M2OS-Mc: An RTOS for Many-Core Processors

David García Villaescusa
University of Cantabria, Santander, Spain
garciavd@unican.es

Mario Aldea Rivas
University of Cantabria, Santander, Spain
aldeam@unican.es

Michael González Harbour
University of Cantabria, Santander, Spain
mgh@unican.es

Abstract
A current trend of industrial systems is reducing space, weight and power (SWaP) through the allocation of different applications on a single chip. This is enabled by the continued improvement of semiconductor technology which allows the integration of multiple cores in a single processor chip, as the processors are prevented to continue increasing their clock rate due to the “power-wall”. The use of Commercial-Off-The-Shelf (COTS) multi-core processors for real-time purposes presents issues due to the shared bus used to access the shared memory. An alternative to the use of multi-core processors are the many-core processors with tens to hundreds of processors in the same chip, using different scalable ways to interconnect their cores. This paper presents the adaptation of the M2OS Real-Time Operating System (RTOS) and its simplified Ada run-time for mesh-based many-core processors. This RTOS is called M2OS-mc and has been tested on the Epiphany III many-core processor (referred in this paper simply as Epiphany), a many-core which has 16 cores connected by a Network-on-Chip (NoC) consisting of a 4x4 2D mesh. In order to have a synchronized way to send messages between tasks through the NoC independently of the core where they are being executed, we provide sampling port communication primitives.

1 Introduction
In the past, the evolution of processors was mostly related to frequency improvement but since the processors reached a power consumption too high to dissipate, the designers have been improving the processor’s performance by having more processing cores executing in the same chip: the multi-core era begun. Multi-cores provide not only better energetic efficiency but a greater performance-per-cost. The applications can be parallelized, being divided into sections that can be executed simultaneously, to take advantage of all of cores in the same multi-core chip.

Multi-core processors with few cores have a shared bus for communications among their cores and the shared memory, as shown in Figure 1. When the number of cores increases, the shared bus becomes a bottleneck and different communication strategies are used. In these processors with a high number of cores, called many-cores, a common alternative is the use of a Network-on-Chip (NoC) based on a 2D mesh, as shown in Figure 1 for the Epiphany

processor [16]. This network has a predictable delay on the communications between two neighbor cores. It also requires less wires than a shared bus and its power consumption is linear with the number of cores. Other NoC topologies have been proposed in other architectures like the torus of Kalray [6] or the ring used by Intel [12].

![Figure 1](https://example.com/figure1.png)  
**Figure 1** Generic multi-core topography (left), *Epiphany*’s topography (middle) and eCore architecture (right).

The many-core mesh connects the tiles using the typical configuration shown in Figure 1, with the core, its local memory and a router connecting the tile’s core with the neighbor routers in the mesh.

As the many-core processors seem to be the immediate future of COTS processors, there is a need to have suitable software platforms that allow the execution of hard real-time applications on such architectures. In order to fulfill that requirement, this paper presents the port of the M2OS RTOS to *Epiphany*, a many-core which has 16 cores connected by a 4x4 2D mesh, with each core having a 32-kilobyte local memory.

M2OS is a small and efficient real-time kernel supporting the non-preemptive one-shot task model [17] [1]. The implementation presented in this paper follows the multikernel paradigm [4], with a different RTOS image running in each core. The small footprint of M2OS makes its adaptation to this architecture feasible. M2OS has been ported to the *Epiphany* many-core processor, in such a way that the resulting M2OS-mc is aimed at running in any other 2D-mesh many-core platform with a minimum amount of changes. We have developed a mechanism to allow exchanging messages between the different tasks of the system, independently of the core where they are executing. This mechanism uses the sampling port implementation presented at Section 5.

M2OS-mc, as well as M2OS, is written in Ada as this language has specialized features supporting low-level real-time, safety-critical and embedded systems programming  

1 https://en.wikibooks.org/wiki/Ada_Programming

A typical Ada application executed on M2OS in the *Epiphany* processor is composed of several tasks running in the different cores, with one or more tasks in each core. Tasks in the same core are executed under the one-shot non-preemptive scheduling policy implemented by M2OS. The tasks’ messages between tasks allocated in different cores will travel through the NoC.

To take advantage of the parallel architecture of the underlying many-core, the response to an external event is typically performed by several tasks (running on the same or different cores) which are activated in sequence (a task’s predecessor activates the next task in the sequence and provides its input data).
M2OS alongside M2OS-mc are available on-line at the website \(^2\), and are distributed under a GPL license.

