Targeting heterogeneous SoCs using MCAPI

Peng Sun, Sunita Chandrasekaran and Barbara Chapman
Department of Computer Science, University of Houston, Houston, TX, 77004, USA
{psun5,sunita,chapman}@cs.uh.edu

***Abstract*—Programming emerging complex embedded systems is a challenge. Embedded applications are complicated enough, hence demanding code reuse and easy adoption. Unfortunately existing software solutions expect programmers to handle most of the low-level details giving rise to a plethora of non-portable proprietary commercial solutions. The need to have industry-standards is becoming more and more critical. The Multicore Association (MCA) offers industry-driven standard-based approaches that provide portable and scalable solutions. In this paper, we use Multicore Communication API (MCAPI), one of the popular APIs used in the embedded industry enabling inter-core communication and synchronization. We have extended the reference MCAPI implementation for a Freescale QorlQ P4080 multicore platform consisting of eight e500mc Power Architecture^{TM} and specialized accelerators such as Pattern Match Engine (PME) and Security Engine (SEC) integrated with Data Path Acceleration Accelerators (DPAA). We establish communication with PME from power cores, using MCAPI, thus abstracting all low-level configurations and function calls.**

1. Introduction

Heterogeneous architectures can be typically referred to an architecture that consists of a variety of different types of computation units. In embedded systems, the computation units could be a general purpose processor (i.e. *CPU*, *ARM* or *Power*), a special-purpose processor (i.e. *DSP* or *GPU*), or even a hardware accelerator (i.e. *FPGA* or *Pattern Matching Engine*).

In this project, our goal is to design and create a portable programming paradigm for heterogeneous embedded systems. We plan to leverage the recently ratified accelerator features of the de facto shared memory programming model OpenMP that may serve as a vehicle for productive programming of heterogeneous embedded systems. To begin with, we have studied the industry standards API i.e. MulitCore Association (MCA) API, that specifies essential application-level semantics for communication, synchronization, resource management and task management capabilities among several others.

For the task accomplished in this paper, we have explored the multicore communication APIs (MCAPI) that is designed to capture basic communication and synchronization required for closely distributed embedded systems. We have considered Freescale QorlQ P4080 as the target and evaluation platform for this work. We identified the main challenge which is to establish communication mechanisms between the P4080 multicore processor and the Data Path Acceleration Architecture (DPAA) and *Security Engine*(SEC 4.0) and *Pattern Matching Engine*(PME) accelerators. The organization of the paper: 2 discusses MCAPI in detail, Section 3 gives an overview of the target platform we used for this work, Section 4 elaborates the implementation details and Section 6 concludes the paper and also discusses the future work.

2. MCAPI

The purpose of MCAPI, which is a message-passing API, is to capture the basic elements of communication and synchronization that are required for closely distributed embedded systems. MCAPI provides a limited number of calls with sufficient communication functionalities while keeping it simple enough to allow efficient implementations. Additional functionality can be layered on top of the API set. The calls are exemplifying functionalities and are not mapped to any particular existing implementation.

MCAPI defines three types of communication:

- *Messages*, connectionless diagrams
- *Package Channel*, connection oriented, uni-directional, FIFO package streams
- *Scalar Channel*, connection oriented, single word uni-directional, FIFO package streams

MCAPI *messages* provide a flexible method to transmit data between endpoints without first establishing a connection. The buffers on both sender and receiver sides must be provided by the user application. MCAPI messages may be sent with different priorities. MCAPI *packet channels* provide a method to transmit data between endpoints by first establishing a connection, thus potentially removing the message header and route discovery overhead. Packet channels are unidirectional and deliver data in a FIFO (first in first out) manner. The buffers are provided by the MCAPI implementation on the receive side, and by the user application on the send side. MCAPI *scalar channels* provide a method to transmit scalars very efficiently between endpoints by first establishing a connection. Like packet channels, scalar channels are unidirectional and deliver data in a FIFO (first in first out) manner. The scalar functions come in 8-bit, 16-bit, 32-bit and 64-bit variants. The scalar receives must be of the same size as the scalar sends. A mismatch in size results in an error.

