Open Event Machine

White Paper

August 2012

Revision Record	
Document Title: Open Event Machine: White Paper	
Revision	Description of Change
0.0.1	Created
0.0.2	Updated after review
1.0.0	Updated for release of SA_OpenEM 1.0.0.1

TABLE OF CONTENTS

1 Scope

Open Event Machine is a library that implements a multicore runtime system. This paper describes the operation and illustrates the performance of the Open Event Machine (OpenEM). The API of OpenEM is defined by Nokia Siemens Networks. TI provides an implementation for its high-performance multi-core DSP.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description

Table 1. Referenced Materials

3 Acronyms

Acronym	Description

Table 2. Acronyms

4 Introduction

TI is providing high-performance multi-core DSP since 2006.

The first generation of multi-core DSP (TMS320C647x) had limited support for multi-core programming. Programmers typically looked at those DSP as "multiple single-core DSP in a single package". The integration effectively allowed achieving more revenue for a given area and power budget and SW reuse allowed keeping the development cost low.

The second generation of multi-core DSP (TMS320C66x) is built around the Multicore Navigator. Those DSP belong to the KeyStone product line. The KeyStone architecture provides an excellent platform for load balancing and allows reducing the cost per channel even more.

The Open Event Machine (OpenEM) is a multi-core runtime system developed for the KeyStone product line. TI’s implementation extensively leverages KeyStone’s multicore infrastructure and especially the Multicore Navigator. The main missions of OpenEM are to enable efficient scheduling, dispatching and load balancing of work across the cores of a KeyStone device.

In addition, the Open Event Machine enables support easy porting of multi-core applications from one KeyStone device to another. It is able to transfer data between global shared and local private memories and optionally manages cache coherency. Finally, it integrates well with many interfaces and accelerators as well as with different Operating Systems.

5 Why a new Multicore Runtime System?

There are already many multi-core operating systems on the market. Although, multi-core operating systems are capable of load balancing work across multiple cores, they are not up to the job in several situations.

In particular when the work is chopped in too small and/or too many chunks, the overhead generated by operating systems will be prohibitive. It doesn't pay off to create a thread for a chunk that takes only a few thousand cycles to execute, nor to create a thread for many hundreds or thousands of chunks.

Some programmers prefer to run their application on the bare metal (i.e. without operating system). Other programmers use one of the many real-time operating systems that don't have multi-core support yet. And still other programmers want to load balance work across cores running different operating systems. Each of them will benefit from a multi-core runtime system.

A lot of research has been done on multi-core runtime systems. And several multi-core runtime systems (OpenMP, OpenCL, TBB, ...) are already available on the market. They typically do not rely on an operating system for the load balancing. They opted for a co-operative scheduler with fibers instead of a pre-emptive scheduler with threads. But, they are also not designed for embedded systems with specific requirements for heterogeneous platforms, real time constraints, non-uniform memory architectures, ...

6 The KeyStone Architecture

The Multicore Shared Memory Controller (MSMC) and the Multicore Navigator are the vital building blocks of the KeyStone architecture.

