The EVI Distributed Shared Memory System

Farshid Besharati, Mahdad Davari, Christian Danheimer Furedal, Björn Forsberg, Niklas Forsmark, Henrik Grandin, Jimmy Gunnarsson, Engla Ling, Marcus Lofvars, Sven Lundgren, Luis Mauricio, Erik Norgren, Magnus Norgren, Johan Risch, Christos Sakalis, Stefanos Kaxiras

Department of Information Technology, Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

Abstract

With the data handled by companies and research institutes getting larger and larger every day, there is a clear need for faster computing. At the same time, we have reached the limit of power consumption and more power efficient computing is also called for, both in the datacenter and in the supercomputer room. For that, there is a great push, both in industry and academia, towards increasing the amount of computing power per watt consumed.

With this shift towards a different computing paradigm, many older ideas are looked upon in a new light. One of these is the distributed shared memory (DSM) systems. It is becoming harder and harder to achieve higher performance and better power efficiency at the same form factor as we have always had. Furthermore, while we have seen a stop in the constant increase of processor speeds, there is a constant increase in network communication speeds. Software implemented DSM is again a viable solution for high performance computing, without the need for sacrificing ease of programming for performance gains.

The goal of this course was to develop such a system, and learn in the process. We chose to work with the Adapteva Parallella boards and design a DSM system there. Over one semester we designed and developed that system.

Contents

1 Introduction

Large companies that handle large amounts of data are constantly looking to find new, more efficient, and cheaper ways to handle all the requests and data that are processed every day. Currently, this is most often handled through interlinked servers working in unison in large data centers. Due to the market interest in this and the potential knowledge gain, the aim of this project is to implement a distributed shared memory system (DSM) using several Parallella boards, each containing 16 cores, to create a power efficient, low cost supercomputer.

The goal was to configure several Parallella boards to form a DSM system which should simulate a unified memory that works as a single set of cache and physical memory, shared by all the cores. The goal was also to create an abstract layer that gives the user the impression of having access to all the cores automatically when running a multithreaded program. Another goal was to gain experience of working in a group developing a bigger system. This includes experience with planning and distributing the workload amongst the participants.

The reason why the EVI system [1] is a DSM system is because of how much easier the programming model is compared to, for example, message passing systems. The participants of this project are familiar with programming in a shared memory system and most of the algorithms are developed for such a system. On the other hand, programming on a DSM does require some fine tuning, as communication costs even more than it does on a normal shared memory system, but nowadays that is something that is expected from any high performance piece of software, regardless of the programming model used.

Finally, we chose the Parallella boards because it is a new architecture and nobody, as far as we know, had done any similar work with them. Even if we weren't sure if the system would actually be a success, the opportunity to do something new was quite exciting.

2 Background

The EVI system has been built using the Parallella platform. To understand the design choices made and the limitations of the system an introduction to the platform is given as background information. Many different types of DSM systems have emerged to give the best performance when running specific types of programs and calculations. What separates these different types of systems are how much focus they put the following three design aspects: consistency, synchronization and granularity of shared data. An explanation of these aspects will also be given below to give background to the project.

2.1 Platform

Adapteva[2] has developed a system on a chip, called the Parallella Board, which consists of two ARM CPUs, and a number of Epiphany cores for parallel programs. The Epiphany cores are capable of executing pthread programs.

The Parallella board has e-mesh connectors on the backside which allows the connection of several boards to increase the size of the onboard mesh. However, at the time of the project, the support for this functionality was extremely experimental and unsupported. The communication part for a DSM system based on this platform needs to be performed via the onboard Ethernet, which is only accessible via the ARM core. This requires an infrastructure for moving data from the Epiphany cores via the ARM cores over the network to remote nodes. The different transfer speeds within the system can be seen in Table 1.

The programming model provided in the SDK released by Parallella relies on the user to explicitly allocate memory and move data from the host ARM system to the Epiphany chip for execution. While this system gives the programmer a large amount of freedom in designing the system, it is very prone to bugs. In a DSM system the addressing also

[1] Named EVI from Epiphany Valid Invalid, since these are the two states we use in the coherence protocol

[2] `http://www.parallella.org`

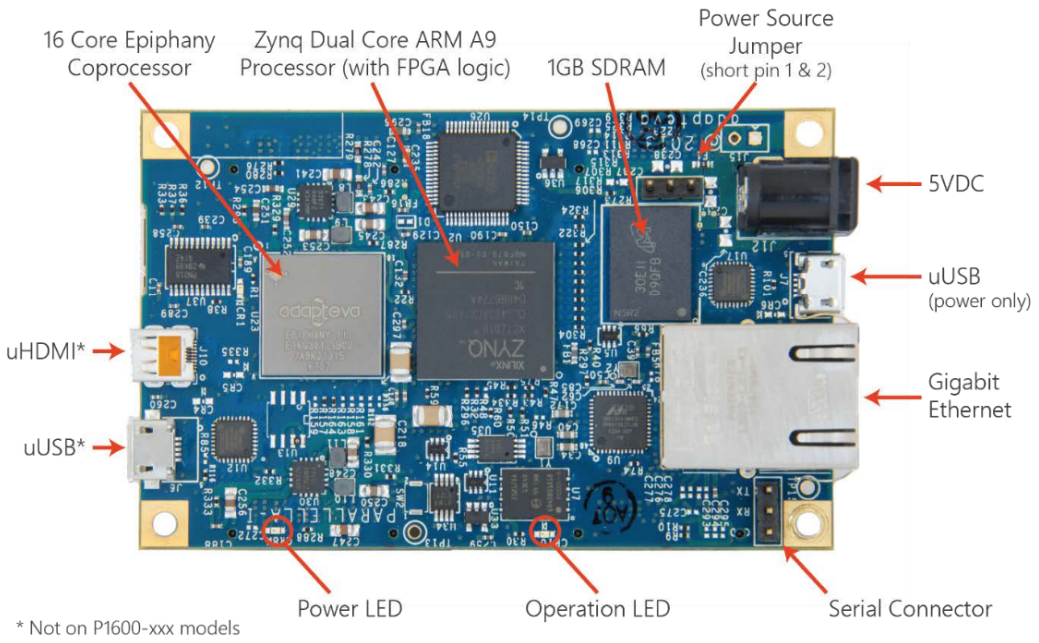

Figure 1: The Parallella board. Image taken from the Adapteva Parallella Reference Manual, Rev 14.09.09

From ⟷ To	Read speed	Write speed
ARM ⟷ eCore	6.40MB/s	14.01MB/s
ARM ⟷ SHDRAM	105.77MB/s	75.88MB/s
ARM ⟷ DRAM	194.46MB/s	194.46MB/s
eCore ⟷ eCore (neighbour)	390.01MB/s	1280.04MB/s
eCore ⟷ SHDRAM	87.69MB/s	241.54MB/s

Table 1: Table of the transfer speeds within the system.

introduces problems, as the same address on different cores will point to different places in memory, and even to memory located on other boards.

To get around both of these problems the implemented system distributes the virtual memory over all nodes within the system, using Open MPI as communication infrastructure. A caching hierarchy is implemented on the ARM side which is tasked with keeping locally cached versions of remotely allocated data to minimize the required network communication (unused network bandwidth can be used for prefetching), and sending the correct data to the requesting Epiphany cores.

The construction of the system requires the implementation of a coherency protocol which provides a consistency model to the programmer. There are several protocols available and currently in use in different systems, and the models used in the EVI system is discussed in Section 2.4.

2.2 Power Usage

We run some basic power usage measurements to ascertain the maximum power the board and our system might require. We measure power consumption under four states: idle, compiling everything [3], running all the tests, and running the bucket sort algorithm on multiple nodes. For each case, we kept the maximum power consumed. This included the power for everything on the board, but not any losses incurred by the power supply.

[3] make -j

Measurement was done by measuring the voltage the board uses (almost 5V) and then amperage going through the power source pins found on the board. The results are visible in Table 2.

State	Max Power Consumption
Idle	2.2 W
Compiling	3 W
Tests	3.8 W
Bucket sort	2.9 W

Table 2: Table over the power consumption during different operation.

Out of curiosity we also measured the power usage after the board has been shut down, and we show no drop in power usage, which might mean that the boards do not really shut down through the operating system.

2.3 Consistency Model

When several threads are working on the same data they might try to write to the same shared memory space at the same time. This might be a problem depending on how strict the program is about executing each operation in order. The consistency model determines in which order the memory accesses will be made, ranging from strict to weak consistency models. If the program requires each read operation to return the most recently stored value, it needs to follow a strict consistency model, otherwise a weak consistency model can be used. The most common models are:

Strict consistency A fetch call will always return the most recently stored value.

Sequential consistency All operations are executed in a common global order on all processors, this order is normally defined by the program.

Weak consistency All operations within critical sections are done sequentially, all other memory accesses may be seen in different order on different processes.

Processor consistency Memory accesses are only guaranteed to be consistent on each node individually.

