

# A Proposal for Efficient CPU-Accelerator Handshake

Gordon T Davis\* Anil Krishna Srinivasan Ramani Jeff H Derby Ken V Vu

IBM Corporation, Research Triangle Park, North Carolina, USA  
gordontdavis@yahoo.com, {krishnaa, sramani, jhderby, kenvu}@us.ibm.com

## ABSTRACT

The use of hardware accelerators to improve performance has been steadily gaining traction. Several processors have started incorporating on-chip accelerators, and in some cases, systems have been augmented with I/O attached accelerators. Three of the most important challenges to overcome in order to efficiently integrate main processor cores and accelerators are – mechanism used to initiate work on the accelerators, passing addresses of parameters, and dealing with page boundary crossings. This paper proposes efficient methods to address each of these challenges. Each method uses instructions already in a processor’s instruction set, thereby reducing hardware cost associated with new instructions and minimizing tool chain changes required. The methods discussed also obviate the need for a memory management unit in each accelerator, thus reducing hardware costs even further.

## 1. INTRODUCTION

In order to increase system performance beyond what can be achieved using conventional processor architectures, some new designs are migrating toward the use of heterogeneous computing systems, sometimes referred to as hybrid computing systems. These systems consist of coprocessors or hardware accelerators [1], in addition to general-purpose processors. In addition to performance gains due to offloading of work to these specialized auxiliary units, system performance may also benefit due to reduced latency for completion of these accelerated tasks. In broad terms, acceleration can be classified as – (a) provided by a functional unit inside the processor core and accessible via instruction set architecture (ISA) extensions [2], (b) bus-attached accelerators, and (c) I/O-attached accelerators, such as those

attachable through a PCIe [3] interface. The focus of this paper is on bus-attached accelerators, although similar notions can be applied to I/O-attached accelerators. In the rest of the paper, the term accelerator refers to bus attached accelerators.

Bus-attached accelerators include hardware coprocessors that perform standard or programmable functions such as cryptography, compression/decompression, regular expression evaluation, XML parsing, Ethernet packet classification, to name a few. Hardware accelerators could also be potentially embedded in existing bus-attached hardware units, controllers, or interfaces, to perform some routine operations in an accelerated manner rather than entirely in software. For example, accelerators can be embedded within memory controllers to perform functions such as block copy, block move, block init, standard string operations, and so on. Latency can be very significant if a task driven by code executing in the processor core must process a chain of pointers in memory. That process works more efficiently if driven by an accelerator in the memory nest (potentially close to the memory controller). In general, hardware accelerators which need to work with data in memory should be located in or near the memory controller unit for optimum operational performance. The concepts presented in this paper apply to these embedded accelerators also.

Three of the most important challenges to overcome in order to efficiently integrate main processor cores and accelerators are – mechanism used to initiate work on the accelerators, passing addresses of parameters, and dealing with page boundary crossings. Each challenge is briefly described below.

Typical interfaces to accelerators that are remote (e.g. separated by a bus) from the main processor

---

\* Retired from IBM Corporation

may involve memory mapped I/O to control accelerator function. In these systems, usually, results must be polled to determine when a hardware function completes, or alternatively, an interrupt is used to signal completion to the main processor. Polling for results wastes main processor cycles and consumes bus bandwidth. The interrupt scheme requires hardware enhancements to generate and route the interrupt signal. Alternately, some designs have added special instructions to the instruction set of the main processor that control accelerator function. These special instructions require a hardware enhancement to implement the new instruction, and additional support in the tool chain to facilitate use.

Either mechanism above needs to convey the acceleration function needed, as well as the addresses of input and output parameters that should be used. In order to access these parameters resident in main memory, the accelerators need to know the real (physical) addresses of these parameters. A couple of approaches are possible. The accelerators can have a memory management unit (MMU) embedded in them, or, the main core can pass real addresses to the accelerators it invokes. If an MMU is present in each accelerator, on a TLB miss or hardware page table walk failure, either the operating system or a hypervisor layer needs to be interrupted to resolve the translation miss. This interrupt handler is expensive, and creates a bottleneck, especially when there are multiple accelerators and several threads in the system. The MMU also needs to continually snoop translation invalidates on the chip bus and act accordingly. The very presence of an MMU also increases hardware cost. On the other hand, if real addresses are passed to accelerators, the accelerator function can proceed till access to a location outside the currently known input or output parameter page is needed, or an address invalidation command to an input or output parameter page is received on the bus.