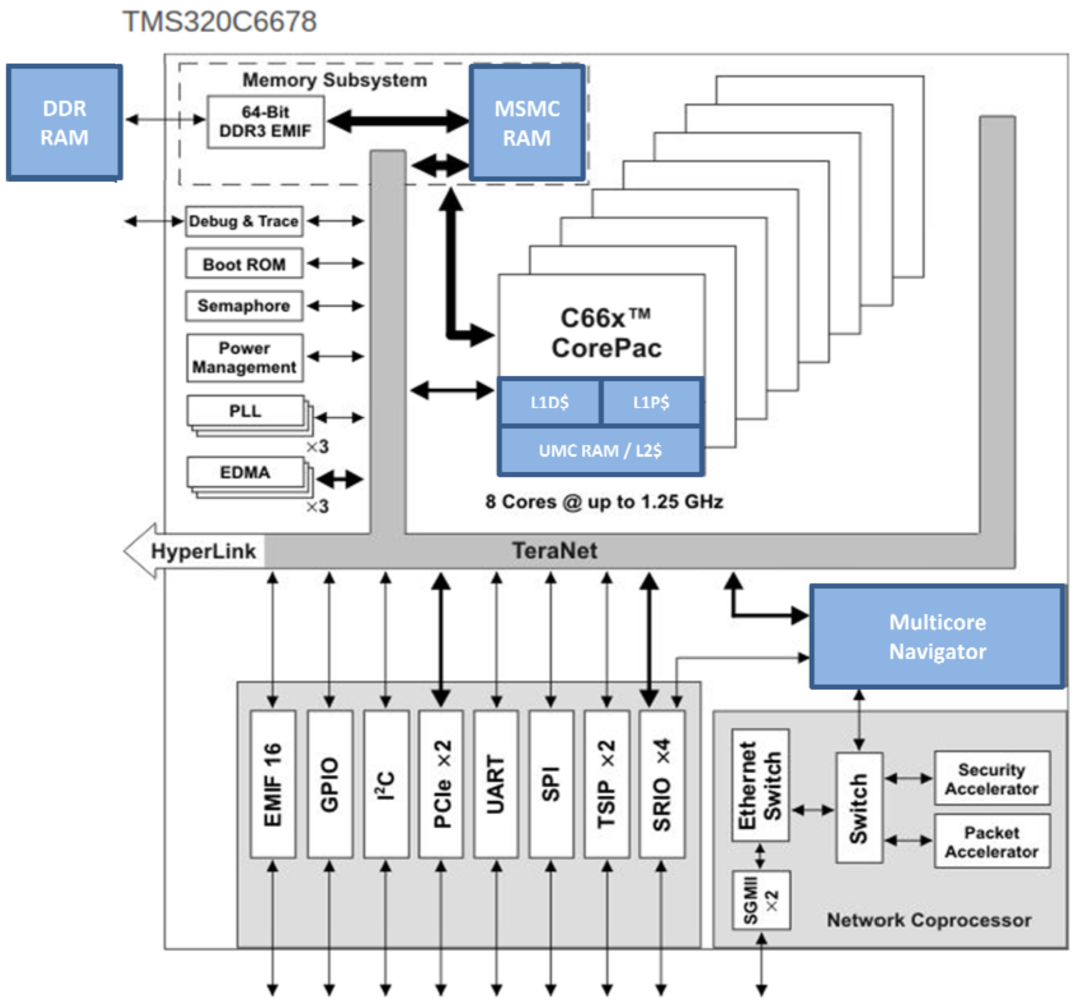


Figure 1: KeyStone Architecture

The MSMC provides access to the shared memories for one or more C66x CorePac and other SoC masters. There are 2 shared memories: on-chip (referred to as MSMC RAM) and off-chip (referred to as DDR RAM). Each C66x CorePac contains one C66x CPU, L1 program and data cache/RAM and L2 unified cache/RAM.

The Multicore Navigator consists of a central Queue Manager and multiple Packet DMA engines. The Queue Manager provides atomic access to thousands of HW queues. The Packet DMA engines use HW queues to transfer packets between memory and SoC masters (interfaces, accelerators) or from memory to memory.

The Multicore Navigator also has Packet RISC engines. They allow extending the basic push and pop services of the Queue Manager with more sophisticated services, like queue monitoring, traffic shaping and packet scheduling.

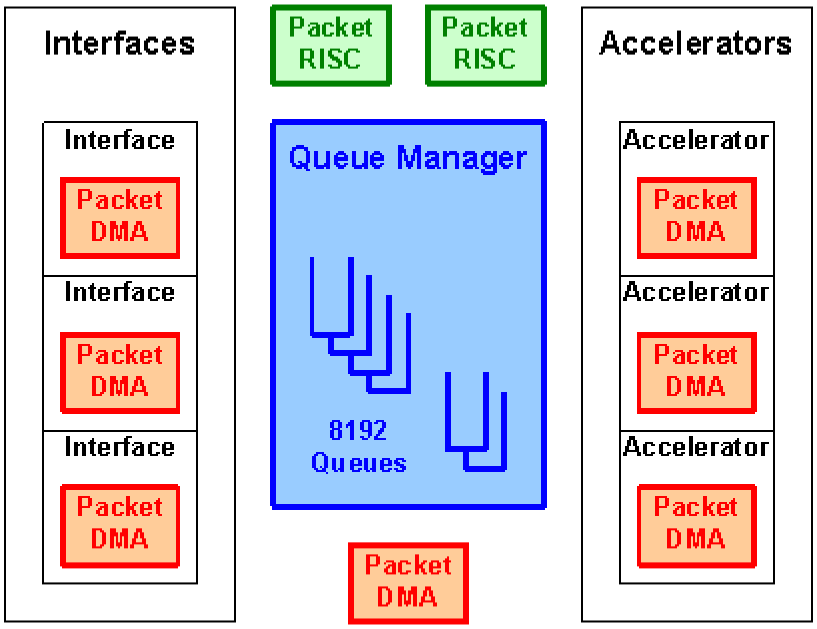


Figure 2: Multicore Navigator

7 Events, Queues and Execution Objects

OpenEM provides services to manage events, queues and execution objects. All objects are shared among all cores and all services are multi-core safe.

Events typically carry the data to process, but they can be data-less tokens as well. The data payload can be stored in a single buffer or scattered over multiple buffers. OpenEM provides services to access each buffer of the event payload.

Each event belongs to an event pool. Each event pool has a free queue. At init time, all events are queued in the free queues of their event pools. OpenEM also provides services to allocate and free events. An event is allocated from the corresponding free queue and freed to the same free queue.

Execution objects encapsulate the algorithm to execute when an event is received. The algorithm is executed by a receive function. The user implements the receive function and registers it with the execution object. OpenEM provides services for creating and deleting execution objects.

Queues connect events (data) and execution objects (algorithms). Each queue is associated with one execution object and all queued events will be processed by this execution object. OpenEM provides services for creating and deleting queues as well as a service for sending events to queues.

Each queue has a context. The purpose of the context is to store persistent data, i.e. data that stays alive before and after the processing of an event.
Each queue exposes attributes that control the scheduling of the queued events. The scheduling attributes will be described in [8].

We can distinguish four states during the lifetime of an event:

- **free**: the event sits in the free queue of its event pool
- **preparing**: the event has been allocated but not yet sent
- **ready**: the event has been sent and is waiting to be processed
- **running**: the event is being processed

In order to go from the ready to the running state, OpenEM needs to schedule and dispatch the event. The scheduler selects a non-empty queue, pops the oldest event and sends it to a dispatcher. The dispatcher looks up the corresponding execution object and calls the receive function. We will further describe the scheduling and dispatching operations in [8] and [9].