Release consistency An extended version of the weak consistency model that uses acquire access to tell the system that it's about to enter a critical section and release access when it exits a critical section.

2.4 Cache Coherence

While the consistency model affects the execution order of a program, cache coherence defines how the shared data is kept up-to-date among the executing cores.

A common way to implement coherency and the way EVI is built, is by using a directory-based protocol, in which the system keeps a directory with the location and status of shared data. When a processor requests shared data from memory, the core asks the directory for permission to read it from main memory to its own cache. When a core modifies shared data, the directory updates or invalidates it on all other caches. Exactly how this works varies between the different types of cache coherence protocols. The VIPS coherence protocol will be discussed further in Section 2.7.

2.5 Synchronization

To keep all data up to date among all cores, a DSM system needs to synchronize the data in its caches. A common way to do this is to enter a critical section where all nodes except one

from executing any read or write operations on some data until the data is released again when exiting the critical section.

When the EVI DSM system calls for a write-back, the changes that have been made to the data in the L1 is compared to the original data in the LLC. The changes are then merged with the unchanged data in the LLC in order to synchronize the memory. The system uses buffers to communicate between different cache levels. When writing or reading data through a buffer, the data is put in a first in, first out buffer queue to avoid data races. The system then uses a mutex lock to enter a critical section that keeps any other node from accessing the buffer. When the buffer is done sending or receiving, the mutex lock is removed.

2.6 Granularity of Shared Data

The granularity of a DSM system defines the size of the chunks of data that the nodes send and receive in order to communicate. The problem here is that implementing a system with large chunks increases the risk of causing contention while smaller chunks will cause more fetches which will increase the amount of overhead.

In our system, we decided to compensate by implementing a granularity of 4KB pages. This granularity means that the MPI communication overhead will be small in comparison with the actual sending of data, while still being small enough to work with. It is also a page size used by several systems and is therefore tested in real systems.

2.7 VIPS

The EVI DSM system's coherence protocol is greatly influenced by a protocol called VIPS[KR13], named for "Valid, Invalid, Private, Shared".

VIPS is a coherence protocol that, much like ours, does not rely on having a shared L1. Instead, all cores communicate by reading and writing to the LLC. Data is classified as private as long as it is only being accessed by a single core. When a core sends a request to read some data stored on a different core's L1, this data becomes classified as shared and is written back to the LLC where it can be accessed by the requesting core. This dynamic write policy uses write-back for private cache lines since these do not need coherence and write-through for shared cache lines.

VIPS-M is a modified version of VIPS where the writes only have to be data-race-free at cache line granularity so that only the part of a cache line that has been modified needs to be transferred. Instead of having a directory to keep track of shared data, VIPS-M uses self-invalidation which invalidates all shared data at the next synchronization point.

3 Methodology

The project was divided into three main phases. Part of the initial phase consisted of research, and was crucial since no previous knowledge of the Parallella board and Epiphany cores existed in the group. A large amount of time was also spent on designing the system. The second phase, in which the system was implemented, was the shortest phase, although implementation of optimizations and bug fixes continued into the third phase. The final phase was mostly dedicated to testing and benchmarking. Implementations were made in parallel to correct errors and make improvements to the system.

3.1 Research and Design Phase

The first couple of weeks of the project was spent researching and learning about the Parallella platform, on which the system was to be built. The aim of the project was to create a distributed shared memory system, in order to decide if the Parallella platform was suitable and capable enough for such a system. Once it became clear that the Parallella board could be used, some initial hardware tests were made.

After everyone had learnt more about the project, several discussions were held to find the optimal method to use. For example, it was decided on which level the implementation should be. This discussion led to a design phase. The design phase of the project continued for a couple of weeks. A lot of design decision were influenced by the Parallella board's memory architecture. Where and how to implement the virtual address space was also an important design topic. The design was done iteratively, and many design faults and mistakes were found in each iteration. Should the implementation of the system had started with one of the earlier versions of the design, a lot of work would have had to be re-done.

For example, when the system originally consisted of three caches, the idea was to have a shared L2 cache in the shared DRAM. Also, the virtual memory was originally correlated to the Linux virtual memory and intended to be structs of offset and size.

3.2 Group Organization

After the design phase, the implementation of the system was divided up into tasks. The main topics of these tasks were coherency, cache, API, memory, MPI, and general system. Parallel to this, a case for the system was designed and built, to make the system more convenient to move and easier to cool. During the run of the project, 3-4 development groups worked on implementing different parts of the system in parallel. Every few weeks, when tasks had been completed, the changes were merged, tests executed and after each successful merge the groups were re-organized.

Each group consisted of a number of members, but we preferred to employ pair programming. This, except from the known advantages of pair programming, also allowed people to move to different groups easily, as long as the driver was already familiar with the module that was being worked on.

Furthermore, a task table on a white board, divided into planned, ongoing and finished, was used to keep track of the progress made. Also, meetings were held every Monday and Wednesday. On Fridays, usually the group would meet up with the supervisors to gain valuable feedback. Most of the meetings were informal in nature, and we just discussed what everyone had been doing, if any new problems have arisen and what we were going to do next. In some occasions though, the meetings where more formal and organized, with specific topics for discussions and written meeting proceedings.

For most of the note-taking and online communication, we used a Facebook group. Anything that we thought might be useful was uploaded there so that everyone could see it, including the meeting proceedings. Also, people that were going to be absent from the group, posted there to let the rest know and organize accordingly. While Facebook provides only basic functionalities, in our case it worked because we only needed to be able to upload posts and get notifications. Also, everyone was already using Facebook, so there was no need for a new platform.

Generally, our organization was very relaxed and informal. There was no leader in the group and all the decisions were taken either by the whole group or the subgroup working on the module affected. We did not have many rules in place and everyone was responsible for their own participation in the group. Similarly, we did not use any of the well known organization methods, such as agile, but did mostly what felt convenient and natural to everyone. This of course does not work in every occasion, and different groups might need a different way to organize themselves, but it did work for us.

3.3 Languages and Software Tools

The system was mostly implemented in C++, as the Epiphany boards are C/C++ programmable. C++ was chosen over C, as this allows the API to redefine the functionality of the standard operators within the system, thereby making it easier to produce an understandable way to program the system. In addition, the use of C++ provides data types such as hash maps and queues, which simplifies the implementation of the system.

Furthermore, Git[4] was used as a version handling tool, and the main repository was

[4] `http://www.git-scm.com`

placed on GitHub[5]. Git was selected because it provides an easy way to branch and merge code in a distributed manner. By using GitHub, the project did not need to spend any time or resources setting up a version control system.

For automated documentation, Doxygen[6] was used. Doxygen allows the programmer to write comments that are very similar to the C commenting standard. This makes it an easy tool to use and learn to use. The comments are written in the header files to allow a fast compilation of the documentation. When compiled, Doxygen gives a good overview of the entire system, which can be easily read as a website.

To be able to perform continuous testing on the system while it was being developed, unit tests were developed using the QUnit framework[7]. QUnit was a good tool for debugging and making sure that changes to the system did not have any unwanted consequences. These tests were run before committing any changes to the source tree.

As more and more parts of the system were completed, the test grew to encompass more and more of the system. These tests were written using the API implemented for the system, in combination with QUnit. Towards the end of the project, this test was added to the test suite and became an integral part of the continuous testing of the system.

To debug the system on the ARM side an extensive logging was used. For more in-depth debugging, GDB[8] and Valgrind[9] were used. The Epiphany cores were debugged by sending flags and data via a specifically allocated debug channel. We also used a modified version of GDB provided for Adapteva, but it provided very limited capabilities.

Finally, it is worth noting that we did not enforce code review, and that later felt as a mistake, at least by some members of the group. Perhaps it would have been better if some software solution[10] was used, in order to assure better code quality. Also, in the context of the course, code review would have allowed everyone to get much more familiarized with the whole system and not just the parts they worked on.

4 Implementation

The implementation of the EVI system is divided into a number of modules which specialize on performing one of the tasks that make up the system. The most important of these are the MPI module, the Coherence protocol module, the Virtual Memory module, the Cache module, the API module and the Memory Management Unit module. These modules are described in greater detail in their respective subsections of this section. In addition to these modules, the system contains a number of smaller modules which handle operations that support the main modules. These include buffer implementations, logging and specialized hash maps. While these supporting modules are important for the operation of the system as a whole, they are not considered defining parts of the implementations and are not described further in this report. Each module is largely independent from the other modules, which allowed for the parallel implementation of the system.

4.1 MPI Communication

The Parallella boards (nodes) communicate via the OpenMPI implementation of the message passing interface. In the EVI system, all MPI communication has been abstracted away from the MMU implementation into its own module, to simplify the MMU implementation. Because of this, the MPI module doesn't have any "intelligence" of its own, but only handles incoming and outgoing requests. In that sense, the input and output of the MPI module can be thought of as raw data which is then handled by the MMU.

