ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

DESIGN AND IMPLEMENTATION OF A RUN-TIME RESOURCE MANAGER FOR MALLEABLE APPLICATIONS ON NETWORK-ON-CHIP (NoC) ARCHITECTURE

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βασίλης Σ. Τσούτσουρας

Επιβλέπων: Δημήτριος Σούντρης

Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

DESIGN AND IMPLEMENTATION OF A RUN-TIME RESOURCE MANAGER FOR MALLEABLE APPLICATIONS ON NETWORK-ON-CHIP (NoC) ARCHITECTURE

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βασίλης Σ. Τσούτσουρας

Επιβλέπων: Δημήτριος Σούντρης
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Ιουνίου 2013

....	....	....
Δημήτριος Σούντρης	Κιαμάλ Ζ. Πεκμεστζή	Νεκτάριος Κοζύρης
Επ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αναπ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2013

....
Βασίλης Σ. Τσούτσουρας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικών Υπολογιστών Ε.Μ.Π.

Περίληψη

Αντικείμενο της παρούσας διπλωματικής εργασίας αποτελεί η μελέτη και η ανάπτυξη ενός διαχειριστή των πόρων, για ένα εξειδικευμένο είδος παράλληλων εφαρμογών που αποκαλούνται εύπλαστες (malleable), για Πολυ-Πύρηνα Συστήματα-σε-Ψηφίδα (Multi-Processor-System-on-Chip, MPSoC) που χρησιμοποιούν Αρχιτεκτονική Δικτύου-σε-Ψηφίδα (Network-on-Chip, NoC). Παρουσιάζεται ένα πλαίσιο (framework) κατανομής πόρων το οποίο κατασκευάζει μια αρχική χαρτογράφηση (mapping) για τις εφαρμογές αυτές και έπειτα τις διαχειρίζεται καθ' όλη την διάρκεια ζωής τους, παρέχοντας τους τρόπους για να μεγιστοποιήσουν την επιτάχυνση τους (speedup). Ο στόχος του αλγορίθμου κατανομής πόρων είναι να μεγιστοποιήσει την χρήση των πόρων, αποφεύγοντας κυριαρχικά (dominating) αποτελέσματα και λαμβάνοντας υπόψη τους τύπους των επεξεργαστών υποστηρίζοντας την ετερογένεια (heterogeneity) της πλατφόρμας. Όλα αυτά επιτυγχάνονται έχοντας συνολικά μικρή επιβάρυνση στην επικοινωνία μεταξύ των πυρήνων. Η εν λόγω δουλειά παρουσιάστηκε στο συνέδριο DAC 2013. (www.**dac**.com/**dac+2013**.aspx)

Στο κεφάλαιο 1, γίνεται μια γενική εισαγωγή καλύπτοντας τα βασικά χαρακτηριστικά ενός Δικτύου-σε-Ψηφίδα. Παρουσιάζεται η έννοια της χαρτογράφησης εφαρμογών και βασικοί ορισμοί για την πλατφόρμα και το μοντέλο εφαρμογών που χρησιμοποιήθηκαν.

Στο κεφάλαιο 2, παρουσιάζονται έξι δημοσιευμένες εργασίες οι οποίες περιλαμβάνουν θέματα πάνω σε σύγχρονους αλγορίθμους χαρτογράφησης κατά την ώρα εκτέλεσης ή αυξάνουν την διορατικότητα μας πάνω στις εύπλαστες εφαρμογές. Ερευνούμε τις βασικές ιδέες των εργασιών και δίνουμε έμφαση στις καινοτόμες συνεισφορές τους.

Στο κεφάλαιο 3, περιγράφεται το πλαίσιο κατανομής πόρων κατά την διάρκεια εκτέλεσης που αναπτύχθηκε στα πλαίσια αυτής της διπλωματικής εργασίας. Εστιάζουμε στα βασικά δομικά του στοιχεία και εξηγούμε πως λειτουργεί ο αλγόριθμος κατανομής πόρων. Επιπρόσθετα αναλύεται η αλληλεπίδραση και επικοινωνία των επί μέρους κομματιών του πλαισίου.

Στο κεφάλαιο 4, αναλύονται οι επιλογές σχεδιασμού που έγιναν για να κατασκευαστεί το εν λόγω πλαίσιο. Έπειτα, εξηγούμε πως αυτές οι επιλογές μοντελοποιήθηκαν σε κομμάτια ενός προσομοιωτή πλατφόρμας δικτύου-σε-ψηφίδα και εν τέλει σε υπηρεσία για πραγματική πλατφόρμα δικτύου-σε-ψηφίδα.

Στο κεφάλαιο 5, παρουσιάζουμε, αξιολογούμε και αναλύουμε τα αποτελέσματα των πειραμάτων που διεξήγαμε συγκρίνοντας το προτεινόμενο πλαίσιο κατανομής πόρων με ένα άλλο αντίστοιχο, τελευταίας τεχνολογίας που και τα δύο απευθύνεται σε εύπλαστες εφαρμογές.

Στο κεφάλαιο 6, συνοψίζουμε τα συμπεράσματα της διπλωματικής εργασίας και προτείνουμε τρόπους και ιδέες για περαιτέρω ανάπτυξη της δουλειάς που παρουσιάστηκε.

Λέξεις Κλειδιά

Σύστημα-σε-Ψηφίδα, Πολυ-Πύρηνο Δίκτυο-σε-Ψηφίδα, Χαρτογράφηση στον χρόνο εκτέλεσης, Εύπλαστες εφαρμογές

Abstract

The purpose of this diploma thesis is the design and implementation of a run-time resource manager, for a special kind of parallel applications called the malleable ones, for a Multi-Processor System-on-Chip (MPSoC) that utilizes the Network-on-Chip (NoC) architecture. A framework is presented that creates an initial mapping for these applications and then manages them throughout their entire life cycle providing ways for them to maximize their speedup. The goal of the resource management algorithm is to maximize resource utilization, avoiding dominating effects and taking into account the type of processors supporting platform heterogeneity. All these are achieved having a small overhead in overall inter-core communication. This work was presented in DAC 2013. (www.**dac**.com/**dac**+**2013**.aspx)

In chapter 1, an introduction is made, covering the basic characteristics of a Network-on-Chip. The concept of application mapping is presented and basic definitions about the NoC platform and the application model used.

In chapter 2, six published works are presented which either involve topics on state-of-the-art run-time mapping algorithms or they give an insight on parallel applications. We investigate the outline of these works giving emphasis on their novel contribution.

In chapter 3, the run-time resource management framework that was developed as part of this thesis is described. We highlight its major components and explain how the algorithm for allocating the resources functions. Additionally, we analyze the interplay and intercommunication of the components of the framework.

In chapter 4, we analyze the design choices that had to be made in order for our framework to be constructed. Then, we explain how these design choices were modeled into components of simulator of NoC platforms and ultimately as a run-time service for an actual NoC platform.

In chapter 5, we present, evaluate and analyze the results of the experiments we conducted comparing our proposed resource management framework to another state-of-the-art resource management framework, both for malleable applications.

Finally in chapter 6, we summarize the conclusions of the diploma thesis and introduce ways and ideas for further development of the work we presented.

Keywords

System-on-Chip, Multi-Processor System-on-Chip, Network-on-Chip, run-time mapping, malleable applications

Ευχαριστίες/Acknowledgements

Για την εκπόνηση της παρούσας διπλωματικής εργασίας θα ήθελα να εκφράσω τις ειλικρινείς μου ευχαριστίες προς τον επιβλέποντα καθηγητή κ. Δ. Σούντρη ο οποίος εμπιστεύθηκε στο πρόσωπο μου την ανάθεση ενός ιδιαίτερα ενδιαφέροντος και απαιτητικού επιστημονικού έργου.
Επιπλέον, αυτή η εργασία δεν θα είχε έλθει εις πέρας χωρίς την πολύτιμη βοήθεια και καθοδήγηση του υποψήφιου διδάκτορα Ηρακλή Αναγνωστόπουλου και του διδάκτορα Αλέξανδρου Μπάρτζα που βοήθησαν τα μέγιστα με τις συμβουλές, τις γνώσεις και την υπομονή τους.

Table of Contents

Chapter 1: Networks-on-Chip

1.1 Introduction

The current trend in VLSI design is to incorporate ever increasing number of working cores into the same IC. The level of integration is so great that allows manufacturers to integrate several Processing Element (PE) units of a great variety of nature, in one chip. The creation of such chips aims at satisfying the ever increasing need of the market from processing power. This is a direct result from the fact that more and more people use computational devices and the application running on them become constantly more complicated.

This approach seems to an effective counterweight to increasing the speed of the PE, but it suffers from the fact that the communication between these processors cannot be efficiently carried out by traditional communication buses without serious bottleneck issues, or point-to-point communication without serious space and energy waste. Performing a great number of computations is not a problem compared to the vast amount of data that need to be communicated for the computations to take place. To make matters worse, communication encounters fundamental physical limitations. On-chip wires do not scale in the same manner as transistors do and the cost gap between computation and communication is getting bigger. The solution to this problem lies in the Network-On-Chip (NoC) architecture.

1.2 Network-on-Chip

Since a bus can no longer satisfy the need of communication between processing elements designers have to come up with an idea to tackle this insufficiency. In large scale computing, for years now people have been connecting machines via a network. Additionally, in recent years designers have created systems with clusters of processing elements communicating through networks. Thus, it came as a reasonable consequence to employ the same principles for on-chip communication.

Before introducing the principles of the NoC design, it is interesting to present the advantages and disadvantages of bus compares to a network-on-chip. They are summed up in table 1.1:

Bus Pros & Cons		Network Pros & Cons	
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.	-	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep submicron process.	-	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	-	+	Routing decisions are distributed, if the network protocol is made non-central.

The bus arbiter is instance-specific.	-	+	The same router may be re-instantiated, for all network sizes.
Bus testability is problematic and slow.	-	+	Locally placed dedicated BIST is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	-	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	-	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	-	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
The concepts are simple and well understood.	+	-	System designers need reeducation for new concepts.

Table 1.1: Pros & Cons of Bus and Network for on-chip communication [1]

In general, the network-on-chip can accommodate the needs of a big number of elements trying to communicate. The bus can be a serious bottleneck for the same number of elements. Only its arbitration and testability is a serious issue and the problem worsens as the number of masters of the bus increases. Such problems are not present on a network. Additionally, a bus may be unable to transport a very big amount of data since it is essentially the only route of data exchange. On the other hand, a network with efficient routing algorithm can deal with the big amount of data by distributing the communication between different routes, thus lowering the traffic and avoiding the congestion of data.

The major drawback of the NoC is that it is in fact a new technology as far as VLSI and system design is concerned. As a result, the existing systems and especially those built around a CPU have to be altered to fit to the new model of communication. There are different needs in terms of synchronization and memory organization. Even the designers themselves may encounter the fact that they will have to adapt they design strategies in a different way compared to what they are used so far. However, the excessive experience on large scale network design will help us avoid pitfalls and reduce the time until NoC technology is mature enough to replace the existing bus-oriented VLSI design.

In its essence, a NoC is a new approach to the System-On-Chip (SoC) model and specifically Multi-Processor-System-On-Chip (MPSoC), which uses elements of Computer Networks for on-chip communication. A NoC consists of several Intellectual Property (IP) blocks, but instead of classical bus-based or point-to-point communications, a more general scheme is adapted, employing a grid of routing nodes spread across the chip.
On comparison to Computer Networks, the NoC consists of the following components:

- *Cores* are Intellectual Property (IP) blocks, usually processors of any kind, containing some local memory. Can also be referred as *tiles* of the NoC.

- *Network Adapters* implement the interface by which the *cores* connect to the NoC.
- *Routing Nodes* are components similar to the routers in Computer Networks. They are in charge of applying the chosen routing protocols.
- *Links* connect the routing nodes, thus providing communication between them, via one or more physical or logical channels.

The network in which the cores are connected consists of The *Routing Nodes* and the *Links* of the NoC. An example of a 4x4 mesh topology NoC is shown in fig. 1.1.

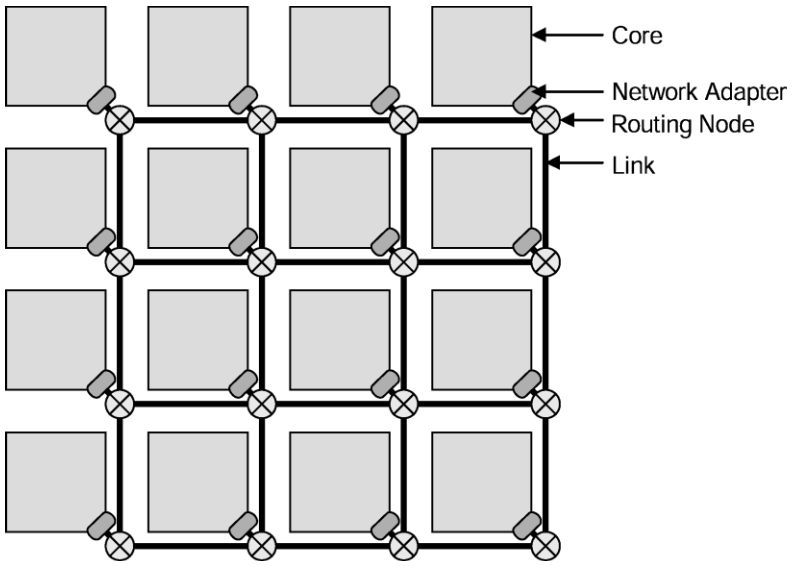


Figure 1.1: Example of a 4x4 NoC in mesh topology. [1]

The new communication model is based on messages. Figure 1.2 presents the design principle for what will happen when a source core wants to send a message to a destination core. This message goes through the network adapter of the source core, which decides the destination, as the core itself isn't aware of the network. Then, the communicated data is forwarded to the core's routing node which, according to the destination, routes it towards any intermediate routing node, which does the same thing. Once the destination node is reached, the data goes in the opposite direction, from the router to the network interface and finally to the destination core. In the figure it is also apparent that the design of the NoC system is directly influenced from the OSI standard that dominates large-scale computer network design. An interesting part is that since on chip communication can be perceived as a subset of inter-node communication in large systems, some layers of the OSI model are merged in one on the NoC.

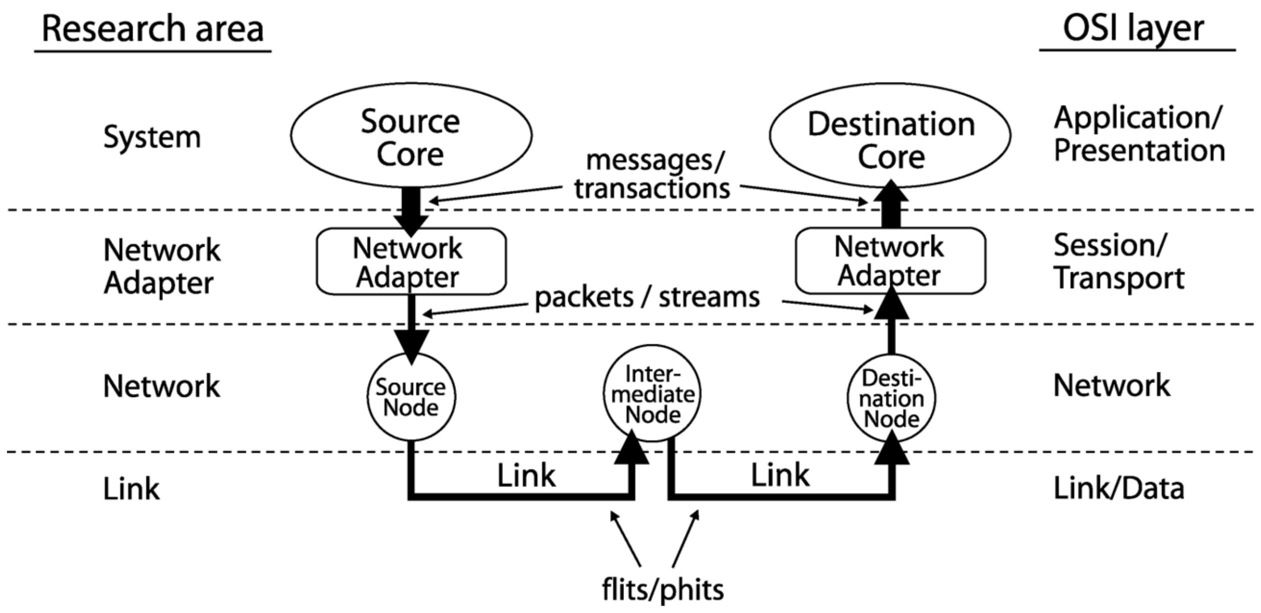


Figure 1.2: Communication between two cores. [1]

1.2.1. Homogeneity and Granularity

As stated before, a core of the NoC can be of various types. This fact gives the designer a great exploration space and a NoC is characterized by its *homogeneity* and *granularity*. In a *homogeneous* one all the cores belong on the same PE *type* while on a *heterogeneous* one more than one types of PEs exist on the same chip. For example, a homogeneous NoC can consist of processor tiles with local memory, and a heterogeneous one can include any of the following: processor-memory tiles, pure processor tiles, digital signal processors (DSP), memory tiles or even reconfigurable tiles like FPGAs. The following figure depicts such a case.

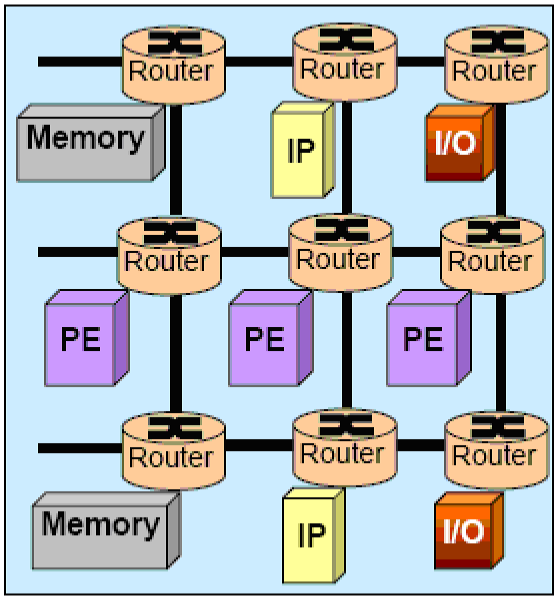


Figure 1.3: NoC heterogeneous platform example.

The second design choice, granularity, distinguishes NoC between *coarse* or *fine grained*, depending on the number of cores per surface. These options give NoCs increased flexibility and higher degree of variety over Computer Networks, which are mostly homogeneous and coarse grained. Examples of all the design choices NoCs are presented in fig. 1.4.

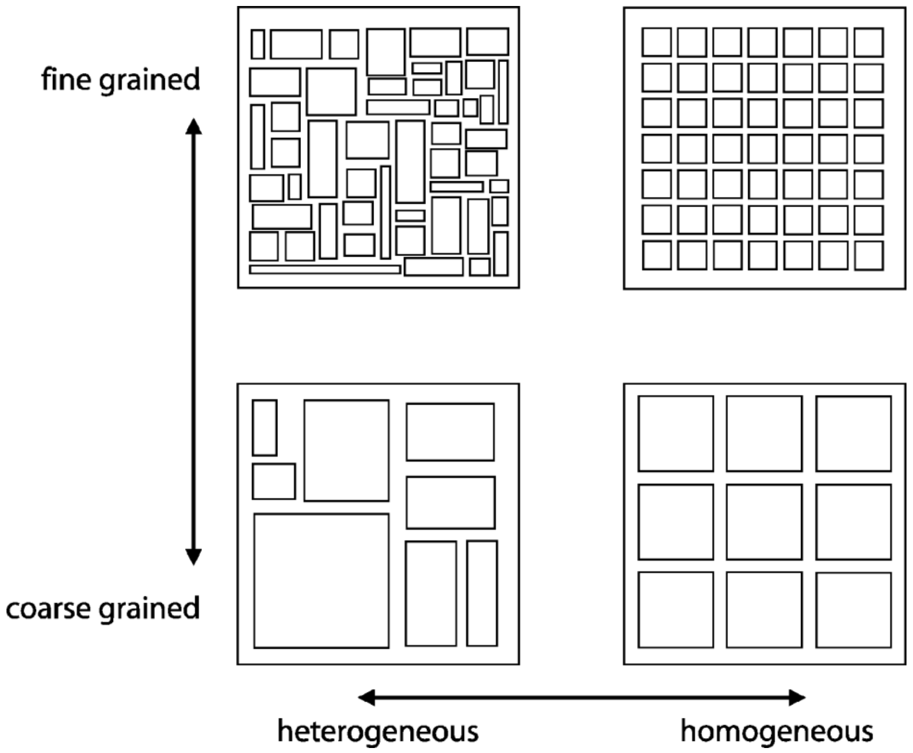


Figure 1.4: Effects of different degrees of homogeneity and granularity of system components. [1]

1.3 Intel SCC platform

As stated, NoC platforms are a VLSI design trend and not a theoretical idea for future use. As a result, many organizations around the world are making effort to perfect this technology and produce it in industrial scale. One of these programs was carried by INTEL Corporation and was named TeraScale Research program. Its first outcome was an 80-core Terascale processor. The 80 core chip featured a tiny, non-IA instruction set and had no compiler, no external memory, no I/O, and no operating system. Consequently, programming this chip was a difficult process that only a few programmers accomplished. [2]

Our NoC platform of interest, the second outcome of TeraScale Research program, is the 48-core SCC processor. It features a well-known x86 processing element employed in each core. This is a significant feature, as explained in [2], since the Linux operating system and C, C++, FORTRAN compilers can be run on this platform, providing a run-time and programming environment which can be used by most programmers and giving the opportunity for other well-known and useful programs to be imported into the platform.

If we take a closer look on the platform we can see the following key elements, as described in [2] and illustrated in figure 1.5 [3]:

- Two blocks, each with a P54C core (second generation Intel Pentium processor), 16 KB instruction and data L1 caches plus a unified 256 KB L2 cache.
- A Mesh Interface Unit (MIU) with circuitry to allow the mesh and the interface to run at different frequencies.
- 16 KB Message Passing Buffer.
- Two test-and-set registers.

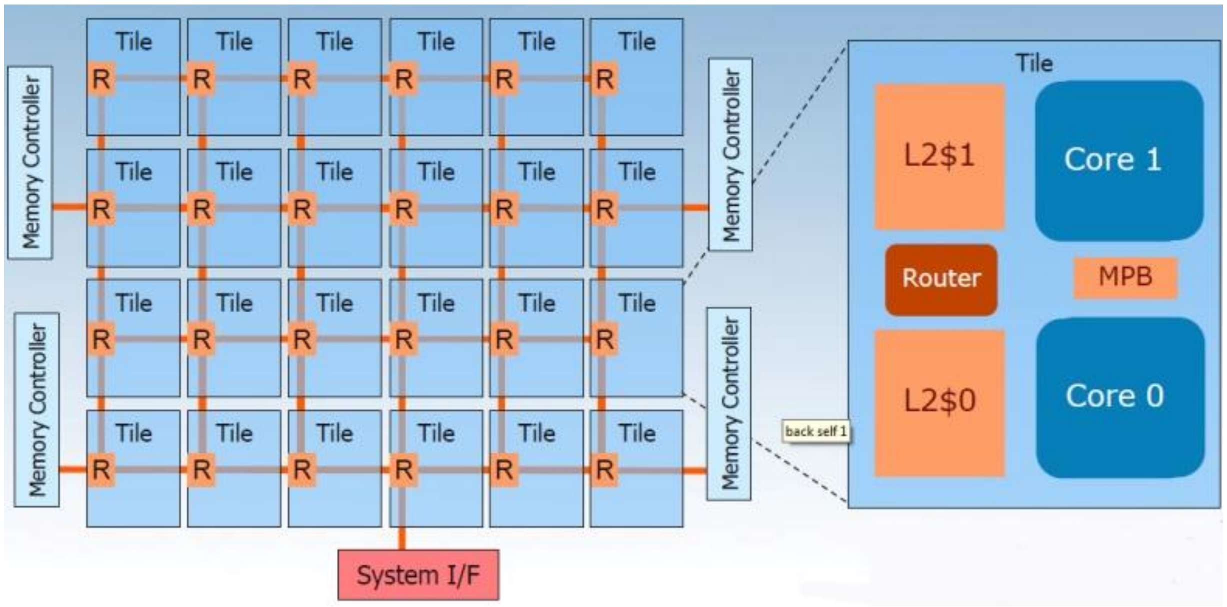
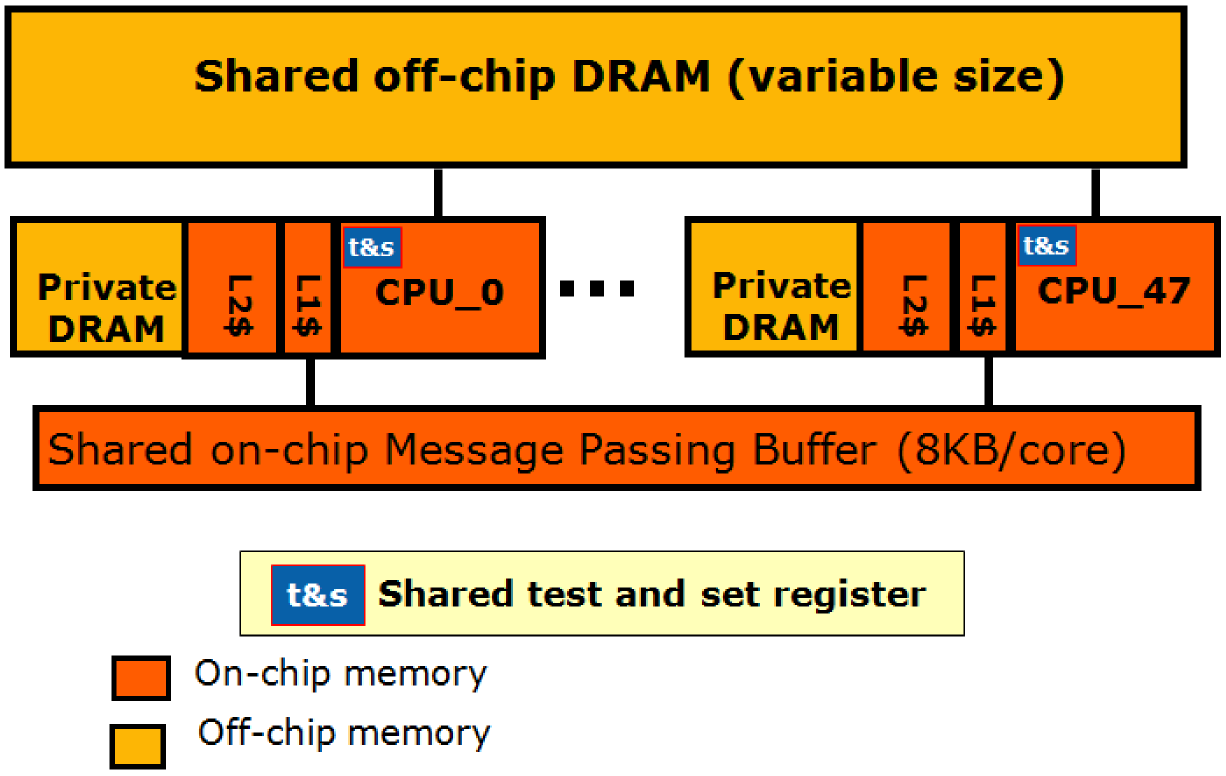


Figure 1.5: Overview of the Inter SCC Platform. [3]

Each tile connects to a router. This router works with the Mesh Interface Unit (MIU) to integrate the tiles into a mesh. The MIU packetizes data onto the mesh and de-packetizes data from the mesh using a round-robin scheme to arbitrate between the two cores on the tile. There is also a great number of off-chip memory which ranges from 16 to 64 GB of DDR3 RAM, controller by four memory controllers. Finally, router is connected to an off-package FPGA to translate the mesh protocol into the PCI express protocol, allowing the chip to interact with a PC serving as a management console. The entire memory architecture of the platform is illustrated in figure 1.6.