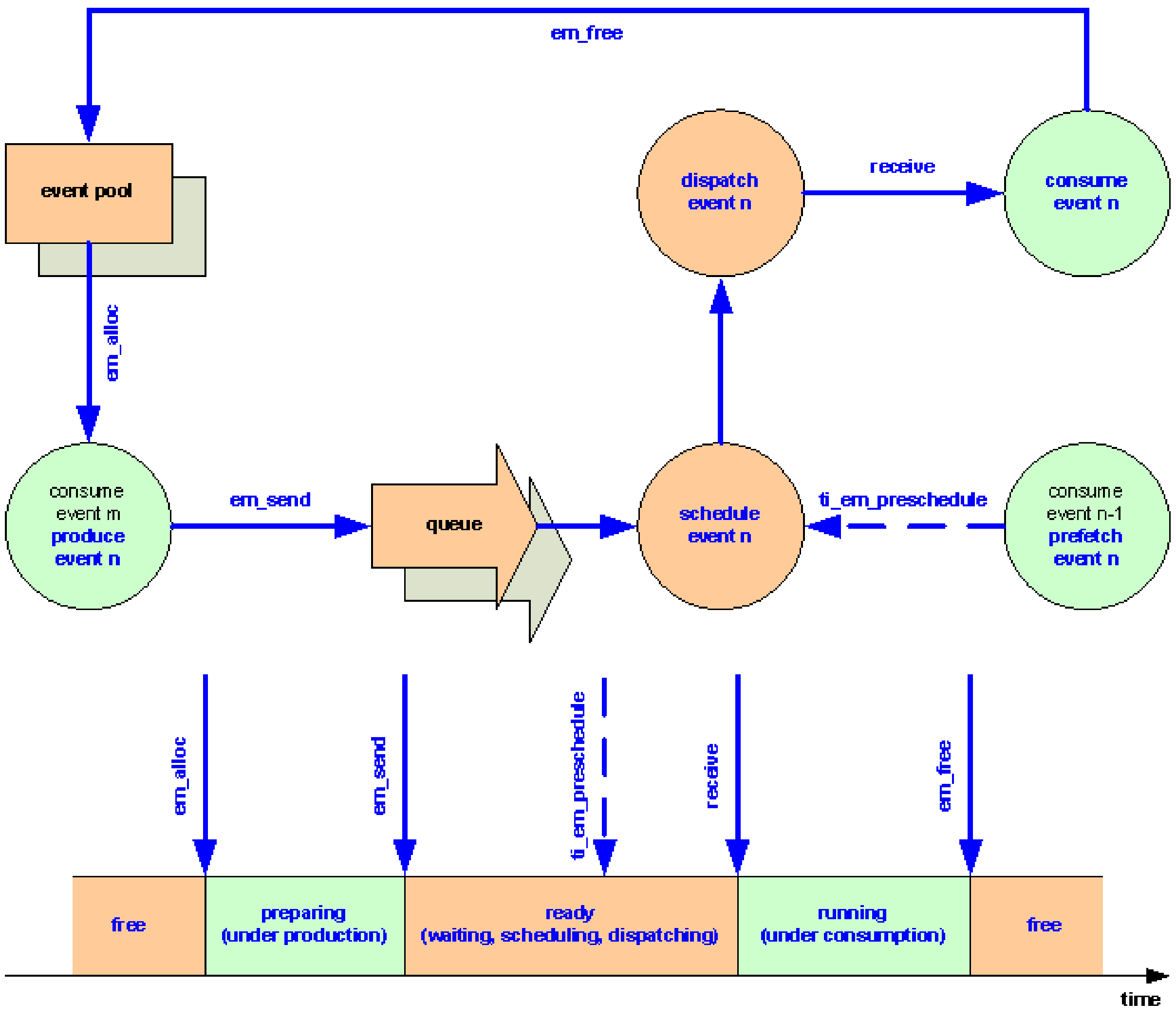


Figure 3: Event Life Cycle

8 Scheduling

There are four criteria to select a non-empty queue: priority, atomicity, locality and order.

- **Priority**
  Each queue has a priority. If two queues with different priorities are not empty, all events in the high-priority queue will be scheduled before any event in the low-priority queue.
  Note that a high priority event will never pre-empt a low priority event!

- **Atomicity**
  A queue is either parallel or atomic. Unlike a parallel queue, an atomic queue does not allow that a core starts processing an event from this queue as long as another core is processing another event from this queue. OpenEM provides a service to notify the end of atomicity. When the scheduler receives such notification, it is allowed to schedule the next event from the atomic queue.
  The user can protect critical sections without using a single mutex. As a result, the user code is more robust and efficient.

- **Locality**
  Each queue belongs to a queue group. Each queue group has a core mask selecting the cores that are allowed to process the events from the queue group. OpenEM provides services for creating and deleting queue groups.
  The user can cover the whole spectrum between static and dynamic load balancing with minimal changes to the user code.

- **Order**
  If two or more events are eligible for scheduling the event that has been ready for the longest time will be scheduled.

There are 2 ways to operate the scheduler: asynchronous and synchronous.

- **Asynchronous**
  The scheduler is deployed on the Packet RISC engines of the Multicore Navigator. Each scheduling operation is triggered by a scheduling request submitted by a C66x core. OpenEM provides a service to submit a scheduling request. Typically, the user will request the next event before the processing of the current event ends. As a result, the scheduling and processing will be pipelined. If the user hasn't requested the next event, OpenEM will automatically submit a scheduling request when the processing of the current event has ended. The scheduler is deployed on the Packet RISC engines of the Multicore Navigator. The scheduling is based on all three criteria: priority, atomicity and locality.

