A Practical Programming Guide to Software Development for Multicore Processors

Markus Levy | President Multicore Association
Rob Oshana | Director, Software R&D, Digital Networking

APR. 2014

freescale
And now for our Feature Presentation.
The Multicore Association

- Established in 2005
- Mission: Improve time to market through the use of industry standards
- Committee-based standards development

Markus Levy
President
Multicore Association
Active Working Groups

- SHIM – Software/Hardware Interface for Multicore/Manycore
- Tools Infrastructure Working Group (TIWG)
  - Tool interoperability for multiple IDEs
  - CE Linux Forum collaboration
  - Chaired by: Brian Cruickshank (TI) and Aaron Spear (VMware)
- Multicore Programming Practices Working Group (MPP)
  - Multicore software programming guide for the industry to aid in improving consistency and understanding of multicore programming issues.
  - Over 4600 downloads
  - Starting up activities on Version 2

Join MCA to Participate and Get a Head-Start
APIs from the Multicore Association

• Multicore Communications API (MCAPI 2.0)
  − Semantic for communication and synchronization between processing cores in embedded systems.
  − Low-level layer for higher-level programming models such as OpenMP

• Multicore Resource Management API (MRAPI)
  − Capabilities required by multicore applications to allow coordinated concurrent access to system resources (i.e. memory, mutexes)

• Multicore Task Management API (MTAPI)
  − Leveraging task parallelism on embedded devices (homogeneous or heterogeneous multicore processors) with dynamic scheduling and task mapping to processor cores to help optimize throughput on multicore systems
Why MxAPI?

- Addresses embedded systems requirements: resource constraints, portability, heterogeneity
- Low hurdle for implementation
- C-API and pure library (no language extension)
- Low-level API: foundation for higher-level programming models

- Spans multiple number of cores, types of cores (CPU, DSP, accelerator), operating systems, physical transports, memory architectures
- Motivation for standardization
  - Commercial and open source implementations available (e.g., by OS vendors): high quality, maintenance, …
  - Support by tool vendors (e.g., correctness checkers)
  - Widespread know-how among software developers
The Multicore Association - Foundation APIs

Support for Wide Variety of Services and Functions
SHIM Introduction

A portable interface between hardware and software in multicore systems
What is SHIM?

**Shim (spacer)**

From Wikipedia, the free encyclopedia

A **shim** is a thin and often tapered or wedged piece of material, used to fill small gaps or spaces between objects. Shims are typically used in order to support, adjust for better fit, or provide a level surface. Shims may also be used as spacers to fill gaps between parts subject to wear.

- Multicore tools, as well as OS/middleware, help applications designed and run optimally on multicore chips while hiding the hardware specifics.
- However, the tools must understand the hardware specifics.
- SHIM provides tools with the specific multicore HW description as standardized, open XML.
Who Benefits From SHIM?

• Development tool vendors, runtime systems (such as operating systems), and semiconductor vendors
  – Today: challenge to support the virtually unlimited number of processor configurations.
  – SHIM will enable easier support of successive generations of SoCs without requiring extensive rework

• SHIM will allow the system developers\textit{ early and quick architectural evaluation including performance} with your application with or without an actual hardware

• SHIM will allow software developers to utilize a complex multicore hardware\textit{ without reading a 1000 page manual}
**Relationship and Use-Cases**

1. **System Configuration**
   - Hardware Modeling
   - SHIM XML
   - Auto-parallelizing compiler
   - Parallelization tool
   - OS/middleware configurator
   - Performance analysis tool
   - 2. Performance Estimation
SHIM Is and Can

• An interface defined as an XML schema
• An extraction of HW properties that matter to multicore tools
• A HW model described from a SW point of view
  − Not a 100% description
• Help tools roughly estimate SW performance
  (with a goal of 80% accuracy)
• Help tools configure themselves and/or auto-generate the HW-specific configuration UI
• Help configure device drivers or HW abstraction layer (HAL)
  − NOT auto-generate HAL
• NOT a tool itself – tools are implemented by various vendors that use SHIM
Hardware Elements Expressed in SHIM

AddressSpaceSet:
Has multiple SubSpaces, containing MasterSlaveBindingSet and PerformanceSet also expressed in terms of Latency and Pitch in cycles

XXXCommunication
Inter-core comm., has PerformanceSet containing Latency and Pitch in cycles

