virtual address pointer with these memory models, data in the device memory domain must be organized as simple record-based data structure. In case a user application wants to send pointer-based data structures to an accelerator, the device driver must perform an expensive serialization operation to copy the pointer-based data structure to device memory.

In summary, the classic device driver model has inherent overheads which can be unacceptable for hardware accelerators. The direct-mapping device driver model reduces these overheads, at the cost of

dividing memory space into two independent domains (an inflexible approach). This generates many serious constraints on its programming model. When adopting direct-mapping device model, programmers may need to put in substantial efforts to rewrite the code to ensure the user application and the device driver don't violate any constraints such as data security and protection.

In the next section we present our new accelerator interfacing model to minimize the software overheads in hybrid environments and enable complex data sharing between applications and accelerators. These optimizations are important in environments where accelerators are connected through low latency hardware interconnects - as in the case of SoCs, and accelerators connected to QuickAssist[1] and Torrenza[2].

3. Proposed Approach: HiPPAI

Our HiPPAI approach addresses the issues presented in the previous section with the following fundamental goals:

1) Eliminate unnecessary system overheads incurred due to domain boundaries and multiple resource addressing modes.
2) Abstract the function interface to the SoC accelerators such that the application development is decoupled from the details of function implementation.

To achieve these goals we base our HiPPAI programming model on two important concepts:

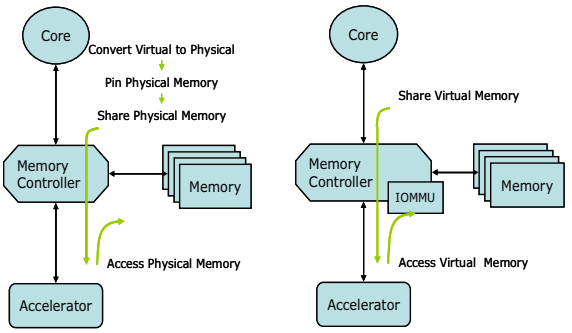
1) Virtual Memory Accelerators: Our programming model introduces the concept of virtual memory accelerators. This concept moves the accelerator function execution to the virtual memory domain. No address translations, system calls or page walks are needed until the hardware resource (memory) is accessed. The only address translation required is enabled through an IO-MMU. This mechanism eliminates all but the absolutely necessary overheads from the classical device driver. Pointer chasing in hardware accelerators also becomes possible through this model since the addressing domain is uniform between the user and the function accelerator.
2) HW/SW function portability: We use standardized interface definitions to couple the function accelerators with user programs. These interfaces decouple the function implementation from program design making it possible to port the user programs between SoCs of varying acceleration capabilities. Further, the discovery, enumeration and initialization of the function accelerators are performed independently by an agent in the operating system. This makes it possible to "virtualize" the accelerator and present multiple interface contexts to different applications.

Our proposed architecture goes beyond just letting the device access virtual memory to place/retrieve data from and allows coordinated data operation accesses from the accelerator and the IA core in a fashion that is transparent to the application. The work presented in this paper differs from the work presented so far in many aspects. While the other researchers have concentrated on architectures and programming models to develop offload aware applications (IBM Hydra [19]), developed techniques to tightly integrate the heterogeneous compute engines like CPU and GPU (Pangaea [20]) and software layers to allow function acceleration in a platform dependent fashion (Merge), our work concentrates mainly on removing the software bottlenecks due to the classical driver model which all of the above models still rely on.

The next two subsections present the components of our HiPPAI programming model in detail.