The most obvious architectural innovation with respect to memory is the message passing buffer (MPB) included on each tile. It provides a fast, on-die shared SRAM, as opposed to the bulk memory accessed through four DDR3 channels. While the processor does not offer any hardware-managed memory coherence, it features a new memory type to enable efficient communication between the cores. This new memory type is called the Message Passing Buffer Type (MPBT).


Shared off-chip DRAM (variable size)
Private DRAM
L2$
L1$
t&s
CPU_0
...
Private DRAM
L2$
L1$
t&s
CPU_47
Shared on-chip Message Passing Buffer (8KB/core)
t&s Shared test and set register
On-chip memory
Off-chip memory

Fighre 1.6: Memory architecture of the SCC processor. [2]

This message passing buffer is 16KB in each tile. Its memory is shared among all the cores on the chip. With 24 tiles, the SCC provides 384KB of message passing buffer. When a message-passing program sends a message from one core to another, internally it is passing the data through the message passing buffers on the chip. The resulting address request gets placed in one of the following three queues:

- A queue to send the request to the router, then to the memory controller, and finally out to DDR3 memory.
- A queue to send the request to the core's MPB.
- A queue to access local configuration registers, which also reside in the MIU.

Data coherence and synchronization between cores is the programmer's responsibility to code appropriately. In order to achieve mutually exclusive memory operations, a test&set register is built in every core. The operation on this register is atomic, providing a solution to the synchronization problem. The SCC processor does not provide a flush operation to help the programmer maintain consistency between views of data in the cache and in shared memory. Data typed as MPBT bypass L2 and go directly to L1. When loading MPBT data it is necessary to invalidate the corresponding lines in L1 in order to prevent reading stale information. In case of writes it must be ensured that the data reaches its final destination before e.g. releasing a lock [2].

In order to make programming easier and to increase the portability and scalability of programs written for the SCC platform, Intel provides a communication environment known as the RCCE. As a matter of fact, SCC and RCCE were developed side by side [3]. RCCE distributes evenly the MPB address space to the 48-cores, designating that each core will have 8KB of memory in this buffer for itself. It provides two basic interfaces for inter-node communication. The first is the “gory” one which is a more low level design and offers the programmer greater control over the SCC in expense of need for explicit synchronization techniques to be employed by the programmer. The second interface is the “basic” one which employs send – recv functions (much like the ones in MPI) and does not bother the programmer with synchronization issues.

In our work, we focused on the gory interface in order to have as much control on the platform as possible and to be able to asynchronously send and receive data between various cores. The first step of every memory operation is to allocate a memory space in MPB of MPBT data type. This is done with the RCCE_malloc function. This is presented in the following figure:

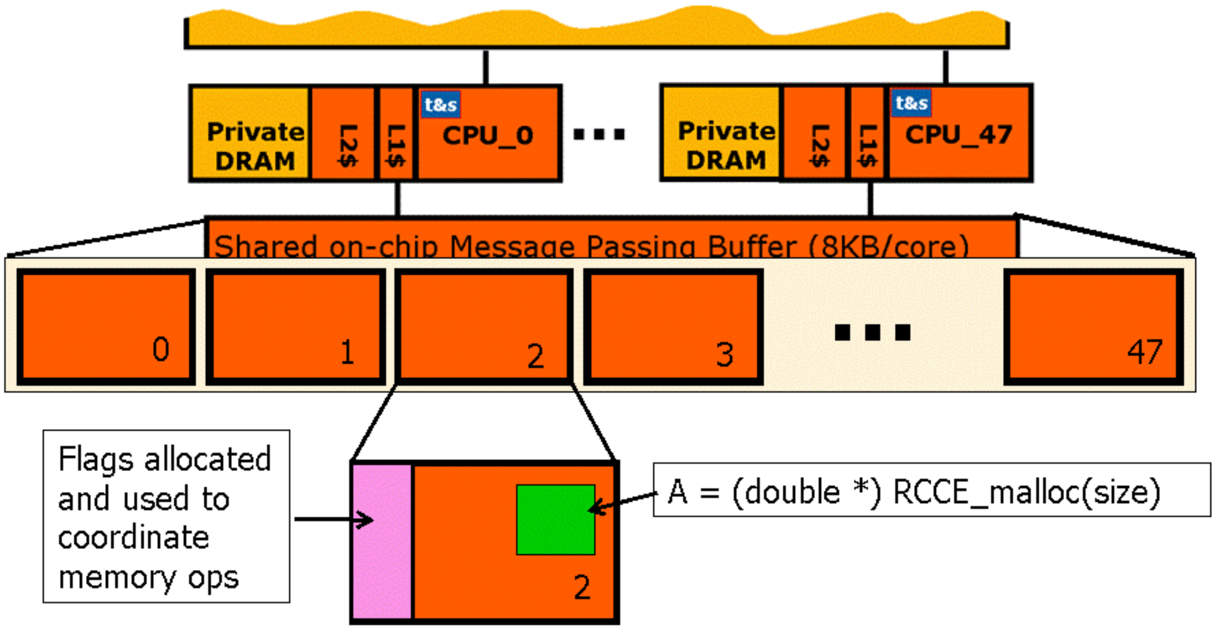

Private DRAM
L2$
L1$
t&s
CPU_0
Private DRAM
L2$
L1$
t&s
CPU_47
Shared on-chip Message Passing Buffer (8KB/core)
0
1
2
3
47
Flags allocated and used to coordinate memory ops
2
A = (double *) RCCE_malloc(size)

Figure 1.7: Symmetric name space model for the MPD as designed for RCCE library. [2]

After the memory space has been allocated nodes can exchange messages trough the functions:

```
RCCE_put(A, buffer, size, ID);
RCCE_get(buffer, A, size, ID);
```

In both functions, A is the space allocated in MPB while buffer is a memory buffer in the local address space of the node. In put function data are transferred from buffer to MPB while in the get function the opposite takes place. In both functions size denotes the number of bytes to be transferred while ID is the number of the target node.

For synchronization purposes the RCCE API provides Boolean flags. Using these flags the programmer can code explicit synchronization techniques. A flag is allocates in the MPB of a node using the RCCE_flag_alloc function. Its modification is managed through the RCCE_flag_write function. The only available values for a flag are RCCE_FLAG_SET and RCCE_FLAG_UNSET representing the Boolean true and false values. The modification of a flag of a node can be performed by any other node but this access is atomic using the test&set register. Finally, an RCCE_wait_until function is provided which forces a node to stall waiting for a specific value of a specific flag.

Despite the fact that the RCCE API provides an abstraction to design programs in high level still it is not deprived of low-level limitations enforced by the architecture of the platform. Hence, details that are typically hidden in mainstream message passing environments such as MPI are fully exposed in RCCE. For example, an array in the MPB must be cache aligned, occupy full L1 cache lines (i.e.multiples of 32 bytes) and fit within the available space in the MPB. Reading or writing in the MPB also only happens in multiples of the cache line. Even flags, are part of the memory so their change of values is bound by the limitations of any other memory operation. This creates the need for even further synchronization techniques as will be presented in chapter 4.

In both provided, application is launched using a program provided by Intel called "rccerun" [4]. The user specifies the number of cores to use from a given subset of cores on the chip. Identical executables are launched on all cores. Every executable is assigned a rank, which is a sequence number ranging from 0 to N-1 (N is the number of participating cores). This rank cannot be changed during the execution of the application and it uniquely identifies both the application and the core it runs on. Finally, all cores can access a disk space in the Management Console which proved to be very helpful in the implementation of the proposed framework on the SCC platform.

1.4 The mapping problem

The concept of *NoC MPSoC* poses a new problem for the designer in a higher level of abstraction, the one of allocation of resources amongst the cores. Up until now we faced the problem of scheduling the tasks in one core or a very small set of cores. In a NoC, resource allocation has a spatial aspect as well. An application comprises of tasks being executed in parallel and every task has to be assigned to a core. The choice of core has to take into account its position as well. This is the reason why we refer to this process as mapping. An idea of what mapping refers to can be seen in figure 1.8.

Calculating a cost efficient *mapping* for a given application in a short amount of time is crucial. The design may have many aspects. Some common goals are to maximize performance, minimize distance between tasks or minimize energy consumption in case of heterogeneous platform. Since the mapping process is an intermediate step between the request for an application to start and the actual initialization of the application, the process must be as fast as possible. As always, there is a variety of design decisions to be made as far as the characteristics of the mapping are concerned. The most important are presented in the following sections.

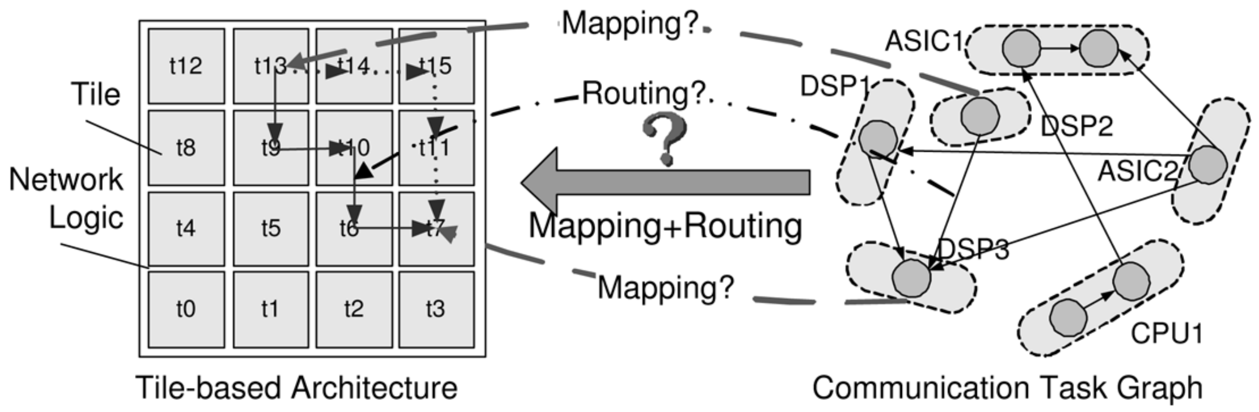


Figure 1.8: Illustration of the mapping/routing problem [5]

1.4.1 Design-time vs Run-time mapping

In the case of Design-time mapping, the nodes an application can run on are designated before run-time. This designation cannot be altered during run-time. Apparently, this technique can be employed only when small amount of application can be run on a NoC and only for specific scenarios. Apparently this is not the case in modern systems.

The alternative of design-time mapping is the run-time one. In this case, the resources are allocated dynamically as application enter and exit the system, providing the necessary flexibility for the platform to function properly. An interesting fact is that run-time mapping is not only the only way to tackle the unpredictability of the incoming applications but has a number of other abilities as well:

- To adapt to the available resources. Those vary over time, due to applications running simultaneously. Run-time information can be incorporated to further reduce the cost of running an application.
- To enable unforeseeable upgrades after first product release time, e.g. new application and new or changing standards.
- To avoid defective parts of a SoC. Larger chips mean lower yield. The yield can be improved when the mapping algorithm is able to avoid faulty parts of the chip. Also aging can lead to faulty parts that are unforeseeable at design-time.
- To be used with reconfigurable hardware where the type of available processing elements can vary over time.

The only downside of a run-time mapping is the extra time it adds to the execution of an application, since it is executed between the request from the OS and the actual execution of the tasks. Hence, it is crucial that the mapping is calculated fast, so that it is transparent and doesn’t burden the system.

1.4.2 Centralized vs Distributed mapping

Apart from defining the moment (run-time or design-time) the mapping occurs one must also define on which tiles the mapping algorithm will be executed. In this case the available design choices are to perform the mapping calculations either on a distributed or centralized manner. [5].

Centralized mapping utilizes one or a small set of cores, *Centralized Managers* (CM), to perform the mapping for every application that arrives. These cores are burdened with the responsibility to calculate the mapping for the whole system. This mapping scheme may cause the following problems [6]:

- Monitoring traffic is increased in volume. During the mapping, since it is performed at run-time, the Centralized Manager needs to collect data from the whole chip, which causes traffic on the wires, possibly stalling the execution of already running tasks.
- High computational cost to calculate the mapping for the whole chip at once.
- Single point of failure. If the Centralized Manager fails for some reason, the mapping can't be performed at all.
- The Centralized Manager becomes a point of *hot-spot* as every tile sends the status of the PE to it. This increases the chance of bottleneck issues around the manager.
- Scalability issues. As NoCs will grow in size, and more Processing Elements will be added, the computational effort of mapping and the traffic it will create will increase exponentially, thus rendering the computation very expensive and the scheme ineffective.

On the distributed mapping scheme, the effort of the computation is distributed, as the name suggests, on several tiles across the chip, *Local Managers* (*LM*) as presented in [3], and they may even change from one mapping to the next. This way, the problems of the Centralized mapping are solved as following:

- Monitoring traffic is decreased in volume. The Processing Elements only need to send the data to their closest Local Manager, and this way they travel less on the chip. When mapping decisions have to be made involving different regions, the communication is limited only between Local Managers.
- The Local Managers only need to perform the mapping computation for the area of the chip they are responsible for, or for some designated tiles. This way the computation demanding problem is divided in less demanding ones.
- There are no issues of single point of failure or hot-spots, since the smaller portions of the computation can be performed on any tile.
- It scales very well with larger NoCs, since all that is needed is some more light-weight Local Managers, whose individual computation effort isn't increased.

The overall differences of the two design choices are illustrated in figure 1.9. Two major downsides occur. First, there is the need for synchronization between the managers, so as to achieve a well-balanced and fair mapping. Secondly, it is reasonable that a centralized approach can provide a better mapping considering the fact that it has an overall view of what is going on in the platform. However, we can achieve results comparable to the optimum mapping with the distributed approach and taking into account its aforementioned advantages, distributed resource management seems to be the only option [7]. The following figure illustrates the principles of centralized and distributed mapping.

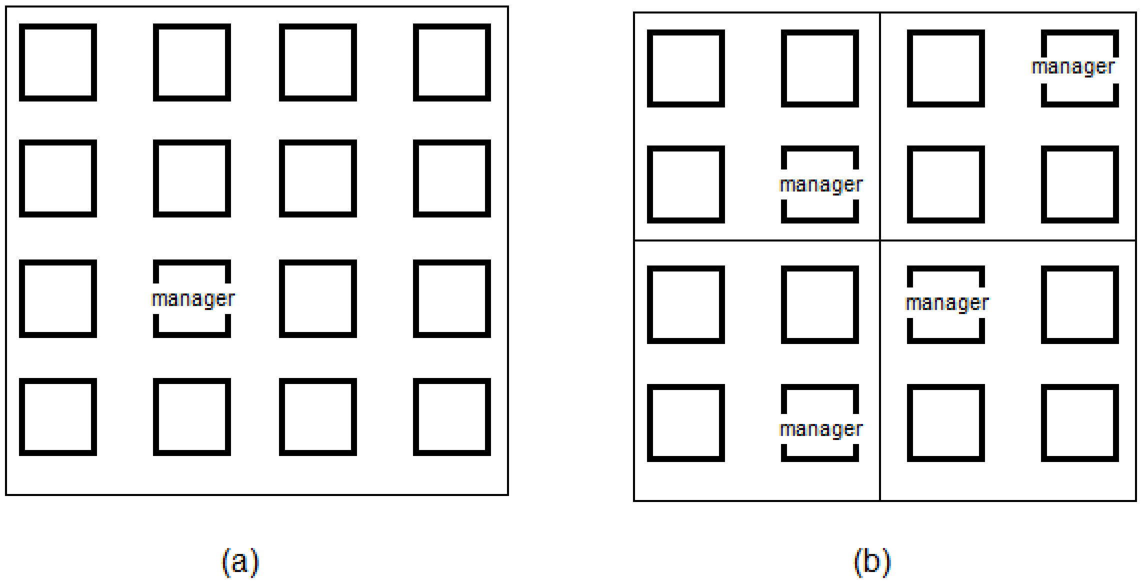


Figure 1.9: Examples of centralized and distributed mapping.

1.5 Parallel applications classification

According to the parallel job classification scheme presented in [9] we can separate parallel applications into three categories based on their capabilities:

- Moldable applications: Parallel applications that can be stopped at any point but the number of processors for such jobs cannot be changed during run-time.
- Malleable applications: These are parallel applications that can be stopped at any point of execution and have flexibility to change the number of assigned processors during run-time. These applications are also called reconfigurable applications.
- Migratable applications: These are parallel applications that can be stopped at any point of execution and can be restarted on processors in a different site, cluster or domain.

Our applications of interest are the malleable ones, and the application model used in this work is the common malleable application model described in [9, 10] and used by other distributed approaches [11]. In this model, each application is described by four parameters W, var, A where W is the workload, var is the parallelism variance, A is the average parallelism. We extend this model with the parameter Q, a set of Processing Elements (PEs) types that the application is supposed to be executed on. The aforementioned model [9] provides a mathematical function for calculating the speedup of an application taking into account its parameters and the number of processing elements assigned to it. The function providing the speedup is:

$$var < 1 \ (low\ variance)$$

$$S(n) = \begin{cases} \dfrac{n \cdot A}{A + \dfrac{var}{2(n-1)}}, & 1 \leq n \leq A \\ \dfrac{n \cdot A}{var \cdot \left(A - \frac{1}{2}\right) + n \cdot (1 - \frac{var}{2})}, & A \leq n \ \leq 2 \ \cdot A - 1 \\ A, & n \geq 2 \cdot A - 1 \end{cases}$$

$$var \ \geq 1 \ (high\ variance)$$

$$S(n) = \begin{cases} \dfrac{n \cdot A \cdot (var + 1)}{var \cdot (n + A - 1) + \ A}, & 1 \leq n \leq A + A \cdot var - var \\ A, & n \geq A + A \cdot var - var \end{cases}$$

The values A and σ for real applications can be obtained by evaluating benchmarks of the intended application running on the target platform [11]. A secondary very interesting characteristic of this application model is the fact the providing that we are aware of the remaining workload of an application, its remaining time is calculated as

$$T_{finish} = \frac{W_{remaining}}{S(n)}.$$

1.6 Objectives and Contributions

We consider that both networks on chip and malleable applications are a key element for future high performance computing both in embedded and general purpose systems. However, as it will be clearer in chapter 2 where related work on field in presented, few work has been performed in the field of resource management for malleable applications on NoC platforms and this work has not been tested on actual NoC platfoms. As a result our goals and contributions are:

i. To propose a resource management framework for malleable applications on NoC platforms with characteristics like efficient mapping, workload awareness of applications and small overhead in inter-node communication.

ii. To test the proposed framework on a NoC simulator. Apart from verification of the design, the simulator itself is an important tool for future development of other applications aimed at NoC platforms.

iii. To utilize an actual NoC platform as a part of our framework evaluation. Besides that, our design methodologies for the NoC platform are individually of interest for other developers since they require special techniques which they can find helpful in the future.

Chapter 2: State-of-the-art Run-time mapping algorithms

2.1 Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles [12]

In this paper, the authors develop a centralized Run-Time manager for heterogeneous NoCs containing fine grained *Reconfigurable Hardware Tiles*. The term fine-grained refers to Reconfigurable hardware which is a type of Processing Elements, exhibiting its own distinct set of properties compared to traditional PEs. Perhaps the most common example of such hardware is an FPGA. Reconfigurable hardware can be reconfigured at run-time, according to the needs of the application, adding more flexibility to the NoC. The term fine-grained refers to the fact that each tile of the grid can change without affecting the nature of the other ones. These tiles are suited for computational intensive tasks, but can only accommodate a single task.

The authors propose a resource management heuristic which consists of a basic algorithm completed with reconfigurable add-ons. The heuristic combines idea from multiple resource management algorithms in a way that can serve the needs of the specific hardware platform. The algorithm is executed on the central Operating System running on a designated tile, called the *Master PE*. The Master PE tile is responsible for assigning resources for both computation and communication to the different tasks (given as input in the form of an Application Task Graph that holds information about for both the properties of the tasks and the inter-task communication). The mapping heuristic will have to find the *best fitting* task for every reconfigurable hardware tile The Operating System maintains a list of PE descriptors, keeping track of the computation resources of each tile, while the communication resources are maintained by means of an injection slot table that indicates when a task is allowed to inject messages onto a link of the NoC. In addition, every tile contains a Destination Lookup Table (DLT), used to resolve the location of its communication destinations.

The Resource Management Heuristic follows the steps given below:

1. Calculate requested resource load. The heuristic calculates the real computation and communication task load.
2. Calculate task execution variance. For every task of the application to be mapped, the execution time variance on the different supported PE types is calculated.
3. Calculate task communication weight. This allows the algorithm to order the tasks of an application based on their communication needs.
4. Sort tasks according to mapping importance. The tasks are sorted in descending priority.
5. Sort PEs for most important unmapped task. This step comes in two phases:
   i. Determine low communication – high performance tasks and their counterparts
   ii. Place together high communication tasks in order to map them close together.
6. Mapping the task to the best computing resource. The most important unmapped task is assigned the best fitting PE. Steps 4 and 6 are repeated until all tasks are mapped.

The authors claim that there are occasions where the aforementioned steps fail to provide a suitable mapping for a certain task. The most common occurrence of this fact is when a task that has great needs in recourses arrives at an instant when the platform is crowded with other applications. An initial step proposed to solve this problem, is to employ the technique of backtracking. One or more previous task assignments are changes in order to solve the mapping problem of the current task. In case backtracking fails there are three options:

- Use run-time task migration
- Use hierarchical configuration
- Restart the heuristic with reduced user requirements

As mentioned earlier, the heuristic was enriched with some add-ons to facilitate reconfigurable hardware platforms. The first addition is taking into account some special properties of this kind of hardware. First internal fragmentation is considered and an effort to minimize it is employed even if this contradicts with the PE priority of a given task. Secondly, the binary state (i.e. 0% or 100% load) and the computational performance of RH tiles are considered. In case a regular PE can accommodate the needs of a task, even if it is not the optimum choice, it is preferred than wasting a reconfigurable tile.

Hierarchical configuration of the tiles involves the use of softcore PEs instantiated on Reconfigurable Hardware tiles. This technique can improve the mapping performance when a task's binary isn't supported for execution on any of the NoC's other PE types or when it is more efficient communication-wise to map a task on a nearby Reconfigurable Hardware tile, rather than a further away PE tile.

As far as task migration is concerned, the authors propose two run-time task migration schemes. The process of task migration refers to relocation a task under execution from one tile to another one. Of course there are a lot of limitations when employing such a technique. For example, in order to overcome the architectural differences between different tiles of the platform, tasks can only migrate at pre-defined execution points in the code, called migration points. Additionally, during the migration process, task communication consistency has to be assured. The two proposed task migration algorithms cover both these aspects. The first mechanism, called general mechanism is depicted in figure 2.1.

The second mechanism is called pipeline mechanism. This stems from the fact that many algorithms are pipelined and at certain points, new and independent information are put into the pipeline. This fact is crucial for allowing the mechanism to move multiple pipelined tasks at once without being concerned about transferring task state. An example can be seen in figure 2.2.

The OS instructs the pipeline input task to continue feeding data into the pipeline until a stateless point is reached. Then the pipeline input task flushes the pipeline. Compared to the general mechanism, there are no unprocessed or buffered messages in the path between the pipeline input and pipeline output.