MasterComponent
Cache, ClockFrequency, CommonInstructionSet, AccessTypeSet

SlaveComponent:
size, RWType

ComponentSet
Nest itself, can contains multiple, but single root

*Note that not all relations are illustrated
64-core SHIM XML Model – Can be created with SHIM editor
MxAPI Taxonomy

• Designed for heterogeneous and resource constrained embedded systems
• Allows the implementation of existing programming interfaces, such as OpenMP
• C-API and pure library (no language extensions), enhances portability
• Commercial and open source implementations available
• Support by tools vendors

API and a semantic for communication and synchronization between processing cores in embedded systems

API for creating low-level lightweight tasks that can delegate their execution to HW or SW nodes being part of a system consisting of homogeneous or heterogeneous processors and/or processor cores

API for application-level management of shared resources in multicore systems
Example: MC-API wrapper for MCC Library
MCC Lib. – Multicore Communication Library

Simple and lightweight library for fast data exchange between applications running on different cores of a dual/multi-core system

Primarily designed for Vybrid platform with Linux running on ARM® Cortex®-A5 and MQX RTOS running on ARM® Cortex®-M4 core (could be ported to different OSes/platforms)

MCC API contains function for messages sending/receiving between endpoints, with both blocking and non-blocking options. Shared RAM, inter-core interrupts and hw semaphores are utilized in this design. All resources are statically configured at the compile time.

```
int mcc_initialize(MCC_NODE node)
int mcc_destroy(MCC_NODE node)
int mcc_create_endpoint(MCC_ENDPOINT *endpoint, MCC_PORT port)
int mcc_destroy_endpoint(MCC_ENDPOINT *endpoint)
int mcc_send(MCC_ENDPOINT *endpoint, void *msg, MCC_MEM_SIZE msg_size, unsigned int timeout_us)
int mcc_recv_copy(MCC_ENDPOINT *endpoint, void *buffer, MCC_MEM_SIZE buffer_size, MCC_MEM_SIZE *recv_size, unsigned int timeout_us)
int mcc_recv_nocopy(MCC_ENDPOINT *endpoint, void **buffer_p, MCC_MEM_SIZE *recv_size, unsigned int timeout_us)
int mcc_msgs_available(MCC_ENDPOINT *endpoint, unsigned int *num_msgs)
int mcc_free_buffer(void *buffer)
int mcc_get_info(MCC_NODE node, MCC_INFO_STRUCT* info_data)
```
MCAPI® Wrapper – Motivation

• To align current MCC implementation with existing industrial standards, a wrapper is created to allow to use MCAPI® functions with the MCC system.
• Users will have a possibility to use both
  – the MCC API for simple/fast/low-overhead communication (MCUs with limited resources), or
  – the MCAPI® standard API (broader functionality, higher footprint) running above the MCC system

• The intention is to implement just the messaging portion of the MCAPI®. Packet and scalar channels will not be implemented as it would require to change the MCC layer and break the backward compatibility. Messages provide flexibility in data exchange between endpoints and has been chosen as the primary and the only way of communication within the MCC library.
MCAPI® Wrapper – High Level Design Overview

MCAPI® WRAPPER
(mca.h, mcapi.c/.h – reuse of headers provided by Multicore Association)

MCC API Layer
(mcc_config.h, mcc_api.c/.h)

MCC COMMON Layer
(mcc_common.c/.h)

MCC OS-SPECIFIC Layer
(mcc_linux.c/.h, mcc_mqx.c/.h)

MQX driver XY

Linux driver XY

mcapi_initialize()
mcapi_finalizer()
mcapi_domain_id_get()
mcapi_node_id_get()
mcapi_node_init_attributes()
mcapi_node_set_attribute()
mcapi_node_get_attribute()

mcapi_endpoint_create()
mcapi_endpoint_get_i()
mcapi_endpoint_get()
mcapi_endpoint_delete()
mcapi_endpoint_get_attribute()
mcapi_endpoint_set_attribute()

mcapi_msg_send_i()
mcapi_msg_send()
mcapi_msg_recv_i()
mcapi_msg_recv()
mcapi_msg_available()

mcapi_test()
mcapi_wait()
mcapi_wait_any()
mcapi_cancel()
mcapi_display_status()
MCAPI® Wrapper – Implementation