- **Synchronous**
  The scheduler is deployed on the C66x cores of the C66x CorePac. OpenEM will schedule the next request when the processing of the current event ends.

The scheduling is only based on priority and locality.

Note that OpenEM does not yet support synchronous scheduling.

9 Dispatching

The dispatcher is deployed on the C66x cores of the C66x CorePac. The user typically calls the dispatcher from within a dispatch loop. The dispatcher is non-blocking. If no event is available, it returns immediately with a negative response. The user decides how to respond to negative responses. He may ignore them, or he may suspend the dispatch loop after a critical number of negative responses.

```
int main(void) {
   ...
   while(1)
      if (ti_em_dispatch_once()!=EM_OK) my_handle_false_dispatch();
   ... }
```

If an event is available, the dispatcher will look up the execution object and call the receive function.

```
em_status_tt ti_em_dispatch_once(void) {
   ...
   if((lvEventHdl=fetch_event())==EM_EVENT_UNDEF) return EM_ERR_NOT_FOUND;
   ...
   receive(lvEventHdl,...);
   ...
   return EM_OK; }
```

There are 2 ways to operate the dispatcher: run-to-completion and co-operative.

- **Run-to-Completion**
  The receive function is supposed to run to completion. Therefore, it should not wait on a condition that depends on the execution of another receive function. Otherwise, deadlocks are very likely to occur.
  The run-to-completion dispatcher works with both asynchronous and synchronous schedulers.

- **Co-operative**
  OpenEM provides services for suspending and resuming events.
  When a running event is suspended, the execution of the receive function is suspended. OpenEM saves the state of the receive function and yields to the dispatcher. The event is now in the suspended state.
  When a suspended event is resumed, the execution of the receive function does not immediately resume. OpenEM brings the event in the scope of the scheduler. The event is now in the resumed state. The execution of the receive function will resume when the event is re-scheduled and re-dispatched. Resumed events have precedence over ready events of the same priority.
  The co-operative dispatcher only works with the synchronous scheduler.

OpenEM does not support pre-emptive dispatchers. Pre-emption allows suspending the current execution without its cooperation. The user should rely on the services of an Operating System if pre-emption is required.

Note that OpenEM does not yet support co-operative dispatching.

10 Putting Everything together

The figure below illustrates a case involving two C66x cores. The run-to-completion dispatcher runs on each C66x core and the asynchronous scheduler runs on a Packet RISC core. There are three execution objects. Each execution object has a receive function that has been implemented by the user. There are many (thousands) queues. Most (if not all) of the queues are atomic. There are also two queue groups: two execution objects (1 and 3) belong to one queue group selecting both C66x cores and one execution object (2) belongs to the other queue group selecting one C66x core (1).

When the scheduler receives a request for the next event from a C66x core, it will select a non-empty queue (based on priority, atomicity and locality), pop the oldest event from the selected queue and send the event to the requesting C66x core.
When the dispatcher finds the next event, it will find the execution object and call the corresponding receive function. During the execution of the receive function, new events may be allocated and sent to the queues.
In the case of atomic events, the receive function (or dispatcher) will notify the end of atomicity to the scheduler.