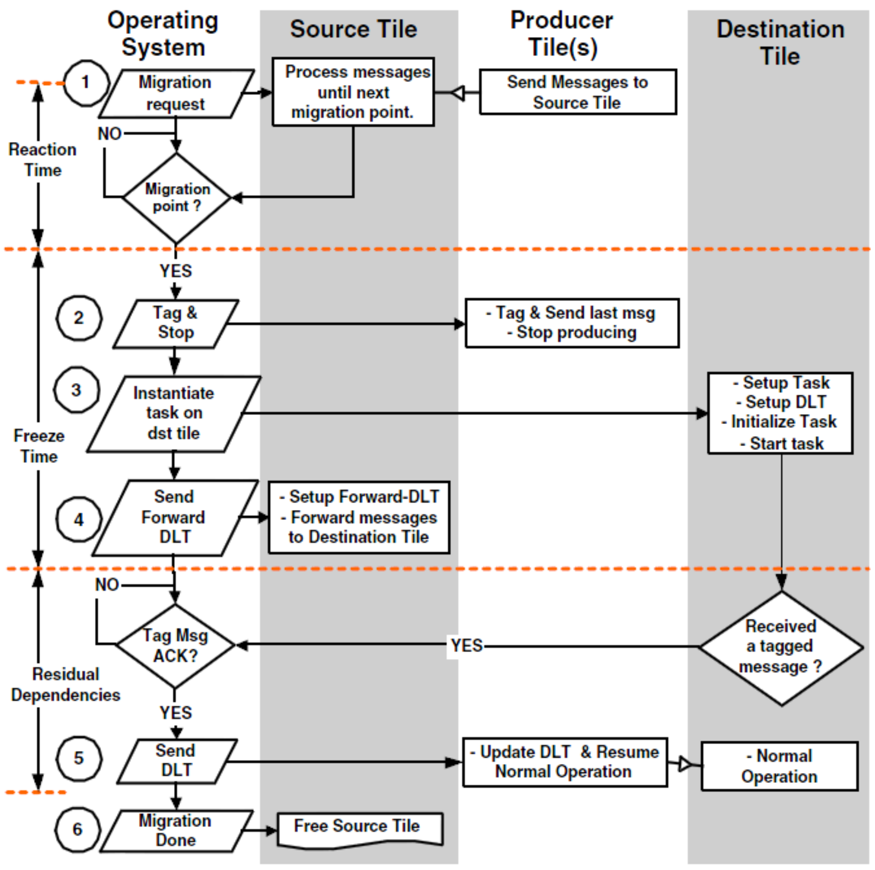


Figure 2.1: General Task Migration mechanism. [12]

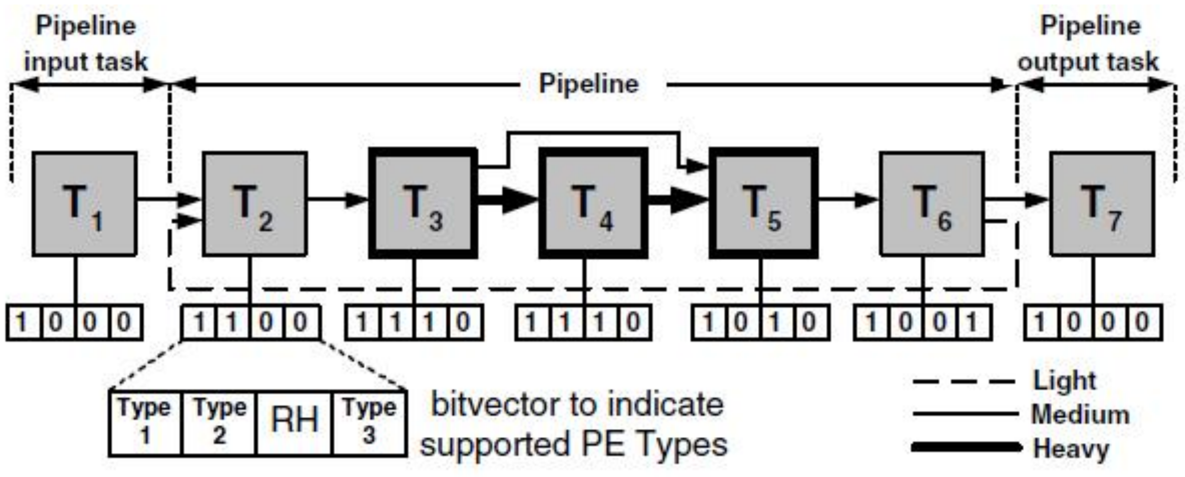


Figure 2.1: General pipeline mechanism. [12]

2.2 ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication [6]

In this work, the authors present an agent-based distributed resource management algorithm for self-adaptive heterogeneous MPSoCs. The negotiation of cores takes place inside a *virtual cluster*. Such a cluster is an area of the platform that can be creates, resized and destroyed at run-time. Additionally, an agent is a computational entity acting on behalf of others. Its main characteristics are:

i. It is a small task close to the system.
ii. It performs resource management.
iii. It must be executable on any PE.
iv. It must be migratable and recoverable.
v. It may be destroyed upon the destruction of the cluster.

A cluster agent is CA, is an agent responsible for mapping operations inside a cluster. A global agent GA is responsible for storing global information such as usage of communication and computation resources for each cluster and this information is used for selection and re-organization of the clusters. An overview of the mapping scheme is presented in figure 2.3.

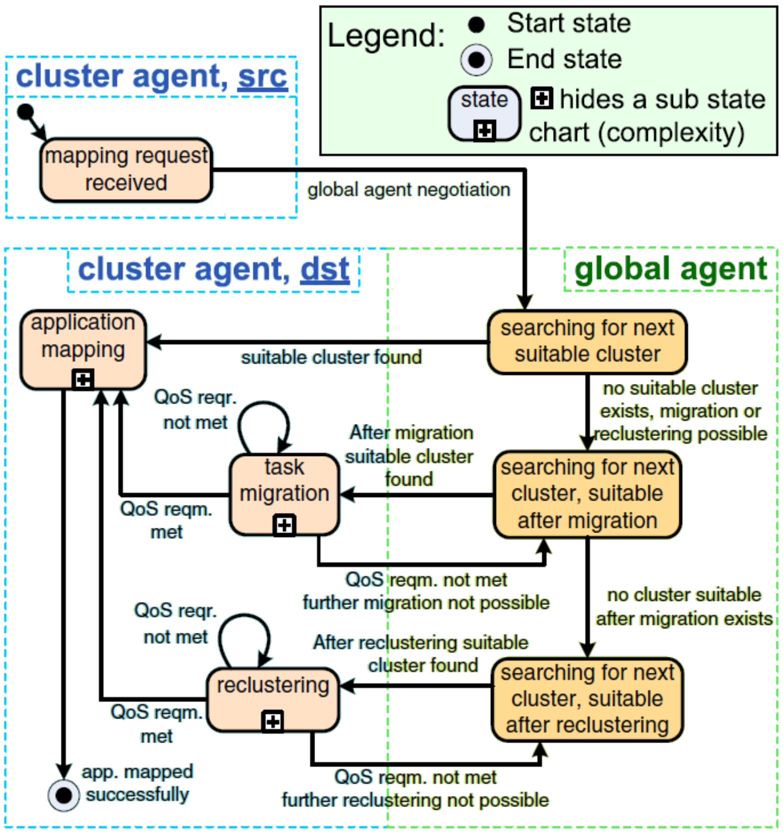

Legend:
Start state
End state
state
hides a sub state chart (complexity)
cluster agent, src
mapping request received
global agent negotiation
cluster agent, dst
global agent
application mapping
suitable cluster found
searching for next suitable cluster
no suitable cluster exists, migration or reclustering possible
QoS reqr. not met
task migration
After migration suitable cluster found
searching for next cluster, suitable after migration
QoS reqm. met
QoS reqm. not met further migration not possible
no cluster suitable after migration exists
QoS reqr. not met
After reclustering suitable cluster found
reclustering
searching for next cluster, suitable after reclustering
QoS reqm. met
app. mapped successfully
QoS reqm. not met further reclustering not possible

Fig. 2.3: Flow of the ADAM algorithm. [6]

When a new mapping request is received from any tile, the Cluster Agent of the tile's cluster communicates with the Global Agent, indicating the request. At this point, the Global

Agent performs the *Suitable Cluster Negotiation Algorithm*, which finds a cluster capable to fit the whole application. The Suitable Cluster Negotiation Algorithm checks if there are enough free tiles in every PE type and resource requirement class for all the tasks in the application.

In case the algorithm fails to find a suitable cluster then a task migration is employed is the most suitable cluster. The point of this task migration is to rearrange the tasks within a cluster in a way that the mapping of the new application will be feasible in the cluster. Still there is a probability that this will fail to produce an appropriate mapping as well. Then the technique of re-clustering is employed. Clusters can be changed in shape, resulting to the probable creation and destruction of some them. This process can benefit running applications since resources can be redistributed in a better way. However, it is an elaborate process that takes into account many parameters of the applications such as the requested energy and memory of the application. The flow of the reclustering algorith is presented in figure 2.4.

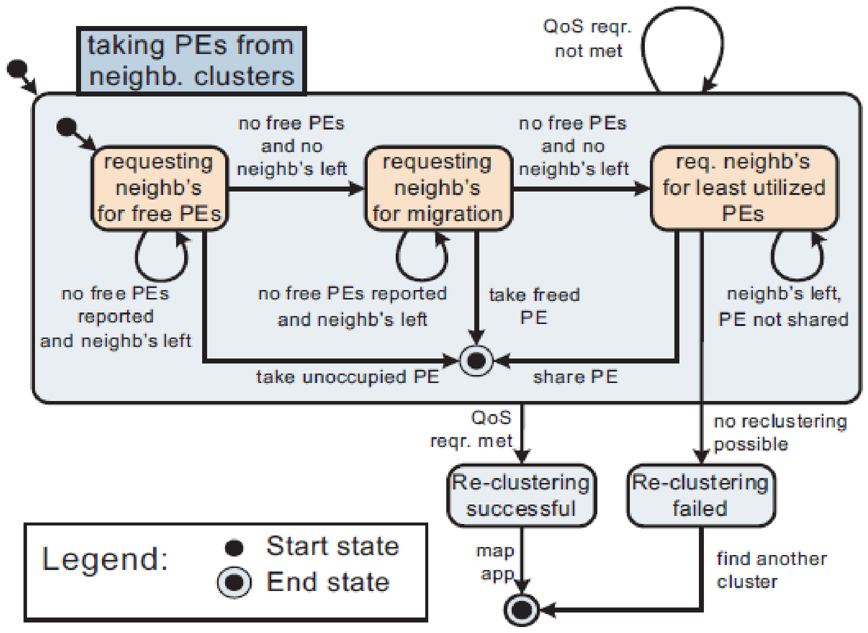

taking PEs from neighb. clusters
QoS reqr. not met
no free PEs and no neighb's left
no free PEs and no neighb's left
requesting neighb's for free PEs
requesting neighb's for migration
req. neighb's for least utilized PEs
no free PEs reported and neighb's left
no free PEs reported and neighb's left
take freed PE
neighb's left, PE not shared
take unoccupied PE
share PE
QoS reqr. met
no reclustering possible
Re-clustering successful
Re-clustering failed
map app
find another cluster
Legend:
Start state
End state

Figure 2.4: The re-clustering process of the ADAM algorithm. [6]

After a cluster that can host the application has been found, that cluster's agent is responsible to perform the *Run-time Mapping algorithm* to appoint every task to a tile to be executed on. The best tile for every task is calculated using a heuristics, checking the tile's position in the cluster (tiles near the center are preferred), the volume of communication on the tile before and after the mapping and the resource requirements for the task to run on any tile.

2.3 A Divide and Conquer based Distributed Run-time Mapping Methodology for Many-Core platforms [7]

In this paper a distributed run-time resource management framework is presented for both homogeneous and heterogeneous platforms. The main idea of the proposed framework is illustrated in figure 2.5.

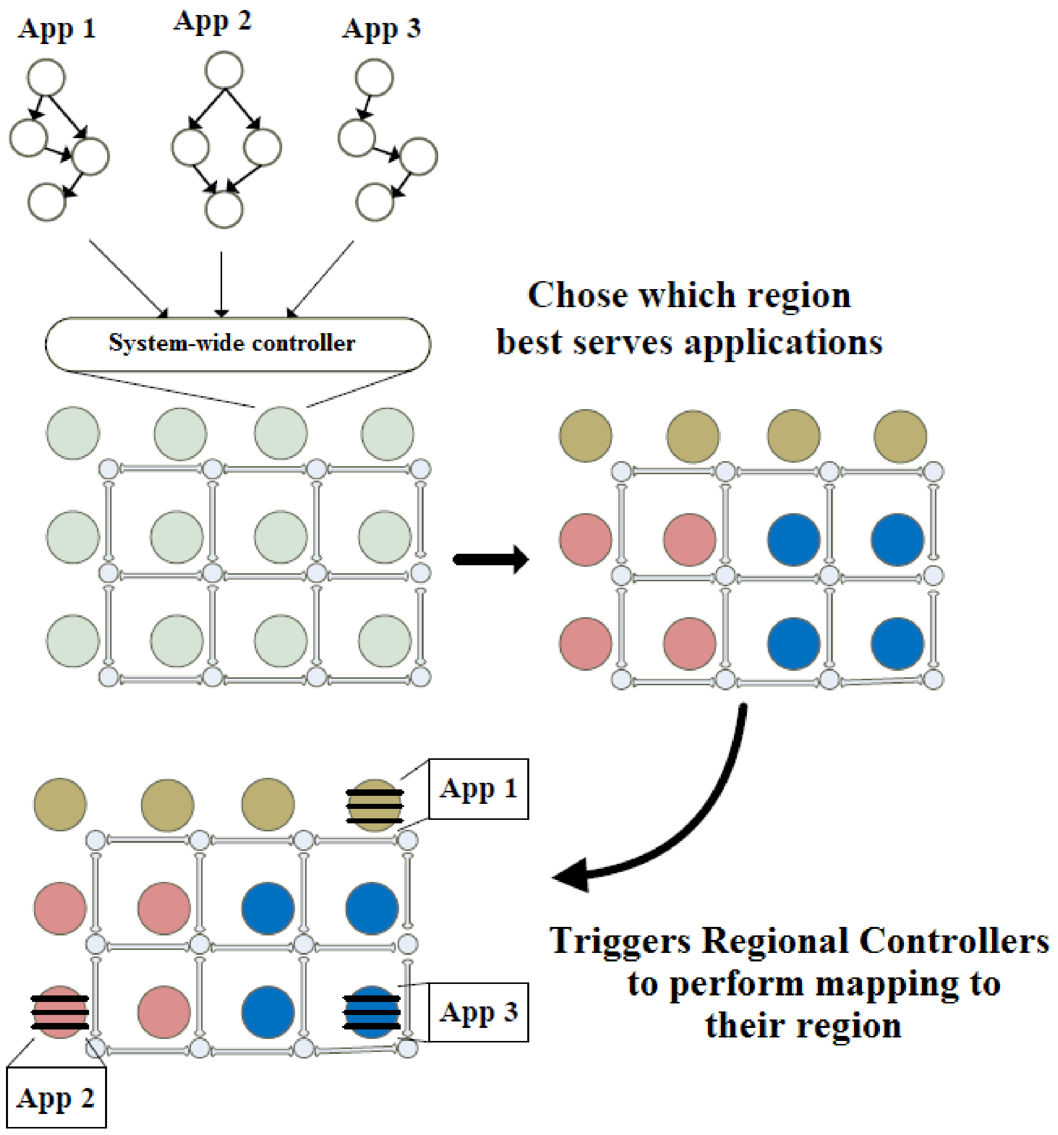

App 1
App 2
App 3
System-wide controller
Chose which region best serves applications
Triggers Regional Controllers to perform mapping to their region
App 1
App 3
App 2

Figure 2.5: Main idea of the run-time manager of [7].

The mapping is carried out in a distributed manner. In order to achieve that, the platform is partitioned in *regions*. A region is a subset of the set of all the tiles on the NoC with no fixed boundaries, and can be reshaped, created or abolished when necessary. The manner in which the partitioning is performed is different in homogeneous and heterogeneous platforms. The overall flow of the resource management algorithm for both homogeneous and heterogeneous platforms is presented in the following figure:

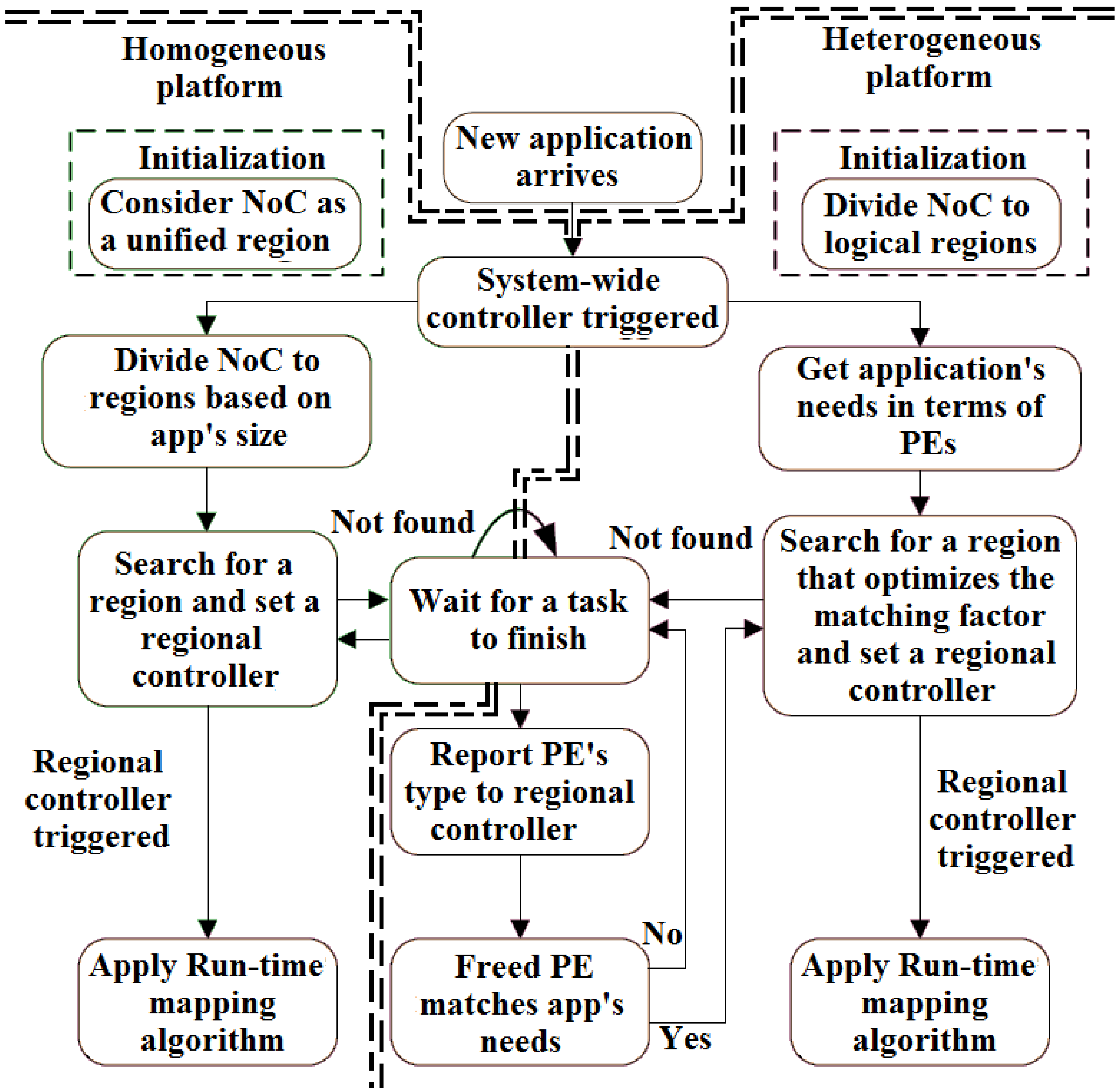

Homogeneous platform
Initialization
Consider NoC as a unified region
New application arrives
Heterogeneous platform
Initialization
Divide NoC to logical regions
System-wide controller triggered
Divide NoC to regions based on app's size
Get application's needs in terms of PEs
Not found
Not found
Search for a region and set a regional controller
Wait for a task to finish
Search for a region that optimizes the matching factor and set a regional controller
Report PE's type to regional controller
Regional controller triggered
Regional controller triggered
No
Apply Run-time mapping algorithm
Freed PE matches app's needs
Yes
Apply Run-time mapping algorithm

Figure 2.6: Flow of the run-time resource management of [7].

Upon arrival of an application, a system-wide controller is invoked. This task is called an "Emperor" task. It is a light-weight in terms of resource demands, it can be executed on any type of processing element of the platform and it is responsible for:

i. getting the requirements of the incoming application
ii. selecting the appropriate region to map the application on
iii. triggering other cores in the region to perform the run-time mapping algorithm. These regional controllers are referred to as"Local Kings".

In addition to the System-Wide Controller, in every region there is a specific node called a Regional *Controller* (*RC*). These tiles are responsible for any action involving the mapping in their respective region. Their code can also be executed on any processing element of the platform. A Regional Controller is responsible for:

- Computing the mapping for the region for which the controller is responsible for.
- Collecting data for the region.
- Communicating and exchanging data with the System-Wide Controller.

The System-Wide controller is responsible for selecting the region on which the application will be mapped on. After that it is responsibility of the Regional Controller to calculate the actual mapping and inform the System-Wide controller. As far as the mapping is concerned, the goal of the algorithm is to compute a fast and efficient mapping that minimizes the energy consumption from the execution of an application. After an initial mapping has been performed an iterative application node swapping process is employed in order to further reduce the total communication cost.

Comparing the RTM algorithm for heterogeneous and homogeneous NoCs its major difference is the way the regions are initialized. On homogeneous platforms during the initialization no regions exist, and are created for each new application, and abolished when the execution finishes its execution. On the other hand on heterogeneous platforms, the NoC is partitioned in regions from the very beginning, with the selection of number and size of the regions left on the designer's judgment and Regional Controllers are appointed on these regions right away. Regions can be reshaped or created to better accommodate new applications, but they can only be abolished if they are an empty set.

Finally, the concept of matching factor is introduced. This designer specified percentage value dictates how strict the mapping is as far as the preferred processing elements of an application are concerned. An MF of 100% dictates that the application can accept only its most preferred processing element, while an MF of 0% dictates that an application can accept any of the processing elements it is able to work on.

2.4 Moldable Parallel Job Scheduling Using Job Efficiency: An Iterative Approach [13]

In this work, the authors present two approaches for a greedy centralized scheduling strategy. Their applications of interest are the moldable ones, parallel applications that are accompanied by information provided by the user. These information are the range of processors the application can run on and run times or the speedup characteristics and constraints of the job.

The main idea of the first approach is that the scheduler associates defines a maximum allowable partition size on the processing elements for an application and afterward uses a greedy scheduling strategy to choose an actual partition size with aim to minimize response time. The fundamental problem of an unrestricted greedy approach to choose partition sizes for all jobs is that most jobs tend to choose very large partition sizes as far as their processing element usage is concerned. As a result the authors employ a fair-share based allocation scheme with system-wide factor which takes into account the weight of each job in order to utilize the recourses in a more fair way. This factor is called an "overbooking factor" (ObF) and it scales up the weight-function of the job determining the upper bound on partition size.

The value of ObF severely affects the average turnaround time of the applications. For example, an increase in ObF:

- Tends to increase the average number of waiting jobs in the queue of pending jobs. Each job's maximum partition is increased and thus the number of jobs that can concurrently run decreases. This causes the average turnaround time of light-weight jobs to increase, since turnaround time of these jobs is overwhelmed by their time on the queue.
- On the contrary, the average run-time of heavy jobs tends to decrease, causing the average response time to also decrease, since run-time dominates queue time.
- When several similarly sized jobs are present, with a high ObF their execution is serialized lowering average response time. An ObF equal to one, would have allowed all them to run concurrently.

In order to further elaborate their scheduling scheme, the authors introduce a system-wide efficiency factor (EF). The efficiency factor limits how much a job's maximum allocation can change from its fair-share limit. The efficiency factor is described by the following relationship:

$$\max\big(1, FairshareLimit * (1 + EF)\big) \leq$$
$$EfficiencyLimit \leq \min\big(Systemsize, FairshareLimit * (1 - EF)\big)$$

The fact that the speedup characteristics of an application are taken into account in its allocation, enables the scheme to take advantage of the benefits of overbooking for tasks that scale well enough. These tasks efficiently use additional processors without wasting processors on jobs that cannot efficiently use them.

The resource management scheme was tested for various sizes of application load for a variety of combination of overbooking and efficiency factor. The conclusion is that the choice of both the factors, not only depends on the relative size of the jobs on the system, but also their overall scalabilities and overall system load. In case of a good overall scalability, a high overbooking factor along with the efficiency based scheme provides the best results. On the other side, in case of poor scalability, a high overbooking factor is detrimental to the average execution time.

The second scheduling scheme presented employs an iterative approach to the scheduling algorithm. This effort stems from the fact that the first approach demands the choice of appropriate parameters like the overbooking and efficiency factors. Ideally, the scheduling policy has to take into account the characteristics of a job mix and function without the need for tuning special parameters of the algorithm. Thus, an iterative backfilling approach is developed to overcome this problem.

The main idea of the iterative approach is that instead of setting an upper bound on job partition sizes, schedules are generated incrementally and iteratively using global information. Initially every job is given a minimal partition of one processor, thus creating a conservative backfilling schedule. Gradually through iteration, this schedule is modified by giving a processor to the job that will benefit the most from it. This benefit is measured in terms of decreasing the runtime of the job. If the offer of the processor to this job decreases the average response time of the schedule then the addition is accepted, otherwise it is rejected.

This approach takes all the aforementioned characteristics of the job mix into account: load, scalability, job size and utilization. The use of turnaround time as the scheduling metric, selecting the job that provide the greater improvement in expected runtime and iteratively selecting next jobs creates a flexible, adaptable algorithm that is able to handle a diverse set of job scalability features. As a result, the iterative scheme performs well across various workloads and small jobs do not receive poor performance in order to improve the performance of large jobs which proved to be a drawback of the first approach.

2.5 Malleable applications for scalable high performance computing [14]

The authors of this work do not provide a resource management but investigate how malleable applications can actually be created and their impact is on dynamic systems and more precisely on a dynamic cluster environment. As stated in section 1.5, malleable applications are those with the ability to accept dynamic changes in the number of computational resources they run on. As a consequence, they provide the middleware which is responsible for the resource allocation, the ability to autonomously reconfigure them as resource availability of the system varies over time.

Besides malleability, the approaches for dynamic reconfiguration of application have involved fine-grained or coarse –grained migration. In the first case, the computational components of an application (actors, agents or processes) are moved to other processing elements, to utilize unused cycles. In the second case, checkpoints are used to pause an application at a certain point of its execution and restart it with a different set of resources to utilize newly available of stop using ones that do not perform well for this application.

Both of these migration approaches fail in many cases and create a bottleneck for the performance of the application. Thus, the concept of malleability has been incorporated into MPI programming framework and SALSA programming language to provide the programmer language level constructs and libraries to produce malleable programs from existing working code or create new ones.

Apart from the existence of a robust programming framework for the development of malleable applications, there is also the need for an effective way for these applications to be allocated on the available resources. A way for this to happen is to extend a middleware to perform such an allocation. In this work, the modular Internet Operating System (IOS) was extended to reconfigure applications autonomously using the concept of malleability. IOS is responsible for the profiling and reconfiguration of applications, allowing entities to be transparently reconfigured at runtime which is achieved by an interface implemented by the applications. An IOS agent is present on every node of the cluster including a decision module which determines when and how to perform reconfiguration of an application.

For the examination of the malleability, two representative applications have been chosen and modified to utilize malleability features. The first is an astronomical one while the second is an application which simulates heat diffusion. Both of them have their unique need in data manipulation and malleability design. The resource availability is altered by making clusters available and unavailable to the applications. Results showed that dynamic malleability is significantly more efficient and scalable than application stop-restart and can be efficiently performed at run-time. Additionally, dynamic malleability compared to migration on a dynamic cluster, runs up to 15% faster on the different configurations tested.

