Universitat Politècnica de València

Master Thesis

Effective Power Saving Method by On-Chip Traffic Compression in NoC-based Embedded Systems

Author:
María Soler Heredia

Advisor:
Prof. José Flich Cardo

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

July 2013

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Tècnica Superior de Ingeniería Informática
Department of Computer Engineering

Master of Science

Effective Power Saving Method by On-Chip Traffic Compression in NoC-based Embedded Systems

by María Soler Heredia

Abstract

As technology advances, multiprocessor systems-on-chip (MPSoCs) increase with the number of components, relying on an efficient on-chip network (network-on-chip; NoC). As the size of the system increases, NoC performance and power consumption become a central issue.

In this project, we design a compression strategy at the NoC level reducing the number of transmitted flits and consequently the energy consumed. The provided mechanism relies on the abundance of memory data blocks filled with zeros in the analysed applications, thus easily compressible by using a zero-elimination strategy. We provide a hardware implementation for both compression and decompression end points at a generic network interface (NI). The mechanisms have been designed in isolated mode in order to make them modular and easily adapted to any NI protocol. Results show the effectiveness of the compression and decompression mechanisms and the low overhead they introduce. The percentage of traffic reduced by the compression strategy (it is reduced by a factor of 3) justifies the added resources.

This work reflects some parts of the main research directions we tackle in the wider PhD framework. In particular, we propose a method for power efficient memory traffic management. The work presented here represents the initial research directions in simulation development, traffic pattern characterization and initial solutions development.

Contents

List of Figures

List of Tables

Acronyms

ACRP	**A**ligned **C**onsecutively **R**epeated **P**attern
ADDR	**Addr**ess
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
AXI	**A**dvanced e**X**tensible **I**nterface
CMP	**C**hip **M**ulti**P**processor
DST	**D**e**ST**ination
FIFO	**F**irst **I**n **F**irst **O**ut
Flit	**Fl**ow control dig**it**
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FT	**F**lit **T**ype
ID	**ID**entifier
MC	**M**emory **C**ontroller
MOESI	**M**odified - **O**wned - **E**xclusive - **S**hared - **I**nvalid
nACRP	**n**on-**A**ligned **C**onsecutively **R**epeated **P**attern
nAnCRP	**n**on-**A**ligned **n**on-**C**onsecutively **R**epeated **P**attern
NI	**N**etwork **I**nterface
NoC	**N**etwork-**o**n-**C**hip
NZ	**N**on-**z**ero
OCP	**O**pen **C**ore **P**rotocol
SAGN	**S**et-**A**side **G**ather **N**etwork
SRC	**S**ou**RC**e

Chapter 1

Introduction

Networks-on-Chip (NoCs) [1] have been accepted as the way to efficiently implement communications on a chip. In the embedded domain, the new multiprocessor systems-on-chip (MPSoCs) multiple cores (different IPs) are integrated and interconnected with a specialized NoC.

Initial multiprocessor systems counted tens of components which could be interconnected with standard buses such as AMBA [2]. However, as technology evolves, the number of processors on a chip increases and the bus has soon become a natural bottleneck, being thus replaced by a network (NoC). During the last ten years, ever since the appearance of the NoC concept, the research community and part of the industry have been exploring and defining NoC systems. The focus was mainly on the efficient implementation of the network to connect all the components. Basically, the research performed on off-chip high-performance interconnects for the last decades has been transferred to the NoC domain. Well stablished concepts as routing, switching, flow control, and arbitration have been adopted and adapted to this new environment. For a complete overview of all these aspects, the reader can see [1].

Moreover, the fast evolution of these systems has overtaken the NoC capacity, both in terms of performance and power consumption. Thus, currently the focus has moved towards optimizing these NoC systems with specialized and customized strategies, specially adapted to the nuts and bolts of this new domain. In particular, the applications that the MPSoC system is running need to be taken into account in order to identify

optimization opportunities. In this work we follow this trend and focus on a particular set of applications and possible future system configurations.

Different techniques have been proposed in the last years to improve NoC performance and power management. Many studies [3, 4] have arisen to find ways to better exploit these capacities on the network. One approach to save power consists in data compression: data streams sent through the NoC can be compressed at injection and decompressed at ejection, saving power and even latency (shorter messages in the average case). This is specially beneficial for long messages, since they allow greater compores-sion rates. In this Master Thesis we have focused on improving the power consumption by compressing the traffic, thus reducing the amount of data to be sent between nodes. We first studied the traffic needs of two representative target applications in order to find a way to optimise the system.

In most MPSoCs nowadays a coherence protocol is needed to provide coherence among caches and with main memory. We thus tackle the compression of NoC traffic assuming a coherence protocol running on top of the NoC. This is becoming a general trend since ARM-based systems may accept coherence systems as the baseline design [2]. Block memory transfers between the memory controller (MC) and the cache hierarchy have been chosen as the compression target. We focused on Main Memory accesses, (meaning most of the traffic on the NoC). In particular Figure 1.1 shows our baseline target system: a 2D Mesh interconnecting multiple processors with an underlying coherence protocol.

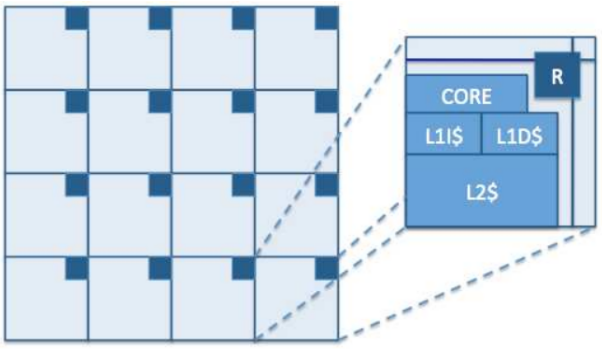


FIGURE 1.1: Target System

This Master Thesis represents the initial research of the overall framework of our planned research work. We can see in Figure 1.2 that the general purpose of this broader research is optimizing memory traffic on NoC-based systems. This is our final target since it is our belief that Main Memory, as an important hot-spot of every NoC-based MPSoC,

is becoming a limitation for their performance. For the present Thesis some parts of this research have been performed, including traces collection, simulation development and our parallel compression technique. Along with these lines of work, we have started working on MC modelling and analysis, as well as in data and control decoupled networks, they being still in progress. Other future work will include studying the benefits of prestablishing circuits to limit traffic latency among other optimization techniques.

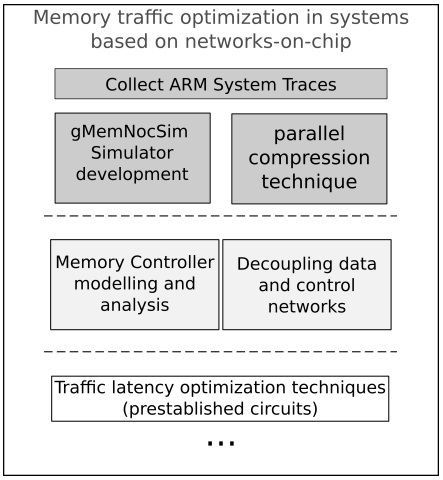


FIGURE 1.2: Global research framework

The rest of the text is organized as follows. In Chapter 2 the Related Work is summarized. Then, in Chapter 3 the Compression Opportunities research s shown; in Chapter 4 we present our main contribution, including the baselines and formats proposed as much as our compression technique; Chapter 6 shows the results obtained are shown together with a description of the final system to close in Chapter **??** by giving our final Conclusions and publication list.

Chapter 2

Related Work

When considering related work, we must distinguish two different subjects, Network Interfaces (NIs) as a whole and proposals of compression strategies for NoCs. Since designing a NI is not within the scope of our research, we have focused on NoC compression strategies. A basic data compression book is [5], which we used to have a general view of the possibilities in compression, comparing the nature of our target data and the needs of each compression philosophy in order to take the most compliant choice. For the sake of organization, this will be extensively discussed in Chapter 3.

Compression of transmitted information does not seem to be a largely explored field. Many examples can be found in the literature on test data compression for NoCs, but on these studies the data is compressed offline, off the network, and only decompression is applied in the NIs. Some examples can be found at [3][6][7][8][9][10] among others. Since compression is not performed in the NoC, the strategies explained in these papers are not applicable to our needs. Other works [11] also deal with compression of the routing table, which again is not what we were looking for. Some papers however study NoC compression strategies, which require special attention. Some examples of these are [12][13][14][15][16][4].

In [12], a real-time compression technique that reduces the amount of bits sent is presented. The USBR technique removes the bits which do not change, and sends some extra information (which bits change and how long the data block is). This technique aims at data streams where the most significant bits are less likely to change than least

significant bits, and thus it is not applicable to the type of data involved in our research (messages that travel between MC and the cache hierarchy).

The authors in [13], integrate compression and decompression with a coherence protocol. Basically, a protocol processing core is included in multi-core nodes to perform both the coherence protocol, compression and decompression. This is a specially interesting techinque since it takes into account the coherence protocol. On the other hand, avoiding unnecessary modification of the existing components and independence from other components or protocols are goals that prevent us from chosing an option that adds an extra core.

In [14] two compression strategies are presented. The first one, named Cache Compression is performed between the L1 and L2 (inside the cache) and is meant for storage (squeezing more data in the same sized cache structure). This strategy is not transmission-oriented. The second one is called Compression in the NIs and consists in compressing the information right before injecting it. It is thus transmission-oriented. The pitfall of this second strategy is that it needs to integrate such modules in every NI, when it is not necessary for us, and that it uses a table-based frequent-pattern compression.

Adaptive data compression for on chip network performance optimization is presented in [15]. It uses a table-based strategy, improving it by using shared tables and pipelining compression and injection of the flits. It is interesting, but the nature of the message characteristic to the applications of interest makes using a table redundant, since the compression target are long strings of zeros.

A similar approach is shown in [16], except that instead of frequent patterns, it is based on frequent values. The authors claim that with a small code-book for end to end communications, that does not need to be the same among different cores, they reduce power consumption in the router up to a 16.7%. This proposal is again not applicable to our needs because the use of a code-book would be redundant.

Finally, in [4] an interesting proposal is presented: Although it uses tables (in fact it needs not only compression tables but also candidate ones, increasing redundancy), it allows to completely eliminate some transmissions (in the best case), sending only a short

message and using matching status bits in the directory, which simplifies and fastens miss handling.

The particular solution discussed here is not compliant with any of these found in previous literature. It is a new proposal based on the massive appearance of zeros in the messages between MC and the cache hierarchy in the applications analysed in the framework of the vIrtical project. Should further study show that this abundance of zeros is an anomaly, and other common applications lack this characteristic, a strategy such as the one presented in [4] would be a good option.

Chapter 3

Compression Opportunities

In this chapter we analyze the compression opportunities for the applications provided to us in the vIrtical Project. We first describe the traces acquisition method, then we briefly expose the main techniques that can be found in previous literature, followed by our analysis results and the conclusions we obtained, which lead the choices we took in the rest of our compression technique design and analysis.