- MCC API Layer stays unchanged.
- No changes in the MCC bookkeeping data structure (shared memory).
- Because MCAPI® Layer includes both blocking and non-blocking calls, a receive task has to be introduced. It will handle requests (data send/recv) from non-blocking API functions. Processing of these requests will be triggered by the CPU-to-CPU ISR (via a LWSem).
- Reuse of headers provided by Multicore Association® (mca.h, mcapi.c/.h).
MCAPI Example: Adding API

Application

.... application code ...

mcapi_msg_recv(subport, buffer, ...);

if(mcapi_status != MCAPI_SUCCESS)
    {handle error}

.... application code ....

mcapi_msg_send(localport, subport, buffer, ...);

if(mcapi_status != MCAPI_SUCCESS)
    {handle error}
MCAPPI Example: Connecting Cores

Freescale i.MX6q

SMP: Linux

Communication between nodes

Tools: Configuration Time is less than 1 hour
MCAPI Example: Connecting Cores

Freescale i.MX6q

AMP: Linux, RTOS & Bare Metal

Communication between nodes on Linux, RTOS and bare metal

Tools: Configuration Time is less than 1 hour
Same Application Source Code
MCAPI Example: Connecting Cores

Freescale Vybrid VF6xx: Cortex-A5 and Cortex-M4

AMP: different cores and different OS

Communication between nodes on Linux, RTOS and bare metal

Tools: Configuration Time is less than 1 hour

Same Application Source Code
MCAPI Example: Connecting Cores

Freescale Vybrid VF6xx: Cortex-A5 and Cortex-M4 (2)

AMP: Between Cores and Processors

Communication between nodes on Linux and RTOS

Tools: Configuration Time is less than 1 hour

Same Application Source Code
TIWG and Freescale tools
• Initiated by Multicore Association, TIWG is promoting interaction of existing and new tools to support migration of legacy sequential applications to multicore platforms and development of new parallel software.

• The TIWG holds regularly scheduled meetings, typically, on a bi-weekly basis.

• Primary Contributing Members:
  - Ericsson
  - EfficiOS
  - Freescale Semiconductor
  - Mentor Graphics
  - Polytechnique Montreal
  - Samsung
  - Texas Instruments
  - University of Huston
  - Wind River

• One important contribution of TWIG to tools standardization is CTF (Common Trace Format).

• The Common Trace Format specifies a trace format based on the requirements of the industry and the Linux community.
Trace IP – Hardware platform from Software perspective

- Software profiler needs require a deep understanding of hardware Trace IP
- The software perspective on Trace IP blocks is very different from hardware perspective
- Software is concern to define a logical flow for configuration and collection of a hardware trace
- Configuration attributes needs to be easy to understand and easy to integrate into a software framework
- Solution is to describe Trace IP into a platform configuration XML file; through data binding those data are easily accessible from software (e.g., LMX for C++, JAXB for Java)
Adopting new standards to CodeWarrior

- CodeWarrior software suite is already using XML files to describe Trace IP
- There is under investigation the adoption of Common Trace Format for Trace events
- We are interested and actively involved into participation to Multicore Association working groups that defines new standards for multicore systems
- We are constantly reviewing working groups drafts for new standards
MPP Use for Multicore Optimization
General Programming Issues

• Fact: C/C++ will be predominant programming language for at least 8 years

• Problem: While we wait for long term research results, the multicore programmability gap is opening rapidly
What Does The Industry Need Right Now?

• Continue with long term research into languages, methodologies, etc
• Short term direction as to how today’s embedded C/C++ code may be written to be “multicore ready” today
• Influence of a group of like-minded methodology experts to ensure completeness, usefulness and industry-wide compatibility
• The creation of a standard “best practices” guide through a recognized, neutral industry body, based on capturing current best practices
Action - Multicore Programming Practices Working Group

• Best practices for writing multicore-ready software using C/C++ without extensions
• Allow embedded software to be more easily compiled across a range of multicore processor platforms
• Framework of common pitfalls when transitioning from serial to parallel
• Consider solutions or avoidance tactics
• Minimize debugging efforts by reducing bugs
MPP - Summary

• Problem To Solve: While we are waiting for long term research results, the multicore programmability gap is opening rapidly

• Action Taken: MPP Working Group – Best practices for writing multicore-ready embedded software