2.6 DistRM: Distributed Resource Management for On-Chip Many-Core Systems [11]

The authors of [11] present a distributed run-time resource management framework for malleable applications. As the name of the work suggests, the framework is called DistRM. The application model used for the examination of the proposed framework is the one proposed by Downey [9] and presented in section 1.5. The framework is distributed in the sense that different cores, dispersed around the grid of cores, are responsible for performing mapping decisions and managing applications. These cores are called agents.

To manage regional information in a distributed way, the authors suggest that there is a directory service distributed to the entire platform. This directory service enables the agents to communicate with other agents without the need of broadcast communication. All available cores are split into evenly distributed clusters and each cluster contains one directory service running on one of the cores. The agents register themselves at the directories corresponding to the cores occupied by the own application.

When a new application arrives on the system a randomly chosen core is triggered to initiate its agent. This is a means of load balancing without the need for global system state knowledge. This core acts as a seed for searching resources and there is a probability that this core will not be chosen for the application. The agent determines the cores the application will eventually run on. The agent searches in randomly chosen areas on the system which are gradually moving away from the core if no suitable core for the application is found. When such a request has to be made in a region further from a specific threshold, then this is consider a far request and the core in the center of the far region is delegated to discover a far request manager. It utilizes the knowledge of the distributed directory service and forwards the request to the agent with most cores in the desired area. Figure 2.7 illustrates the interplay of the resource management agents and applications.

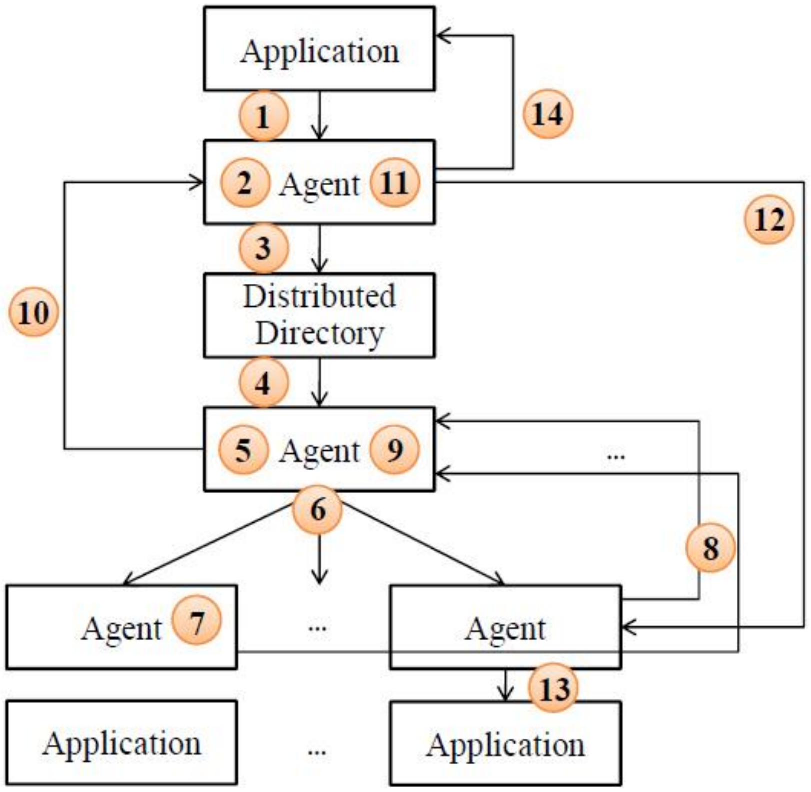


Figure 2.7: Interaction of components of the DistRM algorithm. [11]

The figure shows that an agent can communicate with all other agent for the bargaining of cores to take place. A core is offered to the requesting agent if its gain in speedup is greater than the loss in speedup of the agent offering it. To avoid constant trades of cores between two agents, a core trade takes place only if the gain of the receiver is significantly bigger compared to the loss of the offering agent. In any case of a core offer, the offering agent has to receive a reply dictating if its offer is accepted or not. The agents also communicate with the cores holding the regional information and with the cores of their application in order to reconfigure and resize it.

The unallocated cores of the platform are handled by a special category of agents called the idle ones. These agents handle cores when the platform is initialized or when an application has finished its execution and there is no need for its agent to exist anymore. They are located at fixed positions on the platform and form a regular grid distributed all over the system. When another agent asks an idle agent for a core, the latter always offers one provided that it handles at least one.

Finally the concept of self-optimization of an application is introduced. After an agent has discovered an initial set of cores for its application, it requests more cores in order to increase the speedup of the application. At some point at run-time, an agent sends request for cores to its nearby regions. This idea helps to optimize the mapping of application over time and as a concept it will be further explained in section 3.5. To ensure that requesting additional resources does not happen too often, the delay between two self-optimization processes is doubled after the completion of one.

Chapter 3: Proposed methodology

3.1 Overview

Our proposed methodology is a run-time, distributed recourse management framework. Our interest is in malleable applications. The framework is responsible both for allocating some initial cores for an application to be executed on and to provide a way that the application is managed throughout its execution time. The entire above are performed with intention to minimize communication between and nodes and result into a fair distribution of resources. Another very important aspect of our proposal is that it takes into account both the application characteristics and its current workload. In that way unnecessary functions are avoided and the message count for communication is kept low. The framework is designed to work not only in homogeneous platforms but also in heterogeneous ones.

3.2 Definitions

A many-core platform topology and its communication infrastructure can be uniquely described by a strongly connected directed graph $P(I,N)$. The set of vertices N is composed of two mutually exclusive subsets N_{PE} and N_C containing the available platform's *PE*s and the platform's on-chip interconnection elements (*C*), such as routers in a NoC technology. Each PE can be of a specific type and differ from the other platform types (heterogeneous platform) or all PEs can be of the same type and functionality (homogeneous platform). In our framework $T_{pe}[i]$ for each PE in N_{PE} specifies the type T of the PE with index i. In a heterogeneous platform, the T_{pe} varies for each PE while in a homogeneous platform T_{pe} is the same for all PEs.

In our work, the speedup is calculated in a way that will be presented in Algorithm 2, taking into account the set Q and using the speedup function provided by the application model [9]. As a consequence, for every application, we also define Util[pe[i]] $\in [0, 1]$ that states how good the app(W, var, A, Q) is served by the pe[i] with $T_{pe}[i]$ type.

Finally, we define the region R for each core. This region is defined as the area with center the index of a given node and includes any other node in the Manhattan distance (sum of hops in each direction) with value equal to the size of the platform in the X dimension divided by the number of controller cores in the same dimension. This region is very important, since we consider every core included in this region, to be near the center core. This region is illustrated in the following figure. The center of the area is the core with blue color and side lines. All the cores with blue color are inside the region and the controller cores with blue lines as well:

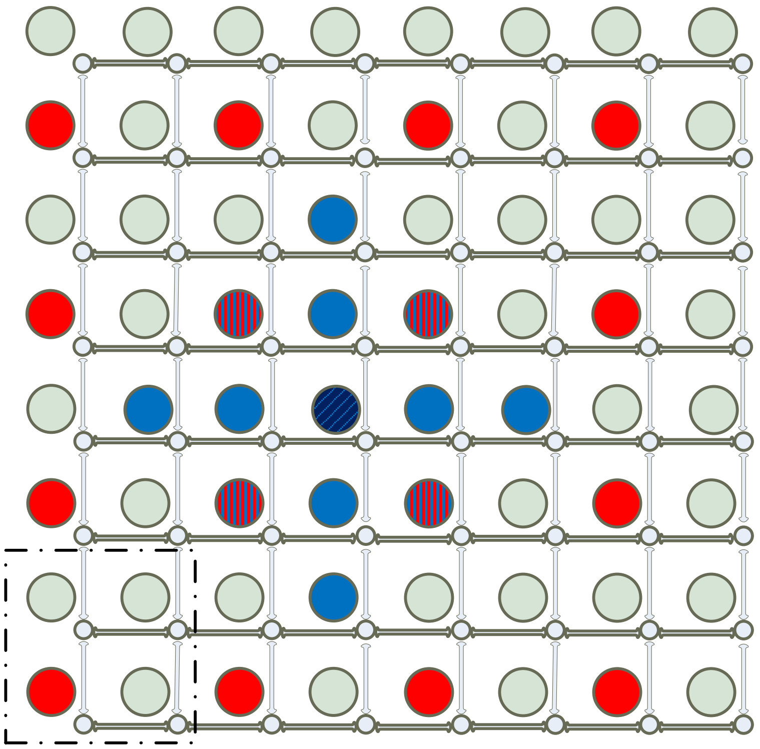


Figure 3.1: The region R of our proposed methodology

3.3 Core Classification

In our framework a core can have one of the following roles:

3.3.1 Controller Core

A controller core is responsible for handling all the unoccupied cores inside a predefined region called cluster. A cluster is defined at the initialization of the platform and cannot be changed during run-time. At the same time, it is responsible for maintaining a list of all the manager cores that possess a core inside this region. We formally call this list a Distributed Directory Service (DDS). This is an essential part of our distributed design since the information of who processes cores inside a cluster is not maintained in a central point but is scattered throughout the platform. This information can be provided to any core who asks for it. In the following figure, one can see two controller cores and their clusters.

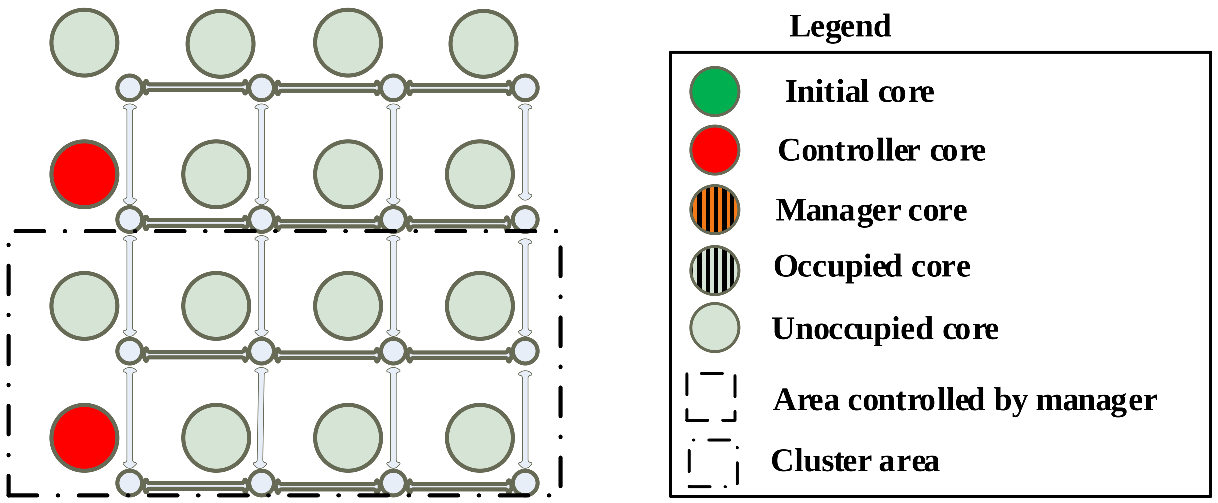


Figure 3.2: Definition of a cluster area

3.3.2 Manager core

A manager manages a single application throughout the application’s life cycle. First, it initializes the application in the sense that it informs the working nodes of the application that he is their manager and he distributes evenly the workload of the application among them. The manager himself does not execute any workload of the application. A manager core and its working nodes are illustrated in the following figure:

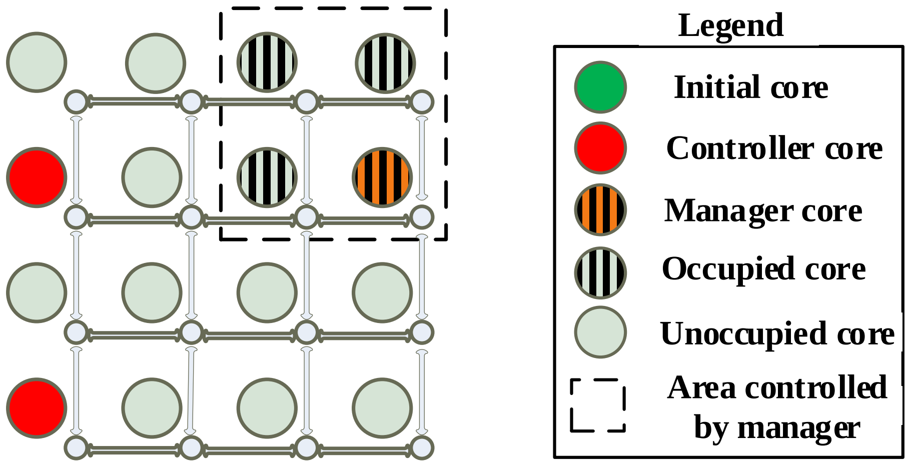


Figure 3.3: A manager and its working nodes. [19]

After its initialization, the manager is responsible for instructing the resizing of the application. A resizing occurs when the application has more or less cores to run on. Instructing the resizing means that the manager must survey all its working nodes in order to find out their remaining workload. When this is done, the total remaining workload must be re-distributed evenly to the new set of working nodes.

Finally, the manager is responsible for asking for additional cores for its application. This process is called self-optimization and will be described in detail in the following section. In this process, the manager asks other controller and manager cores in its region R for additional cores. This implies that in his life time he may offer cores to other applications and resize his application according to their responses.

3.3.3 Initial core

An initial core is randomly chosen and triggered when a new application arrives on the platform. Its purpose is to find the initial set of cores that the application will start running on. It requests cores from controller and manager cores, gathers their offers and determines if at least one core had been offered. In case that no core has been offered, this process is repeated until at least one is found. When the initial set of cores for the newly arrived application has been discovered, the initial manager determines who the manager core of the application will be and instructs him to initialize its application.

Then the initial manager can resume whatever task he was performing before its appointment. A core cannot be an initial manager of two applications at the same time. Therefore, if he is chosen to initialize an application while he is already initializing one, the second one will be stalled until the initialization of the first one is over. In the following figure, an instance of a NoC platform is presented with a newly arrived application in an initial core.

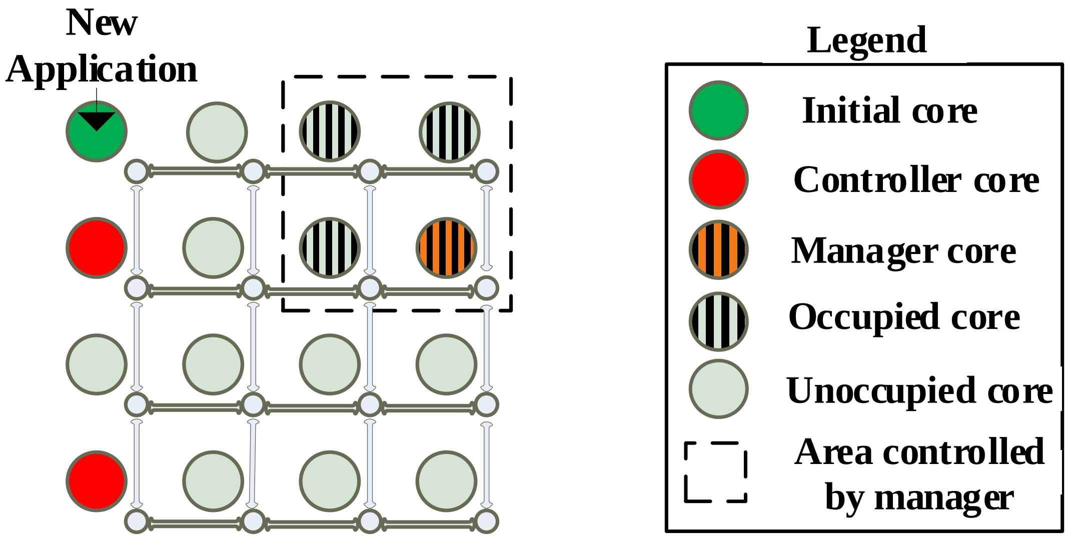


Figure 3.4: A new application arrives on an initial core. [19]

3.4 Gain calculation algorithm

In the following listing the proposed gain calculation algorithm is presented. As it has been mentioned the application model [9] provides a function to calculate the speedup of the application. In this case, the gain calculation of a core trade between two applications is trivial. It is calculated as the speedup of the requesting application with an extra core, minus the speedup of the offering application with a core less.

However, this kind of gain calculation cannot be used for heterogeneous systems. In such systems, given that there is more than one type of cores, every application has its own core preferences. In addition, even if a core is appropriate for an application to be executed on, it may not be the preferable core of the application and this will decrease its speedup. Thus, the need emerged to come up with a way to calculate the speedup of an application in heterogeneous platforms.

The main idea behind the solution is that an application manager will distribute the workload of his application in a way that its speedup is maximized. As stated, we have defined Util[pe[i]] to denote how appropriate is processing element for an application. We can also consider that the speedup provided by the nth core of an application is the difference between the speedup with n cores minus the speedup with n-1 cores.

To calculate the modified speedup of an application first we order the processing elements belonging to it in an ascending order according to their Util[pe[i]] value. Then, the speedup of the i-th PE is calculated as its Util value multiplied with the difference of speedup with i cores and (i-1) cores. The value of the multiplication is added to the total modified speedup. In the following listing we can see the algorithm used whenever an initial or manager core asks for a core offer by another manager or initial core.

```
$offers$ [] = $\emptyset$
while ($gain > 0$) {
    for each $N_{PE} \in ((PE_{src} \cap R \cap Q) - offers[])$ {
    // Actions regarding the calculation of speedup of the destination node
        $PE_{dst} = PE_{dst} \cup N_{PE} \cup offers[]$
        $ord\_PE_{dst} = order\{PE_{dst}\}$
        for $pos = 1$ to $ord\_PE_{dst}.length()$
            $SP_{dst} \mathrel{+}= Util\{ord\_PE_{dst}[pos]\} * (SP[pos] - SP[pos - 1])$
        $gain_{dst} = SP_{dst} - previous\_SP_{dst}$
    // Actions regarding the calculation of speedup of the source node
        $PE_{src} = PE_{src} - offers[] - N_{PE}$
        $ord\_PE_{src} = order\{PE_{src}\}$
        for $pos = 1$ to $ord\_PE_{src}.length()$
            $SP_{src} \mathrel{+}= Util\{ord\_PE_{src}[pos]\} * (SP[pos] - SP[pos - 1])$
        $loss_{src} = previous\_SP_{src} - SP_{src}$
    // Calculate total gain
        $gain\_temp = gain_{dst} - loss_{src}$
        if (($gain\_temp > gain$) || (($gain\_temp = gain$) &&
    $D(manager, N_{PE}) < D(manager, prev\_N_{PE})$))
            $gain = gain\_temp$
            $prev\_N_{PE} = N_{PE}$
    end for
    if ($gain > 0$)
        $offers[] = offers[] \cup N_{PE}$
end While
```

Listing 1: Gain calculation algorithm

The request for cores concerns only a specified area of the platform. In the algorithm, destination node is the one requesting for cores and source node is the one offering cores. The destination node states in which area the offered cores should be located. The main idea of the algorithm is that the source node iterates among its cores and if the trade of a core results in bigger speedup gain to destination node compared to the speedup loss in the source node, then the node can be offered. The one node that its trade will provide the greater speedup gain is the one that is actually offered. Of course, such a core may not exist.

Specifically, in line 1, offers is the set of cores offered from the source to the destination node. Gain in line 2, is the greatest offer in value that has resulted from iteration among all the available nodes of the source. The characterization "available cores" becomes clearer in line 3. A core is available for offer when:

i. It is inside the region in which the destination designated (denoted by R in this case).
ii. It is in the set Q of appropriate cores for the destination application.
iii. Has not already been offered to the destination node.

When a core k meets the aforementioned criteria, in lines 4-8, the modified speedup of the destination application is calculated including the core k to its working set along with the set of cores that have already been placed in the offers set. The gain of the destination node with the core k is calculated in line 8 as the modified speedup calculated with the core k minus the modified speedup without the core k.

Respectively, in lines 9-13 the modified speedup for the source application is calculated excluding the core k and any other offered core from its set of nodes. The loss of the source node with the core k is calculated in line 13 as the modified speedup calculated without the core k minus the modified speedup with the core k.

The gain of this possible core trade is calculated in line 14 as the difference between the gain of the destination node and the loss of the offering node. If this result is greater that the biggest gain at that moment then the core k becomes the preferable core for offer at that moment and the biggest gain becomes the gain of the trade of core k.

In case the previously designated core for trade provides the same gain with core k then the one who has the least distance from the destination node is preferred in order to minimize the distance of their future communication.

When all the available cores of the source node have been checked, if we have a positive gain then the core that was chosen from the iteration is added in the offers set and will be offered to the destination node. A new iteration will begin until there is no positive gain from a core trade. Given this aspect of the algorithm we can describe it as a greedy one. In addition, in a homogenous platform, since the speedup calculation of an application is straightforward, the actions regarding this calculation both in source and destination are much simpler. We just calculate the speedup with or without a core in both cases respectively.

3.5 Self-optimization process

When an application manager is appointed, he is provided with a set of working cores that have been discovered by the initial manager. An important part of our framework is that the manager core can decide to look for more cores in order to increase the number of his working cores. This is called a self-optimization process since its ultimate goal is to maximize the speedup of the application being managed.

Since all managers can initiate such a process, after some time the resources of the platform will have been better shared between the managers. For example, suppose that one application has received a great amount of cores at an area, because at the moment no other applications were active in that area of the platform. When new applications arrive, they may acquire only a small subset of these cores. Obviously, in most cases this distribution of resources is unfair. By means of the self-optimization process, the newly arrived application will acquire some cores of the initial application running in that area and the overall average speedup of all application will increase even though one of them has lost some of its working cores. This can be illustrated in the following figure:

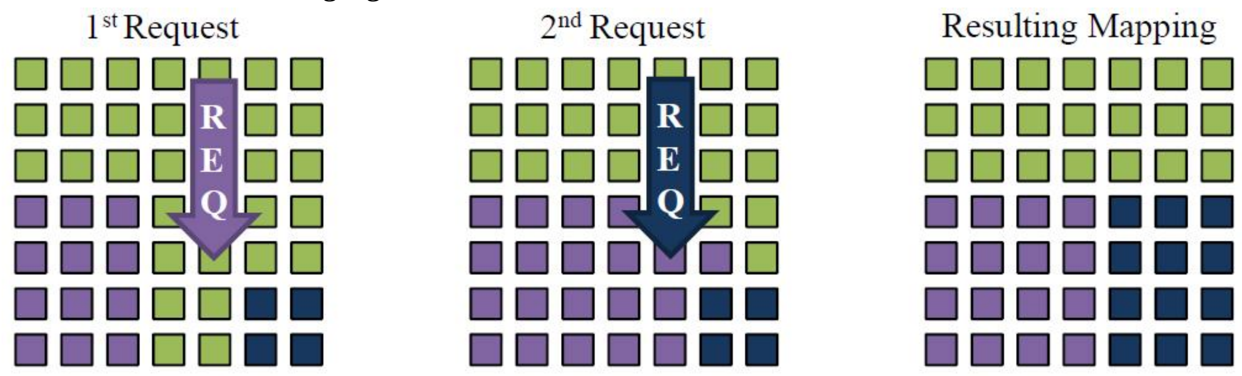


Figure 3.5: More fair resource allocation through self-optimization process. [11]

Additionally, the resource availability of the platform changes dynamically as some applications come to an end. Since there is no central resource management scheme, the existing applications have to claim those new cores without having original information about their availability. Such a case is depicted in the following figure:

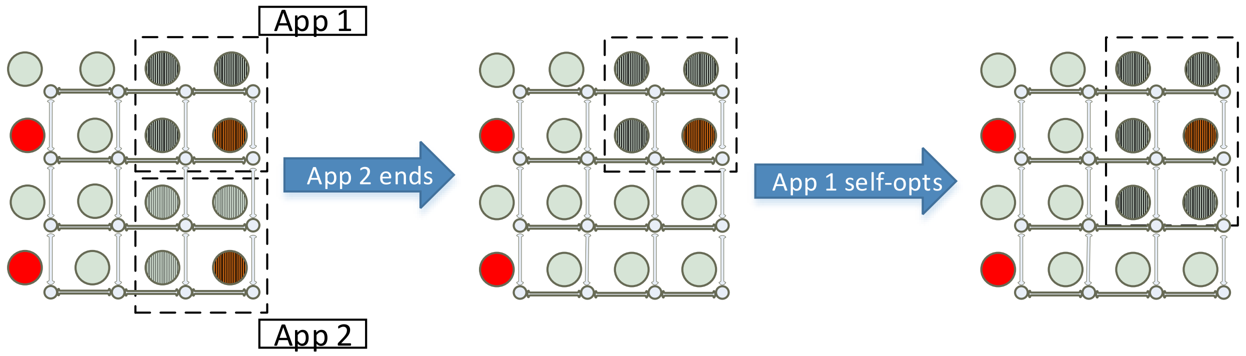


Figure 3.6: Utilization through the self-optimization process of freed resources

However, it is not always the right decision for a self-optimization process to take place. From the time that it is initialized until it is complete, a relatively big amount of time will pass by and a quite complex interplay between manager and controller cores is required. Thus, our design states the following criteria under which a self-optimization process must **not** be initialized.

i. The application being managed has maximized its speedup. The speedup function provided by the application model has in all cases an upper bound. So if an application has reached this bound, it is obvious that there is no reason why it should try to increase its set of working cores.
ii. The application is near to its completion. The application model provides us with a way to be aware of the remaining time units until the workload of the application has been served. It is actually an estimate, since working nodes can be interrupted in order to complete another newly emerged task such as to serve as initial managers. Despite that fact, the estimate gives us a way to know if the completion of the application is near or far. Consequently, if the necessary time for a self-optimization process to take place is much greater than the estimate for the completion of the application, the process will not start even if it has the potential of a great increase in its speedup.

Finally, even if an application meets all the criteria for self-optimization it is not wise to constantly burden the platform with such a process. Thus, between two such processes, a time interval must pass by. This time interval is doubled each time a self-optimization is complete. The only case a self-optimization is imperative is when a manager has no working nodes. It is reminded that the manager core **does not** execute any of the workload of the application. Thus if he possesses no working core he must immediately search for one.