3.1 Trace Acquisition Methodology

Since ARM systems are very broadly used for embedded systems, we chose to comply with their architechture for our study. In these multiprocessor systems, all communications between L1 and L2 are managed internally, so we need to capture communication between L2 and main memory. However, we traced loads and stores at the L1 level due to limitations in FastModels simulator. It nonetheless provides a more complete and accurate model of the system. We also need to know the memory contents in order to analyze the data transferred. We decided tracing when the the applications enter parallel mode. We have obtained traces with five of the many different algorithms in the OpenSSL suite, three hash algorithms (SHA1, SHA256 and SHA512), and two encryption algorithms (AES-128-ECB and AES-256-ECB). We chose these since they are the most relevant ones according to THALES point of view. The environment selected to obtain traces and simmulate can be seen in Figure 3.1.

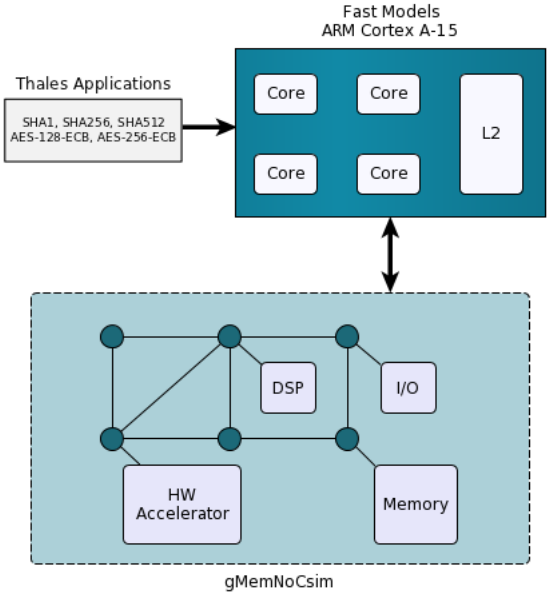


FIGURE 3.1: Traces collection and simulation environment

To obtain the traces we used FastModels and gMemNocSim to analyse them. We will introduce FastModels here and leave all explanations about gMemNocSim for Chapter 4. ARM FastModels simulator provides out of the box programmers view models of the ARM processors. It is thus both functionally accurate and easy to use since ARM processors models are already implemented as an Instruction Set Simulator. We use this simulator to model the quad-core Cortex-A15 MPCore part of the target system and to run on top of this the targeted applications.

The model of Cortex-A15 provided with FastModels is capable of running basic applications, but it does not cover all the requirements of an operating system, which is needed to evaluate and benchmark parallel applications. We thus use a more complex model also provided with FastModels (namely RTSM-VE Cortex-A15) that allows the simulation of both operating systems and applications. In this RTSM-VE model, as seen on Figure 3.2, the cores are connected directly to a Versatile Express platform through a 64-bits AXI bus. This platform includes the Motherboard Express uATX, which has been especially designed to support future generations of ARM processors, and the CoreTile Express daughterboard with the on-board DDR2 SDRAM.

The motherboard provides the following features:

- Peripherals for multimedia or networking environments.

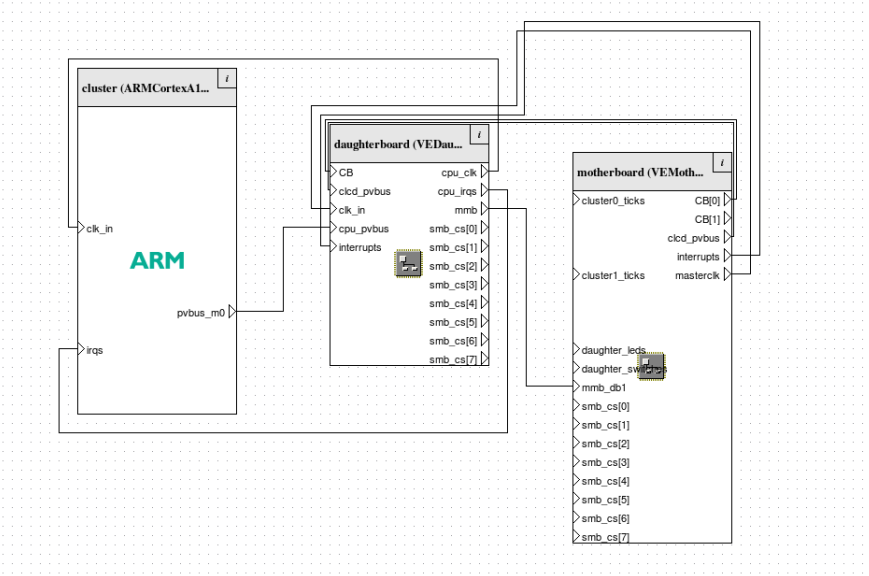


FIGURE 3.2: RTSM-VE Model in Fast Models

- All motherboard peripherals and functions are accessed through a static memory bus to simplify access from daughterboards.
- Consistent memory maps with different processor daughterboards simplify software development and porting.
- Supports FPGA and processor daughterboards to provide custom peripherals, or early access to processor designs, or production test chips.

FastModels supports the use of a Model Trace Interface (MTI) plug-in that permits us to consistently track the execution of the model. Through implementing an MTI plug-in for tracing memory accesses produced by the cores and adding it to the simulation we are able to trace exactly what we needed in the form that we required. The ARM simulator offers other alternative tracing methods but they would either modify the system behavior when using RTSM-VE model or be prohibitively time consuming.

MTI plug-in provides many different sources to trace, but the more verbose the trace obtained is and the more sources are involved, the more it slows down the simulation. Since it takes billions of instructions to boot a Linux system on FastModels, we need to deactivate the output and minimize the number of sources of the tracing until the starting point of the segment of interest is detected. We have achieved an acceptable compromise solution by capturing only the instructions fetched by the cores until we

reach a special nop, and subsequently tracing loads, stores, and fetches until we get to the ending special nop. These special nops were introduced in the code in order to mark out the portion of interest.

The ARM Simulator provides programmers view models with some limitations. On system simulators there is a trade-off between speed and accuracy: very accurate simulators lack in speed whilst fast simulators cannot be totally accurate. FastModels in particular opts for the execution speed thus lacking some features needed for our analysis, such as:

- Instruction timing: A processor issues a set of instructions (a.k.a a quantum) at the same point on the simulation time, and then waits some amount of time before executing the next quantum, being impossible to determine the right time each individual instruction is executed.
- Bus traffic: bus traffic has several optimizations that make it inaccurate. We were able to deactivate some of those optimizations but not all of them, at the expense of making the simulation far slower and still not accurate to the level required.
- It does not support out-of-order execution and write-buffers as architecturally defined: execution on FastModels is only an approximation to execution of architecture and it must be thus considered.

As mentioned above, traces must be obtained in the exchange of L2 and main memory to be used for compression opportunities analysis, including memory contents. Since we could not find a proper MTI source to trace at this point, we used the traces obtained from FastModels (loads and stores) and fed them to gMemNoCsim. gMemNoCsim reproduces the communication between all different levels of the memory hierarchy translating loads and stores into coherent requests to main memory. In turn, main memory contents were obtained in FastModels through the use of CADI debug interface when the starting trace point was detected.

As for gMemNoCsim, it is a cycle accurate event-driven NoC simulator developed at UPV by the parallel architectures group (GAP), which allows high precision in modeling the key elements of NoCs, including all the key design aspects: as topologies, router, routing algorithms, schedulers and flow control mechanisms. In its current version, gMemNoCsim also models cache coherence protocols on top of the NoC.

gMemNoCsim has the capability of working with synthetic traffic, with real traces or it can be fed on-line with the output of a different simulator. As stated before, in our case we feed gMemNoCsim with the memory access traces obtained by running the applications on Fast Models. Thus, gMemNoCsim simulates the memory hierarchy (converting memory loads and stores in memory block accesses flying between L2 cache and main memory over the NoC). This will provide the data needed to analyze compression opportunities.

gMemNoCsim has been enhanced, within the framework of this Thesis, to add support for actual data exchange between all levels of the memory hierarchy. Also, tracing capability of gMemNoCsim has been provided to cope with the needs of compression analysis. Finally, the compression mechanism has been developed in the simulator to analyse the compression rate achieved.

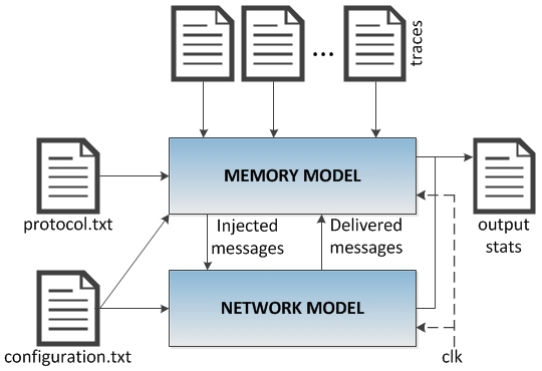


FIGURE 3.3: Simulated Scenario

In Figure 3.3 we observe the scenario we simulated in gMemNocSim. As we can see, we use the traces obtained from FastModels as an imput, customizing the simulation with a configuration file that indicates gMemNocSim the topology, routing, etc, and a coherence protocol file; gMemNocSim implements the network with all the parameters introduced and obtains a file of results as an output.

Finally, we describe the trace and memory files format. In the case of FastModels, we need to obtain traces with the information required for coherency modeling, including the core id (to characterize the number of sharers of the block), the address (to identify the block being accessed), the type of access (to classify the block as data or instructions and to discriminate writes from reads) and the data (for compression analysis). Traces obtained using gMemNoCsim must include address (to analyze whether or not address

compression can be an interesting technique), and data (the entire block contents to analyze compression opportunities). Source and destination have been included for readability.

Traces obtained from ARM FastModels using the MTI plug-in are as follows:

core	address	type	data
0	001ea10c	l	b6f6713c
0	b6f6713c	f	e92d001f
0	bef1de68	s	0020787c
0	bef1de10	bs	bef1eef4,bef1ef10,bef1f984,001a0c9c,00000003

Where type refers to: l (load), s (store), f (fetch), and b applied to l or s (burst load-store). When a burst is detected the data field is extended to the total amount of data exchanged. Both addresses and data are coded in hexadecimal format.

Traces obtained from gMemNoCsim traces are as follows:

Address	origin	destination	data_block
0x00207840	Memory	L2Cache[0][481][0]	2030,69,0,0,1,88,0,0,0,0,52,3,1,4,0
0xdf04a040	L2Cache[0][129][9]	Memory	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

Where origin and destination can be either Memory or L2, including as indexes the node (in case several L2 are present), index and way. Addresses are coded in hexadecimal format and data is coded in decimal format. Memory content obtained from FastModels is as follows: Memory is organized as a vector of words which can be scanned translating

Data
00000000
00000000
03e03dbf
801000c0
00000000
...
...

address to line number. To keep a backup for subsequent executions, a copy of the memory file is made when starting to simulate the model on gMemNoCsim and all memory exchanges are made on the copy.