The paper continues by analyzing the related work in Section 2. In Section 3 the Epiphany processor is introduced. Section 4 discusses the properties of M2OS and exposes its adaptation to a many-core. Synchronization and message exchange between tasks are described and evaluated in Section 5. Finally, Section 6 shows the paper conclusions and future work. An appendix is included with the sampling port interface and a consumer-producer test code.

## 2 Related work

The NoC concept is not something new. It was already presented by Benini in 2002 [5] and it soon got the real-time community’s attention [10].

There have been some projects that brought many-core platforms and real-time operating systems together:

- P-SOCRATES [9], whose purpose is to develop an entirely new designed framework from the conceptual design of the system functionality to its physical implementation, to facilitate the deployment of standardized parallel architectures in all kinds of systems. The tasks follow the OpenMP task model. From that project Erika3, from Erika Enterprise [7], has been developed. It is an RTOS that uses a single image per computer cluster and has a memory footprint of just a few kB. This RTOS runs in the Kalray MPPA-255.

- eSol has developed a many-core real-time OS called eMCOS [8] with a distributed micro-kernel architecture implemented. This micro-kernel is allocated at the cores with minimal functions while more advanced operations are performed through server cores. It claims to support a wide variety of architectures in which Epiphany is not included.

- Altamary ported RTEMS [2] for the Epiphany processor on a Parallella board similar to the one used in this project. As we will see later, the Parallella board has both local memory for each core and a global shared memory. This modified version of RTEMS can be placed in both types of memory. When RTEMS is placed in the shared memory the system is significantly slower than when it is placed in local memory. However, in the latter case it only leaves 5kB for the applications. RTEMS is a relatively complex RTOS that implements several scheduling algorithms.

Several studies have been done for theoretical 2D mesh NoCs [11] [13]. These studies perform scheduling analysis using theoretical many-core processors. With the availability of an RTOS such as M2OS-mc more realistic scheduling analysis could be carried out in the future.

## 3 Epiphany

The Epiphany processor is integrated in the Parallella [14] development board which has the size of a credit card and needs just 5W to work. Apart from the Epiphany processor it also has an ARM dual-core processor (Zynq), which is the central processor on the Parallella board. It combines an ARM dual-core Cortex-A9 with Xilinx programmable logic. Zynq has an Ubuntu adaptation (Parabuntu) that is used as operating system. The Parabuntu OS is used to send the executables to the Epiphany cores (eCores) and it also starts each eCore execution.

\(^2\) https://m2os.unican.es
The *Epiphany* processor is a many-core designed by Adapteva with 16 cores connected by a NoC placed in a 4x4 2D mesh as Figure 1 shows, where every square is a tile that contains a router connected to the neighbor tiles and the execution core. Each core of the *Epiphany* is an *eCore*, whose architecture is also designed by Adapteva, that executes its instructions in order, with a frequency of 600 MHz. It consists of an integer ALU, floating-point unit, a debug unit an interrupt controller, a general purpose program sequencer and a 64-word general purpose register file. Each core has 32kB of local memory. The architecture is supported by the GCC compiler and has libraries for OpenMP and MPI.

The design could grow as it has been shown with a 1024 cores version [15]. Unfortunately, the *Epiphany V* is not available in any development board.

Any *eCore* can access the local memory of the rest of the *eCores* using a range of special global addresses. The synchronized message interface explained in Section 5 takes advantage of that. An *eCore*’s local memory can be written and read without any hardware limitations but the memory size. The process of writing to another *eCore*’s memory is 8 times faster than reading. This is due to the fact that the *Epiphany* processor has independent networks for reading and writing between cores and the one used for writing is much faster.

The *Parallella* board has a shared memory that can be accessed by the *eCores*. This memory access is much slower than a memory access between *eCores* so this method is considered too slow, although it could be useful for other functionalities as it is shown in Section 4.5.

It can be said that the *Parallella* board is a good platform for experimenting with the development of RTOS for mesh-based many-cores.

# 4 M2OS

M2OS [17] [1] is a small real-time operating system that allows running multitasking applications in small microcontrollers with scarce memory resources. This is the case of the *Epiphany* processor, where each of its *eCores* has a 32 kB local memory.

M2OS implements a simple scheduling policy based on non-preemptive one-shot tasks, which requires a very small memory footprint. This policy allows the same stack area to be shared by all the tasks and, consequently, the system only needs to allocate a stack area large enough to fit the largest task stack.