3.1. Virtual Memory Accelerators

Figure 5 illustrates the proposed virtual address domain accelerator model in comparison with the physical memory model used by drivers. With the accelerator working in the virtual address domain, it no longer accesses the physical address space directly. Instead it can access user virtual addresses directly without any address translation in software. When the accelerator tries to access application memory with a virtual address, special hardware called an Input/Output Hardware Management Unit (IOMMU) will intercept the request and automatically translate the virtual address into the corresponding physical address. The IOMMU is a special hardware unit performing address translation from the device driver address space to physical memory address space. As a mature technology for high-end platforms, the IOMMU can be applied in many contexts. First, it can allow legacy 32-bits devices to access 64-bit high memory. Second, it provides memory protection from malicious or misbehaving devices. Originally, this protection capability was devised for virtualization since it allows unmodified native device drivers to be used in a guest operating system.



a) Physical Memory b) Virtual Memory

Figure 5: Comparison between accelerator memory models

The model presented here overcomes the performance bottlenecks presented by the classical models. First, since the device driver runs in user mode, there is no system call overhead. Second, the user application and accelerator share the same user virtual address space. There is neither a need to copy data between different buffers nor the requirement of a user to kernel address translation. Because of this uniform address space, the virtual-address domain device driver model doesn't have the programming constraints of direct-mapping device driver model. The accelerator does not need pre-allocated device memory and it can directly access pointer-based data structures.

3.2. HW/SW Function Portability

Our proposed model incorporates a functional interface that achieves two purposes:

a) *Uniform call semantics:* We propose to implement the software interface to the accelerator as a standardized function call specific to the intended operation of the accelerator. This kind of design decouples the accelerator design from application development. All that an application programmer needs to know is the name of the function call to invoke a specific task that may be accelerated on a target platform. We envision that these names will be globally known and would be derived from available specifications in the field of operations. For example, in the case of image/video processing one could use the function names from the architecture frameworks like OMAP [3, 4] or Intel Performance Primitives Library [12].

b) *Portable:* The proposed design retains the portability for cases where the hardware acceleration is either completely unavailable on the platform or the degree of acceleration varies between different platforms. In Figure 6 we show two different paths a function call from the application can follow depending on availability of the hardware acceleration. The Portable Accelerator Interface (PAI) is responsible for scheduling tasks on the hardware accelerator or diverting the call to a software module if the accelerator is not available. In our set-up the discovery, enumeration and initialization of the hardware is performed by various global agents in operating system before the PAI gets to use its resources. The PAI functionality includes support for user mode interfaces for direct access by the applications. This direct path avoids the unnecessary system call and address translation overheads. In case of our prototype implementation, this was achieved in PAI by mapping accelerator memory mapped IO space into the user domain. In case the PAI does not detect an initialized hardware accelerator, it redirects the function execution to a previously initialized software routine run on the core itself. The PAI can be something as simple as a static library or be hidden under bigger architecture frameworks like OMAP or IPP.

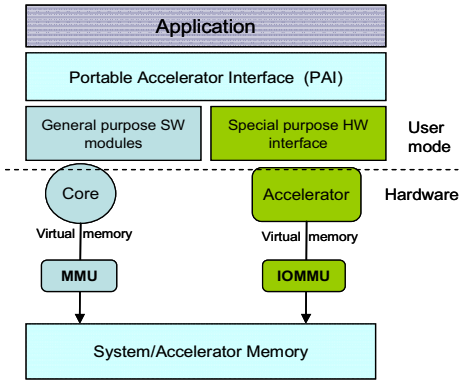


Figure 6: HiPPAI: The Proposed high performance, portable interface architecture

4. MAR Workload

We will use Mobile Augmented Reality (MAR) as a workload to demonstrate the implication of the interface overheads and the benefits of HiPPAI. In the MAR usage scenario, we start with a query image from the camera. The intent is to compare this query image against a set of pre-existing images in a database for a potential match. We focus on still images in this work. The still-image MAR application essentially does the following: (a) acquire/capture the image, (b) recognize objects by computing interest points in the image, (c) match to a pre-established set of images in a database and, (d) display relevant meta-data overlaid on the object in the screen. Our initial analysis shows that the first and last step (image capture and information overlay) are negligible in execution time. Furthermore, for this paper we focus primarily on the image recognition part which is further decomposed into two basic steps

as illustrated in Figure 7: (a) Interest-point detection: identify interest points in the query image and (b) Descriptor generation: create descriptor vectors for these interest points