We use the reference implementation provided by MCA and extend it further to cater to the power processors and its accelerators.

3. Target Platform

The P4080 development system is a high-performance computing, evaluation, and development platform supporting the P4080 power architecture processor. Figure 1 shows the preliminary block diagram of P4080.

A. P4080 Processor

The P4080 processor is based upon the *e500mc* core built on Power Architecture and offering speeds at 1200-1500 MHz.

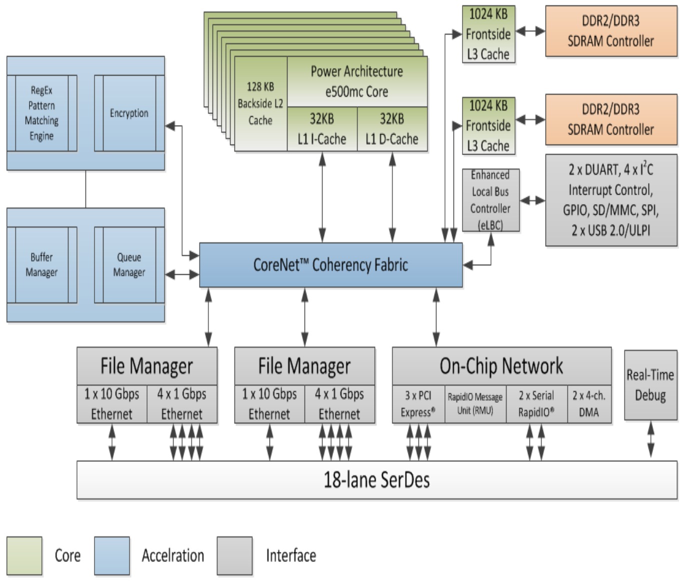
RegEx Pattern Matching Engine
Encryption
Buffer Manager
Queue Manager
128 KB Backside L2 Cache
Power Architecture e500mc Core
32KB L1 I-Cache
32KB L1 D-Cache
1024 KB Frontside L3 Cache
DDR2/DDR3 SDRAM Controller
1024 KB Frontside L3 Cache
DDR2/DDR3 SDRAM Controller
Enhanced Local Bus Controller (eLBC)
2 x DUART, 4 x I2C Interrupt Control, GPIO, SD/MMC, SPI, 2 x USB 2.0/ULPI
CoreNet™ Coherency Fabric
File Manager
1 x 10 Gbps Ethernet
4 x 1 Gbps Ethernet
File Manager
1 x 10 Gbps Ethernet
4 x 1 Gbps Ethernet
On-Chip Network
3 x PCI Express®
RapidIO Message Unit (RMU)
2 x Serial RapidIO®
2 x 4-ch. DMA
Real-Time Debug
18-lane SerDes
Core
Accelration
Interface

Fig. 1: P4080 Block Diagram

It consists of a three-level cache hierarchy with 32KB of instruction and data cache per core, 128KB of unified backside L2 cache per core, as well as a 2 MB of shared front side cache. There are totally eight e500mc cores built in the P4080 processor. It also includes the accelerator blocks known as Data Path Accelerator Architecture (DPAA) that offload various tasks from the e500mc, including routine packet handling, security algorithm calculation and pattern matching. The P4080 processor could be used for combined control, data path and application layer processing. It is ideal for applications such as enterprise and service provider routers, switches, based station controllers, radio network controllers, long term evolution (LTE) and general-purpose embedded computing systems in the networking, telecom/datacom, wireless infrastructure, military and aerospace markets.