The next challenge arises when dealing with page crossings. While performing a function, an accelerator may need to access an address beyond the page it started with. If an MMU is embedded within the accelerator, it can provide this address translation. A page table walk or an expensive page fault interrupt to the OS or hypervisor may be required in some cases. If an MMU is not present, the main core needs to be notified that a new translation is required. Once

the main core provides this, the accelerator can resume.

In this paper, we propose efficient ways to address each of the challenges identified above. Section 2 proposes a scheme to initiate an accelerator function using instructions in a processor core's existing instruction set. Section 3 describes a method to give real addresses of parameters to an accelerator that obviates the need for an embedded MMU in the accelerator. Section 4 details a scheme to handle page boundary crossings during accelerator operation, along with some example scenarios for illustration. Section 5 concludes the paper.

## 2. INITIATING A FUNCTION

This section proposes an efficient method of interfacing to hardware accelerators that uses standard instructions in the existing instruction set. Hence, instructions used to initiate accelerator functions would already be supported by existing tool chains, making it easier for existing application environments to be migrated to a new system that contains accelerators.

In the proposed scheme, a "touch with intent to modify" instruction (similar to *dcbtst* in the PowerPC architecture [4]) is used to initiate an accelerator function. Without loss of generality, we use *dcbtst* to refer to this instruction in the descriptions that follow. A *dcbtst* used to initiate an accelerator function would be differentiated from other uses of the same instruction by a storage attribute of the page that the touched address maps to in the page table. Addresses generated for these special *dcbtst* instructions map to a page designated to control accelerator functions. A hit in this page triggers the hardware accelerator function corresponding to that address to start, as well as identifying the address where the return data (i.e., return code from the accelerator) should be placed. Since the address generated by a *dcbtst* instruction serves to identify the hardware accelerator to be used, system software pre-assigns the real (physical) addresses to be used to initiate functions on a given hardware accelerator on a per software process basis. Each software thread is thus capable of initiating accelerator functions on each accelerator. Note that before initiating an accelerator function, pointers (also referred to as addresses in this paper) to input and output data streams or regions are sent to the accelerator.

Section 3 proposes an efficient method to pass real addresses of the input and output parameters to an accelerator.

One mechanism to implement the triggering of an accelerator based on a *dcbtst* would be for the MMU to send signals to a downstream unit, for example, the bus interface unit, to assemble and send out the actual bus command to initiate the corresponding accelerator. The MMU would do this for *dcbtst* hits in the page reserved for initiating accelerator functions, as described above.

Since these special *dcbtst* instructions are intended to initiate accelerator functions and not a "touch" function, a hit in a page containing these addresses also disables standard functionality of the *dcbtst* (i.e., it does not go to memory for a prefetch operation). Also, the location for the return data must be invalidated automatically in response to execution of the *dcbtst*, if that location exists locally in the processor core. Subsequently, executing a standard load instruction to that address will stall until the requested data is available.

Details of how software drives these accelerators can be hidden from application code, which can issue calls to standard functions like "block copy", "block move", "block init", standard string ops, OpenSSL [5], zlib [6] and many others. The standard library of functions is replaced by a library of equivalent functions that use the concepts proposed here to interface to these accelerators. Once one of these functions is called by the application, a sequence of steps is executed by the hardware-enabled function as follows:

1. One or more parameters may be passed from the application to the system service function, according to standard interface definitions.
2. Pass these parameters to hardware accelerator via memory mapped I/O (or potentially use an enhanced method described in Section 3).
3. Initiate hardware accelerator function via executing a *dcbtst* instruction with address set according to the desired function.
4. Potentially do other useful operations before stalling thread.
5. Execute a load instruction to the same address used in the *dcbtst*. This will cause

the thread to stall until the function is complete.

6. Once the load gets the requested data and the thread continues, return to the calling application with the return data provided by the accelerator hardware.

For cases where the return code is not required (or at least not required immediately), the function could return to the calling application without checking for return data. A separate function could be executed later by the calling application to validate the return code. This would require changes to user code, but would facilitate overlap of other operations on the same thread with completion of the accelerator function.