M2OS is written in Ada and it is the base of a simplified Run-Time System for the GNAT Ada compiler. This RTOS has been developed for *Arduino Uno* and STM32F4. M2OS is intended to be easily ported to different platforms. All the hardware dependent part is encapsulated in a Hardware Abstraction Layer (HAL), which is the only code that has to be modified to port the kernel to a new platform.

The new HAL written for the *Epiphany* uses the *Epiphany Library* (e-lib) to perform low-level functions that are specific to this architecture, such as interrupt and timer management. An Ada interface has been implemented for those functions of the e-lib library that are required by the M2OS kernel. As a result, the e-lib library must be included in the linking instruction.

A deeper analysis the *Epiphany*’s implementation will now be exposed.

## 4.1 Building and loading the application

M2OS is an RTOS written in Ada so taking advantage that the *eCore* architecture is supported by the GNU compiler collection gcc, we have compiled it for the *eCore*, therefore achieving support for the Ada and C languages.
One executable file is generated for each of the eCores. The executable file generated includes both the M2OS and the user code. This executable must be loaded into the different eCores by the Zynq processor. Each eCore starts its execution individually when the Zynq processor sends the corresponding signal.

The linker script used to build the M2OS applications places all the data and code in the local memory of the eCores.

A set of scripts has been produced to automate the cross-compilation of the applications that will run under M2OS and to load the generated executables to the Parallela board.

4.2 HAL

The HAL of M2OS has been implemented for the eCore’s architecture. This layer includes the basic support for context switch, interrupt and hardware timer handling.

- **Context switch.** Under the simple scheduling policy implemented in M2OS the context switch only requires resetting the Stack Pointer to the base position and setting the program counter.
- **Interrupts.** The global interrupts can be enabled, disabled and checked for their status. This implementation was developed thanks to the e-lib library.
- **Core identification.** The e-lib provides primitives for core identification. This service was not included in the M2OS HAL because it is specific of architectures with more than one core. It is part of M2OS-mc now.
- **Spinlock.** The e-lib provides spinlocks to be used among the different cores for non-blocking mutual exclusion synchronization (called “mutex” in the e-lib terminology).
- **System timer.** It follows the “ticker” approach that requires the periodic programming of a hardware timer. In our implementation one of the two eCore’s timers is used to generate an interrupt each 1ms. This interrupt is used to account for the system time. The timer, driven by a 600MHz clock, can only be programmed in one-shot mode, which requires it to be reprogrammed at each execution of the interrupt handler.
- **System clock.** It stores a counter of each system timer interrupt in a 32 bit integer. It has a 1 ms resolution.
- **High precision clock.** Our implementation of M2OS in Epiphany provides a high precision clock by reading the actual value of the hardware timer. This clock has a precision of 1.667 ns and is suitable for intervals up to 1ms (when the system timer resets the value).

4.3 Clock synchronization

*Epiphany* applications are launched from the Zynq processor by loading the application code corresponding to each eCore and sending, sequentially, the start signals to the different eCores of the system. In consequence, each eCore starts its execution at a different instant. For a real-time operating system the timer synchronization at each component is very useful for time awareness, to avoid having significant timer gaps between tasks executing at different eCores. For this purpose M2OS synchronizes all the timers during the start up of the RTOS.

This clock synchronization process is conducted by a master eCore, which is the last one to be started (the 0x0 eCore in the current version). Upon initialization, every other eCore has to wait for the master to send a message containing the value of its timer. After receiving this synchronization message, each eCore updates its own timer with the received value plus the time the message needs to be generated and transmitted through the NoC and the time spent by the eCores executing the required instructions.
4.4 Performance metrics

Different tests are done to measure various mechanisms implemented in a single eCore, which are time measurement, context switch, application size and mutex usage.

The tests in this section execute the required actions a thousand times. This number was chosen to achieve short execution times in which there was no interference from the system timer, which produces an interrupt every millisecond.

- **Reading the clock.** Knowing the time needed for reading the clock is required to get more precise times for the rest of the tests. The result of this test is shown in Table 1. Since the minimum time to read the clock is 81 cycles, from this point we have subtracted this value from all the measurements involving the clock.

- **Mutex.** The time required to lock or release a mutex is constant, as shown in Table 1.