The MPI module is executing in one thread, which handles two tasks, of which the first one handles the sending of requests from the local node, while the second handles messages

[5] `http://www.github.com`
[6] `http://www.stack.nl/~dimitri/doxygen`
[7] `http://qunit.sourceforge.net`
[8] `http://www.gnu.org/software/gdb/`
[9] `http://valgrind.org/`
[10] Such as Phabricator, for example

received from remote nodes. The nodes send and receive `evi_mpi_page_msg` structs. The send thread continuously consumes messages produced by the local MMU via an outbound buffer. The buffer messages consist of a struct which, together with the actual data to be sent, contains a message type code. This code is used by the MPI module to identify which message type should be sent to the remote node. Each message type has an assigned MPI message tag which is set by the sender and used by the receiving node to identify which message type was sent.

For the system to handle incoming requests and pages, the MPI module is continuously probing the MPI input buffer for new messages. When the board receives a new message, the tag of the message is read to identify which message type the system received. The MPI module will then forward the message to the MMU via its incoming buffer. The nodes can send Fetch Requests, Page, Writebacks, Sync/SyncAck and Allocation Allowed messages. The meaning of these requests are briefly described below, and discussed in greater detail in Section 4.3.

All messages except the Fetch messages are sent from one node to one intended receiver. For the Page message this means delivering the page to the node which requested it, and for the Sync message it means sending the sync initialization message to the sync master of the current epoch (currently hardcoded to node 0). Fetch requests, on the other hand, are sent from a node if one of its cores request a piece of memory that is not allocated on the local node. However, the request might be forwarded by other nodes within the system to reach the node on which the memory is allocated. With the current implementation, the maximum number of times a message may be forwarded is 1. The node that initiated the Fetch request will always expect either a Page or Allocation Allowed message in response, depending on if some node within the system has already allocated the corresponding page. This answer is returned by the last node that receives the message.

Scalability is paramount to the usability of the system. Should it not scale well when the number of nodes increase, the system will not be usable. The scalability can be proven to be linear with our system compared to polynomial when using the broadcast implementation. The following calculations show that this is the case.

x = number of boards
y = number of messages per second on communication channel
r = requests per board per second

Broadcast
x boards asking $x - 1$ boards r times per second yields

$$y = r \cdot x \cdot (x - 1) = O(rx^2)$$

Our implementation
x boards asking one board with a maximum of one redirect r times per second yields

$$y = O(2rx)$$

In a system with 1000 nodes sending one message per second, the broadcast implementation will produce 10^6 messages per second, while our implementation only produces 2000 messages per second.

When the system reaches a synchronization point, all data that has been changed needs to be written back to the node that first allocated the data. This is done by sending Writeback messages. Writeback messages will not expect any replies, but expect everything to have succeeded once the synchronization barrier has been reached. The synchronization is performed by sending the Sync message to the designated synchronization master of the

epoch, which is currently hardcoded to node 0, which will send Synchronization Acknowledgement (`SYNC_ACK`) messages in return once every node has reached the synchronization point. This broadcast solution was implemented at the end of the project, as a quick fix to a race condition and is in no way optimally implemented.

Each node within the system executes the exact same code in the MPI module, except the synchronization master, which handles the synchronization points. The synchronization master node is, however, automatically selected at run time, meaning the same binary executes on every node.

4.2 Virtual Memory

The virtual memory implementation in the EVI system is done completely in software, that is, the EVI VM system is built on top of the Linux virtual memory allocated to the executing EVI binary. When this report mentions Virtual Memory it refers to the EVI VM system implemented by the authors. Likewise, physical memory does not refer to the actual physical memory, but to the Linux VM system (which then handles the physical memory).

EVI Distributed Virtual Memory ("Virtual memory")
Linux Virtual Memory ("Physical memory")
Physical Memory

The virtual memory system is implemented as a continuous range of addresses, using 64-bit addressing. The addresses are not locked to one specific node, instead the memory is allocated on the first node to use an address, commonly referred to as first touch allocation, and is common in NUMA systems. The memory system is a very simple implementation. Since the system provides the programmer with a 64-bit addressable memory, the virtual memory makes the assumption that the programmer will never address all the memory. This allows the virtual memory to be implemented with just one counter, called the memory head, which specifies at which address the next memory allocation will occur.

Each time the programmer allocates memory, the memory head is moved forward the corresponding number of bytes to make sure the next allocation is done after the current. To minimize the communication needed between the nodes, the system requires every core to execute the memory allocation function at the same time (that is, at the same point in the code). This guarantees that each node will keep the memory head synchronized with the others, without the need for locks or broadcasts.

The memory head counter can only move forwards, as the system assumes the programmer will never need to use the whole address space. This might seem like a waste of memory, as once an address is allocated, it can never be allocated again. However, the virtual memory is not directly mapped to the physical address, meaning that the virtual memory system can allocate any amount of memory, while only a fraction (or none) of it is allocated in the physical memory. Once the programmer is finished with a piece of allocated memory, the memory is freed. While this will not affect the virtual memory head counter, the physical memory that the virtual address refers to will be freed, allowing the next virtual memory allocation to reuse that part of the physical memory.

The physical memory of the original memory architecture is only 32 bit addressable. However, as the EVI virtual memory system may allocate memory over all nodes present in the cluster, it uses a 64 bit addressable virtual memory to be able to allocate more than 4GB in total.

From the programmer's viewpoint, the virtual memory is completely abstracted from the physical memory. This means that the programmer will never need to worry about how the EVI virtual memory system is implemented, and how it interacts with the underlying Linux system.

4.3 Memory Management Unit

The Memory Management Unit (MMU) is responsible for serving pages of data answering both remote requests received over MPI, and local requests from the L2 cache (and in

extension, the Epiphany cores). While the MMU is distributed, running independently on each of the nodes, all of the MMUs together can be thought of as one unified system that handles the virtual memory. Internally, the MMU peers communicate using MPI to fetch or writeback pages that are not physically stored on the local node. To the rest of the system, all communications with the MMU are done from the LLC or via the system call interface. Each read/write operation on memory is handled transparently to the user, no matter if the memory is allocated locally or on a remote node..

Once a memory request is forwarded from the LLC (cache miss) to the MMU, all the information needed to fulfill the request is stored in the request message. The MMU itself does not retain any state information, it only reacts to the received messages. By running in an almost stateless way, we minimize the possibility of introducing errors in the system since the number of cases to handle is decreased. In addition, this stateless approach allows the distributed memory to function by just forwarding requests to remote nodes without any extra meta data.

The MMU internally consists of three hash maps; the `local_hash`, and the local and global TLBs. Each of these control different parts of the MMU operations and are described below.

The `local_hash` maps virtual addresses to physically allocated pages, that is, handles the translation from the EVI virtual addresses to the addressing used by the underlying Linux system. Each time a node allocates a page locally, an entry in the `local_hash` is added. Consequently, once a local core requests a page that exists in the `local_hash`, the MMU knows that the page is allocated locally, and responds with the requested page without having to request it over MPI.

Each node within the system is assigned a number of address ranges for which that board must keep track of on which node has allocated a specific address. The address ranges for each board are specified by using the MPI rank of each node, the number of boards in the system, and the page size:

$$\frac{\text{Address}}{\text{PageSize}} \% \text{NumberOfBoards} = \text{ResponsibleBoard} \tag{1}$$

With this formula any board can deduce that board 0 is responsible for v-pages with addresses in $[0, PageSize)$. Note that the node is only responsible for knowing where the address is physically allocated – the address can be allocated on any node. The global TLB on each MMU is used to keep track of where the pages that board is responsible for are allocated. The local TLB is used to map received virtual pages to the node on which they are allocated. Each time the MMU receives a requested page from another node, it adds an entry to the local TLB mapping the address of the page to the MPI rank of the sending node. This way, the system will be able to send future requests for the same page directly to the node on which it is allocated, instead of going through the board responsible for that address.

The design with the two TLBs allow the system to allocate pages on any node, while still avoiding broadcast. Should a node request a page that it has previously not used, the node will send a message to the node responsible for the address, using equation (1).

Depending on if this is the first touch of the address, one of two scenarios happen:

- The address has not yet been allocated, and this is the first touch. The remote node adds the requesting node to its global TLB and responds with an Allocation Allowed message. The local node allocates the memory and adds it to its $local_hash$.
- The address is already allocated. The remote node checks its global TLB, and forwards the request to the node on which the address is allocated. That node, in turn, receives the request and sends the page back to the original requester, which will then add the node it received the page from to its local TLB. ...

This design enables each MMU to read a new page with a maximum of two messages over MPI. In turn, this makes the system more scalable compared to the naïve implementation of

requesting pages via MPI broadcasting. The current implementation works as the current system does not support data migration and is restarted for each new execution. This means that once an entry has been added to one of these hash maps, it will not have to be removed during the system lifetime.