B. DPAA

The P4080 includes the first implementation of the PowerQUICC *Data Path Acceleration Architecture* (DPAA). This architecture provides the infrastructure to support simplified sharing of networking interfaces and accelerators by multiple CPU cores. The PowerQUICC DPAA architecture includes the following major components:

- *Frame Manager*
- *Queue Manager*
- *Buffer Manager*
- *Security Engine*(SEC 4.0)
- *Pattern Matching Engine*(PME 4.0)

C. Pattern Matching Engine(PME)

The PME could provide hardware acceleration for regular expression scanning. Patterns that can be recognized or matched by the PME are of two general forms, *byte patterns* and *event patterns*. Byte patterns are simple matches such as ’abcd123’ existing in both the data being scanned and in the pattern specification database. Event patterns are a sequence of multiple byte patterns. In the PME, event patterns are defined by stateful rules. The PME specifies patterns as regular expressions (Regex). The host processor needs to convert Regex patterns into the PME’s specification data path, which means there is a one-to-one mapping between a regular expression and a PME byte pattern. Within the PME, match detection precedes in stages. The *Key Element Scanner* performs initial byte pattern matching, with hand off to the *Data Examination* Engine for elimination of false positives through more complex comparisons. The *Stateful Rule Engine* receives confirmed basic matches from the earlier stages, and monitors a stream for addition for subsequent matches that define an event pattern. Figure 2 shows the general block diagram of PME.

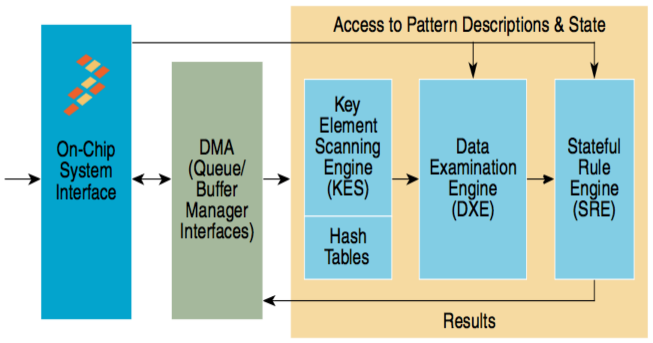
On-Chip System Interface
DMA (Queue/ Buffer Manager Interfaces)
Access to Pattern Descriptions & State
Key Element Scanning Engine (KES)
Hash Tables
Data Examination Engine (DXE)
Stateful Rule Engine (SRE)
Results

Fig. 2: P4080 PME

We explored and analyzed DPAA and its component PME in depth for our work. One of the useful features of MCA is that the API can target bare metal (no OS at all). With this as the motivation aspect, we choose PME for our work; this specialized accelerator is also a bare metal hardware. To add on to the complexity, the accelerator does not offer shared memory support with the host processors, i.e. the power processor cores (e500mc cores) for P4080. As a result, the data movement between the host and the accelerators need to be handled via a *DMA channel* explicitly. We prpose to utilize MCAPI to handle such data transportation and messaging.

4. Design and Implementation

As mentioned earlier, MCAPI is an industry-standard API for inter-core communication within a loosely coupled distributed embedded SOC. It can be treated somehow like MPI, if lighter and designed for the embedded platforms. Nodes are a fundamental concept in the MCA interfaces that map to an independent thread of control, such as a process, thread, or processor. In our implementation phase, using P4080, we set up the PME as a node in MCA and e500mc processors as the other node. Our solution stack is shown on figure 3.

We therefore explored the MCAPI transportation layer that will serve as the communication layer for PME. We modified the ’create end point’ function to cater to the PME