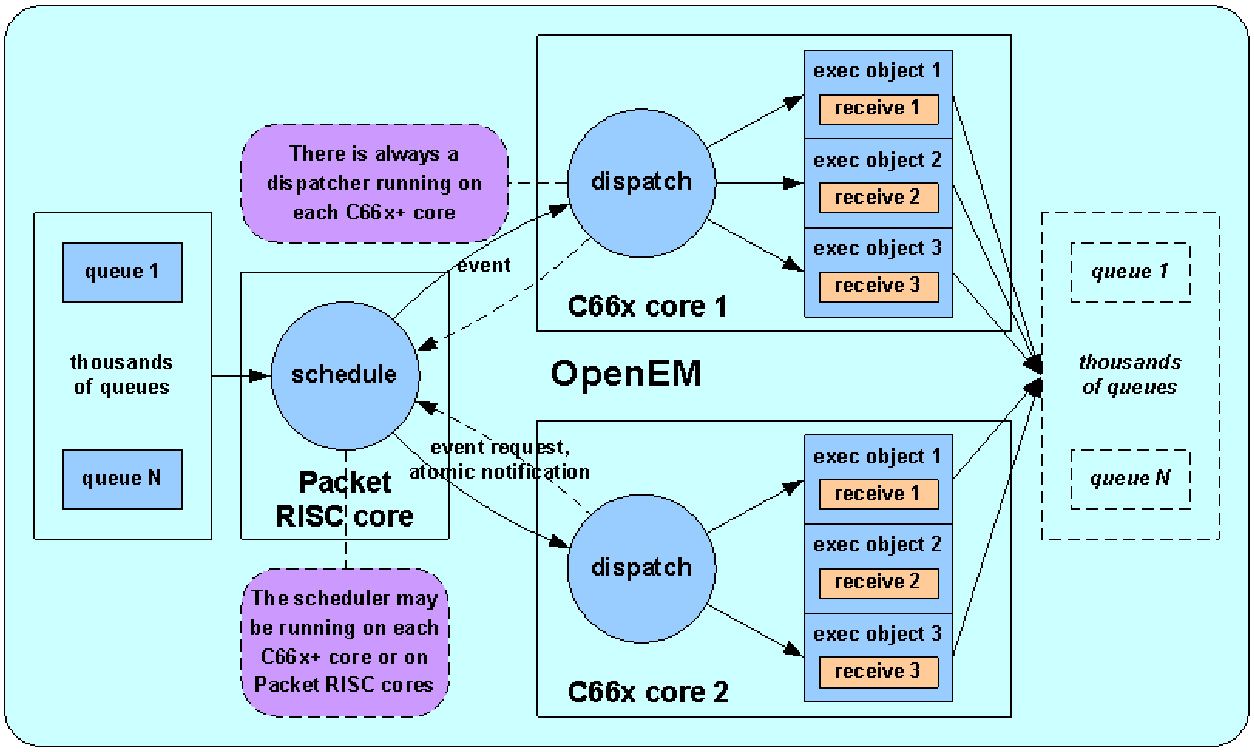


Figure 4: Putting Everything together

11 Load Balancing

There are two distinct strategies to achieve dynamic load balancing: eager and lazy.

- **Lazy**
  Lazy (or passive) load balancing relies on event consumers (C66x cores) requesting the next event when they become ready. There is no explicit load balancer involved. The load balancing overhead is minimal. But the load balancing quality will be sub-optimal, in particular when the execution time of the receive functions varies a lot.

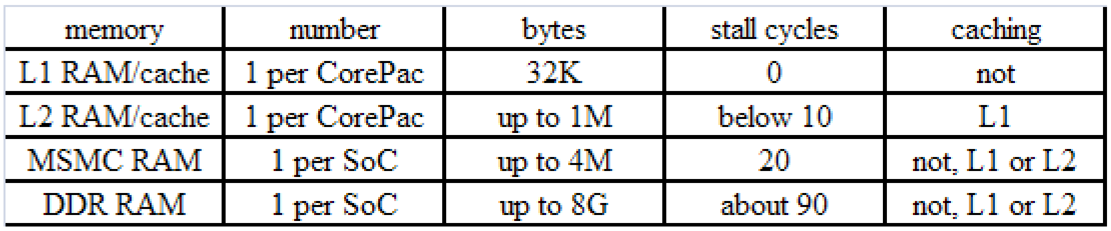
- **Eager**
  Eager (or active) load balancing involves an explicit load balancer. As soon as an event becomes ready, the load balancer will assign it to a consumer (C66x core). The load balancing overhead is typically high. Load balancing decisions are based on estimates of the number of resources required for the processing of each event. The quality of the load balancing can be very high, but also depends on the accuracy of the resource estimates.

OpenEM natively supports lazy load balancing. If eager load balancing is required, the user has to take the load balancing decisions and send the events to queues that are consumer dedicated.

We remind that OpenEM supports static load balancing as well and allows gradually moving from static to dynamic load balancing with minimal changes to the user code.

12 Non-uniform Memory Architecture

The KeyStone architecture offers multiple levels of memory.



memory	number	bytes	stall cycles	caching
L1 RAM/cache	1 per CorePac	32K	0	not
L2 RAM/cache	1 per CorePac	up to 1M	below 10	L1
MSMC RAM	1 per SoC	up to 4M	20	not, L1 or L2
DDR RAM	1 per SoC	up to 8G	about 90	not, L1 or L2

Table 3: KeyStone Memories

Note that there are also prefetch engines between the MSMC/DDR RAM and L1/L2 caches.

In general, free events are not yet associated with a C66x CorePac and the user should map event buffers to shared memory. In many cases, the event buffers would reside in a memory segment with caching enabled.

12.1 Cache Coherency and Memory Consistency

Parallel programmers want to know **who manages the coherency of the caches and the consistency of the memory**. The KeyStone architecture does not provide HW-managed full cache coherency. Therefore, cache coherency needs to be SW-managed and relaxed.

Whenever an event is sent, SW must ensure that the event buffers are written back to memory and invalidated in cache. Whenever an event is freed, SW must ensure that the event buffers are invalidated in cache. In addition, SW needs to think of raising memory fences and flushing prefetch buffers as well.

OpenEM always takes care of memory fences and prefetch buffers. It is up to the programmer whether he wants to manage the cache coherency of event buffers himself or if he wants to delegate this responsibility to OpenEM. The policy may differ from pool to pool, or even from event to event.

Not only event buffers, but also queue contexts require attention. Whenever atomic processing is complete, SW must ensure that the queue context is written back to memory and invalidated in cache. It needs to raise a memory fence as well. It is up to the programmer whether he wants to manage the cache coherency of queue contexts himself or if he wants to delegate this responsibility to OpenEM. The policy may differ from queue to queue.

The tables below summarize the coherency and consistency operations performed by OpenEM.

event operation	memory fence	cache invalidate	cache write-back	prefetch flush
allocate	never	never	never	never
send	always	optional	optional	never
receive	never	never	never	always
free	always	optional	never	never

Table 4: Cache coherency and memory consistency for event buffers

atomic operation	memory fence	cache invalidate	cache write-back	prefetch flush
notify end	always	optional	optional	never

Table 5: Cache coherency and memory consistency for queue contexts

Note that OpenEM does not yet take care of the cache invalidate/write-back of the queue context.

Cache coherency operations can be very costly, in particular for large buffers. OpenEM will skip cache coherency operations whenever possible: for instance, when a buffer has not been accessed, when the next consumer runs on the same core, ...

The programmer should ensure that all event buffers and queue contexts are aligned with a cache line. Otherwise, there is a risk of data corruption due to false sharing.