In addition to handling the page requests, the MMU also handles page writebacks and synchronisations. Writebacks may be sent both from the local LLC and via MPI from remote MMU instances. In both cases, the MMU applies the changes to the local data. The synchronization is implemented using a design similar to a barrier and is implemented in the MPI part of the system. While we are in the critical section of a synchronization, we also stop accepting requests from the LLC and the L1 caches. Practically, we shouldn't receive any requests in any case, since they should be waiting for the synchronization to finish as well. Synchronization is requested by user code as a tool to avoid data races. This operation is necessary, as the coherency protocol EVI is based on requires data race free code. Both of these operations are described further in Section 4.4.

4.4 Coherence Protocol

The coherence protocol of the EVI system is based on the VIPS-M protocol described in Section 2.4, however, the implemented protocol does not distinguish between private and shared data, thereby reducing it to a VI coherency protocol. This greatly simplifies the implementation. However, both protocols require the application to be data race free and allows the creation of a scalable distributed system, as the coherency protocol relies very little on communication between nodes.

The coherency protocol assumes that the executing code is data race free, meaning that two cores are not working on the same data at the same time. Same time in a VI system is defined as within one time epoch, which is the time that elapses between two synchronizations.

The Epiphany cores can only read the data that was written back to main memory at the last synchronization point, that is, at the beginning of the current epoch. Any changes made by one core will only be visible to that particular core until the next synchronization point. Whenever the programmer needs to make all changes visible to the system, the synchronisation syscall must be called. Should the assumption that the code is data race free not hold at this point, the data that is made visible to the system is undefined.

The local data is made visible to the system at synchronization by writeback operations to the memory.The caches have a `dirty_map` that stores information about which data is dirty and which node dirtied it. In the L1 cache, cache lines with the dirty bit set will be written back, after which all cache lines will be set to invalid. In the last level cache the same concept is valid but on a page level. The synchronisation messages are sent across all boards via the MPI module. During synchronisation, the entire system is locked until all nodes are finished.

Write backs occur both during synchronisation and whenever the caches are full and data must be written back. In this case the data must also be dirty, otherwise the data will simply be invalidated and overwritten. During write backs, dirty data is compared to the original data and merged. All nodes that have the original data will keep a copy of the original to perform this difference and merge. Dirty pages are sent across the MPI module to the node that has the original data allocated in physical memory.

4.5 Caches

The EVI system consists of a 2-level cache hierarchy. Each eCore contains its own private L1 cache of 4KB, while the ARM side maintains a shared L2 cache of 16MB. The L2 is, of course, the LLC in our design.

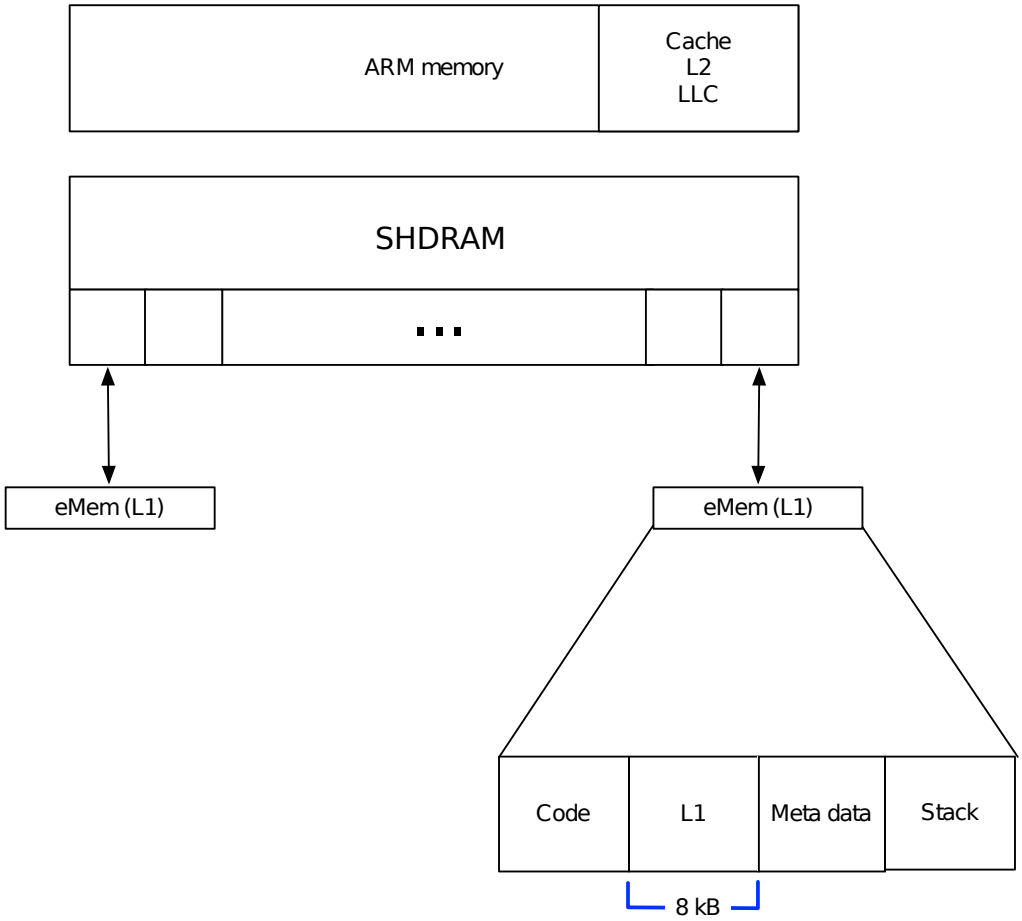

Figure 2: The memory hierarchy

4.5.1 Common Base

All the caches share the same base class implementation, which defines the common cache operations, such as testing and setting the dirty flags. Not all operations can be shared of course, since they require different handling in different levels of the cache.

The base class is a templated function. The template parameters specify the size of the cache, in rows and columns, as well as the associativity. Then, all the space for the cache lines and the metadata is allocated continuously in memory. This means that it is possible to place all the cache data and metadata in a specific place in memory without worrying about where dynamic memory is allocated.

The cache data and metadata implementation is based on hardware caches. Before each cache line, there are bits for the LRU, the dirty, and the valid flags. This was done to improve locality of the data we need to access every time we access a cache line in the cache. It has however the disadvantage of not being able to separate the data from the metadata, in case we would like them placed in different memory parts for example. Also, the cache size has to be a power of 2, with the metadata included, which limits the amount of memory we can use.

4.5.2 Last Level Cache

The LLC is a 4-way set-associative cache with cache line size equal to one page (4KiB). It is stored in the private DRAM of the ARM processor. This means that the ARM processor has optimal access times to it. However, at the same time, the eCores can not access it directly and need to ask the ARM processor for any data they need.

The LLC is run on a separate thread and monitors both the L1 and the MMU for any incoming communication. All the communication between the L1 and the memory system is done through the LLC. As a matter of fact, the only time we bypass the LLC is when an eCore requests his global core ID, which is handled directly by the MMU.

Unlike the L1 cache, the LLC contains a number of auxiliary data structures. First of all, there are data structures for storing pending fetch and write back requests. We do that to avoid blocking the whole cache while waiting for data , that might be needed to satisfy those requests, from the MMU. It also contains clean copies of all the dirty pages in the cache. These are used for creating the differences that we need to send the MMU, as well as detecting which bytes each writeback from the L1 has changed. Finally, it contains a dirty bytes structure which keeps track of all the bytes that are set as dirty in the cache lines, as well as who was that changed them in the first place. This required to detect all the writes that happen in the cache lines, since just comparing with the originals does not detect all of them.

4.5.2.1 Communication

SHDRAM buffers are used to communicate with the eCores. In the shared memory, there is enough space allocated to store 16 flags and 16 message buffers, which are used for passing data. Each eCore can only read and write one message at a time before the LLC can handle it, due to limitations to the buffer size. When the LLC reads a buffer from the L1, it will copy the data and then, depending on the message type, might or might not notify the L1 that the message was handled. Specifically, some cases, such as writebacks, allows the system to instantly notify the L1, while others, such as requests for data, must wait until the system have written the data back into the buffer. On the MMU side, since it is running in the same memory space, communication is much easier and is handled by two thread-safe queues, which allow the LLC to send and receive multiple messages without having to wait for the other side to respond. While running, the cache checks each of those communication buffers, the L1 and the MMU one, and responds to any messages accordingly.

4.5.2.2 Message Handling

The MMU communication is much simpler than the communication with the L1. The MMU will only send two kinds of messages to the LLC, either a `sync_ack` or a memory page that the LLC requested. In the first case, the LLC only has to forward it to all the L1s and it is done. In the second case, it needs to write the page in its cache and then forward any cache lines any L1s have requested from that page to them. Both require some simple copying of data.

The L1 communication can be more complex. The L1 can either send a sync message, a writeback or a fetch to the LLC. Syncs and fetch requests are rather simple. For sync, the LLC just has to count the number of syncs received, and when there are 16 of them, forward the sync to the MMU. For the fetch, the LLC first checks if it has the data requested. If yes, then it just copies the data that the L1 has modified and notifies the L1. We use the clean copy instead of the dirty data for data that the L1 has not changed because the dirty data can appear as data races when doing writebacks. If not, then the request is stored in the pending requests structure and a request is posted to the MMU for the page containing.