- **Context Switch.** The time needed to perform a context switch on an eCore is calculated with an M2OS generic test run in the M2OS-mc. The results when using a delay until operation are shown in Table 1. The context switch has been tested in depth divided in activation and suspension tests, as shown in Table 2. The set of tests consists of:
  - **Activation tests.** Latency since one task opens a suspension primitive and suspends itself until the activated task executes (suspension object, protected object entry without parameters or protected object entry with one parameter).
  - **Suspension tests.** Latency since a task suspends itself until another task executes. Times are measured for different suspension primitives (delay until, suspension object, protected object entry without parameters or protected object entry with one parameter).

- **Application size.** The output of the size Linux command for 2 applications, one with 6 periodic tasks and another one with 2 periodic tasks is shown in Table 3. Each of those tasks just put a message on the console, set a boolean to true, calculate the time of the next activation and delay until that time. It can be seen that the amount of tasks has a small impact on the size of the application.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Latencies for reading the clock and operating a mutex.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>Max</td>
</tr>
<tr>
<td>Clock Read</td>
<td>81 cycles</td>
</tr>
<tr>
<td>Lock Mutex</td>
<td>211.7 ns</td>
</tr>
<tr>
<td>Release Mutex</td>
<td>133.4 ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Context switch tests.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation Tests</td>
<td>Max</td>
</tr>
<tr>
<td>Suspension object</td>
<td>593.5 ns</td>
</tr>
<tr>
<td>Protected object entry without parameter</td>
<td>698.5 ns</td>
</tr>
<tr>
<td>Protected object entry with one parameter</td>
<td>736.8 ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suspension Tests</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay until</td>
<td>596.8 ns</td>
<td>596.8 ns</td>
<td>596.8 ns</td>
</tr>
<tr>
<td>Suspension object</td>
<td>345.1 ns</td>
<td>345.1 ns</td>
<td>345.1 ns</td>
</tr>
<tr>
<td>Protected object entry without parameter</td>
<td>540.1 ns</td>
<td>433.4 ns</td>
<td>453.4 ns</td>
</tr>
<tr>
<td>Protected object entry with one parameter</td>
<td>548.4 ns</td>
<td>548.4 ns</td>
<td>548.4 ns</td>
</tr>
</tbody>
</table>
Table 3 Results of the size command for two applications with 6 and 2 tasks, respectively.

<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>10914</td>
<td>1244</td>
<td>528</td>
<td>12686</td>
<td>318e</td>
<td>six_tasks</td>
</tr>
<tr>
<td>data</td>
<td>10226</td>
<td>1244</td>
<td>208</td>
<td>11678</td>
<td>2d9e</td>
<td>two_tasks</td>
</tr>
</tbody>
</table>

4.5 Console

The console output in M2OS is performed by the console driver, which has to be implemented for each architecture M2OS is ported to. In the Parallella board the system console is managed by the Zynq processor. The eCores do not have direct access to the system console.

The solution adopted is that every eCore writes in a reserved local memory space that is read by the Zynq processor. The reserved memory space of every eCore will be used as a circular buffer into which Put_Line commands write text. The buffer is designed such that a line is never divided. When the final address of the designated area is reached, the next write operation will be done at the beginning of its reserved region, erasing the oldest line or lines. In that way we emulate the behavior of a console. No console input has been implemented.

The console output is thereby printed in the user’s terminal by a specifically-developed software executed at the Zynq, which shows the eCores consoles content by reading the fixed local memory of each eCore where the console driver writes the desired console output.

In case the Zynq tries to read from the memory assigned to a non-initialized eCore it gets content lacking any meaning but the system will not crash.

5 Inter-task messages

A typical application running on the many-core processor consists of a number of end-to-end flows (e2e). Each e2e is a set of tasks (in the same or different eCores) that responds to the same periodic or sporadic event. These tasks must have a way to communicate between them and a mechanism for waking up the next task in the flow at the end of each execution. This requires a way to communicate between tasks in different eCores in a synchronized manner.

The chosen synchronization mechanism is inspired in the ARINC’s sampling ports (SP) and the Ada implementation performed by Garrido [3]. Only one message can be held at a determined sampling port. This means that any new message written in a sampling port will overwrite the previous stored message. Several sampling ports can be mapped at the same eCore.

In order to avoid a task from blocking the eCore we advise application developers to make a periodic polling using a delay until operation that allows other tasks to use the eCore while polling the sampling port. The polling period must be considered to calculate the response time. An example of this approach is shown in Listing 3 in Appendix A.