• Objective Met: Release of MPP guide to meet immediate needs of multicore stakeholders in an open, efficient and effective manner
The Essence of MPP

• Introduction & Business Overview
• Overview of Available Technology
• Analysis and High Level Design
• Implementation and Low-Level Design
• Debug
• Performance Tuning
• Glossary

• So far almost 5000 industry downloads of MPP
1. **Globales**: Avoiding False Sharing

Shared memory systems are achieved with multiple processors sharing one unified memory. Each processor reads or writes to the same location. To guarantee that all processors have a consistent view of main memory, each processor directly maps to the same location. When one processor reads a memory location modified by another processor, the other processor’s cache must be flushed or invalidated, which reduces memory performance.

### 1.1. False Sharing

This situation may occur for independent memory values. For example, assume that two processors read contiguous data values, and each needs all the data it reads.

Each CPU reads its data into its own cache. If the main memory is read, the cache line is written back to main memory. When one processor reads data, it also needs to read from main memory, even though the actual values have not changed.

![Diagram of cache management](image)

This situation is called false sharing and occurs because of data structure layout or interferences with data access schemes implemented.

### 1.2. Poor Memory Access Pattern

An example using `globales` demonstrates a poor access pattern:

```c
void compute_partition(void *args) {
    uint32_t k;
    compute_args_t args = (compute_args_t *) args;
    uint32_t n = args->n;
    uint32_t k = args->k;
    uint32_t di = args->di;

    for (i = 0; i < n; i += di) {
        if (i > n) break;
        // ... compute and write
    }
}
```

In this example, a large array of values is aliased by a single function. The example is simplified and does not show code dependencies between the array items. The data may be partitioned in any way that removes dependence on the array items. The processor reads the data from main memory, modifies the data, and writes the data back to main memory. The problem is that when one processor’s cache is invalidated, the other processor reads the same data and reduces the performance.

### 1.3. Memory Access Optimization

A better approach is to break the data into chunks of contiguous data values.

```c
void compute_partition(void *args) {
    uint32_t n;
    compute_args_t args = (compute_args_t *) args;
    uint32_t k = args->k;
    uint32_t di = args->di;

    for (i = 0; i < n; i += di) {
        if (i > n) break;
        // ... compute and write
    }
}
```
Example: MPP Use for Multicore Optimization
Using IP Forwarding as an MPP Case Study

**Multicore SoC**

- **Ingress Pipe**
  - Receive packets
  - L2 process proto-check
  - Classify Table Hash Table
  - FIB Lookup LPM Table
  - ARP Lookup Hash Table

- **Egress Pipe**
  - Modifying Layer2 header
  - en-queue (Tail drop, WRED)
  - Scheduling among 128 Groups
  - Traffic metering
  - Sending packets

**Algorithm**

- **Task Parallelism**
  - Pool of cores running the control plane. Multiple control tasks can be executed in parallel. OS load balances cores.

- **Data Parallelism**
  - Each core running identical Packet Processing Loop
  - Control Data—ARP, RTSCP, etc.
  - Data Flows—TCP, UDP, RTP, etc.

**Multicore Architecture Model**

- **CPU Running Packet Processing Loop**
  - CPU Running Packet Processing Loop
  - CPU Running Packet Processing Loop
  - CPU Running Packet Processing Loop
  - CPU Running Packet Processing Loop

- **Freescale**
MPP Case Study

- Re-partition the application and optimize it to meet performance goals
- Freescale followed the following MPP guidelines called out in chapter 3
  - Prepare
  - Measure
  - Tune
  - Assess
What We Did

1. Analyzed customer application
2. Partitioned application to multiple cores (parallel and/or pipeline operation)
3. Measured performance to see if it’s as expected (iterate)
4. Collect debug info/statistics to locate bottlenecks
5. Fine-tuned partitioning design based on above collection to eliminate bottlenecks
6. Intelligent use of data path acceleration capabilities for further optimizations
7. After several iterations, we met our performance goal and concluded the exercise
Before Applying MPP Guidelines;
Multicore Partition – Ingress and Egress
## Performance Results – Initial Partition