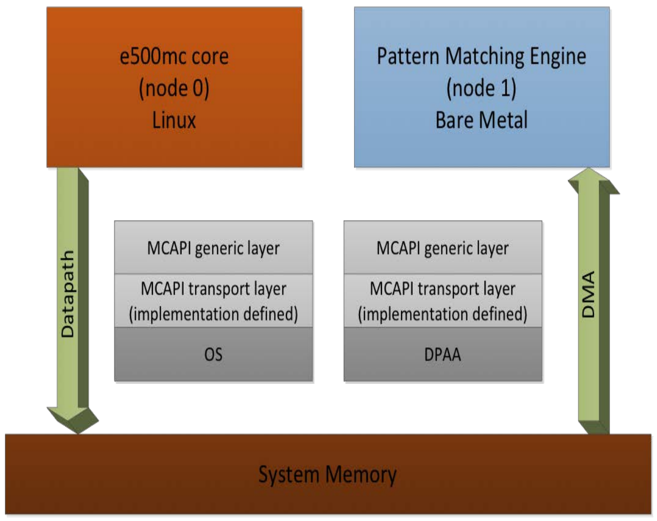

Fig. 3: Solution Stack

platform. We also modified the mcapi_trans_connect_pktchan and mcapi_trans_open_pktchan_recv_i to build the MCAPI transportation channel for PME, that will appear on top of the DMA channel for PME package transportations.

The Pattern Matcher Control Interface (PMCI) library is a C interface used to send and receive pattern matcher control commands to PME. We found that this functionality can be partly abstracted by utilizing the MCAPI massage mechanism , for e.g. mcapi_msg_send and mcapi_msg_recv. To implement the PMCI support within MCAPI transportation layer, we added PMCI shared library into the MCAPI build system. In this connectionless messaging mechanism, the source code makes function calls directly with the PMCI library. The major functions involved in this process include:

- *pmci_open*
- *pmci_close*
- *pmci_set option*
- *pmci_read*
- *pmci_write*

For instance, the pmci_open has been mapped to the MCAPI transportation layer initialization subroutine. The pmci_close has been included in the MCAPI transportation finalize subroutine. The pmci_read and pmci_write functions have been mapped into the MCAPI message receive and send subroutines while the pmci_set option has been included into the MCAPI node and endpoint initializations.

Figure 4 shows that we have set the Power processor e500mc as one node with a dedicated endpoint, and the PME as the other node in MCAPI with another end point. We therefore encrypt PMCI message under the MCAPI communication interface. Unlike the package channel or the scalar channel in MCAPI, sending and receiving messages in MCAPI are connectionless, which means there is no need to build the connections upfront between the nodes. This provides a flexibility to send or receive messages between nodes and endpoints, as well as the possibilities to communicate among multiple endpoints simultaneously.

5. Related Work

MPI [1] is a message-passing library that can be adopted by either shared or distributed memory systems. But MPI is too feature-rich and consumes too much resources (e.g. memory footprints) from an embedded systems point of view, as a result of which the library may not be a good fit for such systems. On the contrary, MCAPI is specifically designed for such systems, consumes lesser resources while still establishing the necessary communication.

There is relatively more work has been done on MCAPI than other MCA APIs. MCAPI debug [2] and verification tools [3] are also available. The reference implementation of MCAPI was created by Freescale Semiconductor Inc. and distributed in the Multicore Association official website [4]. It is supposed to be a ”kick off” point, hence it is a naive implementation. Mentor Graphics [5] created an open source MCAPI implementation based on MCAPI standard 2.0; their implementation supports both the Linux OS and Real-time OS. PolyCore [6] offers MCAPI as part of their multicore programming model solution. Lauri Matilainen [7] adopted MCAPI for their FPGA platforms, and their work was also based on the MCAPI reference implementation.

K.Eric Harper [8] from ABB Corporate Research proposed their effort to build a lock-free MCAPI implementation and ported the MCAPI concurrency runtime to the Microsoft Windows. In their work, the shared memory I/O performance has been increased by using lock-free algorithms.

In our previous work [9] and [10] , we created a portable OpenMP runtime library that uses MCA API as the translation layer thus abstracting low-level details of the target platform and creating an efficient solution.