3.2 Compression Overview

Here we briefly describe the main techniques proposed in literature (you can find more information about them at [5] for example):

- Zero elimination: techniques of this type compress code or data by eliminating long strings of zeros, either by codifying or by just removing them. Removing part of the message (long zero strings) with no further coding seemed most interesting to us (note that the eliminated part must be aligned, the necessary alignment depending on the particular technique).
- LZn: the basic idea of this method is to use part of the input stream as a dictionary, maintaining a sliding window divided in two parts: the search buffer (left, already coded) and the look-ahead buffer (right, to be read). When a match between both buffers is found, a pointer to that possition in the buffers is created and sent.
- Table-based compression techniques: these techniques create tables with highly frequent patterns and avoiding to send the whole pattern by sending the corresponding index of the table instead. There are many different techniques with different philosophies, but they don't require compressible data to be aligned or its occurrences to be consecutive.

As we have seen, there are many different compression techniques and they imply different data analysis. In order to be able to choose the compression technique that best suits our particular case of study, we have obtained a variety of data statistics not to constraint the final decision by applying a specific analysis method. We have thus used three analysis strategies to find repeating patterns:

- Aligned Consecutively Repeating Patterns (ACRP). Patterns in this category are consecutive and aligned to byte size. One example is 000000010000000100000001 where the pattern 00000001 is repeated three times (consecutive repetitions) and is byte aligned.
- non-Aligned Consecutively Repeating Patterns (nACRP).. This category groups those patterns that are still consecutive but not necessarily aligned to any data

size. One example is 11110000000100000000100000001 where the pattern 00000001 is repeated three times (consecutive repetitions) and is not byte aligned.

- non-Aligned non-Consecutively Repeating Patterns (nAnCRP). This category groups those patterns not necessarily aligned to any data size and not consecutive, this is: a pattern that is repeated along the trace but we do not consider if it is consecutively repeated or not. One example is 11110111100100000011111000001111111, where the pattern 1111 is repeated 5 times (not necessarily consecutive) and alignment is not considered.

This study was made before deciding what compression technique was to be used and it is thus general. Byte-alignment was arbitrarily chosen, a different alignment possibly resulting in sligtly different results, but all in all the general conclussions obtained in this part of the research would remain the same. Notice that the ACRP category helps us localizing the occurrence of long strings of aligned zeros (for zero elimination case). However, in order not to lose generality, we localize also all patterns that have the properties referred above. In this case, the nACRP category is also relevant, because it allows us to study the suitability of LZn techniques. In turn the nAnCRP category allows us to assess the convenience of applying table-based compression techniques.

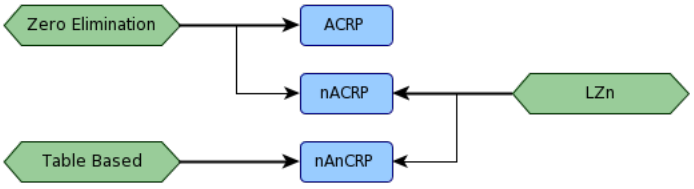


FIGURE 3.4: Relationship between studies made and compression techniques

The relationships between compression techniques and analyzed pattern categories are shown in Figure 3.4

3.3 Analysis Results

First, we analyze in detail, in the three following subsections, the application OpenSSL with the algorithm sha1; and in the forth subsection we compare the obtained results against the other 4 algorithms. Finally, in the fifth subsection, some conclusions are

drawn. In all subsequent analysis, pattern size is one byte, block size is 512 bits (64 bytes) and, when applicable, byte alignment is used.

3.3.1 Aligned consecutively repeated patterns (ACRP)

We first present a graph with general information with the number of occurrences of ACRP patterns. Figure 3.5 shows the results where X axis is the number of consecutive repetitions and Y axis (left) is the total number of occurrences. Notice that Y has a logarithmic scale. The figure also shows the variability (right Y axis) in the number of total occurrences among the different byte patterns that have the same amount of consecutive repetitions. When no red line is present, either only one pattern exists for that category or all included patterns have the same number of occurrences (less likely to happen). Note that in most cases, when no variability is present, all patterns are made of all-zero strings.

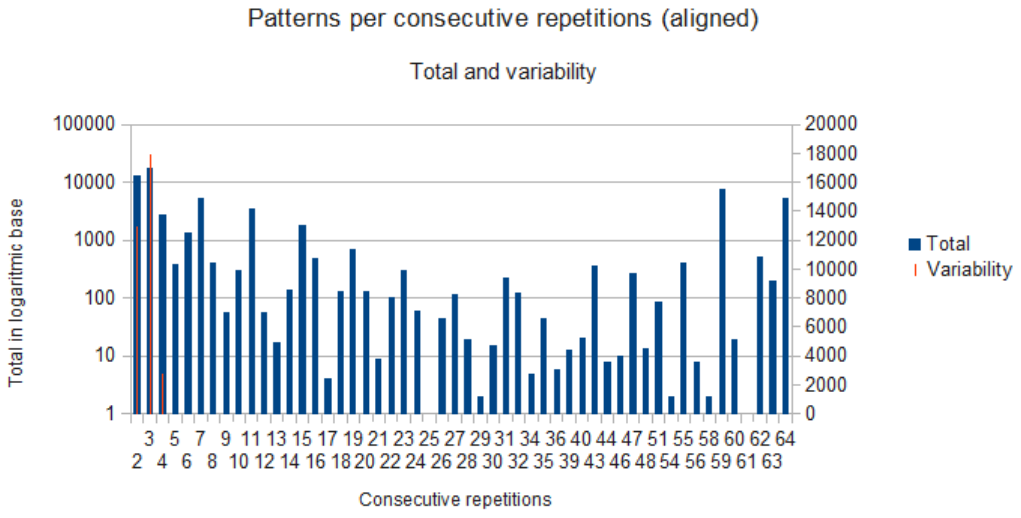


FIGURE 3.5: ACRP total occurrences per number of consecutive repetitions.

The first thing to notice is the large disparity in the number of occurrences of patterns (notice the logarithmic scale). Large occurrences of ACRP patterns occur either with low repetitions or with large number of repetitions. This figure demonstrates that ACRP patterns exist and thus, there is an opportunity to compress those patterns.

In Figure 3.6 we can see the most interesting (most frequent) ACRP patterns found. We select those ACRP patterns with at least 1000 total repetitions. The number of

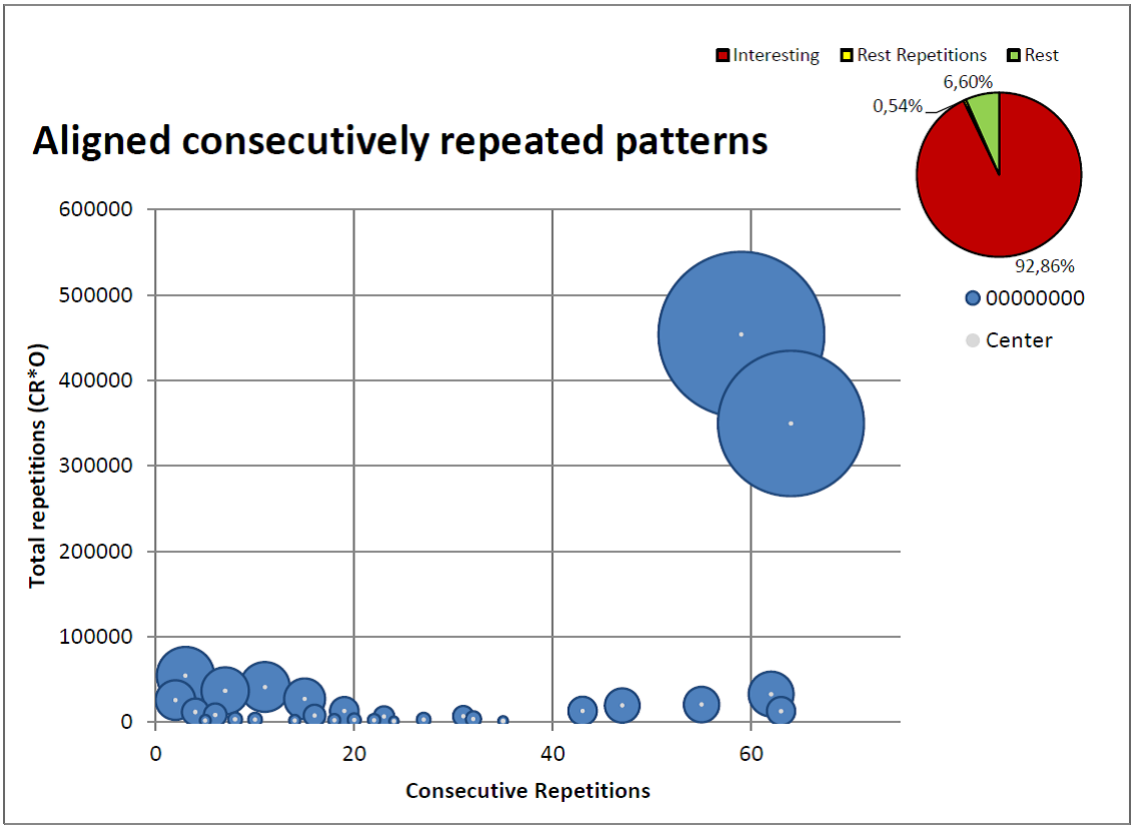


FIGURE 3.6: Non aligned consecutively repeated patterns

total repetitions is obtained by performing the product of consecutive repetitions by occurrences. The bubble size represents the proportion of total data constituted by this pattern. One first outcome is the fact that the set of interesting ACRP patterns represents a large amount of traffic (92.86% of traffic) with only a small number of patterns. The rest of detected patterns represents 0.54%, conformed by many different byte patterns repeated only a few times. This means a large potential exists to compress data traffic. Most interesting is the fact that the largest part of ACRP patterns is made of all-zero streams. Also, the largest set corresponds to long streams of zeros, those represented by the two largest bubbles at the upper right side of the figure.

Since we noticed in the previous graph that most compressible data is composed by long zero strings, we now analyze other alignments of zeros. In this case we care about the block size, as will be the default transmission data size (blocks between caches). Notice that we will compress within individual messages and not between data stretched over several blocks/messages. Figure 3.7 shows the amount of ACRP all-zero patterns for different string sizes (multiple of a quarter of a block) and different alignments (not-block

aligned, aligned to block, aligned to end of a block). As can be seen, the vast majority of ACRP repetitions correspond to large strings (either 48 or 64 bytes in length) and they are practically always block aligned. This will simplify the compression mechanism to be used in the final system.