3.6 Communication Scheme

In the following figure one can see an overall flow of the proposed methodology.

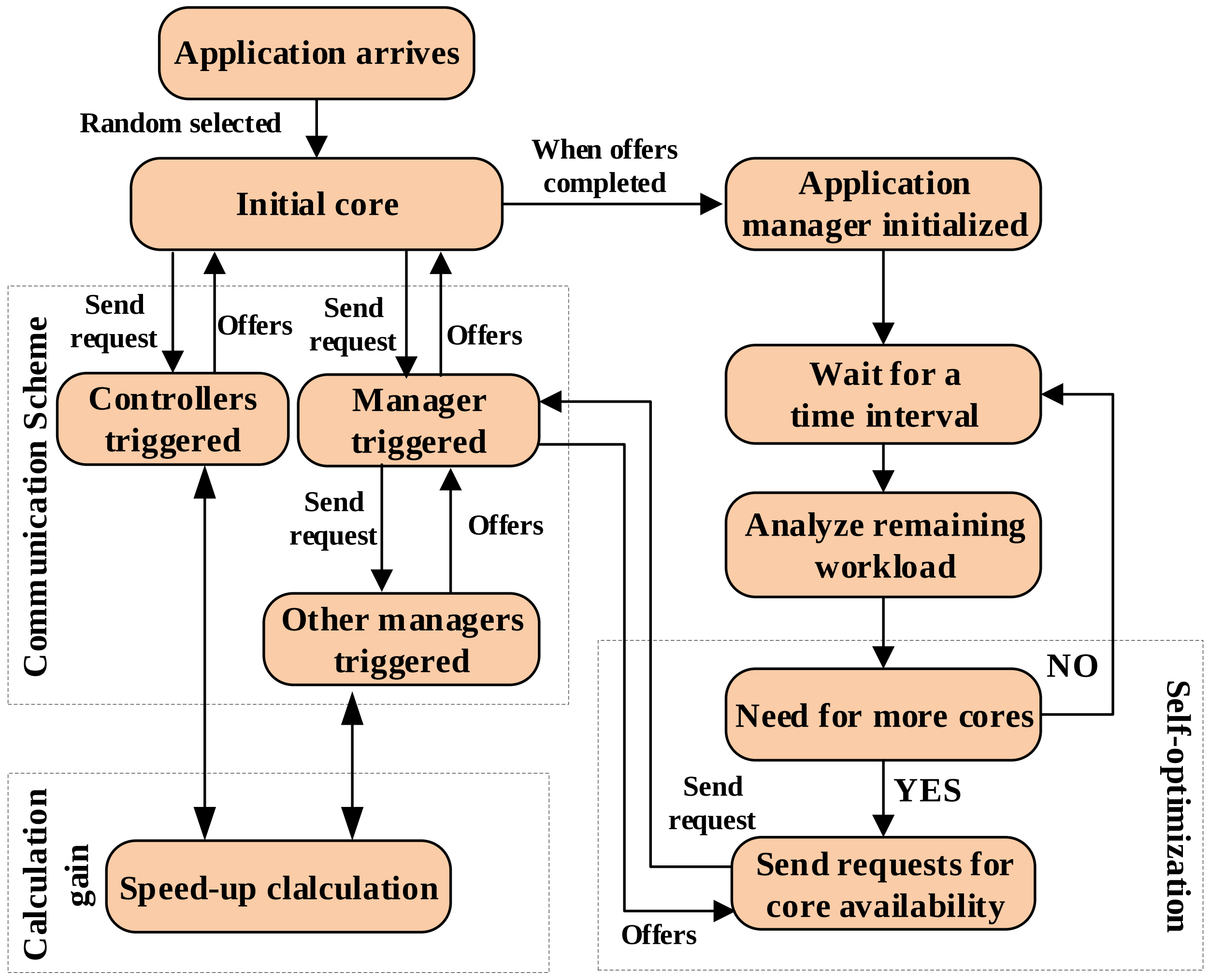


Figure 3.7: Overall flow of the proposed methodology. [19]

3.6.1 Request issuing algorithm

There is a list of steps that need to be complete whenever an initial or manager core has to issue requests for cores.

Step 1: The core issuing the requests must determine the regions in which cores will be requested. There are two types of regions, near and far. The distinction is being made because requests in far regions are handled in a different way compared to requests in near regions. In fact, we consider only one near region, the region R as defined in section 3.2. Far regions are considered only if the platform is big enough. If so, a list of far regions to be checked is produced randomly.

Step 2.1: The requesting core sends a signal to his controller core. He wants to find out which controller cores are responsible for the region R. This is because these controller cores contain the regional information about which manager cores possess cores in this region.

Step 2.2: If there is the need for issuing a request in a far region, the request is sent to the center of that region. He is responsible for handling that request in a way that will be presented later.

Step 3: The requesting node after receiving the response from its controller core made in step 2.1, sends a signal to the all controller cores in region R to find out the list of manager cores in that regions.

Step 4: After obtaining the list of managers from the controller cores, the requesting core issues a request for cores to these managers. The information that needs to be conveyed is its application's characteristics including the number of cores he possess plus the region R.

After these steps are complete, the requesting core pauses for a time interval so that he can accept any offers. Any offers made after the interval has expired, are rejected. After the interval has expired, the requesting core must check its offers. Upon receiving them, they are sorted in descending order according to the number of offered cores. In our implementation he only chooses the best offer but any kind of selection scheme can be employed.

3.6.2 Initial core actions

At some point a new application arrives on the system and node 0 is triggered. Node 0 randomly chooses a node excluding controller cores, to serve as the initial core for this application. The chosen node is signaled for this purpose and it receives a message with the characteristics of the new application. He temporarily stops whatever he was doing and sets out to find at least one core for the new application. For example, at some point the state of the platform is:

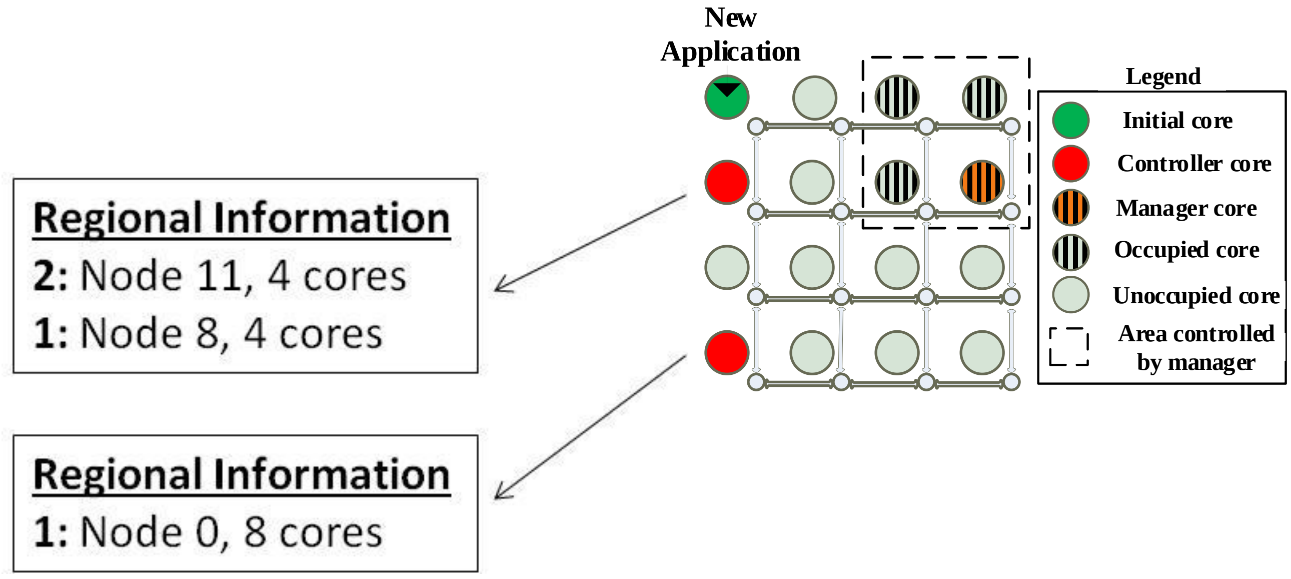


Figure 3.8: An initial state of the platform with DDS information on the controller cores. [19]

The steps involved in the process of finding cores are:

Step 1: He initiates a request issuing algorithm that may include far areas as shown in figure 3.8b.

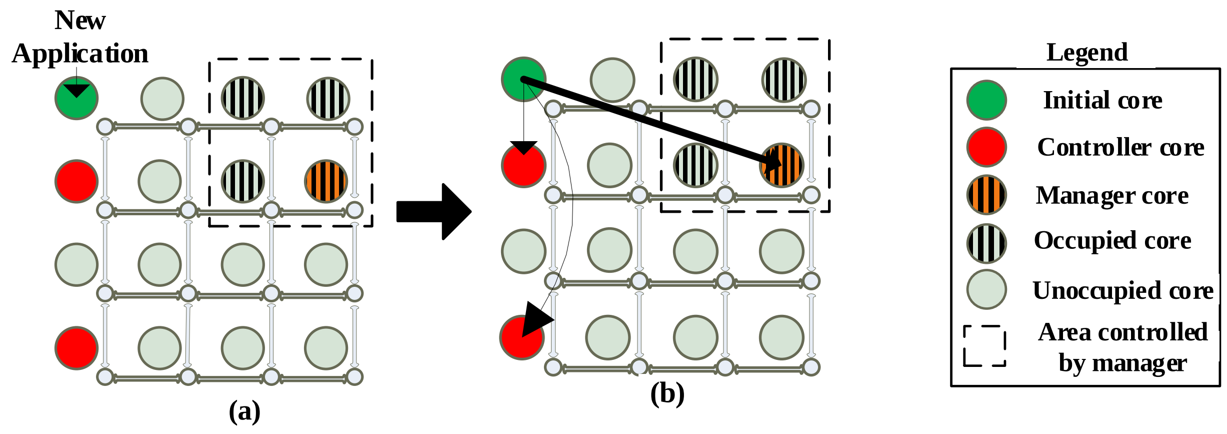


Figure 3.9: Example of the initial core requesting for cores. [19]

Step 2: He checks how many offers he has received. If he has received none, he must start the request issuing algorithm from scratch. This will be repeated until at least one core is found, so that a manager core can be established for the new application.

Step 3: If there is at least one offer, he determines the new manager core. In our implementation, the core that minimizes the distance between all the cores in the acquired set of cores is chosen. This is because, the manager core frequently communicates with his working nodes and we want his messages to travel the least possible distance on the platform.

Step 4: The requesting core must reply to all manager or controller cores that have offered cores to him. His reply can be positive or negative, meaning that he either accepts or rejects their offer.

Step 5: The new manager must be informed that his has to initialize an application. An appropriate signal is sent to him along with the characteristics of his application and his set of working cores, if there are any.

Step 6: The core responsible for assigning initial cores, in our case core 0, has to be informed that the initial core's duties are complete. That is because there may be another pending application for the initial core to initialize.

If the entire above are complete, then the initial core can resume whatever he did before being appointed its duties.

3.6.3 Resizing process

Before introducing the steps of the manager core actions, we must highlight the process of resizing an application whenever the cores assigned to it change. An application can have more cores after a self-optimization process while it can have fewer cores when the manager core offered some of the cores of the application to another manager or initial core.

In any case, we suppose that before the moment of resizing, the manager core of the application has assigned some workload to its working cores. Thus, the manager core has to do the following:

Step 1: Send a message to all his working cores asking for their remaining workload.
Step 2: Monitor the answers of the working nodes until every one of them has responded.
Step 3: Calculate the amount of workload the new set of working cores has to perform.
Step 4: Inform the cores about the workload they have to execute.

We highlight that step 1 is necessary because even if the original workload was distributed evenly among the cores, there is no guarantee that all of them have executed the same, during the elapsed time. This is because one working core can be interrupted to execute another task as for example serving as an initial core.

3.6.4 Manager core actions.

Upon initialization, the manager core has to do the following:

Step 1: Register himself to the Distributed Directory Service so that other initial and manager cores can communicate with him. He signals his controller core, informing about his existence. He also sends his set of working cores to his controller core. It is possible that one of these cores belongs to another cluster. Thus, the new manager core has to be enlisted to another DDS list as well. This decision will be made by his controller core, which is also responsible for informing the respective controller cores. The following figure illustrates the appointment of the manager core by the initial core and the changes this event inflicts on the DDS listings.

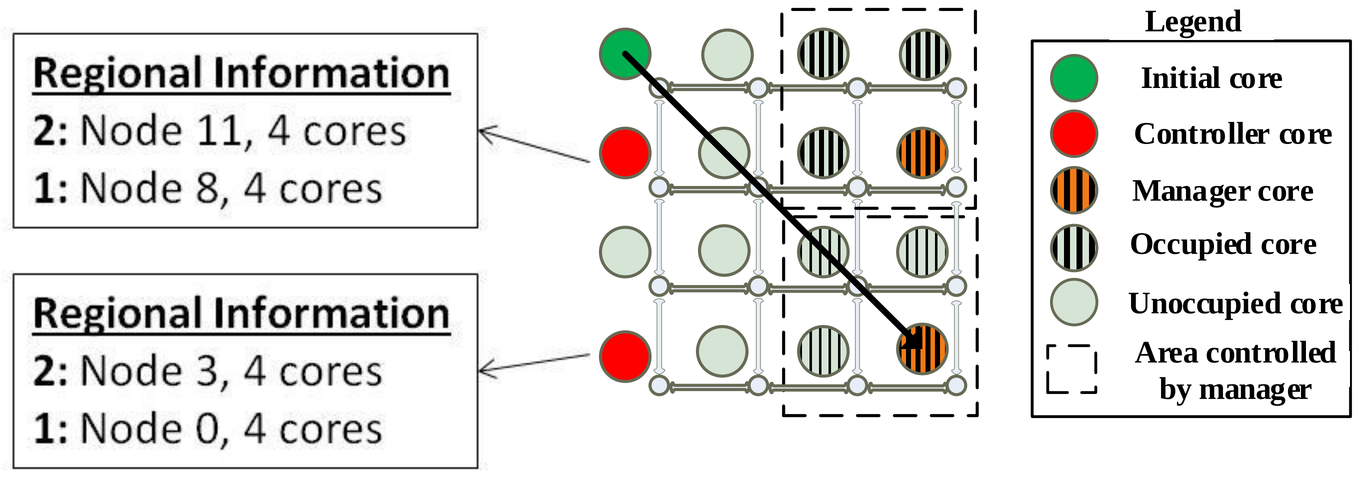


Figure 3.10: Appointment of the manager core by the initial core. [19]

Step 2: The manager core must calculate the workload of his working cores. The workload is evenly distributed among cores.

Step 3: He sends a signal to every one of them and messages them about their workload and his node id to inform them about his being their manager core.

Step 4: He must decide whether he will begin a self-optimization process or not.

Supposing that he decides to do so, the following steps take place:

Step 1: He initiates a request issuing algorithm that **does not** include far areas. If far areas were included, the working nodes would probably disperse around the platform which could prove to be a disadvantage for the communication of the cores, since inter-core communication of an application would involve big distances between cores. Figure 3.9 shows how the manager core asks for new cores in the area near him.

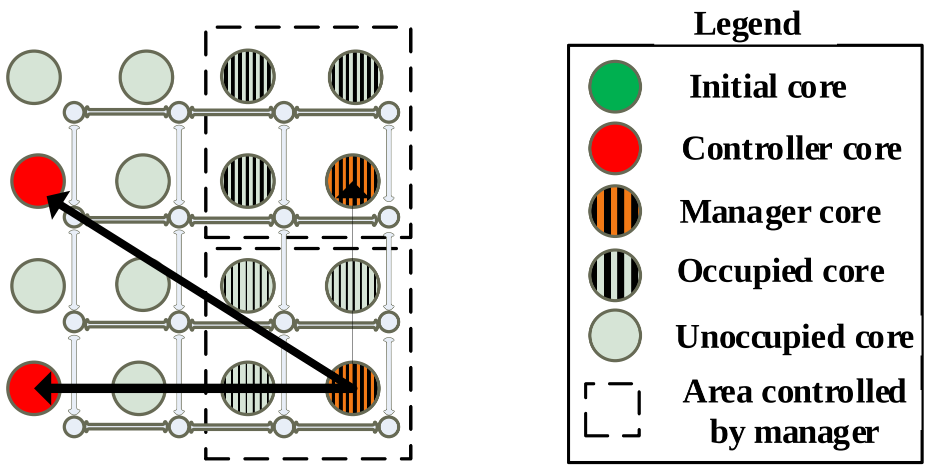


Figure 3.11: The manager core asks for new cores in its near area. [19]

The following steps are valid only if the application has not terminated its execution during the time the request issuing algorithm took place. This is possible, since the time of termination of the application is an asynchronous event that the manager can only estimate the moment he makes the self-optimization decision. The steps are:

Step 2: He checks how many offers he has received. If he has received none, he proceeds to step 7.

Step 3: If there are offers, he chooses the one with the most cores.

Step 4: He initiates a resizing process. While awaiting its completion he goes to next step.

Step 5: He replies to the offers either positively or negatively.

Step 6: He informs his controller core that he added new cores to his set of working cores. He messages the ids of the new cores, in case he needs to be enlisted to a DDS list he did not belong so far. In figure 3.10, the manager core has completed the resizing of its application and the local DDS listing includes this information.

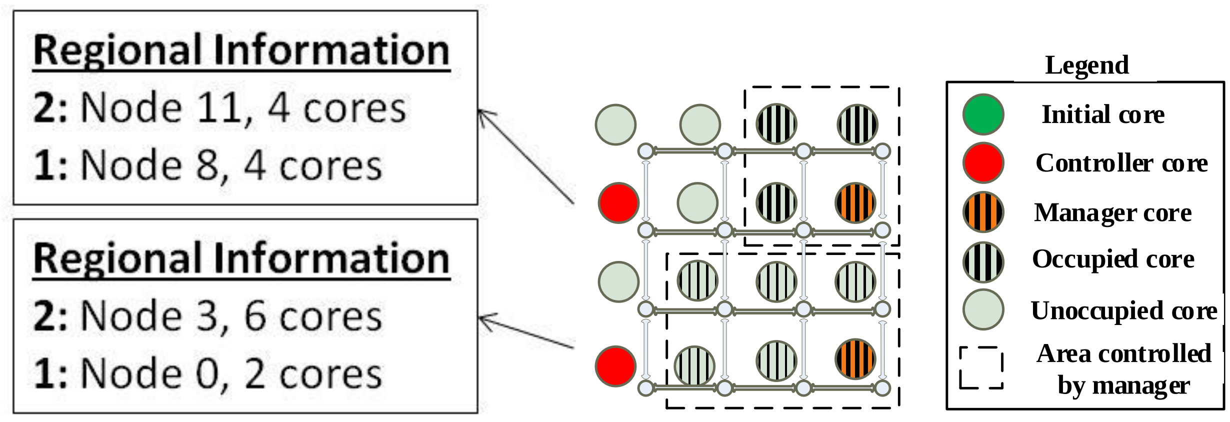


Figure 3.12: DDS listing after a complete self-optimization process. [19]

Step 7: He sets a timer, upon whose expiration; the manager will decide whether he will initiate a new self-optimization process.

In case that the application had ended its workload the time the manager was waiting for offers, all offers are rejected and a negative answer is sent to the nodes that made them. The manager then enters an ending state. Of course, this state is also entered any other time the application finishes.

At this state, the manager must ensure that any offers he has made to other initial or manager cores have been replied. Then he informs his controller core that he wants to be deleted from the DDS listings. The controller core is provided with the set of working cores the manager possessed, in order to eliminate the terminating manager from his DDS list and inform any other involved controllers to do so as well.

If this had been done before getting all replies, we would probably have a controversy. The manager core that accepted the offer would have asked to be added to the DDS listings according to his new set of cores. The ending manager would have asked to be deleted from the DDS listings with the set of cores he owned upon his application completion. Thus for a small amount of time, both managers would possess a common subset of cores. Taking into account that we cannot predict the order in which the signals arrive at cores, a mess can occur in the directory listings, jeopardizing the correctness of the proposed framework.

3.6.5 Far request manager actions

When an initial core issues a far request he sends a signal to the core in the center of the designated far area. The data conveyed are the application characteristics and the designated area. That core, upon receiving the signal and the data, appoints his controller core as a far request manager and feeds him the data. The initial core is also informed that his actual far request manager is that controller core. The far request manager performs the following steps:

Step 1: He finds out which controller cores are responsible for the area designated for the far request.

Step 2: He sends a signal to every one of these cores requesting to find out which manager cores are active in their DDS listing as far as this area is concerned. They are provided with the characteristics of the area.

Step 3: When he gathers the information about the manager cores involved in that area, he sends a signal to both manager and controller cores, requesting core offers. He provides them with the characteristics of the application and the designated area.

After the above steps are complete, the far request manager waits for a time interval to gather offers. Any offers made after the interval has expired, are rejected. After the interval has expired, he forwards the offers gathered combined with his offer to the core that initiated the far request. When that core replies about the acceptance of the offers, he forwards that information to the cores that made the offers. After that, his work as a far request manager is complete. A controller core cannot handle more than one far request offers simultaneously. When he is asked to serve a far request offer while he is already serving one, he only sends his offer to the requesting core.

Chapter 4: Implementation of proposed framework and DistRM

4.1 Design Decisions

In this section the design decisions we made in order to construct and code our proposed resource management framework.

4.1.1 Internal state of nodes

Essentially there are only two distinct categories amongst the nodes of a platform. A node can either be a controller core or a common node. This is because a controller core cannot change its functionality through time. On the other hand, a common node can be a manager, an initial, a working or an idle core. This variety of tasks one core, either controller or common, has to perform mandates the need for the existence of an internal state in the node. From a higher point of view every node functions as a finite state machine. In the following figures we can see these finite state machines.

Apart from the main state of the core, there is a pending state. This means that while the core was executing a certain process, a request occurred for it to serve a new task. That request cannot be served at that time but it has to be served at some point. Therefore, the pending state denotes the process that needs to start the moment the current task comes to an end.

In the following figure we can see the finite state machine from the point of view of the initial core. Note that instead for the word core, the word manager is uses. For example, the state INIT_MAN_CHK_OFFERS stands for Initial Manager Checking Offers. The term AGENT is used for the manager core. Thus, AGENT_SELF_OPT is the state where the manager core initializes a self-optimization process. Every, arrow with a description starting with prefix SIG_, denotes that the core received that signal and changed its internal state.

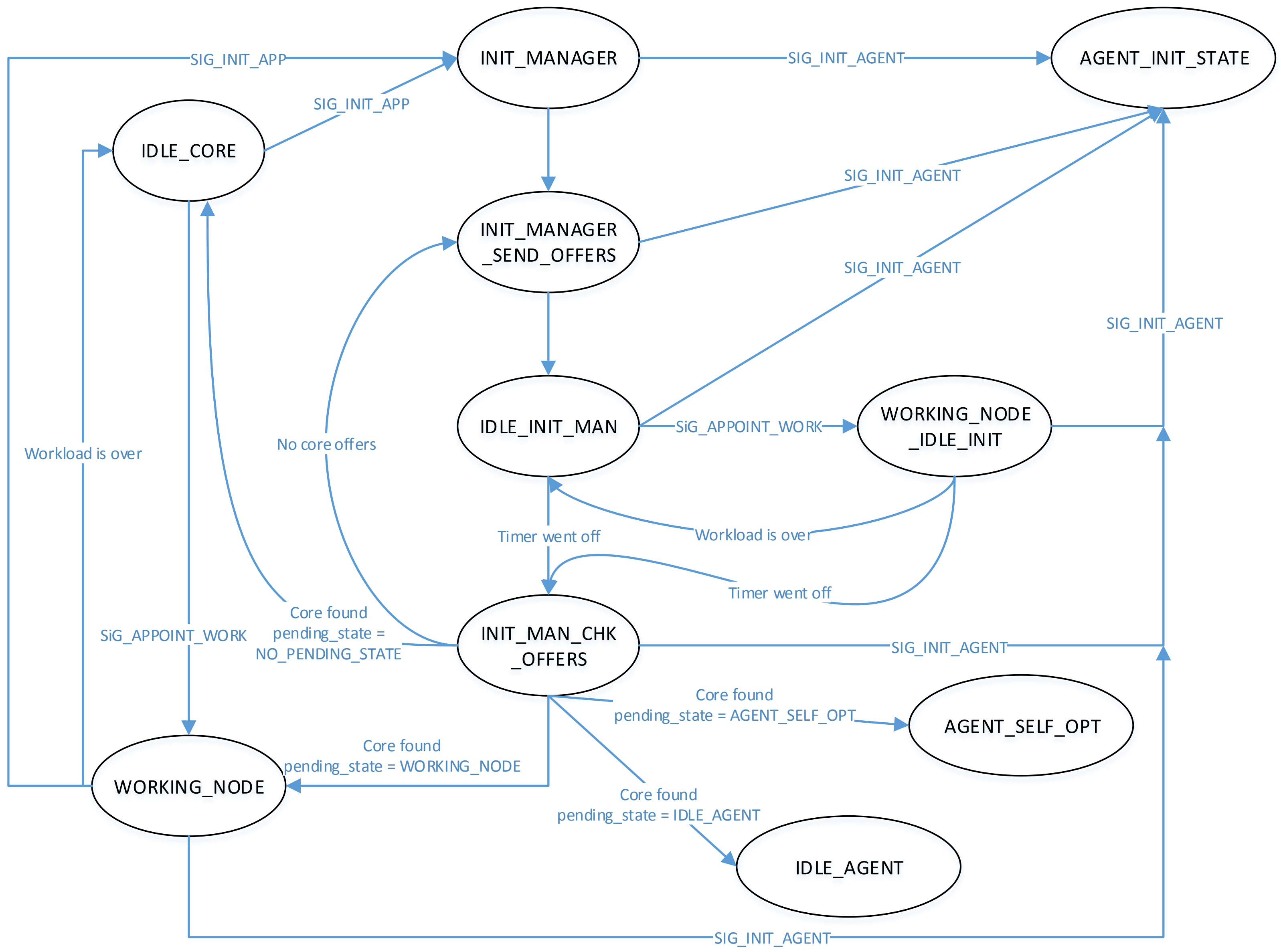


Figure 4.1: Internal states of an initial core

The states presented in the figure, concerning the initial core are:

IDLE_CORE: A state where the core does nothing and waits for a signal.