6. Conclusion

In this work, we discuss how MCAPI can be used to establish communication between heterogeneous multicore cores without the need to be aware of all the low-level details of the platform. We believe that our work can also be extended to similar embedded platforms thus providing a portable solution. This work is currently on-going, we are looking into creating even more abstraction by using the directive-based programming model, OpenMP, considering MCAPI as the translation layer so that the programmer needs to only create a parallel version of the embedded application and the rest is taken care by our OpenMP-MCAPI methodology discussed in this paper.

Acknowledgment

The authors of this paper would like to express our sincere gratitude to the anonymous reviewers. This research has been supported by grants from Freescale Semiconductor Inc. in association with Semiconductor Research Corporation (SRC).

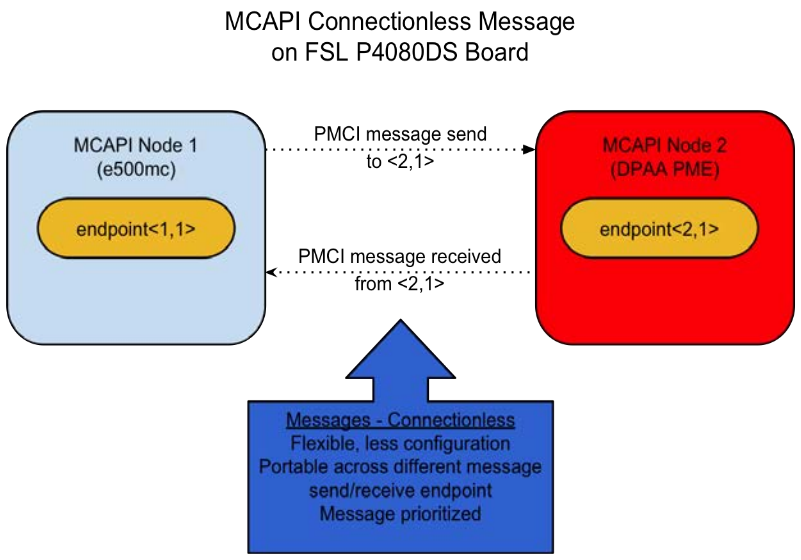
MCAPI Connectionless Message on FSL P4080DS Board
MCAPI Node 1 (e500mc)
endpoint<1,1>
PMCI message send to <2,1>
MCAPI Node 2 (DPAA PME)
endpoint<2,1>
PMCI message received from <2,1>
Messages - Connectionless
Flexible, less configuration
Portable across different message send/receive endpoint
Message prioritized

Fig. 4: MCAPI on PME

References

[1] D. W. Walker, D. W. Walker, J. J. Dongarra, and J. J. Dongarra, "MPI: A Standard Message Passing Interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[2] M. Elwakil and Z. Yang, "Debugging support tool for mcapi applications," in *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pp. 20–25, ACM, 2010.

[3] S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt, "Mcc: A runtime verification tool for mcapi user applications," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pp. 41–44, IEEE, 2009.

[4] "Multicore Association." http://www.multicore-association.org.

[5] "An open source implementation of the mcapi standard." https://bitbucket.org/hollisb/openmcapi/wiki/Home.

[6] "Polycore MCAPI Offers ThreadX RTOS Support."

[7] L. Matilainen, E. Salminen, T. Hamalainen, and M. Hannikainen, "Multicore communications api (mcapi) implementation on an fpga multiprocessor," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pp. 286–293, IEEE, 2011.

[8] K. E. Harper and T. de Gooijer, "Performance impact of lock-free algorithms on multicore communication apis," *arXiv preprint arXiv:1401.6100*, 2014.

[9] C. Wang, S. Chandrasekaran, P. Sun, B. Chapman, and J. Holt, "Portable mapping of openmp to multicore embedded systems using mca apis," in *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pp. 153–162, ACM, 2013.

[10] C. Wang, S. Chandrasekaran, B. Chapman, and J. Holt, "libeomp: a portable openmp runtime library based on mca apis for embedded systems," in *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 83–92, ACM, 2013.