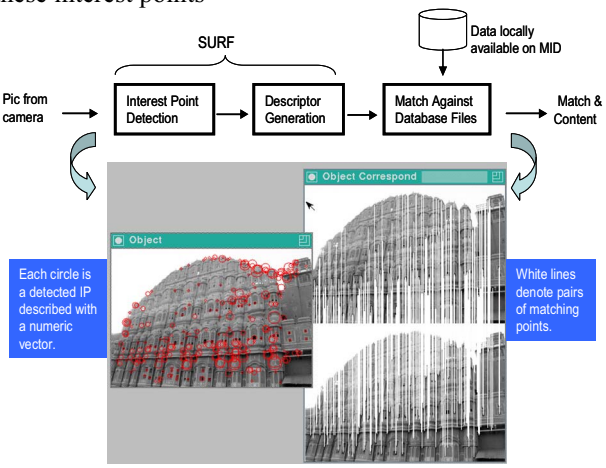


Figure **7:** MAR System Flow

There are several algorithms that have been proposed to detect interest points and generate descriptors. The most popular algorithms amongst these are variants of SIFT (Scale-Invariant Feature Transform)[14] and SURF (Speeded up Robust Features)[15, 16]. We chose the SURF algorithm for our MAR application because it is known to be fast and has sufficient accuracy for the usage model of interest. In addition, previous researchers have also used SURF successfully for mobile phones for MAR. Figure 7 shows an illustration of the use of SURF for detecting interest points and matching against other images in a database. We refer the reader to [15] for a more detailed description. In our set-up we use the FPGA based accelerator to emulate the *Match* and *Fast Hessian Detector* algorithm, which is a computational part of the Interest Point Detection block in Figure 7.

5. Preliminary Evaluation Prototype

In order for us to understand the changes in the software and the hardware requirements, we prototyped the proposed driver model with a virtual memory accelerator. We used the IOMMU capabilities of the platform originally designed to support device virtualization. The following section describes our experiment environment.

We use an Intel Core i7 3.2G processor (Nehalem micro-architecture) and Intel X58 express chipset (Tylersburg) as our experiment platform. It has Intel QuickPath Interconnect (QPI) to connect with the X58 chipset. The Intel X58 express chipset supports Intel Virtualization Technology for Directed I/O (Intel VT-d)[4]. Intel VT-d architecture supports DMA remapping, which is a generalized IOMMU architecture. Figure 8 shows the block diagram for our experiment system. The experiments run on Linux Fedora 8 with the Linux kernel updated to 2.6.28 with a patch to the Intel VT-d code. This patch enables the VT-d IOMMU to be configured with OS page tables rather than the VMM page tables. This allows the address translation from Virtual Address to Physical Address, rather than the translation from Guest Physical Address to Host Physical Address as intended by the virtualization use.

Our prototype implementation employs a PCIe based Altera Stratix II programmable FPGA to emulate the *Fast Hessian Detector* and *Match* accelerators for the MAR workload described earlier. At the time of writing this paper, the FPGA is able to access the image data and write data back into system memory through DMA.

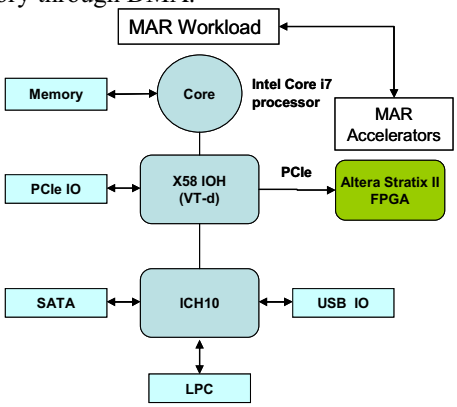


Figure 8: Experiment platform with Intel i7 processor and chipset with IOMMU support