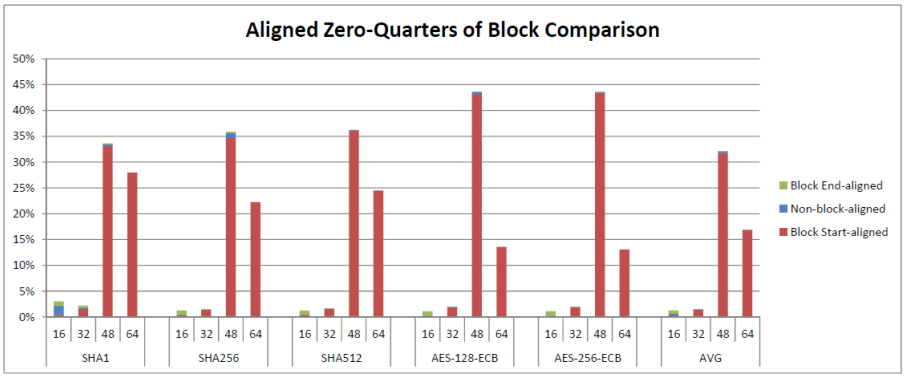


FIGURE 3.7: ACRP quarters of blocks of aligned zero strings

In order to understand what this really means we can see an example on the Figure 3.8. In this figure we observe that data must be at least a quarter of block long to be compressible, but is not mandatory that it is aligned. Although some data compressible with other techniques is not compressible with this one, this technique has the advantage of simplicity and low hardware overhead, so it must be analyzed. To end with Figure 3.8, it is easy to see whether the block is start or end-aligned observing the blue bars.

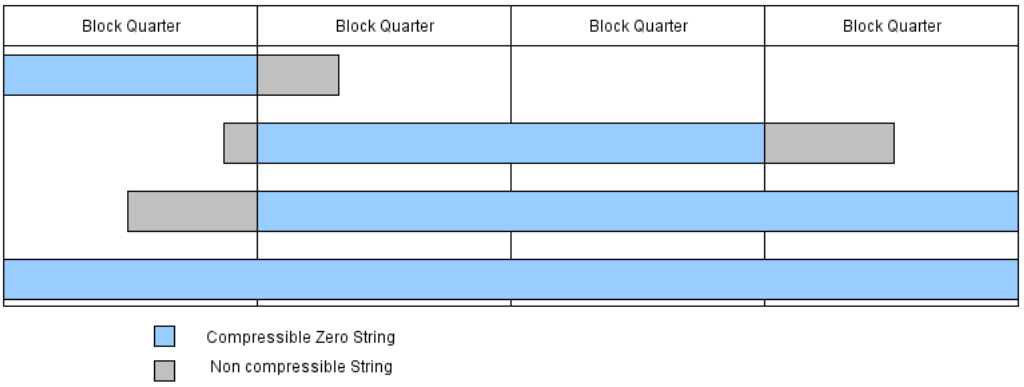


FIGURE 3.8: Examples of block alignments

3.3.2 Non-aligned consecutively repeated patterns (nACRP)

Figure 3.9 shows non-aligned consecutively repeated patterns (nACRP). The X axis corresponds to consecutive repetitions and the Y axis represents the total number of repetitions (the product of consecutive repetitions and the number of occurrences of these, which is the amount of data compressible). The bubble size is determined by the percentage of bytes of the total. The threshold chosen is 1000 total repetitions. With this threshold we capture 93.45% of the patterns. We can observe that compressible zero strings conform 100% of total interesting data according to this analysis. Notice that these results are similar to the ones achieved for the ACRP case, indeed, the only difference is whether patterns are found aligned or not. As we have many aligned patterns, they also are detected in this analysis and, thus, make results quite similar.

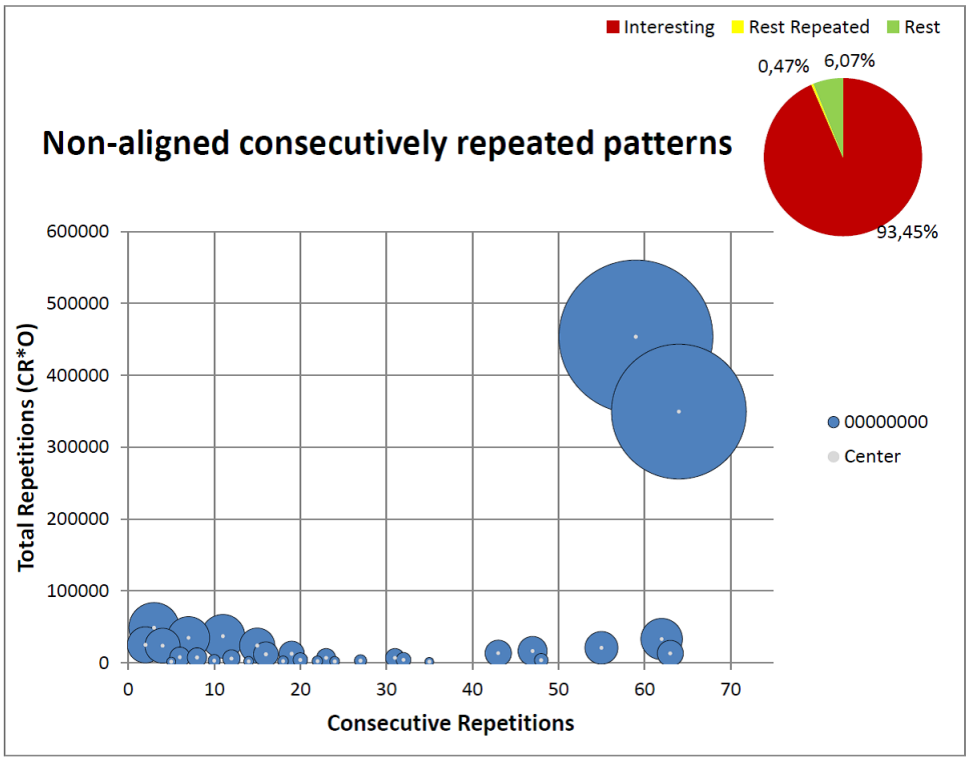


FIGURE 3.9: nARCP total repetitions (consecutive repetitions * occurrences) per pattern

These are the most interesting patterns since byte-alignment is not required for zero-elimination in our particular case (since we need flit-alignment). This will be further explained in Chapter 4.

3.3.3 Non-aligned non-consecutively repeated patterns (nAnCRP)

Figures 3.10 and 3.10 show non-aligned non-consecutively repeated patterns (nAnCRP). The small cake indicates what percentages from total is represented by the big one (96.60% and 95.28% respectively).

Non-aligned non-consecutively repeated patterns are difficult to analyze, since we find a combinatorial explosion when trying to obtain all possible agregations of all patterns (excluding the overlapping ones). There is thus a trade-off between completeness and time consumption needed for the analysis. We chose to select patterns in order with two different priority schemes: first prioritizing 0's strings (Figure 3.10, from 00000000 to 11111111) and then prioritizing 1's strings (Figure 3.11, vice versa). We do this to show that no matter what order we choose to select patterns (even if the scenarios are opposite), zero byte strings are of most importance (97.44x96.60 = 94.13% and 91.06x95.28 = 86.76%) for compression.

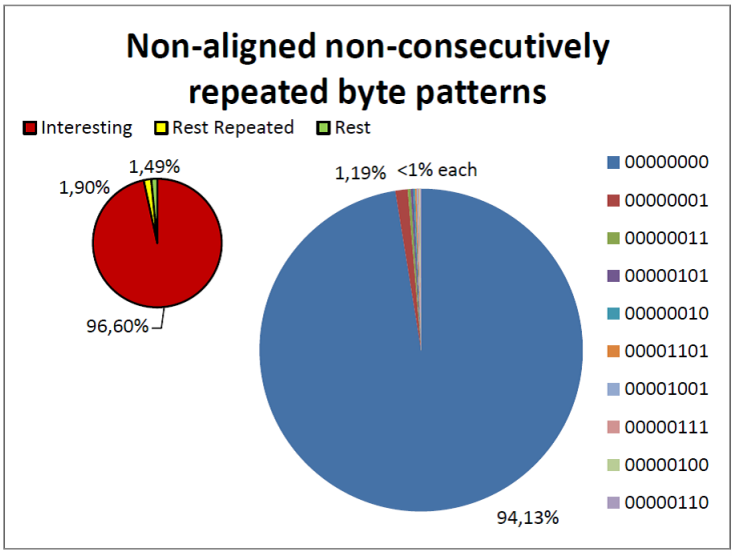


FIGURE 3.10: nAnCRP most repeated patterns (starting by "00000000")

3.3.4 General Analysis of Compression Opportunities for applications

In this subsection we compare the statistics obtained for OpenSSL with algorithm SHA1, that have been profusely analyzed in the previous three subsections, with the other algorithms to be considered, i.e. SHA256, SHA512, AES-128-ECB and AES-256-ECB. We start with a comparison of the general behavior of the data obtained for all algorithms;

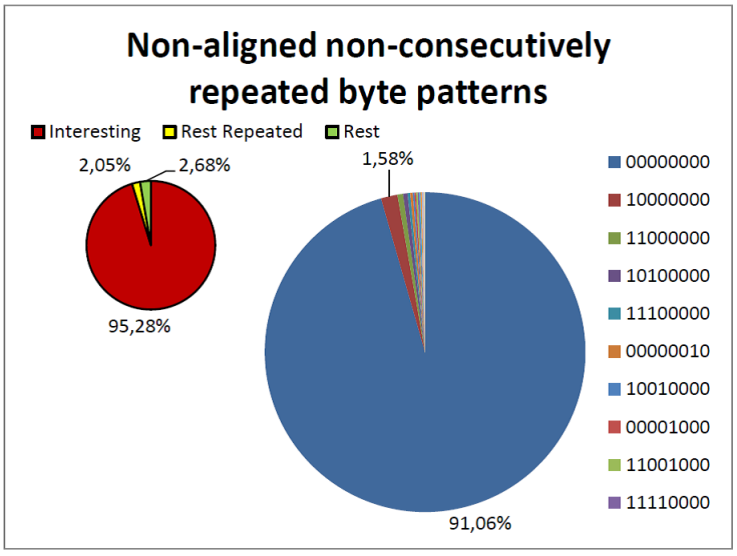


FIGURE 3.11: nAnCRP most repeated patterns (starting by "11111111")

to finish with a comparison of the particular patterns designated as the most interesting according to our criteria.

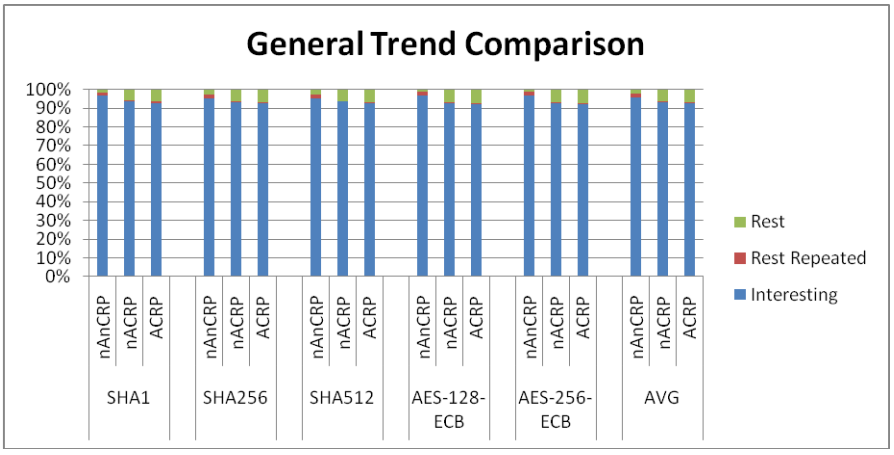


FIGURE 3.12: General Trend Comparison

In Figure 3.12 we can see that the pattern repetition behavior of all the algorithms is very similar for the three repetition schemes analyzed. However, for all the algorithms, nAnCRP exhibits the higher compression opportunities, but notice that this repetition scheme implies a higher complexity. So we must find a compromise solution.