### 3. TRANSFERRING POINTERS

In most cases, one or more pointers must be passed to an accelerator to identify the source data. Another pointer may be required to identify where the processed data should be placed. There may be more than one input stream or region, and likewise, more than one output stream or region that the accelerator needs to be pointed to. For example, a decompression function requires one pointer to the existing data, and a second pointer to identify where the decompressed data should be placed. The controlling processor core works with effective addresses, but if effective addresses are passed to a remote accelerator, a memory management unit will be required as part of the accelerator to enable it to address real memory, adding significant complexity and cost. Real addresses are sometimes passed to accelerators, but this adds complexity to the calling program to do a software translation from effective to real addresses.

What is needed is a more efficient method of passing pointers to hardware accelerators, allowing the processor to deal only with effective addresses, while passing real (physical) addresses to its accelerators.

This section proposes a more efficient method of passing pointers to accelerators. A "data cache block touch" instruction (similar to *dcbt* in the PowerPC architecture) is used to point to source data. Without loss of generality, we use *dcbt* to refer to this instruction in the descriptions that follow. Depending on the type of accelerator, a

couple of variations to the proposed scheme are possible.

In one variation, a *dcbt* is differentiated from other uses of the same instruction by the way the address touched is mapped in the page table. This variation is suitable for accelerators that are used to offload functions such as block copy, block move, block initialization, and string operations. In one approach, the effective address space is duplicated, with one half of the address space being used to support this function. One most-significant (MS) bit of the address indicates whether this is a normal *dcbt* or a special *dcbt* (according to the method proposed here). Both spaces map into the same page in the page table to avoid excessive requirements for TLB entries. If this bit is set, the touch is sent as usual to the memory subsystem, but it is tagged to not be returned to the cache. Instead, it is held in the hardware accelerator, waiting for a subsequent initiation of a compatible hardware function by the same thread. One method for initiating these hardware accelerator functions was described in Section 2. The accelerator will use a programmed block size to determine how much data to consume, starting at the address pointed to by the *dcbt*. An alternative approach to using a MS address bit to distinguish a special *dcbt* from a normal *dcbt* would be to use an alternate instruction encoding for the special *dcbt*.

In another variation, a *dcbt* used to pass pointers to an accelerator would be differentiated from other uses of the same instruction by storage attributes of the page that the touched address maps to in the page table. This variation is suitable for accelerators such as cryptography, compression/decompression, regular expression, etc. Addresses generated for these special *dcbt* instructions map to a page designated to be used for input parameters for accelerator functions. Calling programs coordinate with system software to predetermine the pages to be used for input parameters and to set the storage attributes for these pages to indicate the accelerator each page is meant for. A hit in this page triggers the

address generated by the *dcbt* to be sent to the hardware accelerator corresponding to that page. More than one *dcbt* will be used if there is more than one input pointer to be passed to the accelerator. Subsequently, the calling application initiates the accelerator function using a method such as the one described in Section 2. If an incorrect number of arguments are received before an accelerator function is initiated, one approach to deal with this is to make the accelerator send an exception return code back to the calling routine. Note that in this variation, the process id of the process utilizing the accelerator function should be also passed to the accelerator along with the addresses so that the accelerator can match the input/output pointers to the correct accelerator initiation command.

Likewise, a special "data cache block zero" instruction (similar to *dcbz* in the PowerPC architecture) is used to identify the starting location for the output of the associated accelerator function. If the MS address bit (or instruction encoding) indicates a special *dcbz*, the address is passed to the hardware accelerator. Here too, the address generated by the *dcbz* is used to pass the output pointer to the correct accelerator, instead of allocating and zeroing out the associated cache line - the default behavior of *dcbz*. The hardware accelerator will subsequently use this address as the starting address to hold the output stream generated by the accelerator function. In cases where the target address is not aligned on a standard boundary of an isolated memory access, the first block in that space may be prefetched into the accelerator to facilitate a read-modify-write operation in order to preserve the part of that memory block that is not targeted by the function.

If the hardware accelerator needs multiple input streams, or multiple output streams, or both, multiple *dcbt* and/or *dcbz* instructions will be issued by the library function managing the setting up of pointers for the hardware accelerator.