6. Results

The goal of our experiments is to characterize the major overheads associated with the current driver models described in earlier sections. These overhead include system call, data copy and page pinning overhead. Our experiment prototype is already described in Section 5. The experiments are conducted into two parts. First, overhead associated with conventional device driver software interface for buffer management and user-kernel boundary crossing are evaluated. Secondly, the overhead associated with proposed driver and conventional driver is compared in terms kernel API and function invoked. For each experiment, average, minimum and maximum statistics from several invocations are presented.

6.1. Classic Driver Model Overheads

Figure 9 shows the measured overheads for key kernel API calls to process the data to/from the application. An example of a kernel function API is the *copy_to_user* function which is used to copy a buffer in the kernel to the user space. The overhead

associated with its invocation depends on the size of the data buffer transferred. From Figure 9 we can see that the average amount of cycles consumed by the *copy_to_user* function is around 2100 cycles to transfer 4KB of data. The figure also shows the copy overhead to/from kernel for different block sizes. Note that this overhead may vary based on the kernel and process state, prefetching, caching and other O/S optimization mechanisms (e.g. cache buffers). To minimize the variability in the data, we ran each copy test thousands of times and then averaged the data. Additionally, to minimize the cache effects of copying data, we ensured that a new source and destination buffer block is allocated for each copy operation.

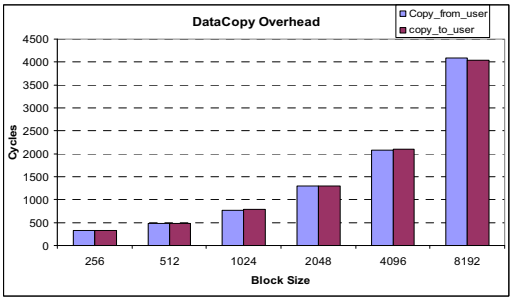


Figure 9: User-Kernel Copy management overheads

Figure 10 shows the overhead associated with key system calls used as a means of communication between user and kernel space. A conventional device driver framework for the accelerators uses these calls multiple times for every operation on the hardware. The results are shown for both one-way (system call into the kernel) and round trip (initiation and completion calls across the user-kernel boundary). We chose these calls for analysis because these are in the data path of an application execution which accesses the hardware accelerators. As an example from Figure 10, we see that an average read call takes about 400 clock cycles to execute on our prototyping system. From Figures 9 and 10 we note that the total overhead for these operations is non-negligible especially with fine grain offloads. Intuitively, if the time taken by the actual function execution in the accelerator is of an order of magnitude larger than the interface overheads, one can justify the cost involved in using the conventional device driver model. But this limits the offload opportunity considerably.

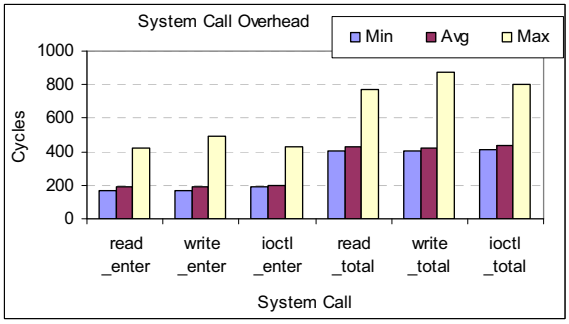


Figure 10: System call overheads

6.2. Classic vs. HiPPAI Overheads

We compare the total overheads of our programming model with that of a conventional device driver in this sub-section.