The complexity increases slightly when a writeback is received, since now data coherence needs to be kept. For that reason, the following steps are executed:

1. Check if the LLC has the page that contains the cache line. If not, the writeback is added into the pending writebacks structure and a request is posted to the MMU for

it. The original page is needed for calculating the difference, before sending it to the MMU.

2. The LLC now has the page and is ready to perform the writeback. First, the LLC checks if a clean copy of the page needs to be created. The clean copies are used for calculating the differences and are stored separately from the cache data. A clean copy needs to be created if the LLC doesn't already have one, otherwise the existing one is fetched.

3. Now the clean copy and the data received from the L1 are compared. This is done by comparing bytes for equality on a byte level. Any bytes found to be dirty are appended to the dirty bytes structure.

4. The LLC iterates over the dirty bytes, and all of them that belong to the L1 doing the writeback are written into the cache. The writeback is now complete.

4.5.3 Level 1 Cache

The L1 cache is much simpler than the LLC. Each private L1 is located in the SRAM of its respective eCore. Unlike the other cache, it is not run in a thread [11] but it is used directly by the API. So the only communication buffers exist to the ARM side. Because of its simplicity, the L1 makes one request at a time to the ARM and then waits for a reply. No coherence operations are done in the L1 and almost anything that could be moved to the ARM side, has been. Since the memory on the eCores is limited to 32KiB and parts of it for the stack and the text [12], the L1 is sized at 4KiB + 1KiB of metadata.

4.6 API

The API is designed to be as transparent as possible. There are however some limitations and quirks, some of which are further described in this section.

4.6.1 API Functions

The API is divided into two parts, one for the Epiphany side, and one for the ARM side. This allows the programmer to execute non-computation intensive operations, like distributing the input data from the ARM side before starting the Epiphany side program, while still retaining complete access to the system.

The API introduces a new pointer type, evi::ptr, used to address data in the virtual memory, which has overloaded the standard C++ operators, thereby allowing the programmer to operate on the data in a manner similar to the way standard $C/C++$ works. This makes the whole caching and virtual memory system largely invisible to the programmer.

In addition to the new pointer format, the API contains the functions as seen in table 3, which may be used by the programmer to collect information about the environment the program is executing in, as well as handle allocations of virtual memory. These functions are largely overlapping between the ARM and Epiphany sides, but with some differences, as detailed in the table below.

In addition to these API functions, the ARM side code may use `MPI::COMM_WORLD.Get_rank()` to identify the global id of the board, which may be useful for performing operations on only one node.

4.6.2 Programming for the EVI System

The programs consist of two parts, the ARM part and the Epiphany part. The system is designed with the following programming model in mind:

1. The data are initialized on the ARM side.

[11] The eCores do not support threads anyway
[12] The program code

ARM-side	Epiphany-side	Description
ptr<T>	ptr<T>	A pointer to a virtual address (`evi_v_addr`) containing data of standard C/C++ type T. Also handles arrays[] and standard pointer arithmetics.
`malloc(size)`	`malloc(size)`	Allocates size bytes in the virtual memory.
`sync()`	`sync(flag)`	Synchronizes the system, thereby making all data changes visible to all cores. Flag controls which mode to use.
	`get_local_coreid()`	Returns the core id on the local board (0-15 if 16 core board).
	`get_global_coreid()`	Returns the global core id, which is unique in the system.
`evi_init()`	`init()`	Sets up the EVI system.
`evi_finalize()`	`finalize(rc)`	Tears down the EVI system. If rc == 0 then synchronization happens.
`run(exe, args)`		Starts execution of the exe binary with arguments args on the Epiphany cores.
`try_wait_all()`		Returns true if all Epiphany cores have finished executing (non blocking).
`wait_all()`		Blocking wait for all cores to finish. Returns 0 on success, otherwise the first non-zero return code.

Table 3: ARM-side and Epiphany-side API functions.

2. The Epiphany program is started. Calling run() which cause synchronization.
3. The Epiphany cores work on the data and write the result in the Shared Memory.
4. evi::finalize is called. When called with an argument of zero (0), synchronization is done once more.
5. The ARM can now read the data from the Shared Memory.

A very basic example on how to use the API is the *test/test_api_load.cpp* and *test/e_test$_a$pi_load.cpp* files, which are presented in Appendix A. Also the two benchmarks provide full working examples.

4.6.3 Limitations

1. The type of the EVI pointers has to fit evenly in a cache line and not exceed the cache line limit. This is due to how we access and write back the data.
2. Pointers to pointers have not been tested.
3. Assigning one pointer to another pointer directly can cause some issues in specific cases. Using a temporary local variable is recommended to solve this problem.

4.7 Previous Implementations

The final design presented in the previous parts of this section was not designed and implemented in a straightforward fashion. Although the final design is in spirit close to the initial design, several versions of the system were implemented and rejected or redesigned during the project. This sections aims to give the reader an overview of the previous implementations and how they helped to form the final design.

4.7.1 Three Level Cache Hierarchy

The initial design of the system featured a three level cache hierarchy, in which the L1 was located in the local memory of the Epiphany cores, and the L2 and L3 in the ARM memory. Furthermore, the L1 and L2 were private, while the L3 was shared between all the cores on the local node. The idea was that the L1 and L2 cache levels would be private, to not have to deal with local coherency when communicating between the Epiphany memory and ARM memory. The only contribution of the L3 within the system would be to handle the local coherency between the cores. The L2 was implemented with a threaded approach, in which each local L2 was running a thread of its own. The L3 was running in one thread of its own.

At this point the system was very slow, and it was theorized that part of this was because of the context switching time that was introduced as overhead. The L2 caches were executing 16 threads out of 23, so a simple test was devised, in which the system was executed on the ARM side with and without the L2 running. When the L2 was not running the execution time for the test dropped by a factor of ten, and it was therefore decided to make the L2 single threaded. However, to further reduce the number of running threads a merge of the L2 and L3 caches into a new L2 was decided. The new L2 cache executes a single thread and handles the local coherency between the Epiphany cores, thereby filling the same functionality as the old L2 and L3 did in a single LLC. The new cache hierarchy is further described in Section 4.5.

4.7.2 Threaded MPI and MMU models

Initially, the MPI and the MMU run in two different threads each. The MPI used different threads for sending and receiving messages, while the MMU used different threads for local and global operations.

This approach caused a number of problems and was removed from the final system. First of all, we discovered that OpenMPI does not really support the `MPI_THREAD_MULTIPLE` thread parallelism level, so the MPI functions are not thread safe. This means that almost every MPI call needs to be protected by a lock. This was an issue, since in the initial design we were counting on being able to call multiple MPI sends and receives at once, as well running a barrier in parallel. After discovering that this is not possible, we redesigned the MPI part to run on just one thread and to not use any blocking MPI calls. Blocking would cause deadlocks, since, for example, we still want to be able to accept writebacks while we are waiting at a sync barrier.

Similarly, we discovered some races between the local and the global MMU threads. While we could easily fix those with mutual exclusion, we decided to reduce the MMU to just one thread, since the more threads the worse the ARM performance became, due to the context switching overhead. The new MPI and MMU implementations are described in sections 4.1 and 4.3 respectively.

5 Benchmarks

To evaluate the system a number of benchmarks are executed on the implemented system. The benchmarks are chosen for their ability to push the performance of the system with a large number of operations in different approaches. The benchmarking algorithms implemented consist of bucket sort and matrix multiplication.

5.1 Bucket sort

The Bucket sort algorithm works by creating a number of buckets, which will contain all elements within a specified range (which is unique for each bucket). Each bucket is then sorted individually using QuickSort, and then put back into the original array.

Our implementation operates in its entirety on the Epiphany cores of the system, although the input data is randomized on the ARM cores. The randomization is done distrib-

uted over all nodes to ensure that all data is not allocated on one board, because of the first touch implementation. Each Epiphany core is in charge of creating its bucket and populate it with the elements from the input array that are within its range (decided by global core id), sort the bucket, and write the elements back to the input array.

Once the input data has been created and synchronized to all nodes, the algorithm only requires one synchronization point, which is before writing back the sorted buckets to the input array. This is due to the fact that each node needs to know at which point in the array to start writing back, which is decided by how many elements all cores with lower id have to write back. In addition to this, the Init and Finalize when the Epiphany cores are started will automatically invoke a synchronization. All other parts of the algorithm are done completely in parallel.

5.1.1 Execution of Bucket sort

The bucket sort benchmarks were executed on 1, 2, 4, and 8 nodes, thereby giving the executing program access to 16, 32, 64 and 128 cores respectively. Since the system is quite slow, the benchmarks were reduced to input sizes smaller than 1638400 elements. The benchmarks were run under the following conditions:

- Number of nodes: 1, 2, 4, 8
- Size of input data:
  - 128
  - 512
  - 2 048
  - 8 192
  - 32 768
  - 131 072
  - 409 600
  - 819 200
  - 1 638 400

5.1.2 Results of Bucket sort

The executions of Bucket sort shows that the speedups gained from adding more nodes starts at input arrays of size 8192, even though the execution time for 8192 seems to slow down after 4 nodes. For input arrays of smaller size, the system suffers from a slight slowdown when more boards are added.