5.1 Implementation

The implemented synchronized messages take advantage of the fact that the local memory of any eCore is accessible from every other eCore.

In our implementation there is a fixed number of sampling ports per eCore, configurable at system configuration time. They are implemented as an array of sampling port records that are placed in the same predefined memory location of each eCore’s local memory. Each sampling port is identified by the identifier of the core where it is allocated and its index in the array.
The sampling port record includes the followings fields:

- **Init**. Boolean used to know if the sampling port has been initialized or not.
- **Mutex**. The spinlock to protect the content.
- **Size**. The size of the protected content.
- **Addr**. Where the content is located using a global memory address of the eCore’s memory.
- **New**. Flag to know whether the content has been modified since the last read operation.
- **Core**. Core where the SP is initialized. This is required by the mutex.

M2OS initializes the Mutex of each sampling port and sets its Init field to False. The access to the sampling port is protected by the spinlock. Any operation on the sampling port must lock the spinlock and release it afterwards.

To wake-up a task, the sampling port used for that purpose must be polled periodically by the task, waiting for the New field to be set to True.

This implementation requires 32 bytes per sampling port, in addition to a user-allocated memory area for the message.

### 5.2 Interface

The interface developed for the sampling ports is shown at Listing 1 in Appendix A. It provides the following functions:

- **Init_Sampling_Port**. This function returns the identifier of the initialized sampling port. It receives the host eCore, the sampling port index at the eCore and the address and size of the shared data. The function initializes the port causing its state to transition from the “initial” state to the “empty” state, passing through the “locked” state. All these states are shown in Figure 2. If the sampling port is already initialized it returns Null_SP_Id. No writing or reading operation over the sampling port are allowed until this initialization is performed, and consequently those operations will return an error indication in that case.
Get_Sampling_Port. A function that returns the identifier of a *sampling port* given the host *eCore* and its index identifier. If the *sampling port* has not been initialized it returns *Null_SP_Id*.

Write_Sampling_Port. A procedure that writes the shared data of the *sampling port* parameter. The content is marked as new. The content to be copied into the shared data is located at the address given as a parameter and has the size also indicated as a parameter. This procedure performs the transition of the *sampling port* from the “empty” (or “new data”) state to the “locked” state and then to the “new data” state. The states are shown in Figure 2. If the write operation has been successful the output parameter is set to *True*. It will be set to *False* (when trying to write into an “initial” state *sampling port* or when the size of the item is larger than the size defined at the *sampling port*).

Read_Sampling_Port. This procedure copies the shared content of a *sampling port* into an address supplied as a parameter. The field *New* is set to *False*. During the read operation the *sampling port* is in the “locked” state passing to the “empty” state once the operation finishes. If the read has been successful the output parameter is set to *True*, it will be set to *False* otherwise (when trying to read from an uninitialized *sampling port*, when the size defined at the *sampling port* is larger than the size of the item where the value is going to be written, or when no data is written yet).

5.3 Usage example

The functionality of the *sampling ports* is described with an example that follows the typical producer/consumer pattern. The producer is a periodic task that produces a data item and writes it in a *sampling port*. This *sampling port* is used by the consumer task to wait for new data and process it.

The consumer, shown at Listing 2 in Appendix A, declares and initializes the data to be shared and then initializes the *sampling port*. Thereafter it continuously iterates waiting for a new value to arrive at the *sampling port* and consuming it. The wait operation is implemented as a periodic poll of the *sampling port*.

The producer, shown at Listing 3 in Appendix A, must wait until the *sampling port* is initialized. This is done by periodically polling the *sampling port* until it is initialized. Then it periodically iterates producing a new data item and writing it to the *sampling port*. New content is written to the *sampling port* regardless of whether the previous content has been read or not.

In this example it can be noticed that the consumer is the one that initializes the *sampling port*, because it is located in its own *eCore* and, since writing through the NoC is eight times faster than reading, we decided to implement the most efficient model.

5.4 Tests

M2OS has a battery of tests that have been successfully passed for the *Epiphany* architecture. These tests include stack management, scheduling, timing events and task handling. To this battery set, we have added other tests such as measuring the latency of sending messages between tasks through the network. These tests will be analyzed in the following lines.