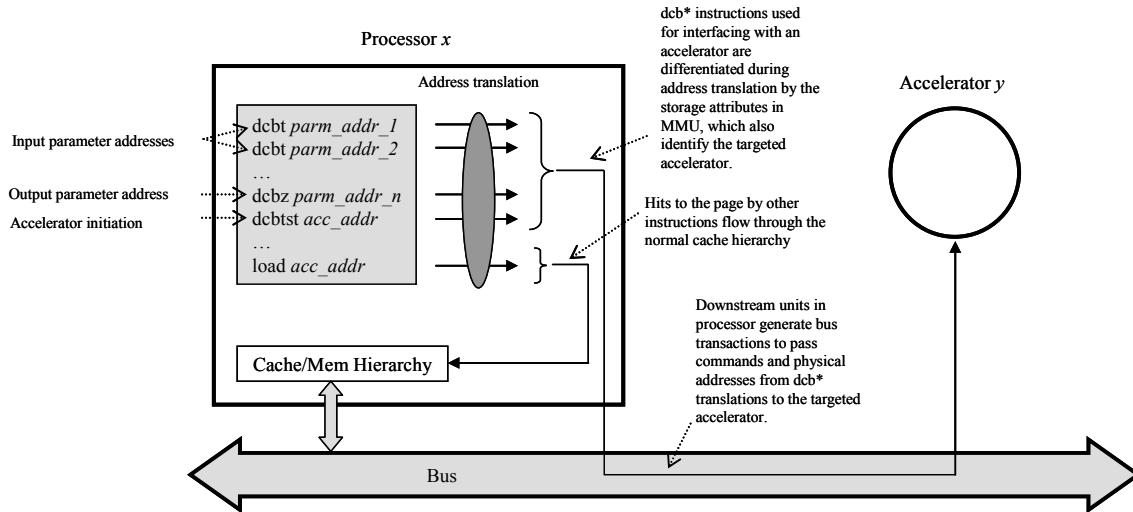


Figure 1. Transferring pointers and initiating an accelerator function.

Figure 1 illustrates the schemes proposed in Sections 2 and 3 to initiate an accelerator after transferring input and output parameter addresses.

#### 4. PAGE BOUNDARY CROSSINGS

As indicated in the previous sections, addresses may be translated from effective addresses to real addresses by the MMU (Effective to Real Address Translation units (ERATs), or Translation Lookaside Buffers (TLBs), or both) before they are passed to a remote accelerator. However, if the remote accelerator encounters a memory page boundary while it is performing a function, the hardware accelerator must be able to locate where the new page resides in physical (real) memory and validate the new page attributes to insure that operations do not violate access rules. Note that even though the current and new memory pages may be contiguous in effective address space, they may not be in real address space. This would seem to force a remote accelerator to operate on effective addresses, requiring it to duplicate at least part of the MMU structure that the main processor is using. In addition, the duplication of address translation resources results in overheads of keeping these structures coherent.

What is needed is a more efficient method of managing page boundary crossings in these hardware accelerators that will enable the accelerators to operate on real addresses and avoid having to duplicate the MMU structure of the processor.

This section describes a more efficient method of handling memory page crossings encountered by these accelerator functions. The key to enabling a hardware accelerator function to work with real addresses is to make the hardware accelerator aware of the page size when this address is passed to it. The size of the page the hardware accelerator is working with is necessary for the accelerator to identify the physical address bounds within which it can operate without needing any address translation services. The page size field can be extracted from the page table entry when it is accessed for translation in the main processor core and passed to the accelerator via a separate physical connection, or potentially via a unique bus transaction on the main bus.

An accelerator monitors pointers used for memory access, looking for a page boundary crossing. Page crossings must be monitored separately for each associated input and output streams, unless restrictions are placed to force all streams to use equal sized pages. Given knowledge of page size, the boundary crossing can be detected via a simple logic circuit when dealing with physical addresses. The accelerator terminates processing once a page crossing is detected, and returns a special code that indicates this. The subsections below discuss two possible scenarios.

##### 4.1 Next physical page is deterministic

The return code identifies the stream or region associated with the offloaded acceleration function. If the calling application or library

thread can compute the next page that the hardware accelerator should work on, then the return code from the accelerator does not need to identify the address that caused the page boundary crossing. This scenario may happen, for example, if the accelerator is working on compressing a chunk of memory that is contiguous in the effective address space. To detect page boundary crossings, the accelerator compares the physical page number of the current page with the physical page number of the next address being accessed. When they

differ, a page boundary has been crossed. When this occurs, the accelerator function stops processing until new input and/or output parameter addresses are provided. The application or library thread remembers which effective page the accelerator is working on and supplies the physical address of the start of the next effective page to the accelerator. This scenario is shown in Figure 2.