Functions in conventional data path of Accelerator	Classical Driver	HiPPAI Model Driver
Memory Allocation	Two buffers are allocated for source and destination	Two buffers are allocated for source and destination
Memory Pinning	N/A	User buffer is pinned to kernel space
DMA Allocation	Needed for allocating kernel buffer to copy user data	N/A
Memory Mapping	N/A	Table Descriptor and Read/Write header memory
Data Copy (User->Kernel)	Copy Application buffer in user space to Kernel space	N/A
Data Copy (Kernel->User)	Copy Application buffer in user space to Kernel space	N/A
DMA Read	Initiates DMA from kernel to FPGA Hardware	Initiates DMA from kernel to FPGA Hardware
DMA Write	Initiates DMA from FPGA Hardware to Kernel buffer	Initiates DMA from FPGA Hardware to Kernel buffer
Kernel Entry/Exit and other kernel overheads	Two Kernel Entry and Two Exits	N/A

Figure 11: Data path of classic driver and proposed interface. Shaded area indicates the onetime setup cost

Figure 11 gives the overall call path during the invocation of DMA read and write calls. Note that shaded area indicates the cost associated to one time setup cost for the DMA transactions whereas other calls indicate the overhead during the data transfer and processing of application workload. This means that a classic driver will incur the overhead of kernel entry and exit along with data copy every time a buffer is being shared between the user level application and the accelerator. These most frequent overheads in the data path are removed in our proposed HiPPAI model. Figure 12 gives the comparison of different overheads for classical and HiPPAI model for setup and DMA execution. The x-axis gives the DMA operation whereas the y axis indicates the cycles consumed. Note that even though the set-up overhead is more in case of the proposed HiPPAI driver, the frequent data-path calls (DMA read and write) have lesser overhead.

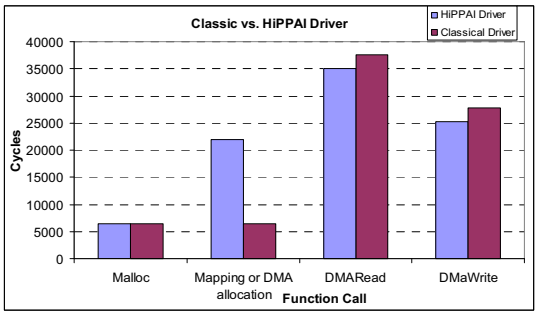


Figure 12: Overhead Comparison between classic driver and HiPPAI model for 4KB data transfer

Figure 13 provides an overhead breakdown of user-level DMA read and write. To evaluate the read/write overhead, we first flush the system cache with dummy writes forcing the workload data to be read from memory every time (instead of reading from the cache).

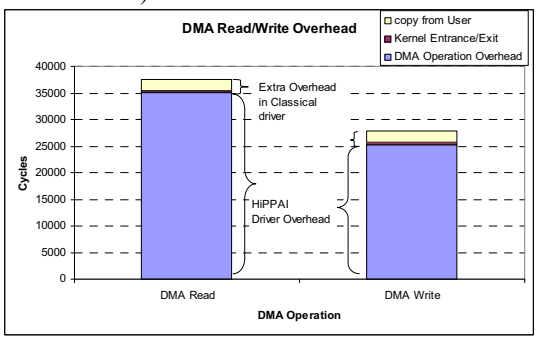
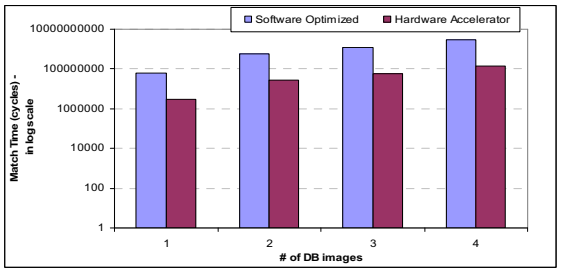