12.2 Pre-loading and Post-storing

OpenEM offers the option to pre-load the event buffers in local L2/L1 RAM. The user specifies for each event whether it needs to be pre-loaded.

Whenever the scheduler has scheduled an event with pre-loading enabled, it will push the event to a TX queue of the Packet DMA engine. The Packet DMA engine will allocate a local event and transfer the event payload from the global event buffers in MSMC/DDR RAM to a local event buffer in L2/L1 RAM and push the local event to an RX queue. The goal is to have the event payload in the local event buffer before the event is dispatched. As a result, the receive function will suffer many less read stalls.

Stale data in L1 cache is automatically updated during a transfer to L2 RAM. The major benefit is that there is no need to invalidate the local event buffer in the L1 cache. In addition, every read has a chance to hit the L1 cache. The read stalls will drop further and may ultimately completely disappear.

If the event payload is scattered, all global event buffers will be pre-loaded in a single local event buffer. This simplifies the event processing for sure. In addition, it is possible to pre-load only the start of the event payload. This is useful if the event payload does not fit into the local event buffer or if only the first part of the event payload is required for processing the event.

Post-storing allows transferring the event payload from a local event buffer in L2/L1 RAM to global event buffers in MSMC/DDR RAM. It will happen each time a local event is sent to a queue. Like for pre-loading, the benefits are many-fold: less (or no) write stalls and no overhead from write-back and invalidate operations.

Note: OpenEM does not yet support post-storing.

Pre-loading and post-storing are only supported by the run-to-completion dispatcher. Furthermore, OpenEM does not allow pre-loading or post-storing the queue context.

13 Seamless Integration

13.1 Operating Systems

OpenEM is able to operate in a heterogeneous OS environment: some cores may have SW running on the bare metal, others may have SW running on an OS and still others may have SW running on another OS. There will be only one dispatch loop on a bare metal core, but there may be one dispatch loop per OS thread on a core running an OS.

There are several reasons why the user would like to have an OS running on at least one of the cores. An OS would offer more services than OpenEM does (memory management, file systems, device drivers, ...). In addition, an OS brings pre-emption into the picture. In a pre-emptive system, a job of the highest priority starts running as soon as it is ready. In a run-to-completion or co-operative system, this job has to wait until a core frees up when a running job completes or yields. Two use cases need special attention:

- **Dispatch loop running in low-priority thread**

If the run-to-completion or co-operative dispatcher cannot guarantee that real-time deadlines are met for high priority events, the programmer may create a high-priority thread to execute those events. The OS will pre-empt the dispatch loop as soon as the run conditions of the high-priority thread are met.

- **Dispatch loop running in high-priority thread**
  If there are low-priority events that occupy a core for a long time, the programmer may create a low-priority thread to execute those events. It is the responsibility of the programmer to suspend the dispatch loop and block the dispatch thread when he observes event starvation. The OS will schedule the low-priority thread when the dispatch thread is blocked. The dispatch loop needs to resume when a new event is available. For that purpose, the scheduler generates an interrupt whenever it schedules an event. When a core receives an interrupt, the interrupt service routine should unblock the dispatch thread. The synchronous scheduler does not support this use case.

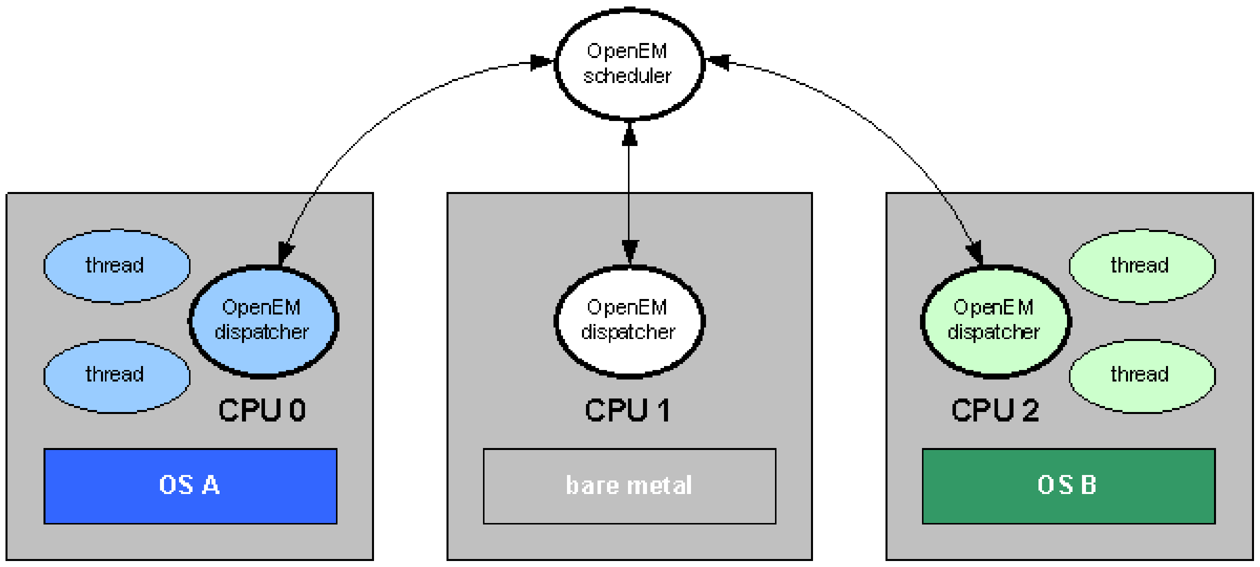


Figure 5: Integration with OS

Note: OpenEM does not yet support multiple dispatch loops per core.