#### 4.2 Next physical page is non-deterministic

The return code may be used to communicate the effective address for which the hardware accelerator is requesting an effective to physical address translation. This effective address need only be passed back to the invoking application or library if it is not expected to be able to determine the effective page the accelerator intends to work on next. This is the case, for example, when the hardware accelerator deals with effective address (such as in a pointer chasing application where the pointer is an effective address that is part of a data structure). Here, page boundary crossing detection is a little more complicated. The hardware accelerator reads a field in a data structure using a programmed offset to determine the effective address of the next data structure. This is the “pointer” in the pointer chasing function. In this case, the accelerator must use its knowledge of the effective addresses of the page boundaries and convert the effective addresses of the pointers it uncovers during the pointer chasing to firstly ascertain that they fall within the same page, and if they do fall within the same page, to calculate the corresponding physical address of the pointer. It can then fetch the next data structure from memory. If the accelerator determines that the effective address of a pointer falls outside the range of the effective page, then it can simply send that effective address in the return code.

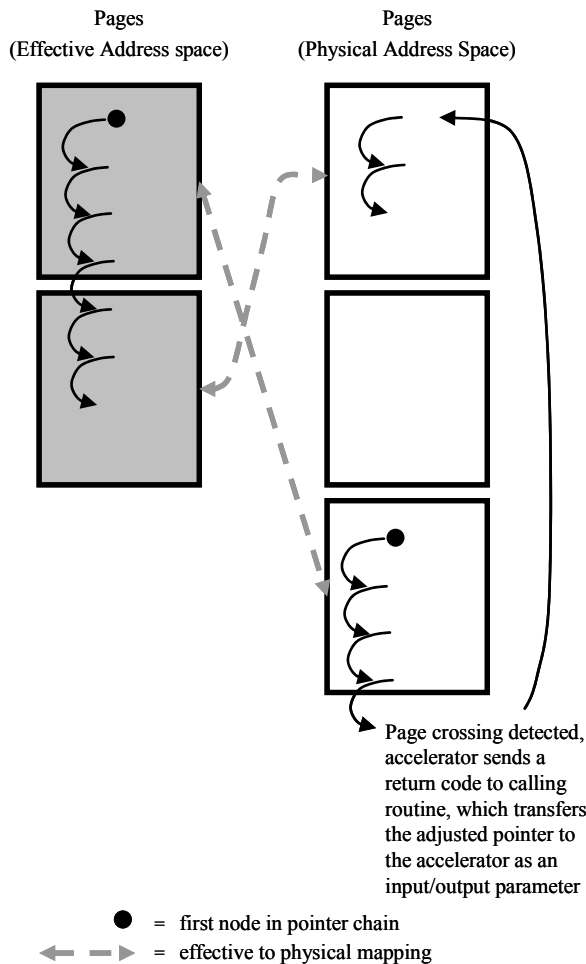


Figure 2. Handling page boundary crossing when the next physical page is deterministic.

Note that when working on the first node in the pointer chain, the accelerator does not yet know if the effective address of the next data structure resides in the same physical page as the one it is currently accessing (the page where the data structure holding the “next” pointer resides). Therefore, to be safe, it sends back a return code to the calling application or library routine with the effective address of the next pointer. The calling routine sends back the physical address of the next pointer. Comparing the physical page of the original pointer to the physical page of the next pointer, the accelerator can know if the next

pointer is still on the same physical page. More importantly, because the accelerator knows the effective address of the next pointer, it discovers the effective page number corresponding to the physical page number it is working on currently. For all subsequent next pointers unearthed, the accelerator can easily identify if they reside on the same physical page or not. On subsequent page boundary crossings, the accelerator sends back a return code containing the effective address to be translated to a physical address. By comparing the effective page number of this address with the effective page numbers of next pointers encountered in the new page, the accelerator can detect further page boundary crossing. This scenario is illustrated in Figure 3.

In either of the scenarios described in the above two subsections, the calling routine tests the return code. This calling routine could be a library that handles communication with accelerators transparently for applications. If the return code indicates successful completion, the calling routine proceeds with the next operations in sequence. If the return code does not indicate successful completion, the calling routine adjusts the original pointer to source data based on the translation implicitly or explicitly requested, and transfers the adjusted pointer to the accelerator as an input or output parameter (using a scheme such as in Section 3). If the accelerator requests only one translation at a time, no identifiers are needed in the return code or response back from