Once the input size is larger than 32768 elements, the speedup as more nodes are added increases. This is probably due to the fact that the overhead for the communication between nodes is reduced to a smaller part of the execution time. Figure 3 presents a graph of the speedup under different input sizes on different amount of nodes.

In addition to good speedups under several conditions, the bucket sort algorithm has a consistently low cache miss ratio for larger inputs, in both the private L1 cache and the shared L2. This means that the algorithm is running at close to the maximum speed the EVI system can deliver. We can also see that the miss ratio decreases as the input size increases, meaning that we get good data reuse once the data has been cached. Table 4 presents an overview of the execution times and cache miss ratios of both the L1 and L2 caches under different input data on different number of nodes.

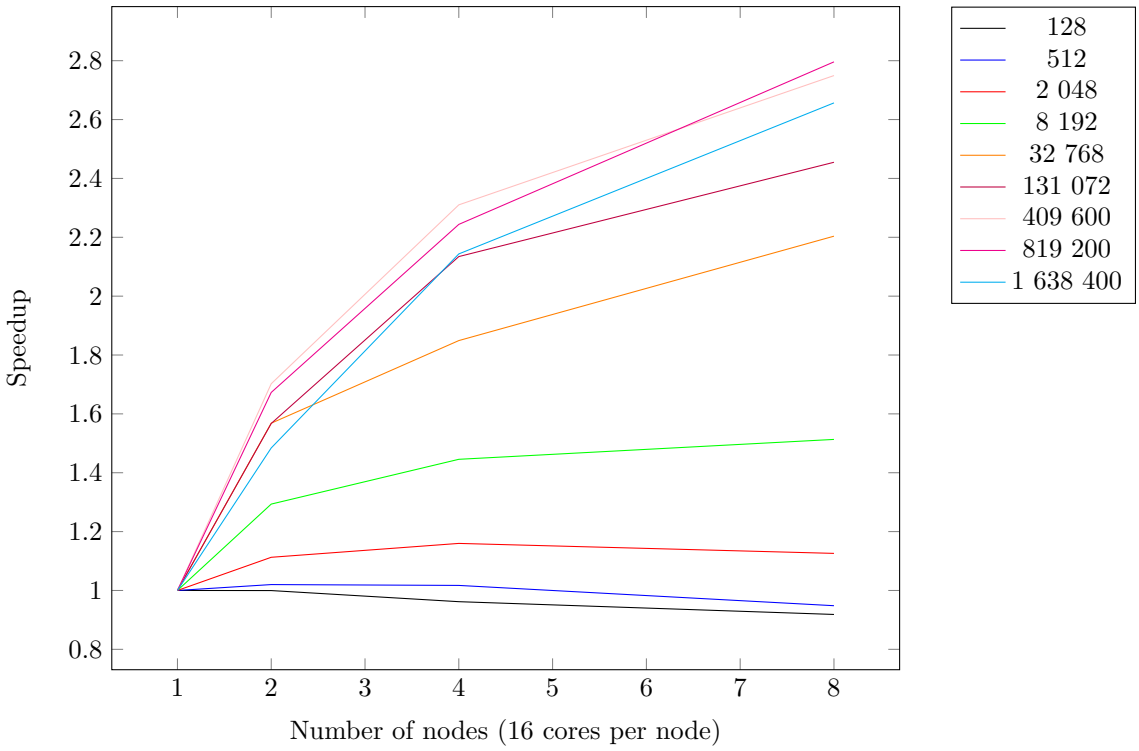

Figure 3: The speedup of Bucket sort on differently sized inputs on different number of nodes.

5.2 Matrix Multiplication

The matrix multiplication algorithm divides the matrix into square chunks which is calculated using normal matrix multiplication. This implies that there are multiple readers on the matrices being multiplied but only one writer to each cell in the result matrix.

Like the bucket sort algorithm, the matrix multiplication runs all calculations on the epiphany cores, while the ARM core generates randomized matrices. In this implementation there are no measures taken to distribute the data onto several boards. Data distribution was not implemented due to the fact that there would be an insignificant performance increase when running with the matrix sizes used. There are only two synchronizations points in the algorithm; after the data has been generated and after all of the epiphany cores has completed their execution.

5.2.1 Execution of Matrix Multiplication

The Matrix multiplication benchmarks were executed on 1 and 4 nodes, thereby giving the executing program access to 16 and 64 cores respectively. Since the system is quite slow, the benchmarks were reduced to input sizes smaller than $1024 * 1024$ elements. The implementation of Matrix multiplication also requires a square amount of nodes, which means that the benchmark was limited to running on 4 nodes, as there were only 8 nodes available. The benchmarks were run under the following conditions:

- Number of nodes: 1, 4
- Size of input data:
  - 32
  - 64
  - 128

# Nodes	Input size	Exec. time	L1 miss ratio	L2 miss ratio
1	128	0.99 s	2.49 %	9.75 %
1	512	1.12 s	1.45 %	2.77 %
1	2 048	1.62 s	1.11 %	1.02 %
1	8 192	4.21 s	0.93 %	0.31 %
1	32 768	17.42 s	0.81 %	0.15 %
1	131 072	71.74 s	0.86 %	0.1 %
1	409 600	255.47 s	0.87 %	0.08 %
1	819 200	519.76 s	0.9 %	0.08 %
1	1 638 400	1024.53 s	0.92 %	0.07 %
2	128	0.99 s	3.25 %	14.6 %
2	512	1.09 s	1.92 %	5.66 %
2	2 048	1.45 s	1.53 %	1.8 %
2	8 192	3.26 s	1.32 %	0.51 %
2	32 768	11.11 s	1.18 %	0.2 %
2	131 072	45.77 s	1.12 %	0.1 %
2	409 600	150.07 s	1.13 %	0.08 %
2	819 200	310.7 s	1.12 %	0.08 %
2	1 638 400	690.29 S	1.13 %	0.07 %
4	128	1.03 s	3.61 %	19.61 %
4	512	1.1 s	2.27 %	7.93 %
4	2 048	1.39 s	1.85 %	2.73 %
4	8 192	2.91 s	1.67 %	0.8 %
4	32 768	9.42 s	1.55 %	0.26 %
4	131 072	33.61 s	1.44 %	0.15 %
4	409 600	110.59 s	1.43 %	0.1 %
4	819 200	231.64 s	1.42 %	0.09 %
4	1 638 400	477.97 S	1.41 %	0.09 %
8	128	1.08 s	3.73 %	23.17 %
8	512	1.18 s	2.52 %	10.01 %
8	2 048	1.43 s	2.09 %	3.75 %
8	8 192	2.78 s	1.93 %	1.01 %
8	32 768	7.9 s	1.82 %	0.43 %
8	131 072	29.22 s	1.74 %	0.26 %
8	409 600	92.92 s	1.7 %	0.19 %
8	819 200	185.89 s	1.69 %	0.18 %
8	1 638 400	385.65 s	1.68 %	0.16 %

Table 4: The execution time and L1 and L2 cache miss ratio for different input sizes on different number of nodes with the Bucket sort algorithm.

- 256
- 512
- 1024

5.2.2 Results of Matrix Multiplication

The graphs show that the EVI system scales fairly well when increasing the number of boards for matrix multiplication. When using four times the number of boards the calculation takes half the time, this concurs with the calculation of the amount of message passing used in the EVI system. In order to improve the hit ratio the second matrix where transposed, this yielded a 10x speed up compared to not transposing it. A plot of the speedup factors is presented in Figure 4.