INIT_MANAGER: A core has received a SIG_INIT_APP signal which denotes that a new application has to be initialized and that core has to act as the initial core of the new application. In that state, an initial core is initialized.

If at this point a SIG_INIT_AGENT signal arrives and the core must also act as the manager core of another application, after completing his initialization as initial core, he proceeds to the initialization of his application as a manager core. After the second initialization he will proceed to the next state, INIT_MANAGER_SEND_OFFERS since his role as an initial core is higher in priority than his role as a manager core.

INIT_MANAGER_SEND_OFFERS: The initial core has completed its initialization phase and starts sending requests for cores in various areas. If a SIG_INIT_AGENT signal arrives he initializes his application and returns to waiting offers.

IDLE_INIT_MAN: After the initial core has sent its requests, he sets a timer and goes into that state waiting for offers. If a SIG_INIT_AGENT signal arrives he initializes his application and returns to waiting offers. If a SIG_APPOINT_WORK signal arrives, this means that the core belongs to an application and must execute some workload. As a consequence it proceeds to WORKING_NODE_IDLE_INIT state.

WORKING_NODE_IDLE_INIT: The core executes workload and also receives offers for core as an initial manager. If a SIG_INIT_AGENT signal arrives he initializes his application and returns to executing workload and waiting for offers.

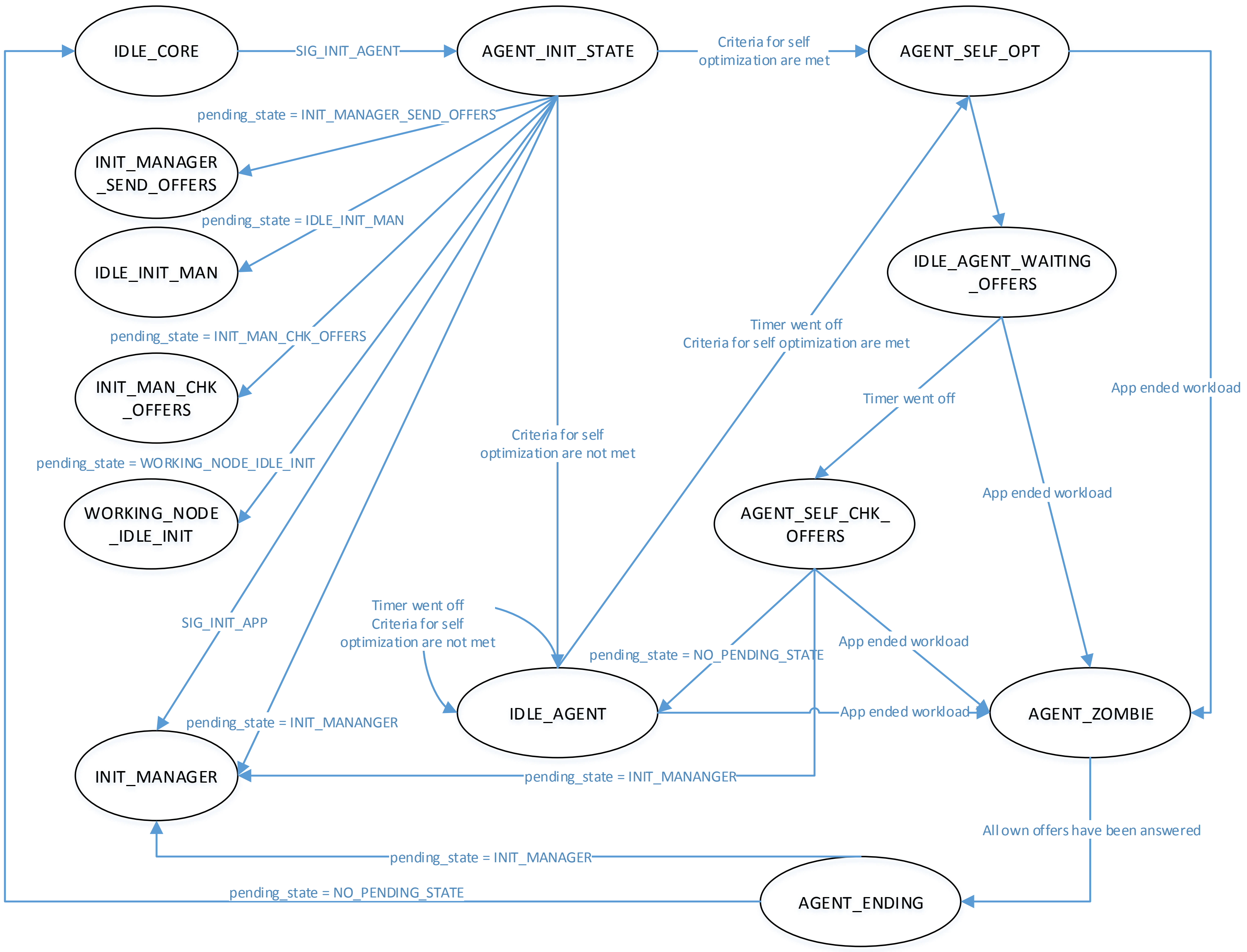
INIT_MAN_CHK_OFFERS: After the timer has expired, he proceeds to this state to check the offers he has received. If he has received no offers, he returns to INIT_MANAGER_SEND_OFFERS state to ask once again for cores since it is imperative that he finds at least one for the new application. If he has received at least one offer he initializes the new manager. After that there are the following possible states:

i. There is no pending task and the core becomes idle.
ii. The core was executing workload when the signal for the initialization of the application came. So, now that the initialization is over, it must return to its previous working state.
iii. The initial core has received a SIG_INIT_AGENT signal and has to initialize his role as the manager core of an application. The same state can occur when the result of the initialization process was that the initial core is now the manager core. This in not unlikely to happen since the near region R where the initial core asks for cores, has himself as center.
iv. The initial core is also a manager core and when the signal for the initialization of the application arrived he was in idle state. Thus, he will return to IDLE_AGENT state.
v. The initial core is also a manager core and when the signal for the initialization of the application arrived he was in a state where a self-optimization process had to be initialized. Thus, will change to an AGENT_SELF_OPT state, which the initial state of a self-optimization process.

The state presented in the figure, concerning a working core is:

WORKING_NODE: A state where the node belongs to a manager core and executes the workload of an application. From this state, he can either change to an initial or a manager core if the appropriate signals arrive or become an idle core when its workload is over.

The states concerning the manager core actions will be presented after the following figure which depicts the internal states of a manager core.



IDLE_CORE
SIG_INIT_AGENT
AGENT_INIT_STATE
Criteria for self optimization are met
AGENT_SELF_OPT
pending_state = INIT_MANAGER_SEND_OFFERS
INIT_MANAGER _SEND_OFFERS
pending_state = IDLE_INIT_MAN
IDLE_INIT_MAN
pending_state = INIT_MAN_CHK_OFFERS
INIT_MAN_CHK _OFFERS
pending_state = WORKING_NODE_IDLE_INIT
WORKING_NODE _IDLE_INIT
IDLE_AGENT_WAITING _OFFERS
Timer went off Criteria for self optimization are met
Timer went off
App ended workload
Criteria for self optimization are not met
App ended workload
AGENT_SELF_CHK_ OFFERS
SIG_INIT_APP
Timer went off Criteria for self optimization are not met
pending_state = NO_PENDING_STATE
App ended workload
IDLE_AGENT
App ended workload
AGENT_ZOMBIE
pending_state = INIT_MANANGER
INIT_MANAGER
pending_state = INIT_MANANGER
All own offers have been answered
pending_state = INIT_MANAGER
pending_state = NO_PENDING_STATE
AGENT_ENDING

Figure 4.2: Internal states of a manager core

The states concerning the manager core are:

AGENT_INIT_STATE: This is the state where the manager core is initialized. This means that he makes his presence known to its controller core, so that he can be enlisted in the appropriate DDS listings. Then he distributes the workload of his application to his working cores. He may come to this state from being an idle core but he may come from any other state of being an initial core. If this is the case, he must return to completing his work as an initial core. If this is not necessary, he must decide whether a self-optimization is a good choice. If it is he proceeds to the AGENT_SELF_OPT state or else he becomes an idle manager in IDLE_AGENT state.

AGENT_SELF_OPT: This is the state that a self-optimization process is initialized and requests for cores in the near region are sent. If during this process the application has not finished its execution, the manager sets a timer and proceeds to IDLE_AGENT_WAITING_OFFERS state to wait for offers by others cores. If otherwise, he proceeds to AGENT_ZOMBIE state.

IDLE_AGENT_WAITING_OFFERS: In this state the manager core is idle and waits for core offers. If the application finishes its workload in this state, he proceeds to AGENT_ZOMBIE state.

AGENT_SELF_CHK_OFFERS: When the timer goes off, the manager core checks the offers he has received. If there is at least one and the application has not finished its workload, he accepts the offers and starts the resizing of the application. If during the self-optimization process a SIG_INIT_APP signal has arrived then this is the appropriate time for the manager core to act as an initial core. If there is not that need, he sets a timer and goes to IDLE_AGENT state. If the application has finished its workload, he proceeds to AGENT_ZOMBIE state.

IDLE_AGENT: In this state, the manager core remains idle in the sense that he is not in the middle of a self-optimization process. It is highly likely that his performing some other task such as sending an offer for cores to another manager or initial core or instructing the resize of his application. If a timer goes off in that state, the manager core checks if the criteria for self-optimization are met. If they are, he proceeds to AGENT_SELF_OPT state. If not, he remains an idle manager. If the application finishes its workload in this state, he proceeds to AGENT_ZOMBIE state.

AGENT_ZOMBIE: This is the state where the manager core waits for all his offers to be answered as explained in chapter 3.6.4. When this happens, he proceeds to AGENT_ENDING state.

AGENT_ENDING: In this state he discards any interactions with his working cores and then informs his controller core for the finish of his application in order to be deleted from the DDS listings. If there is a pending state for the initialization of an application he proceeds to INIT_MANAGER state or else he becomes an idle core.

Finally we have to examine the states of a controller core. In this case, the internal state diagram is much simpler. This is because despite the fact that the controller core almost constantly serves a request, these requests are served inside the signal handling functions without the need the need of a special state. The states are presented in the following figure:

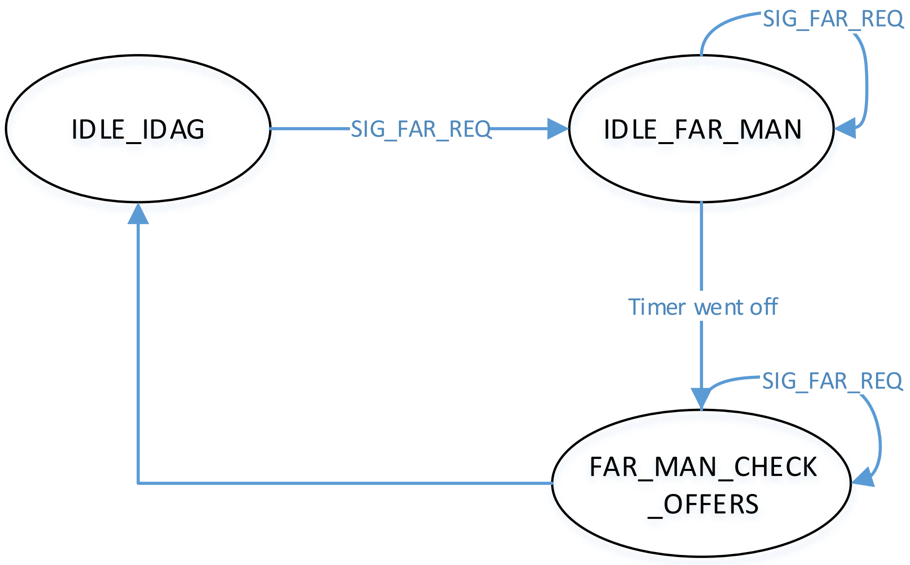


Figure 4.3: Internal states of a controller core

IDLE_IDAG: A state where the controller core is paused and serves requests whenever a signal arrives. In this state if a SIG_FAR_REQ signal arrives, which denotes a far request from a core, he proceeds to IDLE_FAR_MAN state.

IDLE_FAR_MAN: In this state the controller core initially gathers all the necessary information for sending requests. Afterwards he sends the actual requests and sets a timer to wait for core offers. If he receives another SIG_FAR_REQ signal in this state he only sends his offer as described in section 3.6.5. When the timer goes off, he proceeds to FAR_MAN_CHECK_OFFERS state.

FAR_MAN_CHECK_OFFERS: In this state the controller core check the offers he received and forwards them to the cores that initiated the request. . If he receives another SIG_FAR_REQ signal in this state he only sends his offer as described in section 3.6.5. Then he proceeds to the IDLE_IDAG state.

4.1.2 States of an application

An application has also a number of states to help a core, especially the manager one to optimize certain procedures concerning the application. The states are:

RUNNING: This means that the application has been initialized and the working cores are executing its workload.

TERMINATED: The application proceeds asynchronously to this state when all its workload has been executed. Thus, when for example the application comes to an end while the manager is receiving offers, when the time comes for the offers to be checked it is easy to see that the application is finished and proceed to AGENT_ZOMBIE state.

RESIZING: This state dictates that the application is not executing workload and is under a resizing process. This is very important to know, since there is a possibility that the set of working cores of the application may change while it is under a resizing. If a second resizing process begins while another one is on the way, it is definite that it is an unnecessary process and there is a high probability that something will go wrong.

4.1.3 Inter-node communication

From the preceding chapters it is obvious that there is a great deal of inter-node communication. We can distinguish this communication between the part where the appropriate signal is sent and the part where data is exchanged between cores. How exactly a signal is sent, is highly depended on the platform our framework is executed on.

On the other hand, data exchange must happen in a way that ensures that it will complete correctly. Thus, whenever a core A wants to transmit data to core B, it first sends a signal denoting the semantics of its request and then core B must respond with a signal of acknowledgement to show that it is ready to receive data. Only when A receives that signal it sends the data relative to its request.

The signal necessary for our framework to function, along with their semantics are:

SIG_ACK: Signal of acknowledgement for the transfer of data.

SIG_INIT: Signal for the initialization of a core during the initialization of the platform.

SIG_TERMINATE: Signal for the termination of a core during the finalization of the platform.

SIG_INIT_APP: Signal that an application has to be initialized. In other words, a core must act as an initial core.

SIG_IDAG_FIND_IDAGS: A core sends this signal to his controller core when he wants to find out all the controller cores responsible for a region. The controller core receives the signal and sends a SIG_ACK to receive the data. After calculating its response, it sends a SIG_IDAG_FIND_IDAGS signal to the original core to provide him with the necessary data.

SIG_REQ_DDS: A core sends this signal to a controller core when he wants to find out all the manager cores that possess cores in a region. The controller core receives the signal and sends a SIG_ACK to receive the data. After calculating its response, it sends a SIG_REQ_DDS signal to the original core to provide him with the necessary data.

SIG_REQ_CORES: A core sends this signal to a manager or controller core to request for cores in a region. The recipient core receives the signal and sends a SIG_ACK to receive the data. After calculating its offer, it sends a SIG_REQ_CORES signal to the requesting core to provide him with the necessary data.

SIG_REP_OFFERS: When a core has received an offer of cores, he sends this signal to reply whether he accepts the offer or not. The recipient core receives the signal and sends a SIG_ACK to receive the answer.

SIG_INIT_AGENT: A signal sent by the initial core when he wants to initialize a manager core.

SIG_ADD_CORES_DDS: This signal is sent whenever a manager core wants some new cores of his to be added to the DDS listings. The manager core sends this signal to his controller core. The controller core decides whether the manager core has to be added to another DDS listing apart from its own. If this is the case, it sends a SIG_ADD_CORES_DDS signal to any other controller core that needs to be involved in the process.

SIG_REM_CORES_DDS: This signal is sent whenever a manager core wants some cores of his to be removed from the DDS listings. The manager core sends this signal to his controller core. The controller core decides whether some of these cores have to be from another DDS listing apart from its own. If this is the case, it sends a SIG_REM_CORES_DDS signal to any other controller core that needs to be involved in the process.

SIG_INIT_FAR_REQ: A signal sent by an initial core when he wants to initiate a far request in the center of a certain area. The core in the center of this area, replies to the initial core with the same signal to inform him about the actual far request manager.

SIG_FAR_REQ: This is the signal sent by the core in the center of a far region to its controller to inform the latter that it has to act as a far request manager for the core that initiated the far request.

SIG_APPOINT_WORK: A manager core sends this signal to one of his working nodes to notify it that he has to execute some workload.

SIG_TIMER: The signal arriving in a node when its timer has expired.

SIG_CHECK_REM_TIME: A manager cores send this signal to one of his working nodes to find out how much of its workload has been executed. The working node replies with the same signal to provide the manager core with the requested information.

SIG_FINISH: A working node sends this signal to its manager core when it has finished the execution of its appointed workload. Additionally, when a manager core goes into AGENT_ENDING state, he sends the signal to his controller core to inform it about the completion of his work as manager core and be deleted from all the DDS listings he appears in. The controller core decides which other controller cores have to be informed and it sends a SIG_FINISH to them.

SIG_REJECT: This is a signal sent by a core as a reply to a request that it cannot serve at the moment. Of course **not all** requests can be rejected. For example, a request for cores can be rejected with no problem. On the other hand, a signal for the initialization of a manager core cannot be rejected for this would mean that the application would never execute its workload.

SIG_REMOVE_FAR_MAN: This signal is sent whenever an initial core has to discard one of its far request managers, since the latter has reject its role as a far request manager. There is a special signal for this process since the core receiving the SIG_REJECT signal is the core in the center of the far region. This core essentially forwards the rejection to the core that initiated the far request.

4.1.4 Deadlock prevention

Whenever a core sends a SIG_ACK signal in order to receive data, he stalls waiting for the data. It is possible to have a case where core A sends a signal to B and about the same time B sends the same signal to A. For example if A and B request cores from one another. Then, both will send SIG_ACK signals to each other and they will wait infinitely for the response of the other core. A deadlock has presented itself.

Additionally, if we want a design where A sends more than one requests to B at a certain time, we should facilitate the need for a transaction ID to distinguish between the different requests. This would burden the platform with extra information transmitted constantly and since we cannot always predict the arrival of signals on the platform there is a possibility that a latter request could have been served before an earlier one.

Therefore, we decided that at one point node A can have only one interaction with node B. If the need for another interaction arrives before the completion of the first one, it is queued up and the appropriate signal will be sent only when the first interaction is complete. Every node has the following array of interactions in its memory:

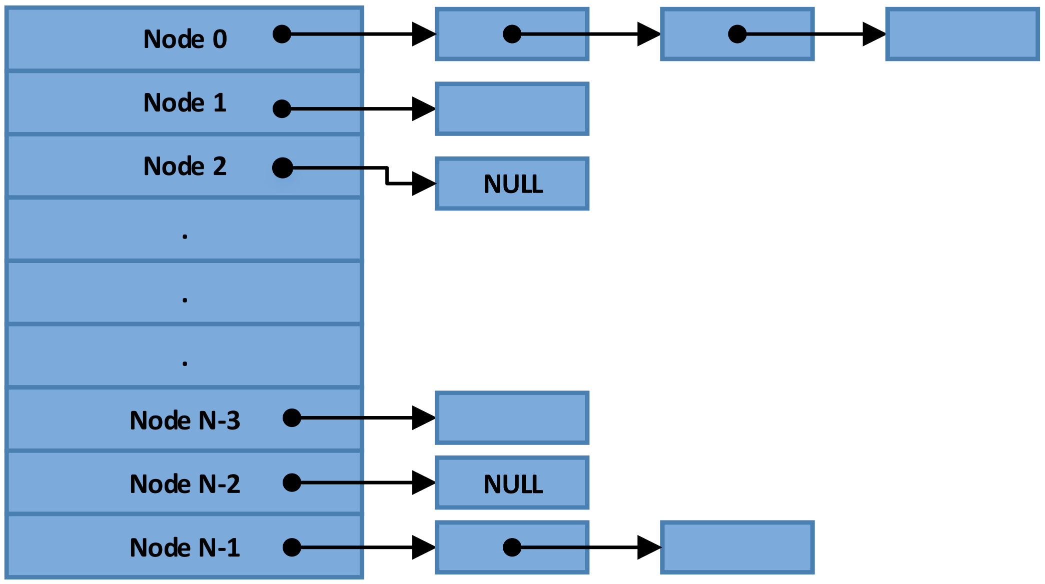


Figure 4.4: Queue of interactions, available in each node

If figure 4.4 represents the memory of node A, the memory line denoted Node 0, is a pointer to the interaction list of node A with node 0. In the figure, node A has an active interaction with node 0, meaning that he has already sent a signal to node 0. It also has two pending interactions with node 0 and the appropriate signals will be sent only when the first interaction is over.

Consequently, in the case presented earlier where the deadlock occurred, both node A and B would understand that there is a deadlock and would abort their requests resulting in a normal operation of the platform.

Of course, there are more sophisticated mechanisms of deadlock prevention. We could have a central task checking for cycles in the interaction graphs of all nodes. However, this violates our distributed design and would probably create a hot-spot on the platform that would require an enormous amount of information to function properly. Perhaps, the most efficient choice in terms of success would be to set a timer whenever there was a need for a data exchange. If the data exchange was not complete when the timer expired, it would seem that a deadlock has occurred and the transaction has to be aborted. The drawback of this implementation is that it requires more sophisticated node design and probably the existence of more than one active timer. Taking into account that there are NoC platforms with no operating system support, it is probable that this design would have made our framework impossible to be executed on these platforms so this choice was also rejected.

4.1.5 Actual workload execution

As stated earlier, a working node has to execute some workload, designated by its manager core. This workload can be transformed in time units of work using the type provided in section 1.5. We decided that the working node will indeed execute some workload that will last as time units as the type dictates.

The opposite option was that a timer would count the necessary time units indicating that the task is complete upon its expiration. This option not only complicates the design due to the existence of another timer but also renders the working node idle as far as actual task execution is concerned. Thus, while the node would theoretically execute workload, he would also serve any incoming requests at the same time. This of course is not the case in any real life scenario.

When a working node actually executes a task, whenever an incoming request arrives, the task has to be interrupted in order for the request to be served. Thus, the execution is actually delayed and the delay is easily taken into account in the final experimental results. In many cases, especially in small platform sizes, this delay is not insignificant at all and can be the leading cause for differences in the total execution time of different applications.

The task that is executed is a trivial one. In our case two nested for loops with a dummy variable increment. This is quite helpful since this kind of task can be interrupted at any time without any considerations regarding its memory footprint or sophisticated ways to pause and resume the task. However, in future implementations any task could take the position of the trivial one.

The following figure depicts the transition of an idle core to a working one and the interruption of its work to perform actions as an initial core.

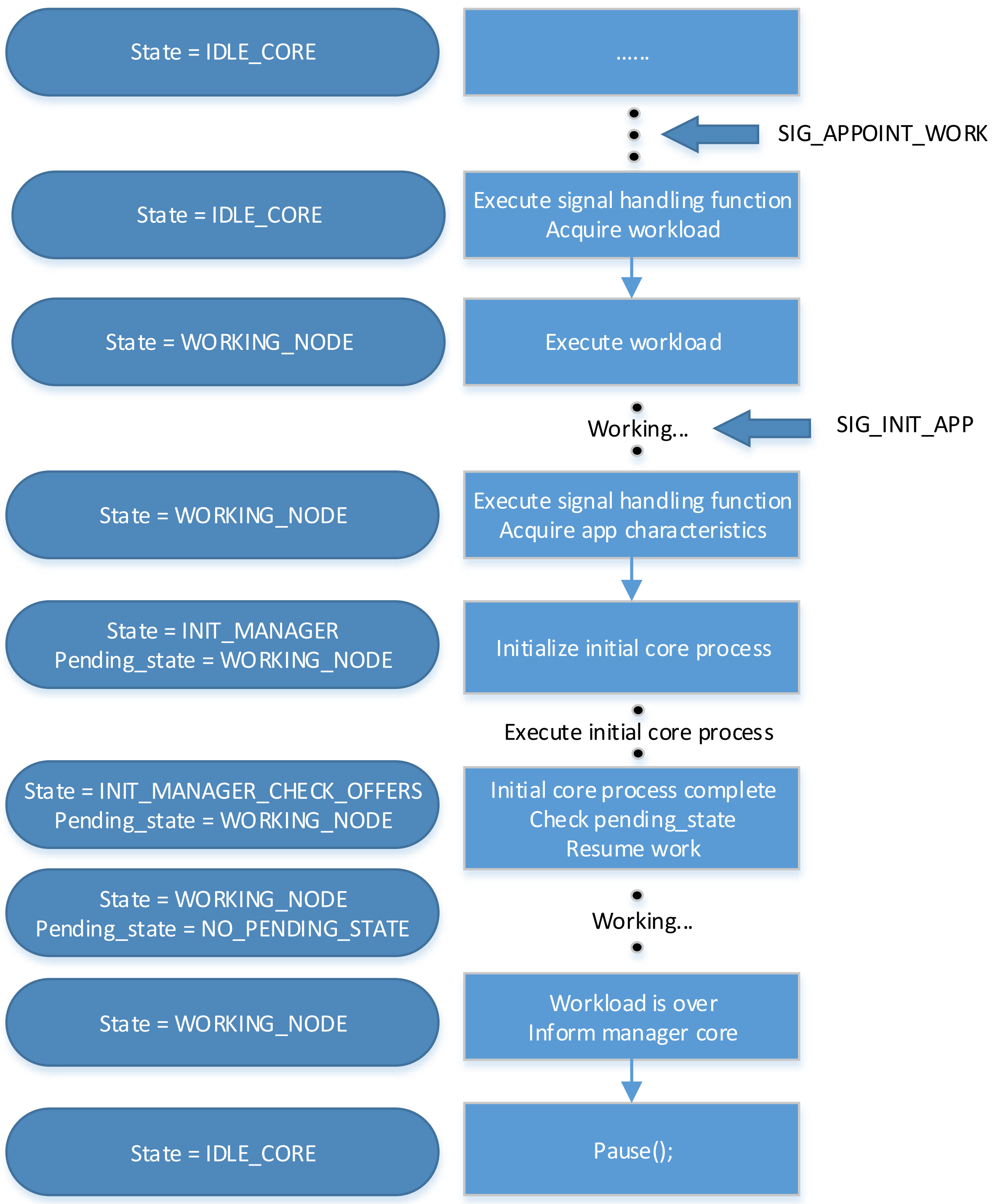


Figure 4.5: A working node interrupted to serve an incoming request.