Figure 13: Classic driver vs. HiPPAI model DMA read/write overhead

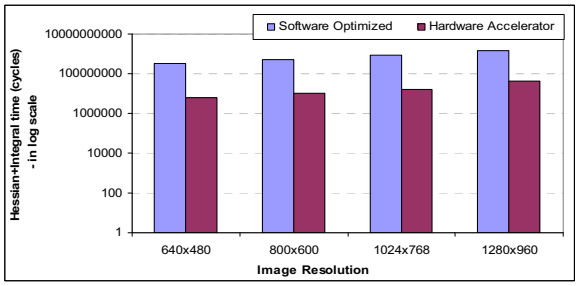
For the base value, we measured the overhead associated with HiPPAI for DMA read and DMA write using a block size of 4K bytes. For this case, DMA read takes an average of 35K cycles whereas DMA write takes 25K cycles. Note that DMA read and write overhead is significantly different because the DMA read call must transfer the data from system memory to the FPGA whereas the DMA write is posted to the chipset which may perform the actual write to system memory at a later time.

6.3. Benefits of HiPPAI on Fast Hessian offloads

We executed the Match and Hessian + Integral Image calculation algorithms of MAR on an optimized software stack as well as on our FPGA accelerator. Figure 14 shows the execution times in cycles, on a logarithmic scale, for these MAR functions. We observe from the figure that there is an appreciable reduction of total execution times when the MAR functions are accelerated in hardware. The actual measured speedup for the match algorithm was close to 20X in situations where a given image is matched against a database of 1 to 50 images. Similarly, the maximum speedup possible with Hessian Detect + Integral Image calculations was close to 14X.



(a) Match Algorithm execution time



(b) Fast Hessian Detect execution time

Figure 14: MAR hardware acceleration

Further, to estimate the performance gains obtained through HiPPAI, we analytically compare the execution times of the direct device model with the HiPPAI model using the experimental data gathered from our prototype. For example, from the data used to plot Figure 14(b) we observe that it takes 3.2M cycles to do fast hessian calculations for image size of 640x480 bytes. For this image size, a total of approximately 307.2 KB of data have to be transferred to the FPGA and 1.2 MB of data has to be transferred from the FPGA. For a classic device driver, this operation involves 2 memory allocations (*malloc*), 2 DMA buffer allocations in OS kernel, 2 copies from user/kernel space, and the overheads for actual DMA Read and DMA Write. We show the comparative overhead added by the classic driver and the HiPPAI to this execution time in figure 15. From Figure 15 we observer that HiPPAI achieves an overhead reduction of approximately 1.6M (1.6M to 0) cycles. The percentage overhead is dependent on the offloaded function granularity. For the Fast Hessian Accelerator which takes 3.2M cycles this reduction translates to 12% less overhead (12.3% to 0% going from classic to HiPPAI). Note that the working set size (size of the input image size) and the accelerator compute time impact the overall overhead considerably. Note that we have not considered the overhead associated with FPGA initialization cost for this analysis since this is a one time cost.

Function in Data Path	Classic Device Driver	HiPPAI Driver
Kernel Buffer Allocation	978750	0
Data Copy (user-> kernel) +Kernel Entry/Exit	124983	0
Data Copy (kernel -> user) +Kernel Entry/Exit	509833	0
Total Interface Overehad Cycles (O)	1613566	0
a: DMA Read	2628450	
b: DMA Write	7579800	
c: Fast Hessian Function	3200000	
Time in accelerator (T = a+b+c)	13408250	
Overhead for Hessian Accelerator (O/T)	12.03%	0

Figure 15: Data path execution cycles for Fast Hessian Accelerator

The data presented in this section shows the efficacy of the HiPPAI in interfacing fine grain accelerators in SoCs. Very low overhead interfaces provided through HiPPAI allow small and medium size functions to be offloaded to accelerators. Further, a portable interface specification using a virtual memory based addressing model and user mode access provides better programming flexibility. It also allows complex data structures to be shared between application and accelerators. These functional benefits combined with the performance advantage demonstrated in this paper make HiPPAI apt for future SoC accelerator interfacing.