13.2 Interfaces and Accelerators

OpenEM interworks seamlessly with CPPI-compliant interfaces and accelerators. CPPI is the protocol used by Packet DMA to transfer data between memory and HW. CPPI-compliant HW is able to allocate and send events without SW intervention. It is transparent for the event consumer whether the event has been produced by HW or SW. Likewise, CPPI compliant HW is able to receive and free events without SW intervention. It is transparent for the event producer whether the event will be consumed by HW or SW.

Note: KeyStone products may still have some interfaces and accelerators that are not CPPI-compliant.

14 Benchmarks

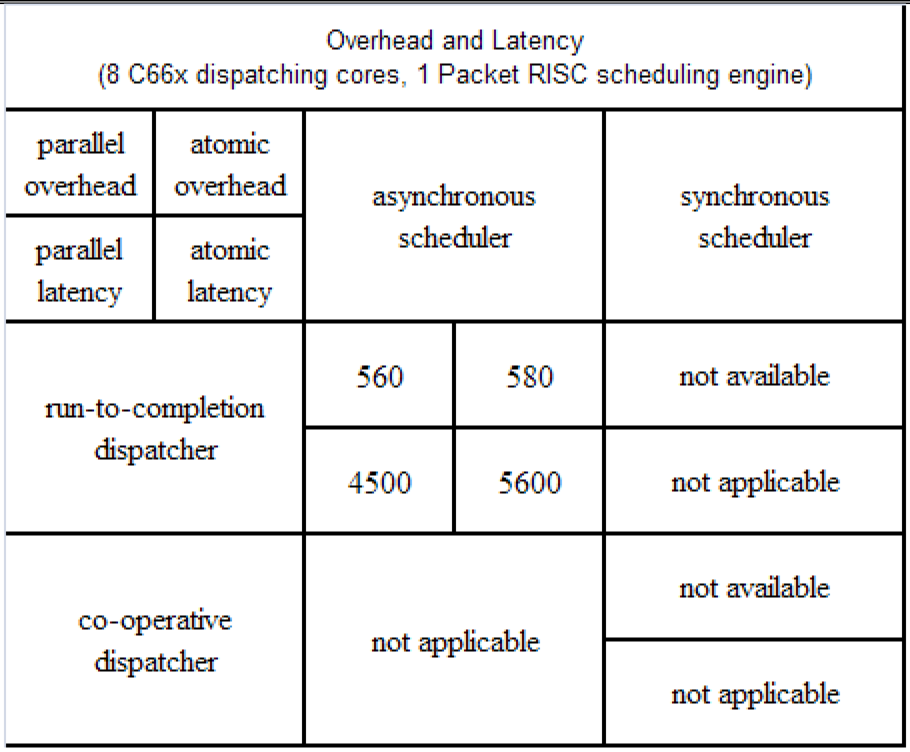
14.1 Micro Benchmarks

Micro benchmarks focus on the performance of OpenEM. There are two performance metrics:

- **Overhead**
  The overhead measures all C66x cycles spent by OpenEM during the live time of an event. It includes cycles for allocating, sending, dispatching and freeing the event. If the scheduler is synchronous, the overhead includes cycles for scheduling as well. Overhead caused by cache coherency operations is not included since it depends on the event payload sizes and the NUMA architecture. The overhead should be low compared with the event processing time. We will consider that the overhead is acceptable if it stays below 10% of the processing time.

- **Latency**
  If the scheduler is asynchronous, the overhead is not sufficient as performance metric. The latency measures the C66x cycles elapsed between the instant when the dispatcher requests the next event and the instant when the dispatcher calls the receive function. If the processing time of the current event is shorter than the latency, the dispatcher will stall. Therefore, the minimal processing time should be longer than the maximal latency. Note that the latency depends on the number of C66x cores running the dispatcher and the number of Packet RISC engines running the scheduler.

The table below shows the measured overhead and latency of the OpenEM implementation with the asynchronous scheduler running on one Packet RISC engine and the run-to-completion dispatcher running on eight C66x cores. Measurements for the synchronous scheduler or co-operative dispatcher are not yet available.



Overhead and Latency (8 C66x dispatching cores, 1 Packet RISC scheduling engine)				
parallel overhead	atomic overhead	asynchronous scheduler		synchronous scheduler
parallel latency	atomic latency			
run-to-completion dispatcher		560	580	not available
		4500	5600	not applicable
co-operative dispatcher		not applicable		not available
				not applicable

Table 6: Overhead and Latency

The table indicates that **OpenEM is able to efficiently support event processing times of 6K cycles or more**.

14.2 Macro Benchmarks

Macro benchmarks focus on the performance of an application running with OpenEM. The performance metric is the speedup, which is defined as follows.

$$speedup = \frac{sequential\ execution\ time}{parallel\ execution\ time}$$

We assume an ideal sequential execution without memory read/write stalls, which implies that all data is in L2 RAM. However, the parallel execution will be impaired by four sources of degradation:

- **OpenEM overhead**: cycles spent by OpenEM during the live time of an event (see 14.1).
- **OpenEM stalls**: cycles spent by OpenEM dispatcher waiting for the next event (see 14.1).
- **NUMA overhead**: cycles spent to maintain cache coherency and memory consistency or cycles spent to pre-load and post-store (see 12).
- **Memory stalls**: cycles spent waiting for read data or write status.