In the reminder of this subsection, we show a comparison of the most interesting patterns for each algorithm and each technique. We consider interesting patterns those with a number of total repetitions (consecutive_repetitions * number_of_occurrences) of 1000 or more. We first analyze in Table 3.1 and Figure 3.13 nAnCRP that includes a larger

variety of patterns and then show statistics for nACRP (Table 3.2) and ACRP (Table **??**), where 00000000 is the only pattern that reaches the aforementioned threshold.

TABLE 3.1: Comparison of Most Interesting Patterns (nAnCRP)

(nAnCRP)	SHA1	SHA256	SHA512	AES-128-ECB	AES-256-ECB
00000000	94,13	93,78	93,98	93,37	93,26
00000001	1,19	1,34	1,28	1,55	1,60
00000010	0,18	0,24	0,26	0,26	0,32
00000011	0,28	0,34	0,37	0,29	0,28
00000100	0,10	0,11	0,10	0,10	0,10
00000101	0,24	0,32	0,33	0,44	0,43
00000110	0,09	0,10	0,00	0,00	0,00
00000111	0,11	0,11	0,12	0,11	0,14
00001001	0,13	0,13	0,13	0,12	0,13
00001011	0,00	0,00	0,10	0,00	0,00
00001101	0,16	0,09	0,08	0,12	0,08
10011010	1,58	0,08	0,09	0,22	0,14

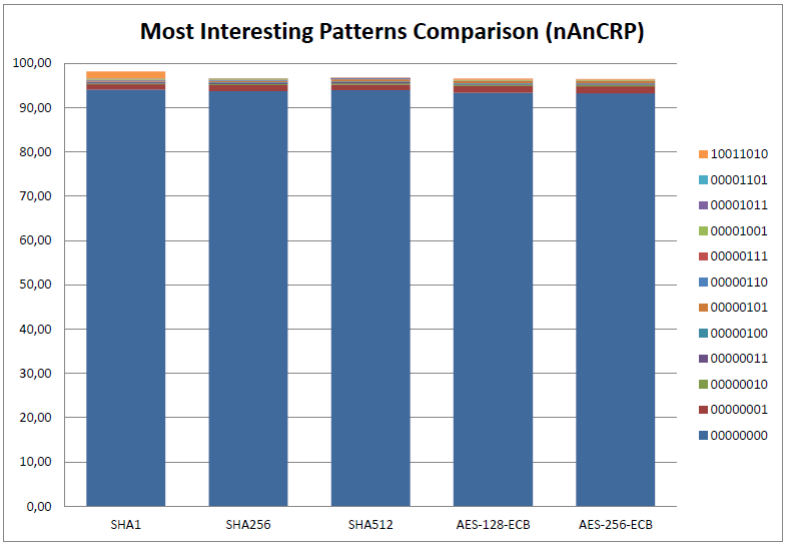


FIGURE 3.13: Most Interesting Patterns Comparison (nAnCRP)

In Table 3.1 we have highlighted those patterns that were not included among the interesting ones for an algorithm by assigning them a zero value and writing them in red color. The interesting patterns are quite similar for all algorithms and the percentages represented by them are very similar as well. Zeros represent over 90% of the data in all cases, so it can also be induced that the compression techniques to consider hereafter should be zero-elimination techniques.

Table 3.2 and Table **??** show that the proportion of zeros in both repetition schemes (nACRP and ACRP) is very high and very similar. We can deduce that most zero strings are long (consecutive repetitions of zero byte strings) and many of them aligned to byte (since percentages for ACRP are just a little bit lower than for nACRP).

TABLE 3.2: Comparison of Most Interesting Patterns (nACRP)

(nACRP)	SHA1	SHA256	SHA512	AES-128-ECB	AES-256-ECB
00000000	93,45	93,26	93,42	92,83	92,74

3.3.5 Conclusions for compression opportunities

From the previous analysis we have different conclusions:

1. The most important streams of data are made of all-zeros. This means that those streams should be the primary focus for compression opportunities.

2. All-zero elimination methods should be prioritized, analyzed and researched.

3. Non-aligned consecutive patterns are the most interesting pattern and zero-elimination the most interesting technique.

4. Non-consecutive patterns, although largely found in the applications analyzed, will be largely covered for the all-zero streams. Indeed, most of the non-consecutive patterns are made of all-zero streams.

Chapter 4

Compression Mechanism

In the present chapter we present the main contribution performed in the framework of the present Master Thesis. This includes a section with the foregoing considerations, where we define formats and the baseline NI along with the compression mechanism.

4.1 Foregoing Considerations (Formats, Baselines...)

We start this Chapter with an overview of the Baseline Network Interface we have selected. We continue with a description of the strategy we follow to compress messages, including packet formats.

4.2 Baseline Network Interface

A basic NI has been taken as a baseline, thus not adopting any current NI technology: we only take into account the existence of a decoupling buffer for the injection of packets, and a reception buffer for the ejection of packets. Communication between the core and the NI is not affected and is completely orthogonal to the provided technique.

Two different solutions may be applied: parallel and sequential compression. Parallel compression is more resource demanding, but it is also a better informed technique, allowing for a more effective compression. In parallel compression the end node transfers a whole memory block to the NI (512 bits in our case) plus some extra information, so the

NI needs to allocate a message-size buffer, with the corresponding buffering overhead. At ejection, however, the NI needs to reconstruct the message so some extra latency is incurred before delivering the complete message to the end node.

In this research two different steps have been effected: a study of the characteristics and patterns of messages between MC, and the cache hierarchy; and a study of a particular technique to compress these messages. The solution we propose is independent of the use of virtual channels. Moreover, an NoC with different channels can be built up by physically replicating one VC-less network. Also, the baseline flit format will be specified and customized to the compression solution provided. Compression information needs to be injected together with the packets so that it can be decompressed at destination. Also, information for the coherence protocol must be included (source node address and memory address). Flit size is assumed to be 32 bits.

With these assumptions, we achieve a high compression rate, thus reducing long-message traffic by a factor of 3, by introducing an overhead of 8.91% in area and 24% in power. However, the large compression factor achieved allows to cancel and even obtain positive power reductions.

Figure 4.1 shows the schematic of the NI used as a baseline. It implements two ports connected to the network, one in each direction (for injection and ejection). Therefore, injection and ejection of messages are supported by the NI. The NI has one buffer where messages to be injected are allocated by the injection logic. The injection logic is driven by the node connected to the network (through the NI) and is specific of the protocol used (AMBA, OCP, etc.) Likewise, the NI has an ejection buffer where messages received from the network are temporarily stored and delivered to the node. The logic to read from this buffer is also protocol dependent. The size of both buffers will be customized to different configurations, both in number of slots (1, 2, 4 & 8) and slot width (32 & 558).

The NI has three extra logic blocks, all of them implemented in the solutions provided. The header builder logic is in charge of preparing the header of the flits of the message to be injected. At both sides of injection/reception flow control logic is also implemented: the Stop&Go protocol. Finally, as shown in the figure, we add two logic blocks for the compression/decompression solution. These blocks are intimately linked to the

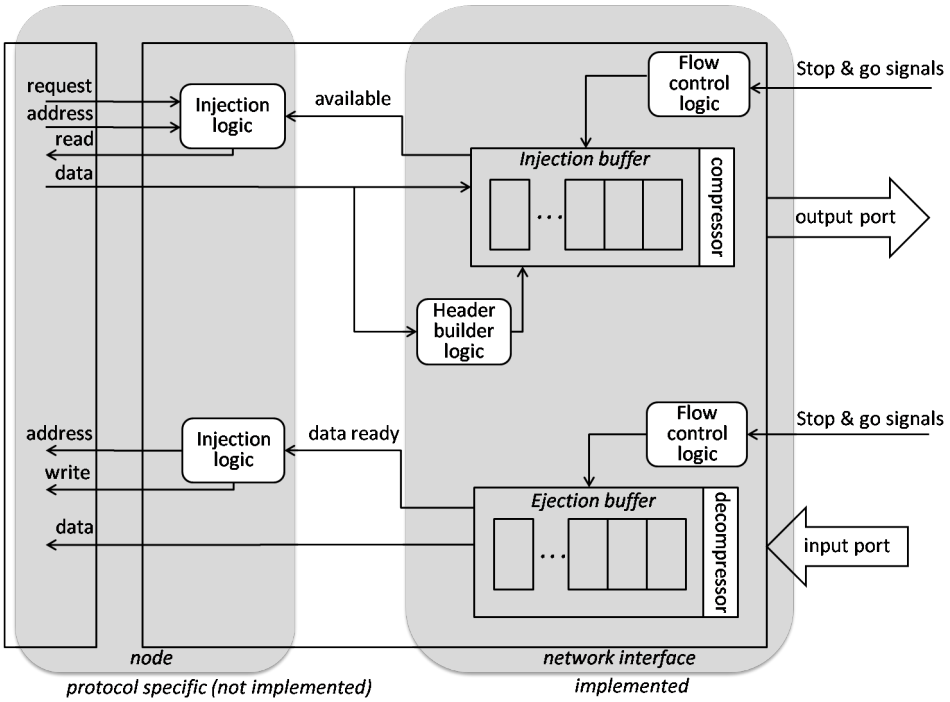


FIGURE 4.1: Baseline NI

buffers. We analyze the overhead of such solutions in terms of area overheads, operating frequencies, and power consumption.

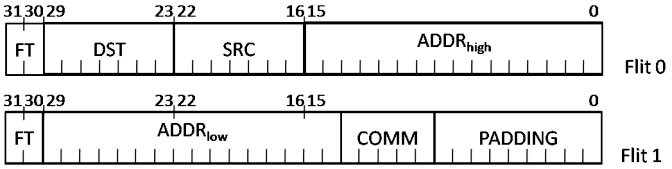
The NI evaluated in this work does not support virtual channels. Indeed, injection and ejection buffers consist of a single FIFO queue each. Virtual channels are an interesting addition to a network on chip, mostly for performance issues (boosting network throughput) but also for correctness reasons. For instance, in a request-reply protocol, both types of messages need to be decoupled by traveling through different virtual channels. Otherwise, protocol-level deadlocks can be induced. To fully support virtual channels and to decouple request messages from reply messages, we opt for a replication of the NI. Indeed, only the standard (not implemented) part shown in the above figure will be replicated. The network can also be replicated providing separate resources for each traffic class. In addition, the results obtained in terms of area must be multiplied by the number of virtual channels needed in order to get an approximate estimation. Indeed, most of the area needs of the NI are caused by the buffers, which need to be replicated for the support of the request-reply protocol.