Table 4 shows the time required to execute the operations described in Section 5.2. The times for the Get_Sampling_Port and the Write_Sampling_Port operations are measured for a *sampling port* allocated in an *eCore* at one hop distance from the calling task. It can be seen that both writing and reading a *sampling port* have a linear increment in relation to the size of the message.
Table 4 Sampling port latencies for different message sizes.

<table>
<thead>
<tr>
<th>Latency of</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init_Sampling_Port</td>
<td>556.8 ns</td>
<td>556.8 ns</td>
<td>556.8 ns</td>
</tr>
<tr>
<td>Get_Sampling_Port</td>
<td>146.7 ns</td>
<td>146.7 ns</td>
<td>146.7 ns</td>
</tr>
<tr>
<td>Write_Sampling_Port (8 bytes)</td>
<td>596.8 ns</td>
<td>596.8 ns</td>
<td>596.8 ns</td>
</tr>
<tr>
<td>Read_Sampling_Port (8 bytes)</td>
<td>873.5 ns</td>
<td>873.5 ns</td>
<td>873.5 ns</td>
</tr>
<tr>
<td>Write_Sampling_Port (40 bytes)</td>
<td>1247 ns</td>
<td>1237 ns</td>
<td>1237 ns</td>
</tr>
<tr>
<td>Read_Sampling_Port (40 bytes)</td>
<td>2364 ns</td>
<td>2364 ns</td>
<td>2364 ns</td>
</tr>
</tbody>
</table>

The process of sending a message requires locking and unlocking a remote mutex. The latencies therefore depend on the latencies in the NoC, which in turn depend on the distance, in hops, the message needs to travel. A test sending an 8-byte message comparing different locations and different hop distances is shown in Table 5, where it can be seen that the timing is also linear in relation to the distance. We have performed the same test for reading a message from another eCore placed at different distances.

Table 5 Write_Sampling_Port and Read_Sampling_Port latencies for different hop distances in the NoC.

<table>
<thead>
<tr>
<th>Write</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
<th>Read</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 hop</td>
<td>596.8 ns</td>
<td>596.8 ns</td>
<td>596.8 ns</td>
<td>1 hop</td>
<td>873.5 ns</td>
<td>873.5 ns</td>
<td>873.5 ns</td>
</tr>
<tr>
<td>2 hops</td>
<td>631.8 ns</td>
<td>631.8 ns</td>
<td>631.8 ns</td>
<td>2 hops</td>
<td>958.5 ns</td>
<td>958.5 ns</td>
<td>958.5 ns</td>
</tr>
<tr>
<td>3 hops</td>
<td>666.8 ns</td>
<td>666.8 ns</td>
<td>666.8 ns</td>
<td>3 hops</td>
<td>1044 ns</td>
<td>1044 ns</td>
<td>1044 ns</td>
</tr>
<tr>
<td>4 hops</td>
<td>701.8 ns</td>
<td>701.8 ns</td>
<td>701.8 ns</td>
<td>4 hops</td>
<td>1129 ns</td>
<td>1129 ns</td>
<td>1129 ns</td>
</tr>
<tr>
<td>5 hops</td>
<td>736.8 ns</td>
<td>736.8 ns</td>
<td>736.8 ns</td>
<td>5 hops</td>
<td>1214 ns</td>
<td>1214 ns</td>
<td>1214 ns</td>
</tr>
<tr>
<td>6 hops</td>
<td>771.8 ns</td>
<td>771.8 ns</td>
<td>771.8 ns</td>
<td>6 hops</td>
<td>1299 ns</td>
<td>1299 ns</td>
<td>1299 ns</td>
</tr>
</tbody>
</table>

In order to complete the performance analysis of the sampling ports, a round-trip scenario has been created. This scenario involves two cores with one task and one sampling port in each. The task in the first core sends a message to the sampling port allocated in the second core. A task waiting for that message sends it back to the sampling port allocated in the initial core. Table 6 shows the latencies measured for this round trip for different distances between the cores. In this test, when either of the two tasks has to wait for a message sent through a sampling port it does so by spinning continuously, so that there are no context switches or delays.

We can see that there is a dependency on the size of the message, an that the dependency on the distance between the eCores is linear.

Table 6 Round-trip latencies for messages of 8 bytes (left) and 40 bytes (right).