The figure below shows the multi-core speedup for a synthetic application. The application has three parameters: the event execution time, the amount of input data and the amount of output data. The application uses eight C66x dispatching cores and one Packet RISC scheduling engine to processes 1024 events. Pre-loading transfers the input data from DDR RAM to L2 RAM and post-storing (implemented outside OpenEM for the time being) transfers the output data from L2 RAM to DDR RAM.

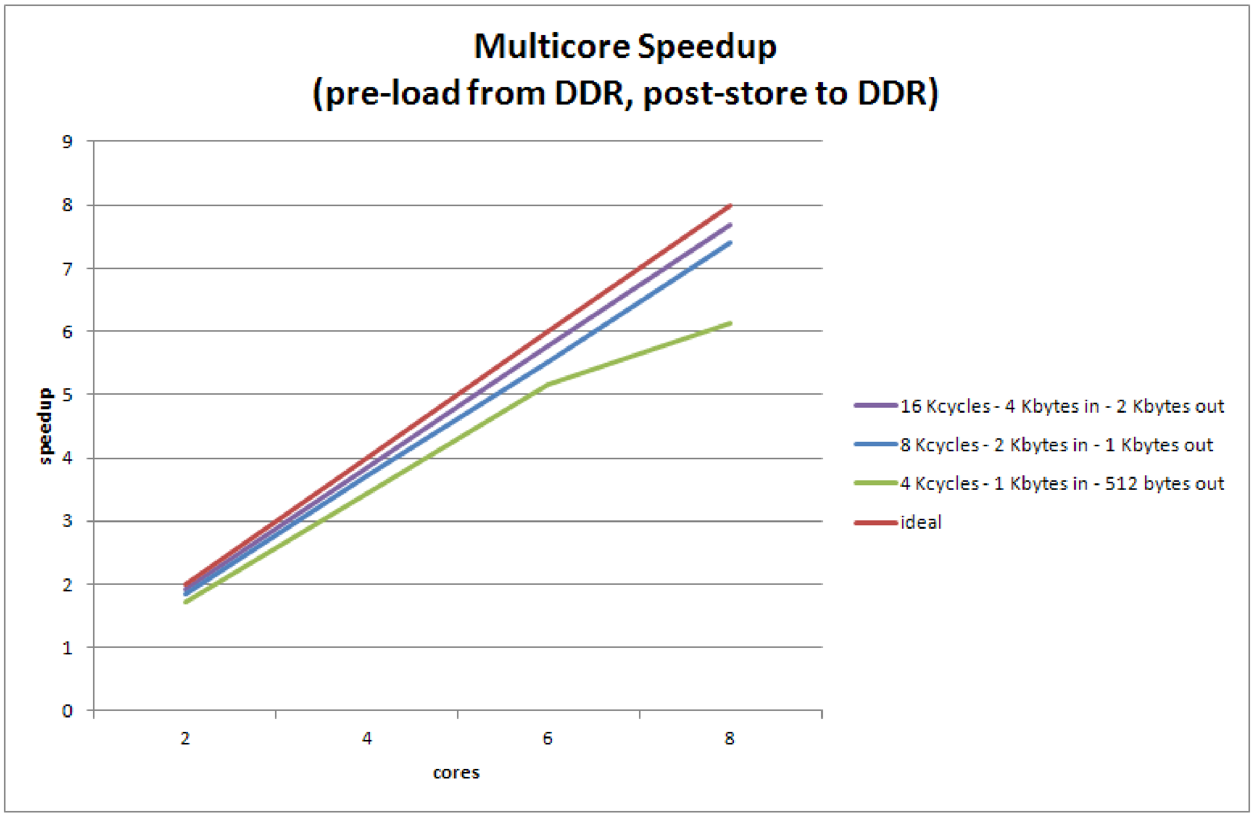


Figure 6: Multicore Speedup

The figure illustrates that OpenEM enables a **close-to-ideal (90%) multi-core speedup for applications that process an event in 8K cycles, consume 2K bytes of input data and produce 1K bytes of output data**. As expected, the multi-core speedup will degrade if the event processing time decreases, but also if the amount of data consumed or produced increases. The former is due to the OpenEM overhead and stalls, the latter due to the NUMA overhead and memory stalls. When lots of data are consumed or produced, the Packet DMA throughput becomes the bottleneck and the NUMA overhead may be lower if pre-loading and post-storing are switched off.

15 Conclusion

OpenEM is a true multi-core runtime system offering dynamic load balancing. It comes with an asynchronous scheduler and run-to-completion dispatcher considering priority, atomicity, locality and order. A synchronous scheduler and co-operative dispatcher are planned in order to address a wide variety of concurrency patterns.

The provided services are low-level, but don't prevent building higher-level runtime systems above OpenEM. Prototypes of OpenMP and OpenCL runtime systems are currently under development.

OpenEM is able to manage lots (thousands) of events with low (less than 600 cycles) overhead. As such, it can achieve a high (more than 90%) multi-core efficiency for events taking as few as 6000 processing cycles.

An application relying on OpenEM services can be deployed on a variable number of cores without changing the application code. The programmer can opt for static load balancing for an early deployment and smoothly move to dynamic load balancing for later deployments. Easy porting between different KeyStone devices is guaranteed.

OpenEM understands the inherent challenges of an embedded environment. It runs on the bare metal as well as with a mix of Operating Systems. It deals with a Non-Uniform Memory Architecture and allows interworking with various HW producers and consumers.