Our compression solution focuses on memory blocks. Therefore, short messages triggered by the coherence protocol are not targetted, meaning that compression logic needs to

be applied only to the channel used by long messages carrying memory blocks. This motivates also for a design of the NI in isolation of the virtual channel requirements, since the final solution will require a NI for the request layer with no compression logic built-in and a NI for the reply layer with built-in compression logic. Since we provide results for different configurations, it will be easy to obtain the final requirements for a system implementing a request-reply protocol with and without the compression mechanism.

4.3 Baseline Packet Format

The NI assumes the packet format shown in Figure 4.2 and Figure 4.3. The first one corresponds to short messages generated by the coherence protocol. The protocol will trigger a request (coded in the COMM field) to a specific destination end node (coded in the DST field) and with a 32-bit address (coded in the ADDR field). The format also codes the source end node sending the message (coded in the SRC field). Notice that a short message will be compounded in two flits injected through the NoC.



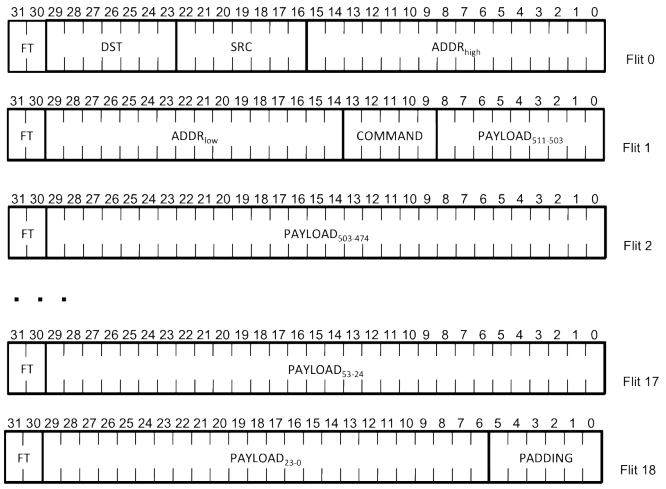
FT	Description
00	Non-valid
01	Tail
10	Payload
11	Header

FIGURE 4.2: Short packet format (baseline configuration)

Each flit includes two bits to code the flit type. Only two types are possible for short messages: Header (11) and Tail (01). Although we could use only one bit to code the type, we prefer to use two bits in order to make it compatible with long message formats.

For long messages more data needs to be injected, thus packet format is made of up to 19 flits. The first flit is identical to the short packet format one, including destination, source and part of the address. The second flit also includes the remainder bits of the address, and instead of the padding, a 5 bit coherence command and 9 bits of the payload (the most significant bits). The payload is the memory block that is being transmitted, (512 bits). Flit 1 to Flit 18 transport part of the payload (memory block). The final flit

includes a pading of 6 bits. In long messages, three flit types are used: header (11), tail (01) and payload (10).



FT	Description
00	Compressed payload (or non valid)
01	Tail
10	Uncompressed payload
11	Header

FIGURE 4.3: Long packet format (baseline configuration

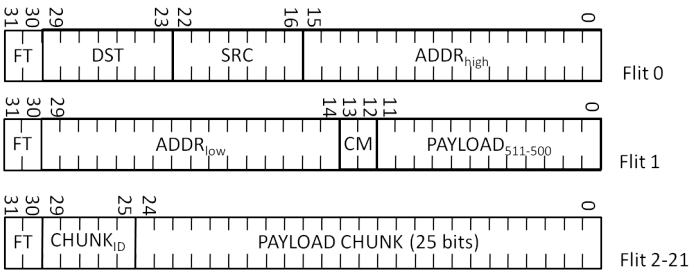
4.4 Parallel Compression

In this section we develop the strategy to compress messages with a parallel approach, where the whole message is simultaneously accessed by the NI. Long packets carry a memory block, which is the target of our compression strategy. 64-byte memory blocks are divided in 20 chunks of 25 bits each and a remainder set of 12 bits (the highest order bits of the memory block). 25-bit chunks have been selected as this allows a perfect match with both flit and packet width. If all the bits of a chunk are set to zero, then the chunk is not transmitted over the network. In order to support chunks and align them to flits, we need to define the packet format for long messages.

In Figure 4.4 we see the packet format for long messages: FT is a two bit flit-type field in every flit; DST and SRC (7-bit each) are respectively the destination and source of the message; $ADDR_{high}$ is made up of the most significant half of the memory address (16 of 32 bits), while $ADDR_{low}$ corresponds to the least significant half; CM is a command

of the of the coherence protocol; $CHUNK_{ID}$ is the flit-number (2-21) and PAYLOAD is the part of the message the flit carries.

If a memory block with all its bits set to zero arrives, the strategy will only inject into the network flits 0 and 1. This is the maximum compression achievable with this strategy. Notice that these two flits carry extra fields (address, source, destination), which are typically non-zero.



FT	Description
00	Non valid
01	Tail
10	Payload
11	Header

FIGURE 4.4: Long packet format (memory blocks) for parallel compression

Figure 4.5 shows an example where a memory block is processed and 4 flits are injected. Flits #2 and #3 correspond to two chunks with some of their bits different from zero.

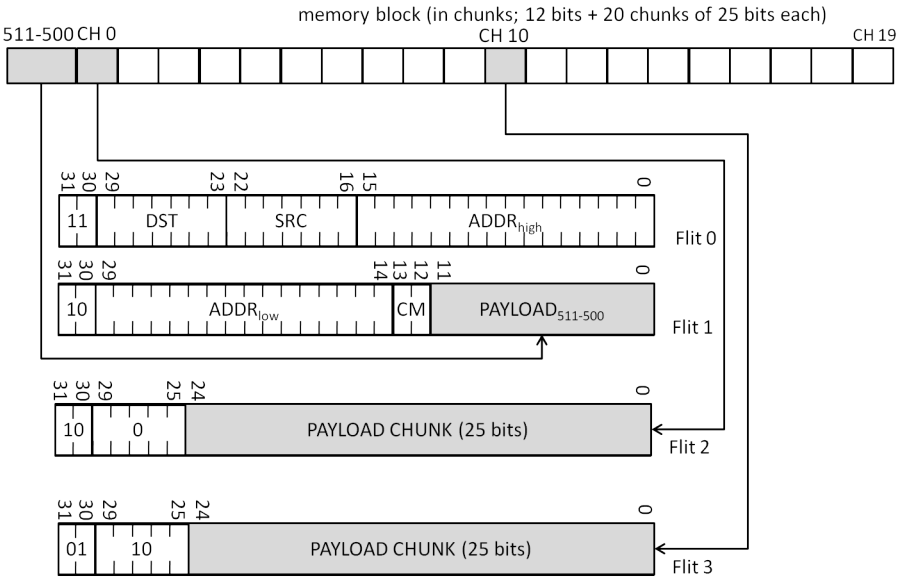


FIGURE 4.5: Example of parallel compression

Thus, for a memory block with all its bits set to zero, the strategy will only inject into the network flits #0 and #1. This is the maximum compression rate that can be achieved with this strategy. Notice that the two flits carry other information fields (address, source, destination), which are typically non-zero fields.

4.5 The Resulting Parallel Compression Mechanism

Figure 4.6 shows the logic details for the parallel compression implementation. In this approach we make the slot width of both injection and ejection buffers equal to the size of the message. Thus, when a node requests to inject a new message, the injection logic copies the entire message into the injection buffer (if there is any free slot).

When arriving to the NI from the node, messages are injected together with control information (destination node, source node, and memory address) thus adding 558 bits in total (packet). The whole packet is treated as payload, although the first 60 bits (the first two flits) are never compressed. Those bits correspond to the source, destination, address, and the 12 most significant bits of the message.

The buffer slot is logically divided in chunks of 25 bits, excluding the aforementioned 60 bits (chunks 0 & 1). Chunks #2 to #21 are sent to the OR stage where all-zero chunks will be detected. The resulting output of the OR stage will be used to compute the flit type as well as to select the next flit to inject into the network. This selection drives the multiplexer shown in the figure. Upon an injection of a flit, the associated output of the OR stage is reset in order to allow the injection of the next flit for the packet. Let us describe each stage details.

Figure 4.7 shows the OR stage. First, all the bits of the message are exposed to the OR stage, and a 22-bit register stores the output of the OR gate for every chunk (notice that the first two chunks are not computed with an OR gate but are always set to one). The register (Nz) will keep account of the chunks that need to be injected (i.e. are non-zero). This register is written once per message injected when the message is exposed at the head of the queue (in the first slot), and can be driven by the write signal of the buffer. The Nz register output is then fed to the FT/ID selection logic (Figure 4.8). This logic selects the next flit to inject by means of a priority encoder with all the Nz bits as input. The encoder selects the most significant bit set to one. The Select signal is then used to

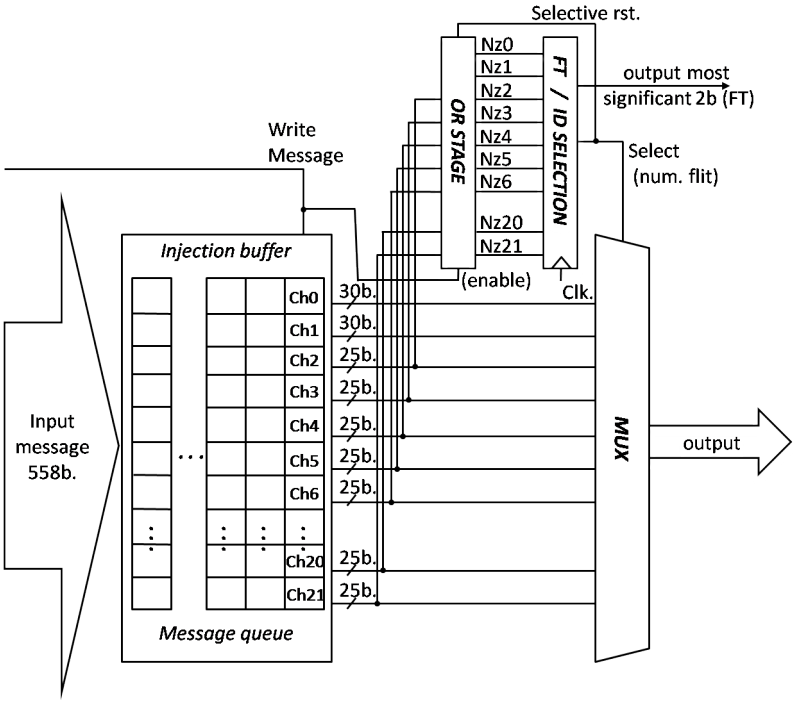


FIGURE 4.6: Injection NI with compression (general view)

drive the multiplexer at the injection port, as well as to reset the appropriate Nz bit so that, at the next cycle, a different chunk is selected (if any).