The graphs shows a great speedup. However, there are only two data points. Since the algorithm required a square formation of boards and there were only eight boards available

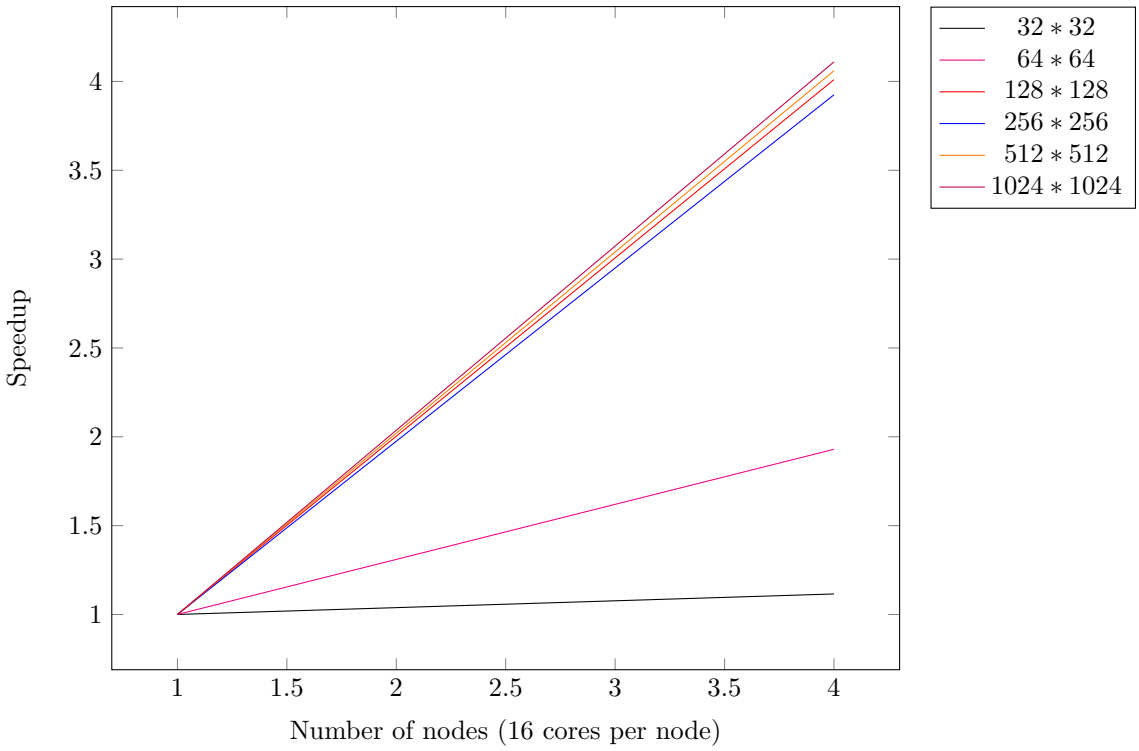

Figure 4: The speedup of Matrix Multiplication on differently sized inputs on different number of nodes.

allowing for a maximum of two data points, one and four boards. The execution time, and cache miss ratios for different sized inputs are presented in Table 5.

# Nodes	Input size	Exec. time	L1 miss ratio	L2 miss ratio
1	32	0.83 s	1.06 %	0.55 %
1	64	1.95 s	0.5 %	0.19 %
1	128	15.34 s	3.53 %	0.06 %
1	256	122.7 s	3.71 %	0.03 %
1	512	1061.89 s	4.13 %	0.01 %
1	1024	10093.2 s	5.07 %	0.00 %
4	32	0.74 s	2.38 %	2.08 %
4	64	1.01 s	0.95 %	0.74 %
4	128	3.82 s	1.43 %	0.22 %
4	256	31.27 s	3.75 %	0.06 %
4	512	261.55 s	4.15 %	0.02 %
4	1024	2456.54 s	5.07 %	0.00 %

Table 5: The execution time and L1 and L2 cache miss ratio for different input sizes on different number of nodes with the Matrix Multiplication algorithm.

As seen in the miss ratio columns the matrix multiplication is really successful in utilizing the memory layout of the system. The data also implies the necessity to take the memory layout into account. Another conclusion from the data is that there is a need to improve the common case, as even with a large ratio of L1 hits the execution time is still to long for the system to be useful.

6 Discussion

In general, we can see from the benchmarks that the system works extremely slowly, so we can presume that it can not be used for real life applications, at least at its current state. There are however many opportunities for optimizations that we have not implemented due to time constraints. Several of these optimizations are further described in Section 7.

When looking at the results of the bucket sort benchmark, we can observe that a program that has good data locality will also have good miss ratio. Even though the system is slow, this indicates that at least the theory behind our system is solid. In these cases we only need to focus on optimizing the L1 hits to get a noticeable improvement of the system.

One thing that became apparent when we tested our system with our benchmarks, is how sensitive it is to data placement. When executing a matrix multiplication with a size that aligned with our hashing function in the L1 the performance naturally plummeted. This is a common problem with low associativity, but it is especially noticeable on our system as the speed of the L2 is so slow. But as long as the user is are aware of this limitation the user should be able to write their code in a way that works around this.

In retrospect, the hardware we chose is not the most suitable for our system. The hardware and programming model by Adapteva works very similar to an accelerator: The data is loaded on the Epiphany chips on startup, the Epiphany chips work on them and then the ARM reads them back. For that reason, communication between the ARM and the Epiphany chip is slow and there is almost no synchronization capabilities. Our system on the other hand, requires constant communication between the ARM and the Epiphany cores. At the same time, the simplicity of the eCores prevents some very important optimizations, such as zero-overhead hits in the L1, which can, for example, be implemented on Linux systems using memory protection and mapping. This presented us with some interesting problems to solve, but really limited the possible solutions and the performance of the system.

While the system is slow on the current hardware, the scaling factor is quite impressive. With this in mind, the system could be repurposed into a general DSM system, by removing all Epiphany specific parts and running it on generic PC hardware. This might combine the scalability of the implemented system with the significantly faster hardware, at which point it might actually provide a fast enough platform with a better programming model than message passing.

The power consumption seems to be lower than 5 W; around 2 W when idle, and around 3 W when executing the benchmarks. When performing the system tests for the system the power consumption increases to almost 4 W, which might be explained by higher I/O usage, as the system tests consist of fifteen independent tests which each have to be loaded from disk.

7 Future work

There are several improvements that can be introduced to the system, should the project have had more time allocated. Most of these are related to reducing the latency of the system.

Prefetching

The first major improvement to the system is the implementation of prefetching. As the benchmarks show, the system produces cache hits in many cases. A common access pattern is to iterate over an array, and the introduction of prefetching would increase the amount of cache hits even further. As an example, each cache line can contain 16 integers, and without prefetching, the L1 cache is going to miss each time we touch a new cache line, that is, every sixteenth element. Should the system fetch four cache lines at once, the miss ratio can be reduced to every 64th element.

Memory protection mechanisms

Currently, the system is compiled in as a library in the user code, meaning that the system will only run for the duration of a single executed process, after which the entire system will be destructed. This means that memory protection between processes is not an issue, since only one user process is running on the system at one time. However, the system does currently not differ between allocated and unallocated memory, in fact, the allocation function is only used to distribute the same address to each executing core. This means that a user process which accesses unallocated memory will not terminate due to memory access policy violations (or segmentation faults). This makes debugging harder, as the user may unintentionally overflow buffers and write into unallocated memory. Thus, a check at each memory access if the memory is allocated would ease development.

The introduction of memory protection mechanisms would however need to be designed in a scalable way. Since the virtual memory is distributed over all nodes within the system, the memory may be allocated remotely. The simplest way to introduce these features would be to add a message type to the MPI protocol which allows the MMU to respond with a Unallocated error message. The system may then terminate the user program or allow it to continue according to some system policy.

Statically allocated variables

In the current system, the only way to have shared memory variables is to allocate them dynamically at runtime. It would be convenient to have statically allocated variables, but implementing it would be impossible. The current system depends on using our custom pointer types and does not intercept accesses done through the normal memory system. For that reason, we are not able to intercept and redirect accesses to anything that does not use our addresses and pointers.

Shared L1

The data transfer speeds between the ARM core and the Epiphany cores is comparatively low to the transfer speed between two Epiphany cores. This means that there is a penalty to pay each time the L1 cache produces a miss and the request is forwarded to the ARM memory. To reduce the communication between the ARM and Epiphany memory, a shared L1 could be introduced. This would allow the L1 to be distributed over all the local core memory on the Epiphany side, thereby increasing the available L1 size. This would mean more hits in the L1, and faster transfer rates from the core on which the cache line is stored.

However, there are several problems that arise with this implementation. Firstly, the Epiphany cores can only run one thread at a time, this means that a memory request from one core to another can not be handled by the core on which the cache line is allocated. This means that there will, potentially, be sixteen concurrent accesses to the same cache line, which introduces race conditions. This leads to the second problem; the Epiphany platform does not provide any memory protection, and these race conditions can not be handled by interrupts and locks.

Certain applications may even suffer penalties from a distributed shared L1 cache. This is mainly due to the reduced space for local data, as part of the core memory must be reserved for the distributed L1, which means more evictions due to conflict misses. If each core is exclusively working on a piece of memory, the cost of retrieving the data from a remote node after each eviction may be more costly than retrieving the data once from the L2 on the ARM side. This cost becomes even more substantial if the time epochs are longer, as the data may be kept in the local L1 for a longer time.

Better memory barrier

The current synchronization implementation introduces a great cost, as the entire system is locked during synchronization. In programs that need to synchronize often, this may introduce a significant overhead in execution time. In addition, the current sync implementation

relies on a broadcast like algorithm where one node collects the synchronization messages from all nodes, and then sends an acknowledgement to all nodes once the synchronization message has been received from each node. This means that the synchronization is one of the least scalable parts of the system.

A solution to this problem would be to implement epoch tags on all stored data. This would mean that the only operation needed to do at synchronization is to increase an epoch counter and ignore any data that does not match the epoch the node is currently in. In more detail, the owner of a page of memory would block read calls from requesters that have a higher epoch value than themselves. At the same time, changing from a global root to a tree-like structure for any operations that require broadcast should further decrease the work each node has to perform and increase performance.