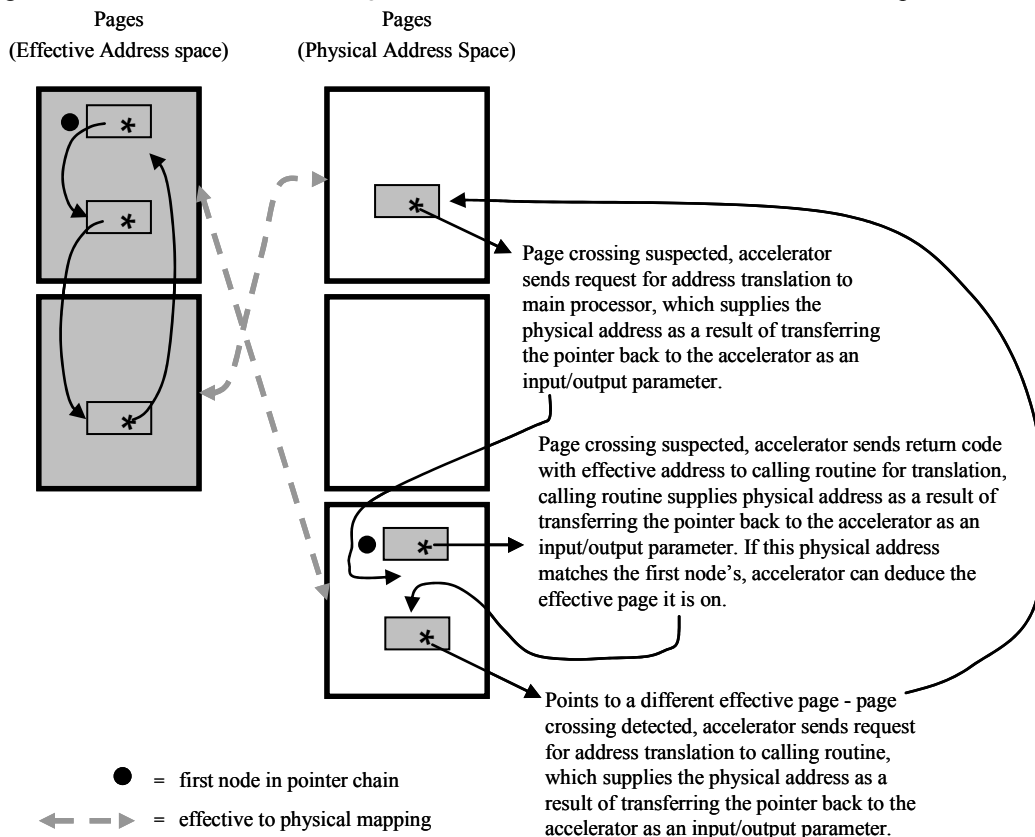


Figure 3. Handling page boundary crossing when the next physical page is non-deterministic.

the calling routine to identify the particular parameter or accelerator function instance uniquely. This process repeats until the requested offloaded function has been completed.

Optionally, a small hardware state machine could be implemented close to the main processor to monitor return codes from accelerators and iterate as described above, but without involvement of the processor. The return code would only be allowed to propagate into the processor core when that return code indicates completion of the requested operation.

## 5. CONCLUSIONS

This paper proposed efficient methods to handle three of the most important challenges to overcome in order to efficiently integrate main processor cores and accelerators – mechanism used to initiate work on accelerators, passing addresses of parameters, and dealing with page boundary crossings. Each proposed method uses instructions already in the processor’s instruction set, thereby reducing hardware cost associated with new instructions and minimizing tool chain changes required. The proposed schemes to initiate accelerator functions and pass pointers to accelerators require enhancements to the storage attributes in page tables, so that pages used for these accelerator related operations can be identified and instruction behavior can be modified accordingly. The scheme proposed for dealing with page boundary crossings eliminates the need for an MMU embedded in each accelerator, thus reducing hardware cost and overhead associated with keeping all the MMUs synchronized. The paper also proposed that in addition to implementing common algorithms such as cryptography in accelerators, common operations can also be implemented as hardware accelerator functions in controllers and interfaces (for example, block operations in memory controllers). In all of the scenarios, system libraries can be adapted to make use of the efficient methods proposed in this paper, completely transparent to software that uses these libraries.

## 6. REFERENCES

1. Sanjay Patel and Wen-Mei W. Hwu, “Accelerator Architectures”, IEEE Micro, July-August 2008.

2. <http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm>
3. <http://www.pcisig.com/specifications/pciexpress/>
4. <http://www.power.org>
5. <http://www.openssl.org>
6. <http://www.zlib.net/>