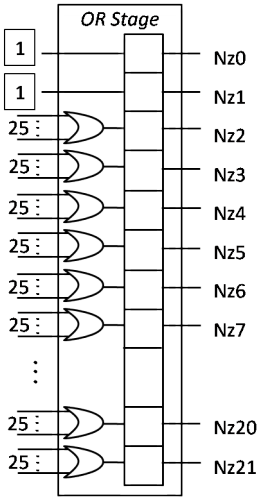


FIGURE 4.7: OR stage of compression

In addition, the FT/ID selection logic in Figure 4.8 also computes the flit type (see the table in Figure 4.4). The most significant bit of the flit must be set to one for header and payload flits, and set to zero only for a tail flit (notice that a flit can not be header and tail at the same time since compressed packets consist of two flits minimum). Similarly,

the second bit of the FT field (bit 30) has to be set to 1 for a tail or header flit and to zero only for payload flits. See the table in Figure 4.3. Therefore, the complexity lies only in computing which of the chunks is the tail (last flit to be sent). This is achieved by an additional priority encoder, this time with the Nz bits input in reverse order. Hence the last chunk with the Nz signal set (meaning the chunk is non-zero chunk) is identified: with a comparator of the two outputs of the priority encoders, upon a match, the last flit being injected is identified as the tail flit.

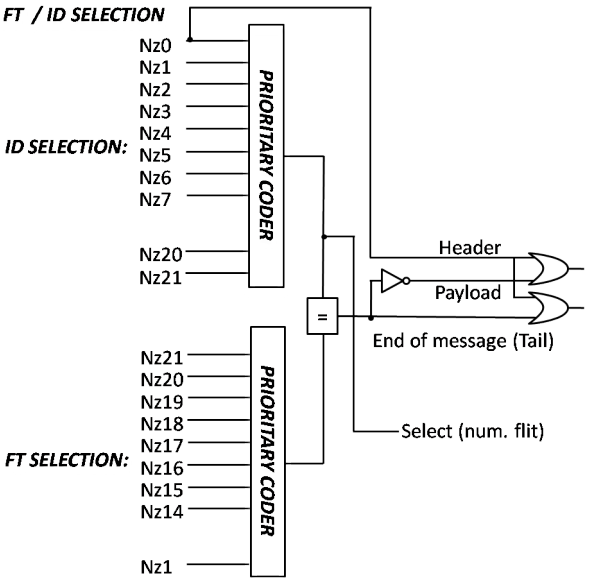


FIGURE 4.8: FT/ID selection stage of compression

The decompression strategy, shown in Figure 4.9, is much simpler since we do not need to calculate anything. When the first flit arrives (with the FT field set to head) the 30 remaining bits are placed at the beginning of the reception buffer and the 25-bit chunks are reset. The second flit is also automatically copied right after. If the second flit is not the tail, all subsequent flits are placed according to their flit address (bits 29 to 25 of the flit). The tail flit activates the end of message signal to indicate that the message is complete and the buffer slot is full.

Notice that untransmitted chunks need to be coded at destination as zeros. Thus, the receiving slot is initialized every time to all zeros.

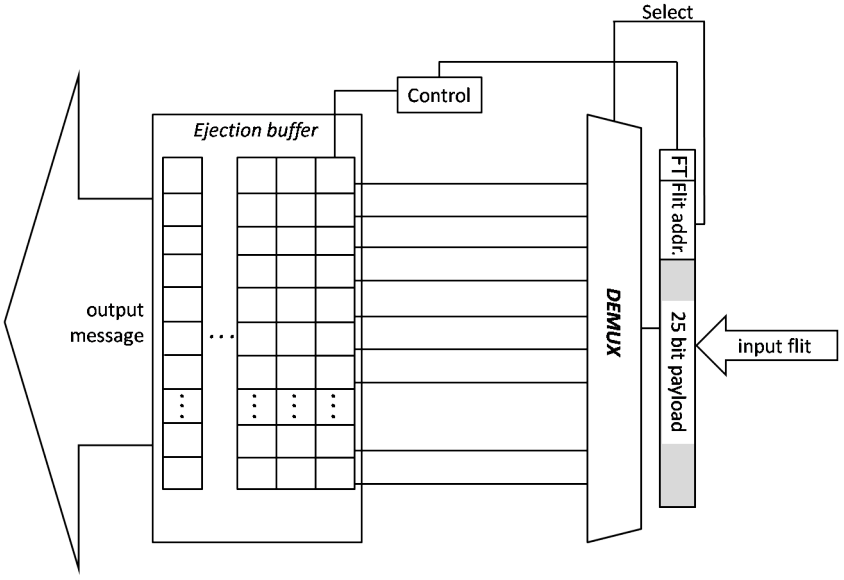


FIGURE 4.9: NI ejection with compression mechanism

Chapter 5

Evaluation Results

In this section we provide the results of analysing both performance of the compression strategy, and the overheads of the implemented solutions. Therefore, we present the results in two separate parts. In the first part, we provide compression effectiveness by calculating the percentage of traffic being really compressed. For this, we use our simulation platform and inject the application memory access traces obtained in the vIrtical framework. In the second part, we focus our attention on the implementation overheads, showing results for area overheads, operating frequencies, and power consumption estimations of the implemented modules.

The combination of both parts indicates the effectiveness of the compression solution. After the evaluation, the discussion section is focused on the combination of both parts.

5.0.1 Compression Rate Achieved

One important aspect of the compression mechanism is its effectiveness. For this, the compression rate achieved has been analysed by simulation with gMemNoCsim. Four cores are attached to the same router. The cores are modelled by private L1 caches. Also, an L2 cache is implemented and attached to the same router. The memory controller (MC) is attached to a neighbour router.

An invalidation directory-based coherence protocol with MOESI is implemented. Flit size is set to 32 bits, short messages sized to 64 bits (2 flits) and long messages sized to 512 bits (block size).

Figure 5.1 shows the number of flits injected into the NoC (the traffic between the L2 cache and the memory controller). There is a significant traffic reduction in all the application cases analyzed. On average, traffic is reduced by a factor of 3.5.

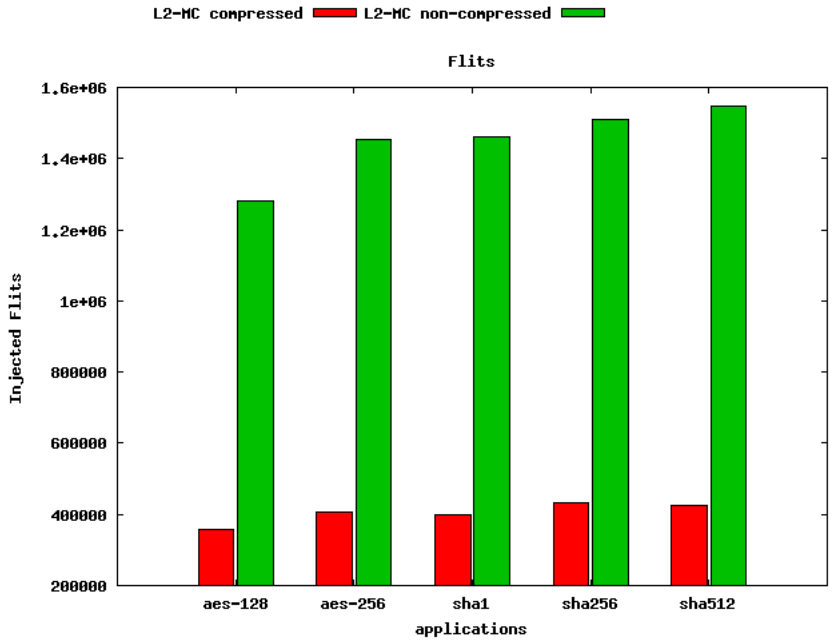


FIGURE 5.1: Number of flits injected into the NoC

The impact in execution time is negligible as shown in Figure 5.2, since memory blocks accesses between the MC and the L2 bank are not in the critical path of the application.

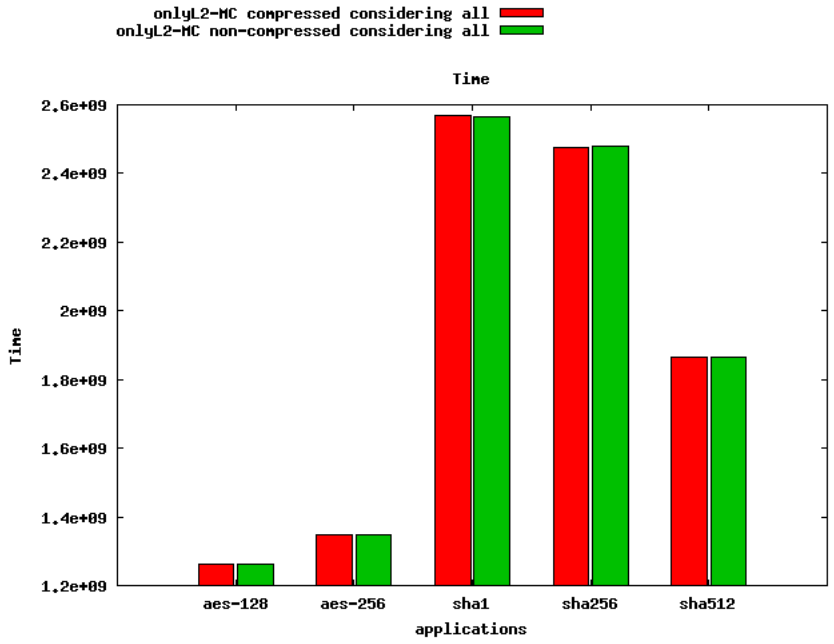


FIGURE 5.2: Execution time of application cases

5.0.2 Implementation Overheads

The compression design has been synthesized using the 45nm technology open source Nangate [45nm. FreePDK] with Synopsys Design Compiler 7 [17]. Table 5.1 shows the area for two models of the injection part of the NI, one with no compression logic added (baseline) and one with the compression logic. We show results for different number of slots (ranging from one to eight). Notice that slot width is set to 558 bits (a whole large message and its header per slot).

TABLE 5.1: Area overheads. Injection. 558-bit slots

	No compression		Compression		Overhead
Slots	Area	/1 slot	Area	/1 slot	
1	6587.09	1	7768.6	1	17.94%
2	12156.7	1.85	13313.9	1.71	9.52%
4	23444.7	3.56	24789.3	3.19	5.74%
8	45839.2	6.96	47911.5	6.17	4.52%

As we increase the number of slots we achieve larger area overheads. Indeed, the buffer is the most demanding area resource. This trend is seen in both injection logic implementations.

For the ejection part of the NI, see Table 5.2, we see different trends. As the number of receiving slots increases (each slot is 558 bits wide), the area overhead increases. When looking at the compression overheads, it ranges from no overhead for one slot to 14% for the 8-slot solution.