Better synchronization primitives

Currently, we only support a full memory barrier for synchronization. This is too heavy a synchronization method and quite unnecessary in most programs. Ideally, we would like to have locks that guarantee mutual exclusion and implement release/acquire semantics. In the VIPS protocol specifically, release allows us to not invalidate any of the clean data we have, so we don't have to fetch them back next time we need them.
Atomics would also be a very nice synchronization primitive, implementing similar release and acquire semantics for loads and stores, but they are much less necessary than locks, especially since it would be very hard to achieve better performance.

Read and write over cache line boundaries

A concern in the current system is that the cache implementation does not support reads and writes over cache line boundaries. This means that the whole data primitives must fit on one cache line. For example, to be able to store an integer in the cache system, the programmer must ensure that the pointer to the integer is aligned in such a way that the data is not stored over the end of the cache line. This also applies to structs, which in the current implementation may not exceed the size of a cache line (64 bytes). Implementing a solution is fairly simple, since writing back variables into the cache is done through an internal write function, which can be made to perform partial writes over multiple cache lines. The only reason why this wasn't implemented in the current system is because we find it to be a very low priority improvement and we were preoccupied with other challenges.

Change the entry point to avoid init() and finalize() in the epiphany side code

As a simple way to achieve transparency, we would like to change the entry points of our code to custom functions, allowing us to call the init and finalize functions ourselves, without the user needing to do so. This however is almost impossible to do in a portable way and it can cause quite a few problems when done incorrectly.

Other small API improvements

The API is implemented to be as transparent and similar to regular C syntax as possible. There are, however, still some quirks to working with the API.

1. Pointers to pointers has not been tested and we are not sure if it works
2. Fixing some of the more obscure bugs, such as how assigning the value of one pointer to another does not always work properly
3. Allow allocating on specific nodes rather than first-touch allocation

VIPS-M

EVI stands for Epiphany Valid Invalid, since it implements the VI coherency protocol. Naturally an improvement to the system would be to upgrade the system to use the full VIPS-M protocol. This was not feasible within the assigned time frame, but would be a logical next step for the system. The VIPS-M coherency protocol is further described in Section 2.7.

Free function

The API does not currently have a function to free allocated memory. This is because freeing is quite complex to do in a distributed way, without at the same time causing performance issues. Furthermore, many applications allocate a large amount of memory in the beginning and then work with it until they finish, so implementing a free function was assigned a low priority.

New SDK

The current implementation does not use the latest SDK for the Parallella board, since it was released in the middle of the project. That version adds support for signalling the ARM CPU from the eCores, which could be used to avoid the constant polling when sending data between them.

Adapteva also has plans for the next version of the SDK which include the addition of atomic operations. These could be used for leveraging the eMesh network and perhaps implementing a shared L1 cache.

FPGA

Memory accesses to the DRAM from the eCores go through the FPGA circuit. We could use this to have the FPGA as the controller of the L2, instead of managing it completely in software. It should be possible to have the FPGA keep a cache directory and move data between the DRAM and the SRAM, much faster than the software implementation can.

8 Conclusions

The Distributed Shared Memory system implemented has been shown to scale very well when more nodes are added. However, the system is too slow to be considered for general use. There are two main reasons for this, the first being the fact that the system is implemented in high level software, meaning that even when the system gets a cache hit, a large amount of cycles are used to retrieve the data. The second reason has to do with cache misses, in the sense that the platform the system is implemented on has very slow communication channels between different parts of the system. The local memory in every core, which holds the L1 cache, can only store around 8 kB of data, and once the system misses in the L1, communication to the L2 is very slow.

The report presents a large number of improvements and future work topics that may increase the usefulness of the system. However, the execution speed is still the major issue, and while it might be improved somewhat via the implementation of prefetching algorithms, the big change that needs to happen is the use of hardware which provides better communication channels. The current platform is designed to take input data, process it, and return one single answer, which puts much lower requirements on memory transfer speeds than the continuous fetch and writeback approach that the implemented system requires.

Finally, we would like to acknowledge that, even if the system designed and implemented was not a success, the course in it self was. We started with completely unfamiliar hardware and worked, in a group, to design a system on top of it. With the many challenges faced

throughout the duration of the course, each one of us expanded both their knowledge and their experience in a wide range of Computer Science related subjects, ranging from cache coherence and C++ templates to software design and splitting tasks between multiple programmers. To conclude with, we can say that while the end product did not meet our initial hopes and expectations, the course definitely did.

References

[KR13] Stefanos Kaxiras and Alberto Ros. "A new perspective for efficient virtual-cache coherence". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM. 2013, pp. 535–546.

Appendices

A Example Code

```
#include "QUnit.hpp"

#include "init.hpp"
#include "evi_api.hpp"
#include "perf.hpp"

#include <chrono>
#include <thread>

struct args_t {
    // Equals operator is required for all types used with the EVI pointers
    bool operator==(const args_t &other) const {
        return data_x.start == other.data_x.start &&
               data_y.start == other.data_y.start &&
               results.start == other.results.start && n.start == other.n.start;
    };
    // We are using evi_v_addr because passing pointers directly is untested
    evi_v_addr data_x;
    evi_v_addr data_y;
    evi_v_addr results;
    evi_v_addr n;
};

int main(int argc, char *argv[]) {
    QUnit::UnitTest qunit(std::cerr, QUnit::normal);
    // Initialize the EVI system
    evi_init(argc, argv);

    // Allocate shared memory for everything we are going to need
    evi::ptr<args_t> args = evi::malloc(sizeof(args_t));
    args->data_x = evi::malloc(sizeof(int) * 32);
    args->data_y = evi::malloc(sizeof(int) * 32);
    args->results = evi::malloc(sizeof(int) * 32);
    args->n = evi::malloc(sizeof(int));

    // Set the addresses in the arguments struct
    evi::ptr<int> data_x = args->data_x;
    evi::ptr<int> data_y = args->data_y;
    evi::ptr<int> results = args->results;
    evi::ptr<int> n = args->n;
    // Set the data
    for (int i = 0; i < 32; ++i) {
        data_x[i] = i;
        data_y[i] = i + 5;
        results[i] = -2;
    }
    *n = 2;

    // For debugging information
    comm_shdram dbg(CHANNEL_DEBUG);

    // Not required
    evi::sync();
    // epiphany executable to run
    char exe[] = "e_test_api_load.srec";
```

```
    // Run the executable on the epiphany cores
    evi::run(exe, args);

    // Wait for execution to finish
    int rc = evi::wait_all();

    QUNIT_IS_EQUAL(rc, 0);
    std::cout << "Epiphany is done" << std::endl;

    // Read performance data (ie cache hits and misses)
    auto prfctr = evi::get_perf_counters();

    std::cout << "L1 hits  : " << prfctr.l1_hits << std::endl;
    std::cout << "L1 misses: " << prfctr.l1_misses << std::endl;
    std::cout << "L2 hits  : " << prfctr.l2_hits << std::endl;
    std::cout << "L2 misses: " << prfctr.l2_misses << std::endl;

    // Check the results
    for (int i = 0; i < 32; ++i) {
        QUNIT_IS_EQUAL(data_x[i], i);
        QUNIT_IS_EQUAL(data_y[i], i + 5);
        QUNIT_IS_EQUAL(results[i], i * (i + 5));
        QUNIT_IS_EQUAL(results[i], data_x[i] * data_y[i]);
    };
    // std::this_thread::sleep_for(std::chrono::seconds(5));
    // QUNIT_IS_EQUAL(*args, 12*3);

    // Shutdown the EVI system
    evi_finalize();
    return qunit.errors();
}
```

Listing 1: test_api_load.cpp

```
#include "evi_api.hpp"

struct args_t {
    bool operator==(const args_t &other) const {
        return data_x.start == other.data_x.start &&
               data_y.start == other.data_y.start &&
               results.start == other.results.start && n.start == other.n.start;
    };
    evi_v_addr data_x;
    evi_v_addr data_y;
    evi_v_addr results;
    evi_v_addr n;
};

int main(void) {
    // Initialize the EVI system and read the input argument
    evi::ptr<args_t> args = evi::init();
    // Get the local ID of the core
    int id = evi::get_local_coreid();

    // Get pointers to all the data
    evi::ptr<int> data_x = args->data_x;
    evi::ptr<int> data_y = args->data_y;
    evi::ptr<int> results = args->results;
    evi::ptr<int> n = args->n;

    comm_shdram dbg(CHANNEL_DEBUG);
```

```
28
29      // Do some calculations
30      int x, y;
31
32      results[id] = data_x[id] * data_y[id];
33      x = data_x[id + 16];
34      y = data_y[id + 16];
35      results[id + 16] = x * y;
36
37      // This is done by finalize
38      // evi::sync(cache_l1::SYNC_BLOCKING);
39
40      // Sync and signal the ARM side that we are done
41      evi::finalize(0);
42      return 0;
43  };
```

Listing 2: e_test_api_load.cpp