4.1.6. Input from text files

We have decided that the bulk of the input in both the simulated system and the actual platform will be through text files. This provides us an easy way to run different scenarios without either inputting information by hand or constantly creating scenarios dynamically at run-time. Especially the latter option would prove to be catastrophic since it would be impossible to debug the framework. All text files are produced using the Python programming language.

We have a text file with information about the incoming application on the system. A small segment from this file is presented below:

```
115   360   35.67      9.54   0.46
150   648   2.36       18.98  1.13
196   662   111.92     2.28   1.17
225   607   53.57      8.57   0.15
290   349   2092.53    1.03   1.69
```

The first integer is the time at which the application arrives. The second one is the initial core to whom the application will be assigned. The third number, a real one, represents the workload the application must execute. The fourth number is the average parallelism (A) and the fifth one the parallelism variance (var) [9].

The second type of text files is the one containing the areas in which the initial and manager cores ask for cores. This is also for debugging reasons. In the proposed framework these text files include only far regions whenever the platform allows their existence. In the DistRM case [11], these text files include all the near and far regions of an initial and manager core.

4.1.7 Heterogeneous platform simulation

In the case of a heterogeneous platform simulation every core is of a specific type. This type is defined upon initialization and cannot be changed at run-time. Every information conveyed about a core includes its type.

We also have a predefined number called MF. This stands for matching factor. It defines how strict the limitations as far as the preferred processing types of an application are concerned. For example MF equal to 1 means that the application can accept only the element that suits 100% its needs. An MF of 0.5 means that the application is willing to accept an element that suits only 50% its needs. Of course this results to a penalty in the speedup of the application. The speedup of the application is calculated by the algorithm in section 3.4.

Additionally, every application has a number of acceptable processing elements and each one of them has a different priority. These characteristics are created randomly and are written in the input text files of the application. A segment from this enhanced input text file is:

```
115   33   35.67      9.54   0.46   4 2 0 3 1
150   30   2.36       18.98  1.13   2 1 0
196   20   111.92     2.28   1.17   4 3 0 2 1
225   23   53.57      8.57   0.15   1 1
```

```
290    8      2092.53       1.03    1.69    4 3 1 2 0
```

The first five numbers represent the same information as in the homogeneous case. The sixth number represents the number of acceptable processing elements for the application. The following numbers are the acceptable processing elements in descending order of preference.

4.2 Implementation of NoC simulator in C programming language

Before incorporating our framework for resource management in an actual NoC platform, it was crucial that we developed a simulator in a common x86 system. This simulator provided the following:

i. A way to get insight on the NoC and refine the framework whenever this was necessary.
ii. An easier way to code the framework since more debugging tools could be used during the development.
iii. Experimental results to test the superiority of the proposed framework compared to DistrRM [11].
iv. Experimental results validating the scalability of the framework in platform with a great amount of cores up to 1024 (32x32).
v. Experimental results simulating heterogeneous platforms.

The C programming language was chosen for the development of the simulator. This was actually the only option since C is the only programming language fully supported by Intel SCC platform. The simulator ran on top of a standard Linux operating system.

4.2.1 Implementation of nodes

The main concept is that every node runs as a distinct process. This is the closest to an actual NoC platform since the exchange of data in that way in not a trivial process. There is an initial process which is triggered when the NoC simulator executable is triggered. This process represents node 0. This node forks all the controller cores. Node 0 is a controller core itself. All the controller cores fork the nodes they are responsible for.

The knowledge as to which exactly these cores are results from the knowledge of the dimensions of the platform and the number of controller cores in each direction. In fact, a controller core knows all the other controller cores and the controller core responsible for each node. For scalability and safety reasons, this knowledge is only available to controller cores.

The only information shared among processes is the correspondence of a node number to the actual PID of the Linux process. In that way, all the calculations inside a node can be performed using node numbers and at the time a signal must be sent it is a trivial and fast task to find the correct PID of the process it is addressed to. For this memory segment, POSIX shared memory is used.

The following figure illustrates the creation of controller cores by node 0. We suppose that there are totally M controller cores on the platform.

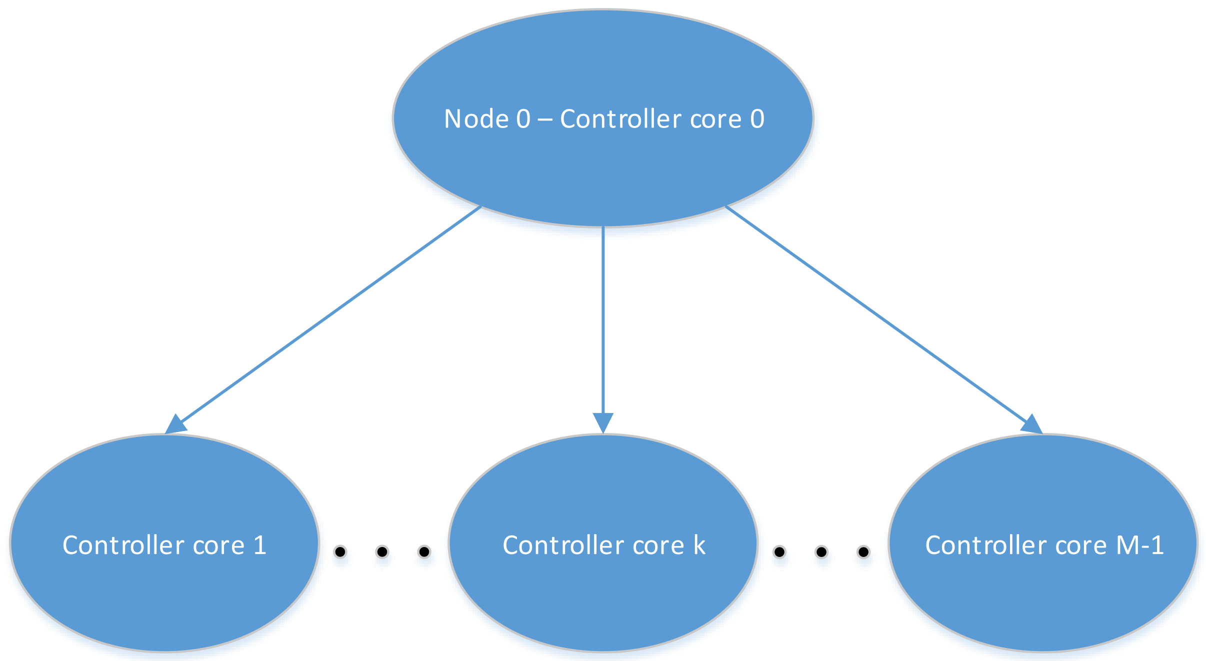


Figure 4.6: Creation of controller cores by node 0

Figure 4.7 illustrates the creation of common nodes by node 0. We suppose that node 0 is responsible as controller core for N nodes in its cluster. A cluster is rectangular in shape so Node 0 does not create all the common nodes from 1 to N-1. Instead it creates only those in its cluster. Pid_num is the array of the shared memory where PIDs are stored.

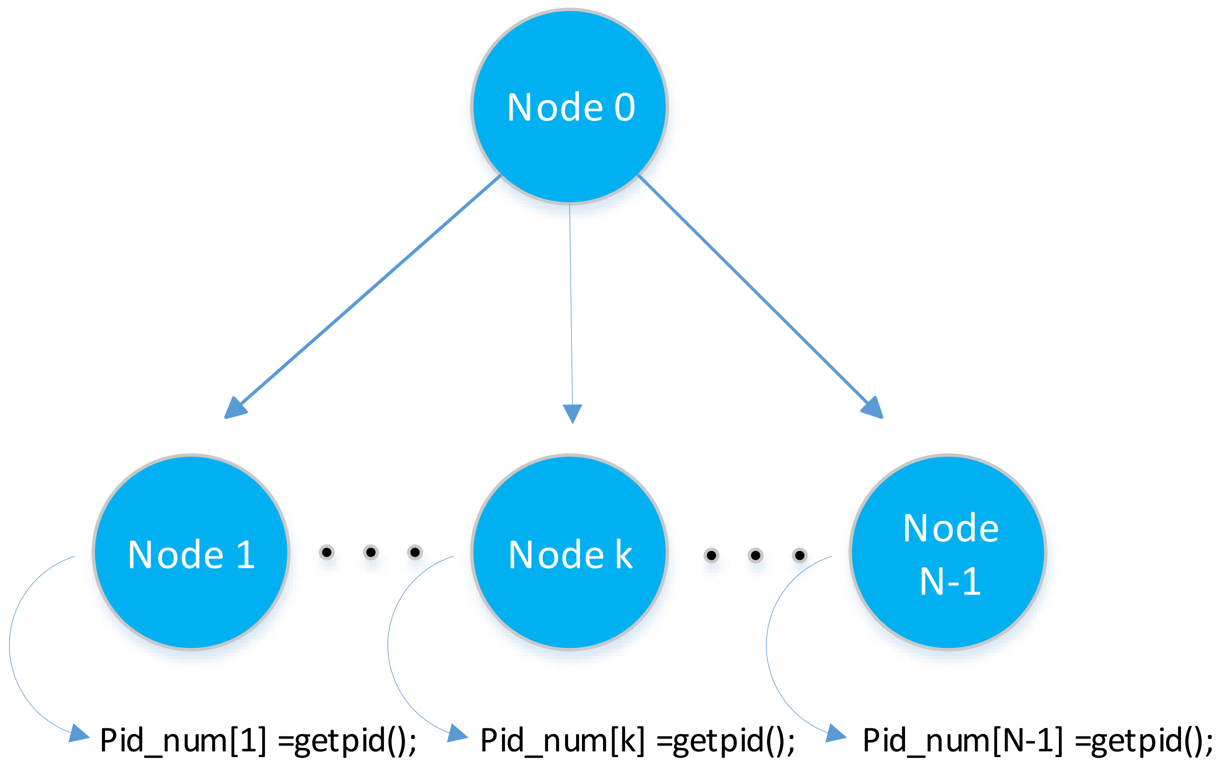


Figure 4.7: Creation of all nodes inside the cluster of controller core 0

4.2.2 Inter-node data exchange

Different nodes exchange data via a Linux named pipe. Every node possesses its own name pipe. Whenever node A wants to send data to node B, it writes data to the named pipe of node B and afterwards B reads them. The synchronization of this data exchange will be described in detail in the following sub-chapters.

4.2.3 Inter-node synchronization

For synchronization purposes POSIX semaphores are used. Every node has two of them. The reason why will be apparent when process inter-communication amongst nodes is explained. These semaphores are also stored in shared memory and the semaphores of one node can be accessed by any other node. The following figure illustrates the communication of two nodes with a named pipe illustrated as FIFO queue. The locks in the figure represent the semaphores of each node.

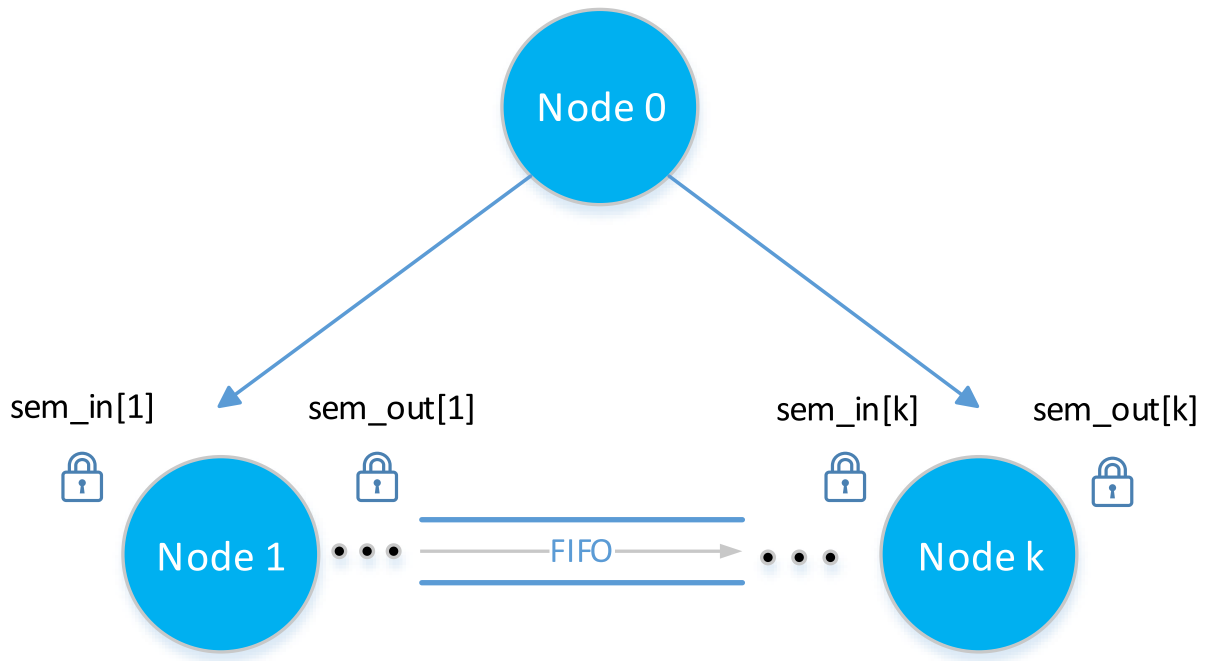


Figure 4.8: Communication of two nodes via a Linux named pipe

4.2.4 POSIX real-time signals

For inter-node exchange of signals POSIX real-time signals are used. The special characteristics of these signals compared to traditional Linux signals are:

i. The programmer can define their name leading to more comprehensible code.
ii. More than one of these signals can be queued up for a process if the first one has not been handled. In standard signals only one signal of a given type is queued and the other signals of the same type are ignored.
iii. Order of delivery of real-time signals is guaranteed to be the same as the sending order.

For the establishment of signal handling functions sigaction() was used. This type of signal delivery can accommodate some special characteristics like an extra integer accompanying the delivery of the signal but these features were not used to achieve easier transition from the C-simulator to the actual platform.

The following example illustrates how signal, semaphores and pipes are used when node A has a request to be served by node B. Node A sends the hypothetical signal SIG_X. Node B receives the signal and enters the respective signal handling function. Node B needs to receive data from A and at the moment B is ready it sends a SIG_ACK signal to A. After B has sent the signal it waits on its sem_in semaphore.

This semaphore exists to solve the following problem. At the moment B sends a SIG_ACK signal, it is highly probable that another node, for example node C, has probably sent another SIG_ACK signal to A. Both B and C would block on the opening of the pipe. Node A would handle one of the two SIG_ACK signals but at the time it opened the other end of its pipe both B and C would unblock. Thus, data read by both B and C would be corrupt. So before opening its writing end of the pipe, A increases only the semaphore of the recipient leading to its unblocking from the waiting on its sem_in semaphore.

After the above, the data exchange takes place. B performs all the necessary calculations and both A and B must exit their signal handling functions. There comes the need for the existence of the second semaphore, sem_out. Sometimes, node A closes its pipe so fast that B has not read all its data. Additionally if instead of using sem_out we used sem_in again, both A and B would come to a deadlock if A entered a signal handling function from a signal sent by B.

4.2.5 POSIX timers

POSIX timers are provided by Linux for an efficient way to have a timer per process. A timer is set and upon its expiration, a signal is sent to the respective process. This signal is handled by the appropriate signal handling function as any other common signal.

4.2.6 Internal state of nodes and text files

The internal state is a simple global variable and text files are stored in the hard drive of the system running the simulator. Figure 4.9, illustrates how timer expiration affects the internal state of a node.

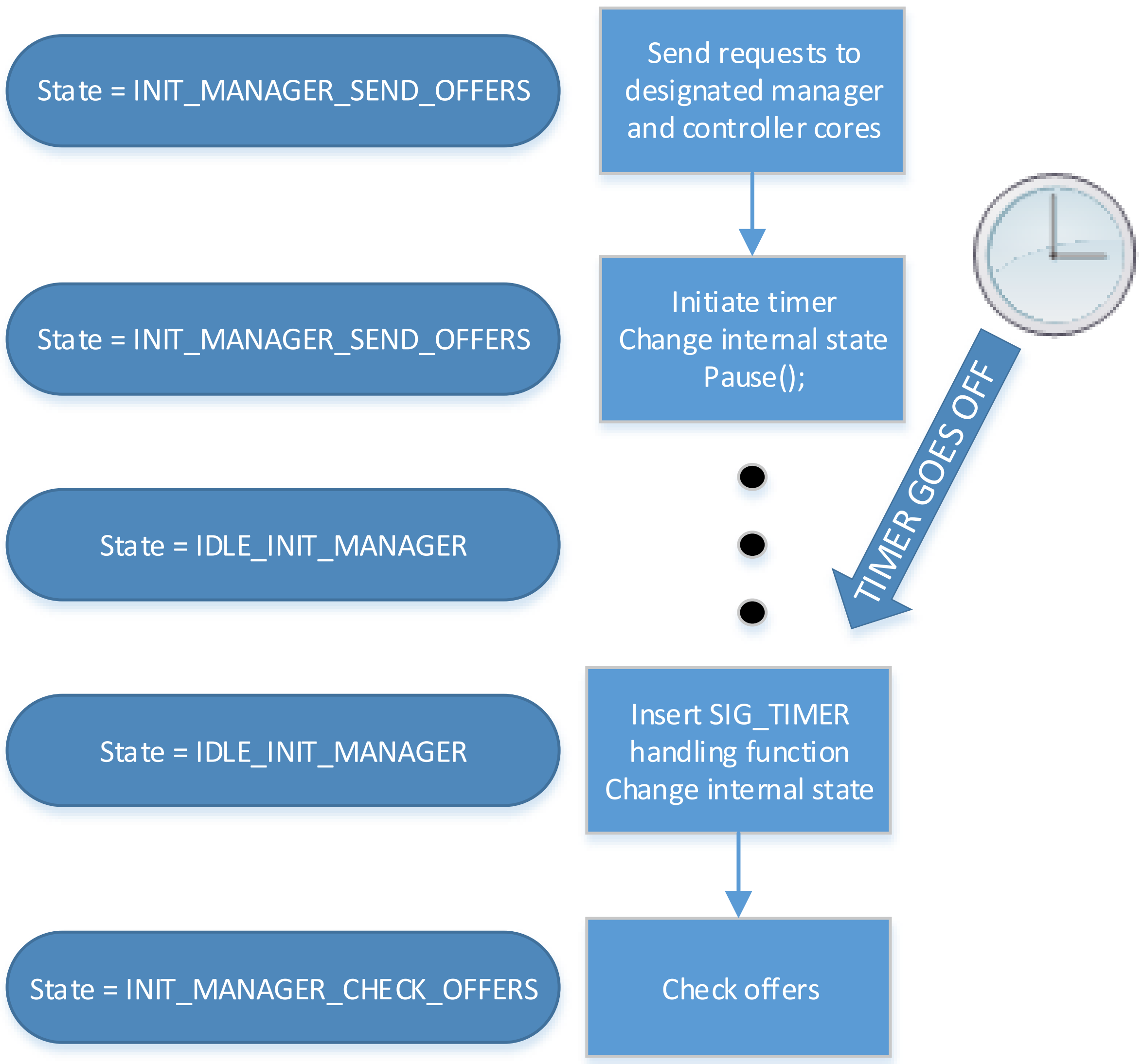

State = INIT_MANAGER_SEND_OFFERS
Send requests to designated manager and controller cores
State = INIT_MANAGER_SEND_OFFERS
Initiate timer
Change internal state
Pause();
State = IDLE_INIT_MANAGER
TIMER GOES OFF
State = IDLE_INIT_MANAGER
Insert SIG_TIMER handling function
Change internal state
State = INIT_MANAGER_CHECK_OFFERS
Check offers

Figure 4.9: Change of the internal state of a node due to a timer expiration

Figure 4.10 illustrates how the input file is processed and how it determined that a new application has arrived on the system. In node 0 a timer is set representing a time unit. Upon its expiration, a counter of the elapsed time units is increased and the input text file is checked. If the file dictates that an application has arrived at that moment, an appropriate signal is sent to initialize the designated initial core. Then the timer in node 0 is set again for the next time unit and this takes place until the input file reaches its end.

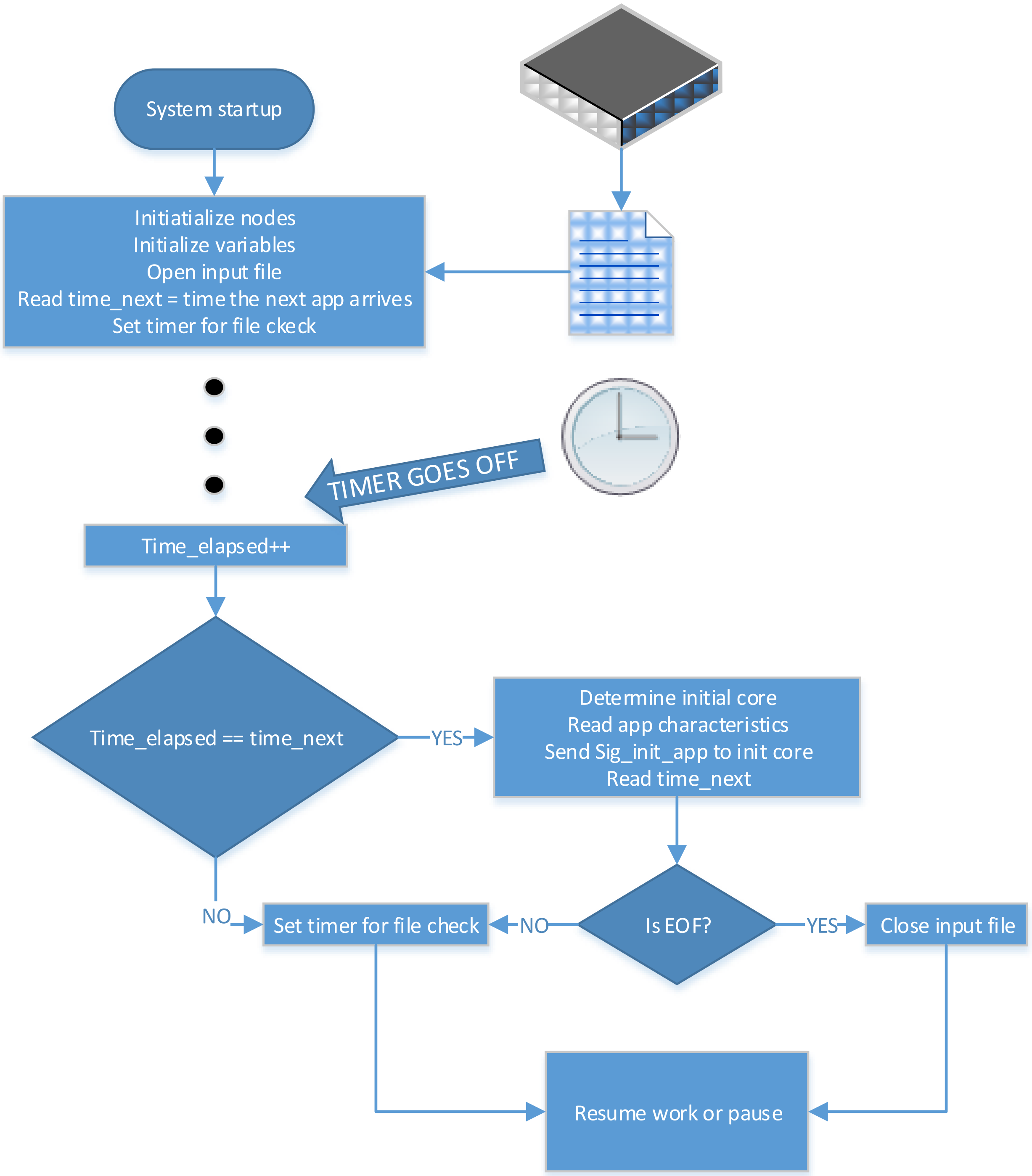


Figure 4.10: Necessary steps node 0 has to take in order to parse the input file

4.3 Implementation of the run-time service in Intel SCC platform

After our simulator was working correctly and our framework had come to its final form, we decided to test its functionality on an actual NoC platform. Intel SCC platform was an excellent choice since it possesses all the necessary programming interfaces to make the transition feasible and efficient. Additionally, the fact that on every core runs a Linux OS, allows for a great amount of the code of the framework to be imported with no changes. For this to be more apparent, we will present the equivalent of all programming elements we used for the C-simulator. The SCC platform was programmed in C programming language using the RCCE API provided by Intel [3]. More specifically the gory interface of the API was used.

4.3.1 Implementation of nodes

Apparently there is no need for any special programming structure for this transition. The platform itself appoints a node id to each node. We use the provided RCCE program rccerun[4] to distribute the final executable to all nodes.

4.3.2 Inter-node data exchange

As stated in chapter 1.3 Intel has provided the Message Passing Buffer for the exchange of data amongst nodes. The RCCE API makes the data exchange even easier. The fact that we can write to this buffer only in multiples of cache line, provides us with enough space to convey all the necessary information in a fast and efficient way.

We create in the MPB of **each** node an array of integers in which data will be stored in every data transaction. Since at one given moment a node can receive data from only one other node and these data are immediately processed, we do not need very big space in memory. The actual size of this array is 96 bytes (3 * cache line size * sizeof(integer)) and it is enough for all necessary actions of both the proposed framework and DistRM[11]. Figure 4.11 illustrates this array of integers reserved for data exchange in a random node. The same scheme applies for all nodes.

4.3.3 Inter-node synchronization

As stated in chapter 1.3 RCCE API provides flags for synchronization purposes. These flags were useful for the construction of an overall synchronization scheme. However, since we have a real time application with a high amount of data exchanged, these flags proved to be inefficient for all our synchronization needs.

This stems from the face that the flag is also stored inside the message passing buffer thus it is subjected to the same rules of any memory operation as far as its completion is concerned. Thus, we needed explicit synchronization techniques. We introduced delay in certain points of the code and after some tweaking all the synchronization requirements were met. The delays were a simple busy looping since we did not want anything more complicated. Their use will be apparent in the following section.

4.3.4 Inter-node signal exchange

This was the most challenging part of the transition in the SCC platform. The problem is the following. Even if every node runs its own Linux OS there is no way provided by the RCCE API to send a Linux signal to the Linux system located on a different core. As an equivalent in a computer network, it would be like trying to send a signal among two different computers belonging to the same network.