TABLE 5.2: Area overheads. Ejection. 558-bit slots

	No compression		Compression		Overhead
Slots	Area	/1 slot	Area	/1 slot	
1	5710.4	1.00	5741.8	1.00	0%
2	11168.3	1.95	11788.5	2.05	6%
4	22334.2	3.91	25120.6	4.37	12%
8	45823.3	8.02	52134.6	9.08	14%

Now, let us turn our attention to power consumption. Power has been estimated by using Cadence Encounter 7 [18] (both for Place&Route and measurements). Table 5.3 shows the power consumed by the injection part of the NI with and without compression capabilities (baseline vs. compression). As we can see, power consumption trends keep

similar to the ones achieved when no compression is included. It has to be noted also that the power consumption achieved by injecting less flits onto the network is not accounted in these tables.

TABLE 5.3: Power consumption. Injection. 558-bit slots

Slots	No Compression	Compression	Overhead
1	3.104	5.455	75.74%
2	7.733	12.62	63.19%
4	16.14	25.72	59.36%
8	34.39	50.86	47.89%

Finally, for power consumption, Table 5.4 shows a comparison of power consumption for the ejection part of the NI, when no compression and compression is used. Trends are similar and power consumption overhead is negligible.

TABLE 5.4: Power consumption. Ejection. 558-bit slots

Slots	No Compression	Compression	Overhead
1	6.231	6.475	4%
2	13.32	12.59	-6%
4	24.93	28.03	12%
8	56.22	57.07	1%

In Table 5.6 we show the minimum periods necessary for each buffer to work properly. These results were obtained by using PrimeTime [19].

TABLE 5.5: Timing overheads. Injection. 558-bit slots

Slots	No compression	Compression	Overhead
1	0.63	1.44	128.57%
2	0.66	1.74	163.64%
4	0.77	2.04	164.94%
8	0.92	2.46	167.39%

TABLE 5.6: Timing overheads. Ejection. 558-bit slots

Slots	No compression	Compression	Overhead
1	0.50	0.55	10%
2	0.65	0.67	3%
4	0.68	0.78	15%
8	0.77	0.83	7%

The previous results show the area and power overheads of different parts of the NIs, also for different configurations of number of slots. Now, we need to compound those results for a possible NI being used by the coherence protocol. This means, the NI will be made of different components, servicing different message classes of the coherence protocol. In detail, the protocol triggers short and long messages and only long messages, carrying memory blocks are subject to be compressed. This means, the NI needs to be built with different injector and ejector configurations.

Table 5.7 shows the area overheads and power consumption of the NI with no compression mechanism included but for slot sizes of 32 bits. The table shows the results for both components, injection and ejection. These components will be used to handle short control messages of the protocol.

TABLE 5.7: Area and power. Ejection. 32-bit slots

Slots	Area	Power
1	314.32	0.309
2	655.20	0.826
4	1321.25	1.791
8	2643.48	3.589

With all these results, now we can build the overheads of a final NI with the following characteristics:

- Injection of long messages with compression facility enabled (1 slot)
- Ejection of long messages with compression facility enabled (1 slot)
- Injection of short messages with no compression facility (2 slots)
- Ejection of short messages with no compression facility (2 slots)

For the number of slots at each component we need to consider the flit size. Indeed, wide slots will need several network cycles to inject the whole message, while thin slots will need fewer cycles. Therefore, it is reasonable to use few slots for long messages and more slots for short messages. We select one slot for large messages and two slots for short messages. With all these considerations, we derive the results by adding the different overheads previously presented. Table 5.8 shows the results achieved.

TABLE 5.8: Overhead of the whole NI.

NI component	Area	Overhead	Power	Overhead
Inject long	7768.6	17.94%	5.455	75.74%
Eject long	5741.8	0%	6.475	3.91%
Inject short	655.20	0%	0.826	0%
Eject short	655.20	0%	0.826	0%
Total	14820,8	8.91%	13.582	24%

As we can see, the overhead is significant but the potential reduction of the number of flits injected in the network (Table 3.2) suggests that the global power consumption will be as well reduced. To corroborate this, we have derived the approximate power consumption of the network per flit for both the presented baseline NI implementation and our compression-enhanced implementation. In the graph in Figure 5.3 we can see the difference in power consumption for different injection rates. In every graph, the X-axis corresponds to the compression rate [0:1] and the Y-axis corresponds to the difference in power consumption between our solution and the baseline (we have taken as a baseline a NI with one injector and one ejector of 2 32-bit slots each). We have only taken into account the long message network, since the other network is exactly the same in both cases and would be nullified.

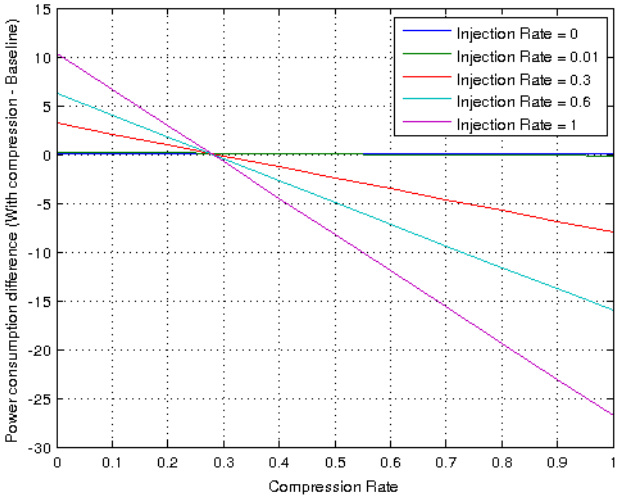


FIGURE 5.3: Study of the difference in power consumption of our solution and the baseline depending on compression-rate for different injection rates

In the graph in Figure 5.3 we can see that depending on long message injection rate, we can save different amounts of power, but in any case the compression rate needed is

0.3 (we need to send at least 30% less flits in order to get some power benefit). Since our compression rate is much beyond the needed 30%, it is assured that with data that follows the pattern of the applications studied some power would be saved.

Chapter 6

Conclusions and Publications

In this final chapter we will state our conclusions and relate the publications resulting from this research.

6.1 Conclusion

In this Thesis we have provided the results of our recent research, including those obtained in the frame of the vIrtical project. Our goal was to design and analyse compression strategies at the NoC level, saving communication costs, by reducing the number of transmitted flits. The provided mechanism relies on the abundance of memory data blocks filled with zeros, thus easily compressible by using a detection strategy.

Our compression mechanism avoids sending flits without information (i.e. not sending flits that only consist of zeros). This technique is both easily applied and highly efficient. This compression scheme is only applied to long messages (a full memory block, 512 bits), not to short messages (of only 2 flits).

The results displayed in the previous section show the effectiveness of the compression and decompression mechanisms and the low overhead they introduce. The percentage of traffic reduced by the compression strategy (a factor of 3.5) justifies the overheads in resources for compression and decompression. As we can see in Table 5.8, the area overhead for the compression and decompression mechanisms required for a system with coherence support is 8.91% whereas the added power consumption is 24%.

In the graph in Figure 5.3 we can see that depending on long message injection rate, we can save different amounts of power, but in any case the compression rate needed is 0.3 (we need to send at least 30% less flits in order to get some power benefit). Since our compression rate is much beyond the needed 30%, it is assured that with data that follows the pattern of the applications studied some power would be saved.

6.2 Publications

The results obtained during the completion of this Thesis include:

- Compression opportunities identified and analyzed in deliverable *D 1.1 Acceleration, memory system opportunities, and virtualization requirements in selected industrial applications* in the vIrtical Project.
- Publication in Summer School 2012 of title *Memory coherence and compression in the vIrtical Project* presenting a Poster of the same title.
- Compression strategy designed and analyzed in deliverable *D 4.1 Hardware support for compression mechanisms at the NoC level* in the vIrtical Project.
- *Effective On-Chip Traffic Compression in Embedded Systems* a paper accepted in JP013 (Jornadas de Paralelismo 2013) to be presented in September.
- A paper submitted to OMHI2013 (On-chip memory hierarchies and interconnects: organization, management and implementation) titled *Power saving by traffic compression on the NoC*. The resolution of this Workshop is pending and will be resolved July 8th 2013.

6.3 Acknowledgements

This work has been supported by the VIRTICAL project (grant agreement n 288574) which is funded by the European Commission within the Research Programme FP7.

References

[1] J. Flich and D. Berlozzi. Designing network on-chip architectures in the nanoscale era, 2010.

[2] ARM Limited. Amba axi and ace protocol specification, 2013.

[3] A. Chandra and K. Chakrabarty. Frequency-directed run-length (fdr) codes with application to system-on-a-chip test data compression, 2001.

[4] B. An, M. Lee, K. Yum, and E. Kim. In *Efficient Data Packet Compression for Cache Coherent Multiprocessor Systems*, DCC, 2012.

[5] David Salomon. Data compression: the complete reference, 2004.

[6] M. Tehranipoor, M. Nourani, and K. Chakrabarty. Nine-coded compression technique for testing embedded cores in socs. *IEEE Transactions on VLSI Systems*, 13: 719–731, 2005.

[7] L. Ciganda, F. Abate, P. Bernardi., M. Bruno, and M. Sonza Reorda. An enhanced fpga-based low-cost tester platform exploiting effective test data compression for socs. In *International Symposium Design and Diagnostics of Electronic Circuits & Systems, 12th. DDECS '09*, pages 258–263. IEEE, 2009. URL http://iie.fing.edu.uy/publicaciones/2009/CABBS09.

[8] J. Dalmasso, E. Cota, M. Flottes, and B. Rouzeyre. Improving the test of noc-based socs with help of compression schemes. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '08, pages 139–144, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3170-0. doi: 10.1109/ISVLSI.2008.86. URL http://dx.doi.org/10.1109/ISVLSI.2008.86.

[9] V. Froese, R. Ibers, and S. Hellebrand. Reusing noc-infrastructure for test data compression. In *VTS*, 2010.

[10] S. Chaki, C. Giri, and H. Rahaman. Binary difference based test data compression for noc based socs. In *ISVLSI*, pages 114–119. IEEE, 2012. ISBN 978-1-4673-2234-8. URL http://dblp.uni-trier.de/db/conf/isvlsi/isvlsi2012.html#ChakiGR12.

[11] M. Palesi, S. Kumar, and R. Holsmark. In *A method for router table compression for application specific routing in mesh topology noc architectures*, SAMOS, 2006.

[12] S. Ogg and B. Al-Hashimi. In *Improved Data Compression for Serial Interconnected Network on Chip through Unused Significant Bit Removal*, VLSID, 2006.

[13] L. Vittanala and M. Chaudhuri. In *Integrating Memory Compression and Decompression with Coherence Protocols in Distributed Shared Memory Multiprocessors*, ICPP, 2007.

[14] R. Das, A. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. Yousif, and C. Das. Performance and power optimization through data compression in network-on-chip architectures. In *HPCA*, 2008.

[15] J. Yuho, K. Yum, and E. Kim. In *Adaptive data compression for high-performance low-power on-chip networks*, MICRO 41, 2008.

[16] P. Zhou, Y. Zhang, J. Yang, and Li Zhao. Frequent value compression in packet-based noc architectures, 2009.

[17] *Design Compiler User Guide*, 2011.

[18] *Encounter User Guide*, 2011.

[19] *Time User Guide*, 2011.