<table>
<thead>
<tr>
<th>8 bytes</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
<th>40 bytes</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 hop</td>
<td>2794 ns</td>
<td>2322 ns</td>
<td>2777 ns</td>
<td>1 hop</td>
<td>5846 ns</td>
<td>5568 ns</td>
<td>5615 ns</td>
</tr>
<tr>
<td>2 hops</td>
<td>2787 ns</td>
<td>2366 ns</td>
<td>2776 ns</td>
<td>2 hops</td>
<td>5841 ns</td>
<td>5579 ns</td>
<td>5603 ns</td>
</tr>
<tr>
<td>3 hops</td>
<td>2904 ns</td>
<td>2479 ns</td>
<td>2882 ns</td>
<td>3 hops</td>
<td>6051 ns</td>
<td>5720 ns</td>
<td>5768 ns</td>
</tr>
<tr>
<td>4 hops</td>
<td>2944 ns</td>
<td>2559 ns</td>
<td>2936 ns</td>
<td>4 hops</td>
<td>6160 ns</td>
<td>5786 ns</td>
<td>5845 ns</td>
</tr>
<tr>
<td>5 hops</td>
<td>3062 ns</td>
<td>2656 ns</td>
<td>3036 ns</td>
<td>5 hops</td>
<td>6161 ns</td>
<td>5770 ns</td>
<td>5866 ns</td>
</tr>
<tr>
<td>6 hops</td>
<td>3132 ns</td>
<td>2731 ns</td>
<td>3097 ns</td>
<td>6 hops</td>
<td>6190 ns</td>
<td>5891 ns</td>
<td>5903 ns</td>
</tr>
</tbody>
</table>
Conclusions and future work

We have ported the M2OS to the Epiphany many-core, with an implementation designed to allow the adaptation for other many-cores. We have also implemented a synchronization mechanism inspired on the ARINC-653 sampling ports.

The resulting port, called M2OS-mc, has passed the whole battery of tests included in M2OS, so we can conclude the port works. Tests to check the functionality and performance of the developed synchronization mechanism have also been developed and the system has passed them. The performance metrics done for the sampling ports shows an acceptable efficiency.

At the current stage, M2OS-mc is a fully functional prototype however, our intention is to continue its development in several aspects:

- Develop a new kind of synchronization port inspired on the ARINC queuing ports. This kind of port implements a fixed-size queue of data and allows a consumer task to suspend on an empty port until new data is written, which would avoid the need for polling.
- Develop a system model that allows us to perform a schedulability analysis of the applications using M2OS-mc for Epiphany.
- Develop task allocation algorithms that allow us to improve the response times of the end-to-end flows that form the applications.
- Extend the application model by allowing some eCores to execute parallel workloads programmed with OpenMP while other eCores execute the real-time tasks.

References


### A Listings

```vhdl
type SP_Index is range 0 .. Sampling_Ports_Per_Core-1;
type SP_Id is private;

function Init_Sampling_Port (C : in E_Lib.Core; Id : in SP_Index;
                           Addr : in System.Address; Size : Interfaces.Unsigned_32)
  return SP_Id;

function Get_Sampling_Port (C : in E_Lib.Core; Id : in SP_Index)
  return SP_Id;

function Write_Sampling_Port (SP : in SP_Id;
                             Orig : in System.Address; Orig_Size : in Interfaces.Unsigned_32)
  return Boolean; -- Successful

procedure Read_Sampling_Port (SP : in SP_Id;
                          Dest : in System.Address; Dest_Size : in Interfaces.Unsigned_32;
                          Successful : out Boolean; Is_New : out Boolean);
```

**Listing 1** Sampling port Interface
task Consumer is
   -- Declare and initialize data
begin
   SP := Init_Sampling_Port (Current_Core, SP_Index, Data'Addr, Data'Size);
   loop
      loop
         Read_Sampling_Port (SP, Data'Address, Data'Size, Success, Is_New);
         if not Success then
            -- Error;
         end if;
      end loop
      Next_Polling_Period := Next_Polling_Period + Period;
      delay until Next_Polling_Period;
      -- Consume data
   end loop
end Consumer;

Listing 2 Consumer

task Producer is
begin
   loop
      SP := Get_Sampling_Port (Core_Target, SP_Index);
      exit when SP /= Null_SP_Id;
      Next_Polling_Period := Next_Polling_Period + Period;
      delay until Next_Polling_Period;
   end loop;
   loop
      -- Produce new data
      Write_Sampling_Port (SP, Data'Address, Data'Size, Success);
      if not Success then
         -- Error
      end if;
      Next_Activation := Next_Activation + Task_Period;
      delay until Next_Activation;
   end loop;
end Producer;

Listing 3 Producer