7. Summary and Future Work

In this paper we presented the challenges associated with using classical drivers for hardware accelerators on SoC platforms. We showed that these drivers come with a high performance cost due to multiple system calls and address translations. To avoid these costs we proposed a new HiPPAI Virtual Memory based accelerator interface that allows programming of accelerator functions from the user space. To enable this, HiPPAI utilizes a hardware IO-MMU for translating virtual addresses to physical addresses. Such a model allows the user application and the hardware accelerator to operate in the same addressing domain. As such, the HiPPAI model avoids the need for system boundary crossings through syscalls and the need for address translations between the user and kernel domains. Further, this also allows complex data structure sharing between the application and the accelerators and enables function portability, decoupling application design from the degree of hardware function acceleration availability on any target platform. Our programming model presented in this paper incorporates a portable accelerator interface that decouples the function implementation from the application making it possible to migrate between SoCs with varying degree of accelerator support.

The future work in this area includes completing the FPGA emulator for offloading all of the MAR algorithm or selective building blocks depending on the experiment requirements. We are also working to finalize the interface specification that addresses performance and portability in the general case regardless of specific functions being offloaded.

References

[1] Intel® QuickAssist Technology Accelerator Abstraction Layer – http://download.intel.com/technology/platforms/quickassist/quickassist_aal_whitepaper.pdf

[2] AMD's Holistic Vision for the Future: Torrenza Enables Platform Innovation http://www.instat.com/promos/07/dl/IN0703889WHT_DahenUd9.pdf

[3] OMAP35x Applications Processors - Product Bulletin. http://focus.ti.com/lit/ml/sprt457a/sprt457a.pdf

[4] OMAP™ 3 family of multimedia applications processors http://focus.ti.com/pdfs/wtbu/ti_omap3family.pdf

[5] D. Abramson, et al., "Intel® Virtualization Technology for Directed IO", Intel Technology Journal: http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art02.pdf

[6] Y. A. Du, O.Tickoo et al, "High Performance Accelerator Interfacing for Emerging SoC Platforms." IOM-2009.

[7] AMD and HP: Protocol Enhancements for Tightly Coupled Accelerators, http://www.hp.com/techservers/hpccn/hpccollaboration/ADCatalyst/downloads/AMD_HPTCAWP.pdf

[8] T. Blank, "A survey of Hardware Accelerators Used in Computer-Aided Design," IEEE Design and Test of Computers, August 1984

[9] W. J. Dally and R. E. Bryant, "A Hardware Architecture for Switch-Level Simulation," IEEE Transactions on Computer Aided Design, Vol. CAD-4, No. 3, July 1985.

[10] G. Catlin and W. Paseman, "Hardware Acceleration of Logic Simulation Using Data Flow Architecture," Proceedings of the 1985 IEEE International Conference on Computer Design. Santa Clara, CA, November 1985

[11] "Silicon Solutions Carves Niche in VLSI Design," Electronics, August 12, 1985.

[12] Intel Performance Primitives, http://software.intel.com/en-us/intel-ipp/

[13] OpenCV, http://sourceforge.net/projects/opencvlibrary

[14] D.G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints." International Journal of Computer Vision, 2004.

[15] H. Bay, T. Tuytelaars, and L. Van Gool, :SURF: Speeded up robust features." ECCV, 2006.

[16] OpenSURF, http://code.google.com/p/opensurf1

[17] Intel Quick Assist Technology, http://www.intel.com/technology/platforms/quickassist/index.htm

[18] T. Rintaluoma, O. Silven, and J. Raekallio, "Interface Overheads in Embedded Multimedia Software," Lecture Notes in Computer Science, Springer-Verlag, 2006

[19] F. E. Powers. Jr., G. Alaghband, "Introducing the Hydra Parallel Programming System", Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures, Cambridge MA USA 2006.

[20] H, Wong, A. Bracy etal, "Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor", Proceedings of the 17th international conference on Parallel architectures and compilation techniques, Toronto Canada, 2008