<table>
<thead>
<tr>
<th>Module</th>
<th>Instructions</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx()</td>
<td>130</td>
<td>107</td>
</tr>
<tr>
<td>L2i()</td>
<td>43</td>
<td>66</td>
</tr>
<tr>
<td>Classify()</td>
<td>322</td>
<td>470</td>
</tr>
<tr>
<td>Ip_route_lookup()</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>Arp()</td>
<td>162</td>
<td>174</td>
</tr>
<tr>
<td>Queue_enqueue()</td>
<td>180</td>
<td>660</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>922</strong></td>
<td><strong>1564</strong></td>
</tr>
</tbody>
</table>

**Egress-core:**

<table>
<thead>
<tr>
<th>Module</th>
<th>Instructions</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 ColorBlind_srTcm() total</td>
<td>163</td>
<td>326</td>
</tr>
<tr>
<td>Queue_deque_pq/drr ()</td>
<td>318</td>
<td>654</td>
</tr>
<tr>
<td>Queue_deque ()</td>
<td>555</td>
<td>1216</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>555</strong></td>
<td><strong>1216</strong></td>
</tr>
</tbody>
</table>
After Applying MPP Guidelines; Multicore Partition – Ingress and Egress
## Performance Results – After Re-Partition

<table>
<thead>
<tr>
<th>Module</th>
<th>Instructions</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx()</td>
<td>130</td>
<td>107</td>
</tr>
<tr>
<td>L2i()</td>
<td>43</td>
<td>66</td>
</tr>
<tr>
<td>Classify()</td>
<td>322</td>
<td>470</td>
</tr>
<tr>
<td>Ip_route_lookup()</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>Arp()</td>
<td>162</td>
<td>174</td>
</tr>
<tr>
<td>Qman_enqueue()</td>
<td>120</td>
<td>118</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>862</strong></td>
<td><strong>1022</strong></td>
</tr>
</tbody>
</table>

### Egress-core:

<table>
<thead>
<tr>
<th>Module</th>
<th>Instructions</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qman_poll()</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Queue_enqueue()</td>
<td>143</td>
<td>151</td>
</tr>
<tr>
<td>3 ColorBlind_srTcm()</td>
<td>163</td>
<td>192</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue_deque_pq/drr()</td>
<td>318</td>
<td>340</td>
</tr>
<tr>
<td>Queue_deque()</td>
<td>352</td>
<td>375</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>535</strong></td>
<td><strong>566</strong></td>
</tr>
</tbody>
</table>
Case Study Conclusions

• Partitioning and optimizing multicore software requires a disciplined process

• Use an iterative approach to achieve faster time to market, or ‘time to performance goals”

• MCA Multicore Programming Practices can serve as a useful guide for developers involved in all phases of multicore software development
What Is In Store For Version 2?

• In the process of aggregating interested participants.
• Exploring additions that could focus on automotive, safety critical, GPUs, OpenACC, etc.
• More industry case studies
• More practical examples of how to use the MPP techniques
Member Involvement Required to Move Forward
Multicore Association Value Proposition

Time-to-market pressures force multicore software solutions into narrow vertical markets, limiting the viable market for silicon and also preventing software reuse.

There was no widely accepted multicore programming model outside those defined for (1) SMP systems, or (2) very specific application domains.

A flexible, general purpose multicore programming model that supports both homogeneous and heterogeneous architectures:

- Provides the ability for programmers to transition to multicore programming
- Allows companies to create custom software for their chips
- Allows tool-chain providers to maximally leverage their engineering
- **End-users no longer have to craft custom infrastructure to support their programming needs**
Check Out This Cool Book

1. Overview of Embedded and Real-Time Systems
2. Embedded Systems hardware/software co-design
3. Specification and Modeling techniques
4. Architecture and design patterns
5. Real-Time building blocks
6. Hardware interface to embedded software
7. Programming and Implementation Guidelines
8. Software Reuse by Design in Embedded Systems
9. Embedded Operating systems
10. Linux and OSS
11. Software Performance Engineering
12. Optimizing Embedded Software for Performance
13. Optimizing Embedded Software for Memory
14. Optimizing Embedded Software for Power
15. User interfaces for embedded systems
16. Integration and testing techniques and quality
17. Software Development Tools

18. Multicore Software Development

20. Intellectual Property Issues with Embedded Software
21. Managing embedded software development
22. Agile development for embedded systems
23. Embedded Software for Automotive Applications
24. Embedded Software Development for Storage and I/O
25. Embedded Software for Networking Applications
26. Multiple Case Studies