Our proposed solution arises from the fact that two different nodes can have only one interaction at a given moment. In addition, a signal is nothing more than an integer that upon its arrival a signal handling function is called. Taking all these into consideration, we create an array of integers in the MPB of **each** node called a signal array. The size of the array ideally would be equal to the number of the nodes. The k-th element of this array represents a signal sent by node k. As a result, when node A wants to signal node B, it writes an appropriate integer to the A-th position of the signal array of B. The following figure illustrates how we partition the MPB memory space of a node in order to facilitate our need for inter-node signaling and data exchange. The figure illustrates the memory space of node 2 but the same scheme applies for all nodes of the platform.

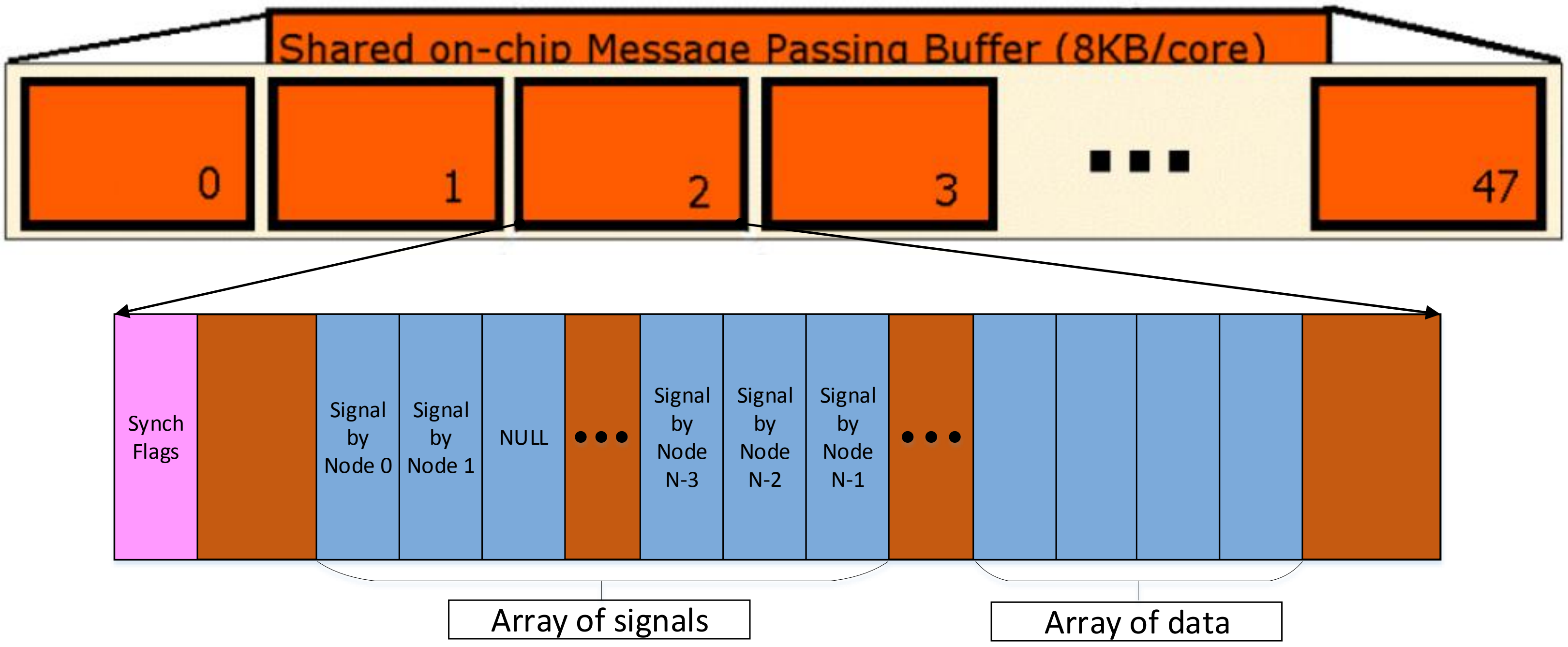


Figure 4.11: Partitioning of the MPB memory space of a node to achieve inter-node signaling and data exchange capabilities

For synchronization purposes we use a flag called flag_signals_enabled. Every node has its own flag. Every time this flag is set, the node is ready to accept signals. Periodically it unsets this flag and checks for signals. This is more comprehensible with an example. We suppose that A wants to send a signal SIG_TYPE_X to B. The process is depicted in the following figure:

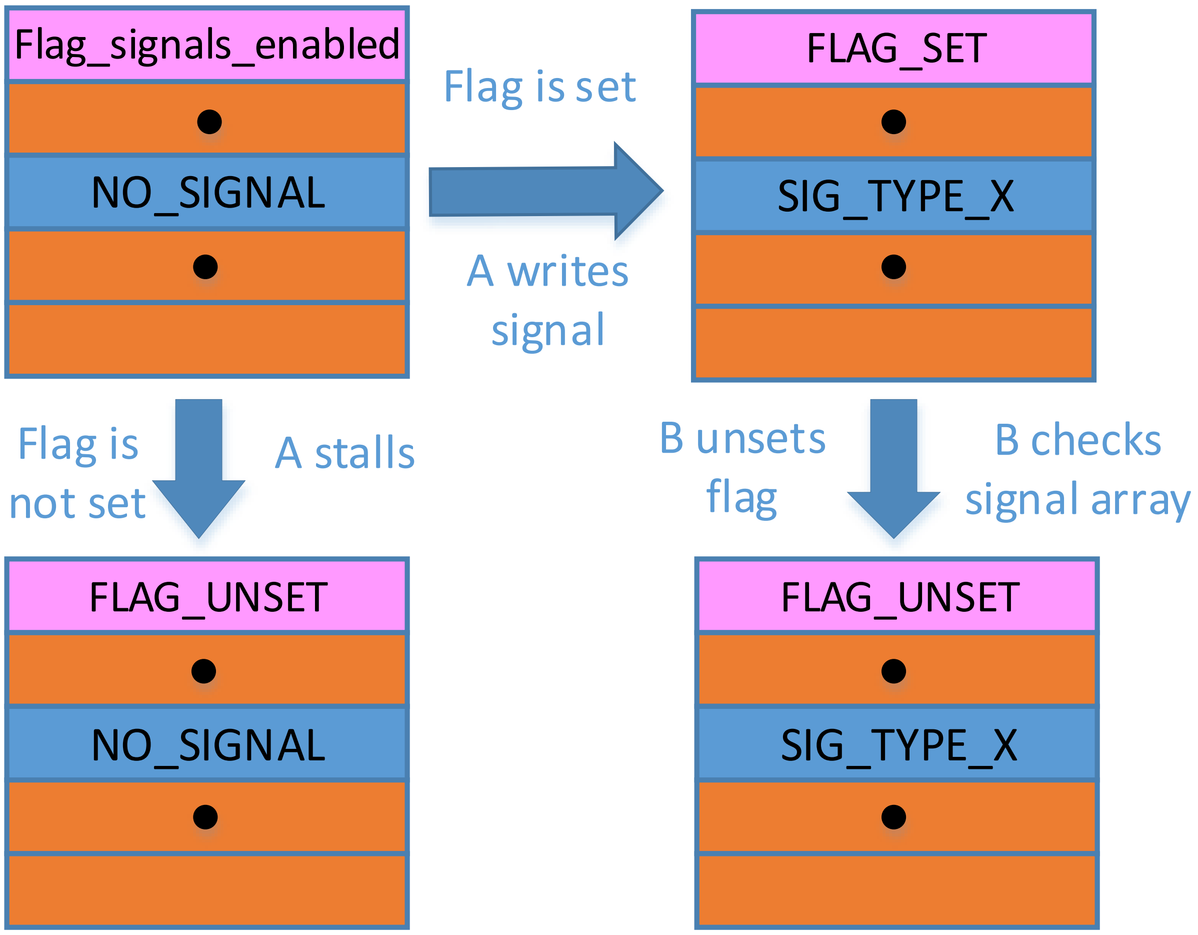


Figure 4.12: Process of node A sending a SIG_TYPE_X signal to node B

First A checks the flag_signals_enabled flag of B. If it is not set, A stalls until B sets the flag. If it is set A writes the integer SIG_TYPE_X to signal_array[A] of B. After some time B unsets its flag and checks its array. When he reads the integer written by A he sets his flag once again in order to be able to receive signals while he is in the signal handling function. Then he triggers the appropriate signal handling function of SIG_TYPE_X. After the completion of this function, he restores signal_array[A] to NO_SIGNAL.

The design proved to be successful. There are only two complications. The first is that there is no possibility of writing only one integer in the MPB memory space. Every read or write is performed in multiples of cache line size. Thus, the size of the signal array is multiplied by the cache line size which is 8 integers long. However, this extra space can accommodate some special features of the implementation, like special cases where an extra signal has to be sent on top of an already sent one.

The second and most important complication is the one of synchronization. The flag we use provides an initial barrier but is not always adequate since it is a part of memory as well. A scenario where the synchronization fails is the following. Suppose that at time t_0 the flag is unset. A little time earlier from t_0 some node has read the flag as set and has commenced a memory operation to send a signal. If the node checking the signal array starts reading from memory immediately after unsetting the flag, there is a high probability that corrupt data will be read. As a consequence, we introduce a delay before checking the signal array. In that way, all memory operations have enough time to finish and no corrupt data are read. The amount of the delay is small and has been designated by experimentation. Figure 4.13 illustrates this whole process.

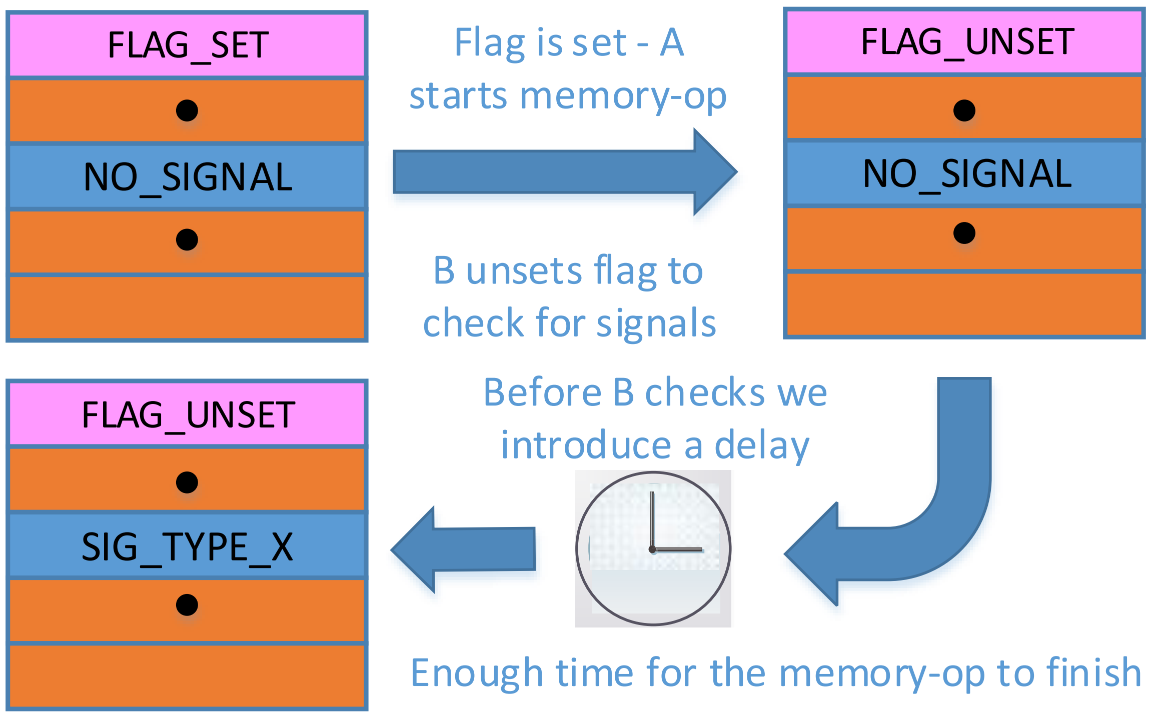

FLAG_SET
●
NO_SIGNAL
●
Flag is set - A starts memory-op
B unsets flag to check for signals
FLAG_UNSET
●
NO_SIGNAL
●
Before B checks we introduce a delay
FLAG_UNSET
●
SIG_TYPE_X
●
Enough time for the memory-op to finish

Figure 4.13: Delay before reading signal_array until all memory operations on it are complete

4.3.5 POSIX timers

Since every node runs its own Linux system, POSIX timers are available. A timer upon expiration sends a signal to the process it belongs to. Consequently, there is no need for any change to the code written for the proposed framework in the C-simulator since the signal is sent inside the same Linux system.

4.3.6 Internal state of nodes and text files

Internal state as a global variable of the C programming language functions exactly the same way in the SCC platform as in the C-simulator. As far as text files are concerned, as stated in chapter 1.3 all nodes can access a folder of the hard disk of the Management Console (MCPC) of the platform. Due to that, we store all the input files in a folder of that system and we process the input files exactly in the same way as described in the C-simulator.

4.4 Implementation of DistRM [11]

The implementation of the DistRM resource manager [11] used exactly the same principles described for the proposed framework. In fact the backbone of both frameworks is exactly the same. Their main difference is the areas in which they search for cores and the way they handle self-optimization. Both of these aspects are directly connected to the algorithm and not the implementation.

Chapter 5: Experimental results

5.1 Evaluation methodology

Our intention was for our evaluation methodology to resemble a real-life scenario as much as possible. Thus, we wanted to test both our simulator and the actual platform in a way that would provide the ability to take into account various aspects of both the input and the specifications of the platform either the real one or the one under simulation.

The basic idea of our evaluation strategy both for homogeneous and heterogeneous platforms, is that a number of applications enter our system at random times and the workload of each one of them has to be fully executed before it is perceived as complete and exits the system. If and only if all applications enter and exit the system is a test case considered successful. The input is directed to the system as described in section 4.1.6 and derived from the application generator provided by [10].

The simulation was performed for a variety of platform sizes varying from 6x6 up to 32x32. Each test case dealt with a number of applications. Our proposed framework is compared to a state-of-the-art runtime resource management framework called DistRM, described in section 2.6. The number of applications varied from 8 up to 64. However, experimentation showed that a number of applications less than 32 are too small to produce significant conclusions in the case of the NoC simulator. On the contrary, in SCC platform even small number of applications is taken into account to provide an insight on how both frameworks scale their resource management in reality.

A lot of metrics could be used to evaluate the two frameworks. We concluded that the best of them are four:

i. **Message count**. This is the **total** number of message exchanged by all nodes throughout a complete test case scenario. These messages are needed for resource allocation, self-optimization and application execution.

ii. **Message size**. This is the total amount of data, measured in bytes, exchanged by all nodes throughout a complete test case scenario.

iii. **Average speedup**. This is a metric introduced by [11]. It results from the following type:

$$Average\ Speedup = \frac{Sum\ of\ workload\ of\ all\ applications}{Sum\ of\ turnaround\ time\ for\ execution\ of\ all\ applications}$$

This metric has an upper bound in each test case because there is a maximum speedup every application can achieve. This upper bound for the metric could be achieved if we had infinite resources to provide to all applications. This of course is impossible. However, greater results of this metric are directly related to better resource allocation thus providing greater speedup values to all applications.

iv. **Computational effort.** The computational effort for each node is the number of times it had to initiate the algorithm for offering cores. The computational effort as a metric is the sum of the computational effort of all individual nodes. It is denoted as computational effort since the greater the number for this metric, the more calculations the nodes had to perform in total for resource allocation purposes.

5.2 Results in simulator for homogeneous platforms

Simulations were executed on a standard Linux system with a x86 Intel Quad Core processor and 2 Gb of available RAM memory. Every simulation was executed five times and the mean value of the measurements was kept as the final measured value. It is important to point out that the application characteristics of the input and their workload is identical for every test case, for every possible grid size. What are different amongst the test cases of **different grid sizes are the randomly chosen initial cores**. This is very important to help us explain the measured values. However, **the randomly initial cores are the same for the test cases of both resource management frameworks**.

Table 5.1 presents the measured values for message count for all applications and all grid sizes. The column "differ." is the difference of the proposed framework compared to DistRM:

Message Count									
	32 applications			48 applications			64 applications		
	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.
6x6	2473	8390	72.3%	3954	12799	69.1%	5302	18517	71.7%
8x8	2692	7534	64.3%	4065	10863	62.6%	5463	14916	63.4%
12x12	5351	10517	49.1%	7760	15658	50.4%	10092	21145	52.3%
16x16	5840	10178	42.7	8389	15214	44.9%	10926	20166	45.8%
20x20	6266	9301	32.6%	9099	13862	34.4%	11737	18360	36.1%
24x24	7196	9609	25.1%	10329	13706	24.6%	13304	18267	27.2%
28x28	7447	9379	20.6%	10775	13531	20.7%	13823	17783	22.3%
32x32	7975	9718	17.9%	11638	13533	14%	14985	17545	14.6%

Table 5.1: Message count of the compared resource management frameworks for all grid sizes

Figures 4.1 and 4.2 illustrate the measured values:

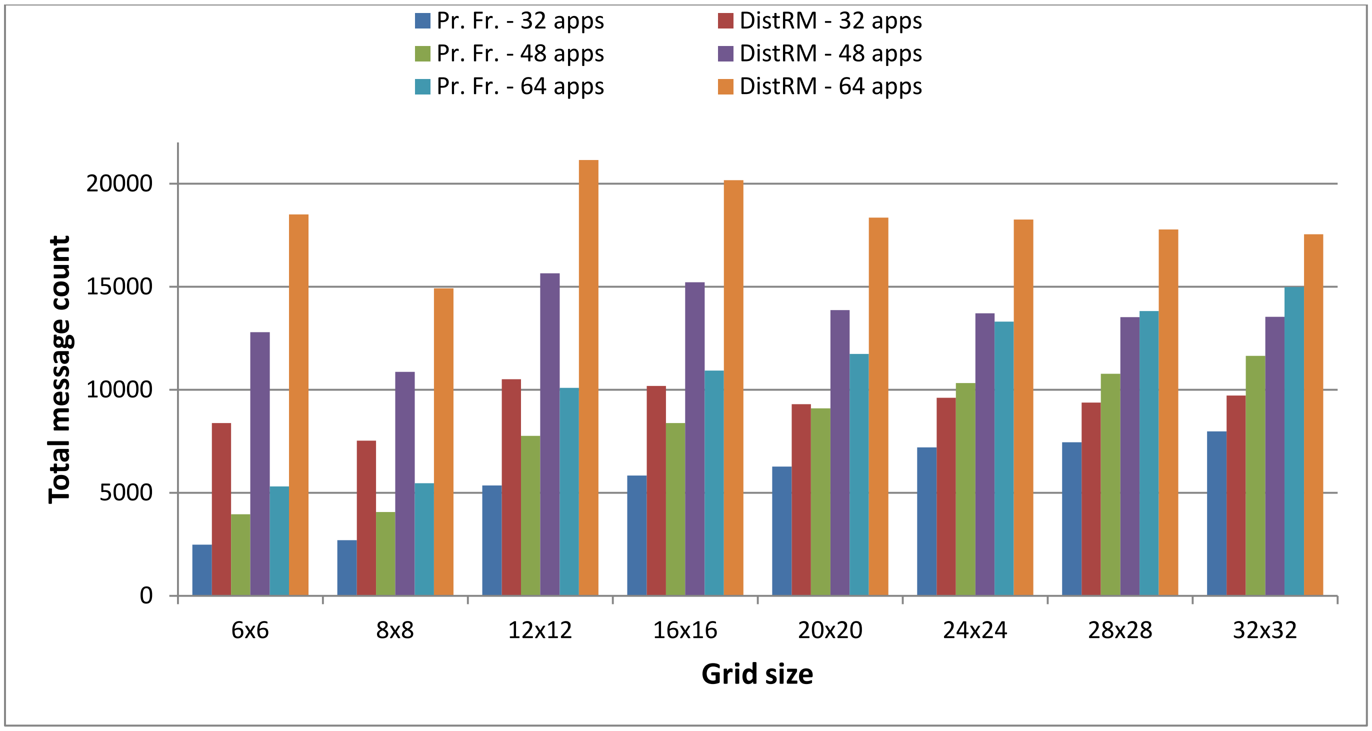


Figure 5.1: Message count of the compared resource management frameworks for all grid sizes

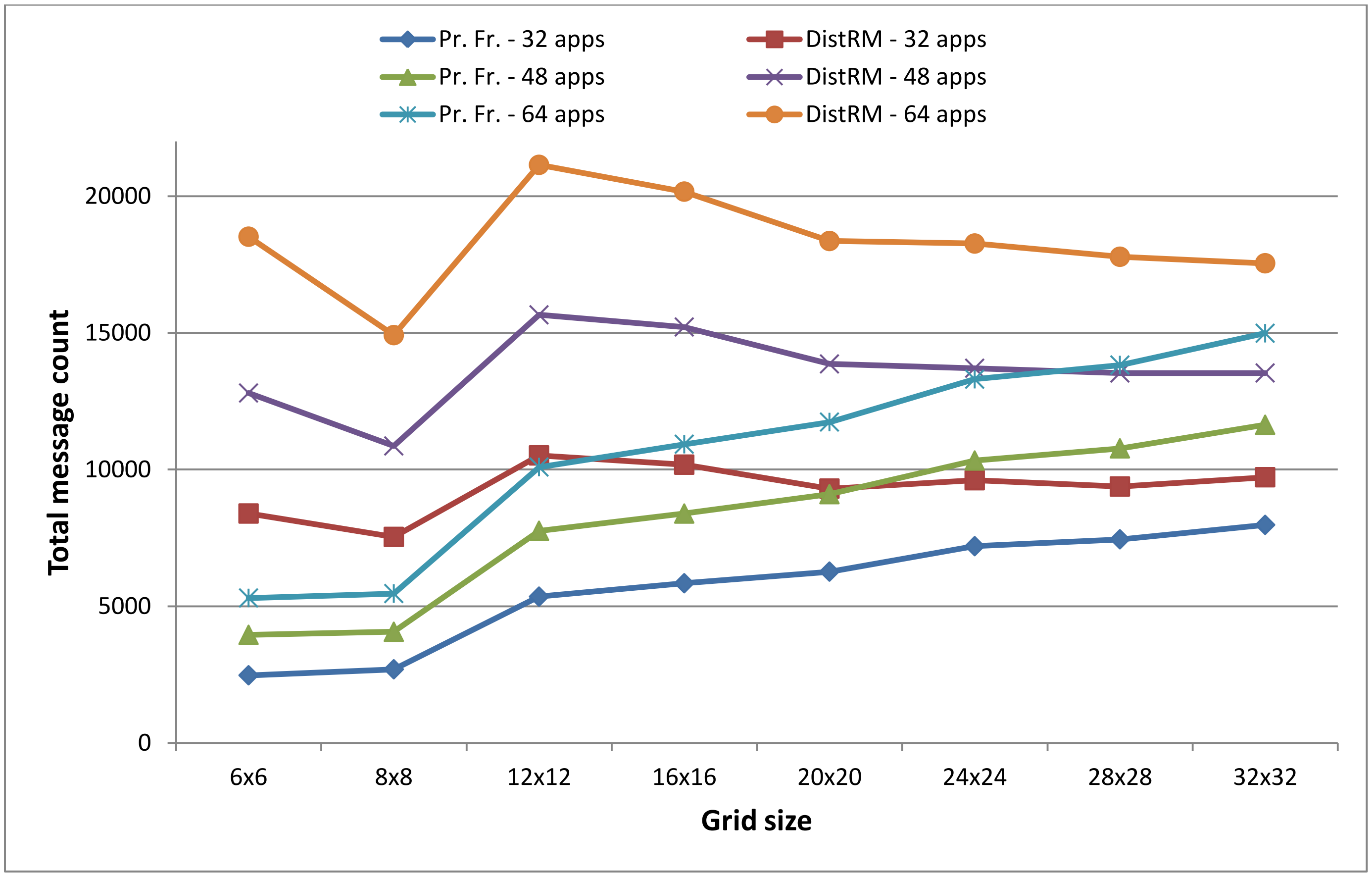


Figure 5.2: Message count of the compared resource management frameworks for all grid sizes

We can see that as far as message count is concerned our proposed framework uses much less messages compared to DistRM framework. This difference ascends up to 70% in small grid sizes. The explanation of this is simple considering the different way the two frameworks choose the areas in which they will search for cores. DistRM asks for cores in many, small areas while our approach uses one big area around the requesting core. This accounts for the extremely big difference in messages in small grid sizes, since the areas that DistRM searches in are with high probability overlapping, leading to searching cores in almost the same area multiple times. In greater grid sizes, the problem of overlapping areas is not so crucial. However, our approach initiates a self-optimization process under certain circumstances. This helps the reduction of exchanges messages in all grid sizes.

An interesting observation concerning the two approaches is the trend line of each one for the same number of applications but for different sizes of the platform. Our framework has an ever increasing number of messages as the size of the platform increases. This is to be expected since, the size of region R increases, applications have more working cores to communicate with and initial and manager cores have more respective cores to negotiate cores with. This is will be clearer when we examine the relationship between exchanged messages and average application speedup. On the contrary, the number of messages exchanged by the DistrRM framework is not proportional to platform size. For example, the messages exchanged for a 12x12 grid are much more compared to those of the 8x8 grid. This is because the DistRM approach is highly susceptible to the position of the agents that initiate the applications. If by chance they happen to be closely together, they will probably fight for the same resources leading to a steep rise in the number of conveyed messages.

Table 5.2 presents the measured values in bytes for message size for all applications and all grid sizes. The column "differ." is the difference of the proposed framework compared to DistRM:

Message size									
	32 applications			48 applications			64 applications		
	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.
6x6	17343	64657	73.2%	27684	92290	70%	37036	134068	72.4%
8x8	19516	55515	64.8%	29718	79893	62.8%	39780	109224	63.6%
12x12	42069	76167	44.8%	61458	114336	46.2%	79393	154757	48.7%
16x16	45590	74708	39%	66907	113103	40.8%	87916	150118	41.4%
20x20	48458	67021	27.7%	72444	102144	29.1%	94788	136416	30.5%
24x24	55516	68100	18.5%	83008	100088	17.1%	109104	135020	19.2%
28x28	54483	62975	13.5%	83588	95997	12.9%	110240	128276	14.1%
32x32	56877	63142	9.9%	90107	92636	2.7%	120196	123223	2.5%

Table 5.2: Message size in bytes of all the conveyed messages of the compared resource management frameworks for all grid sizes

Figures 5.3 and 5.4 illustrate the measured values:

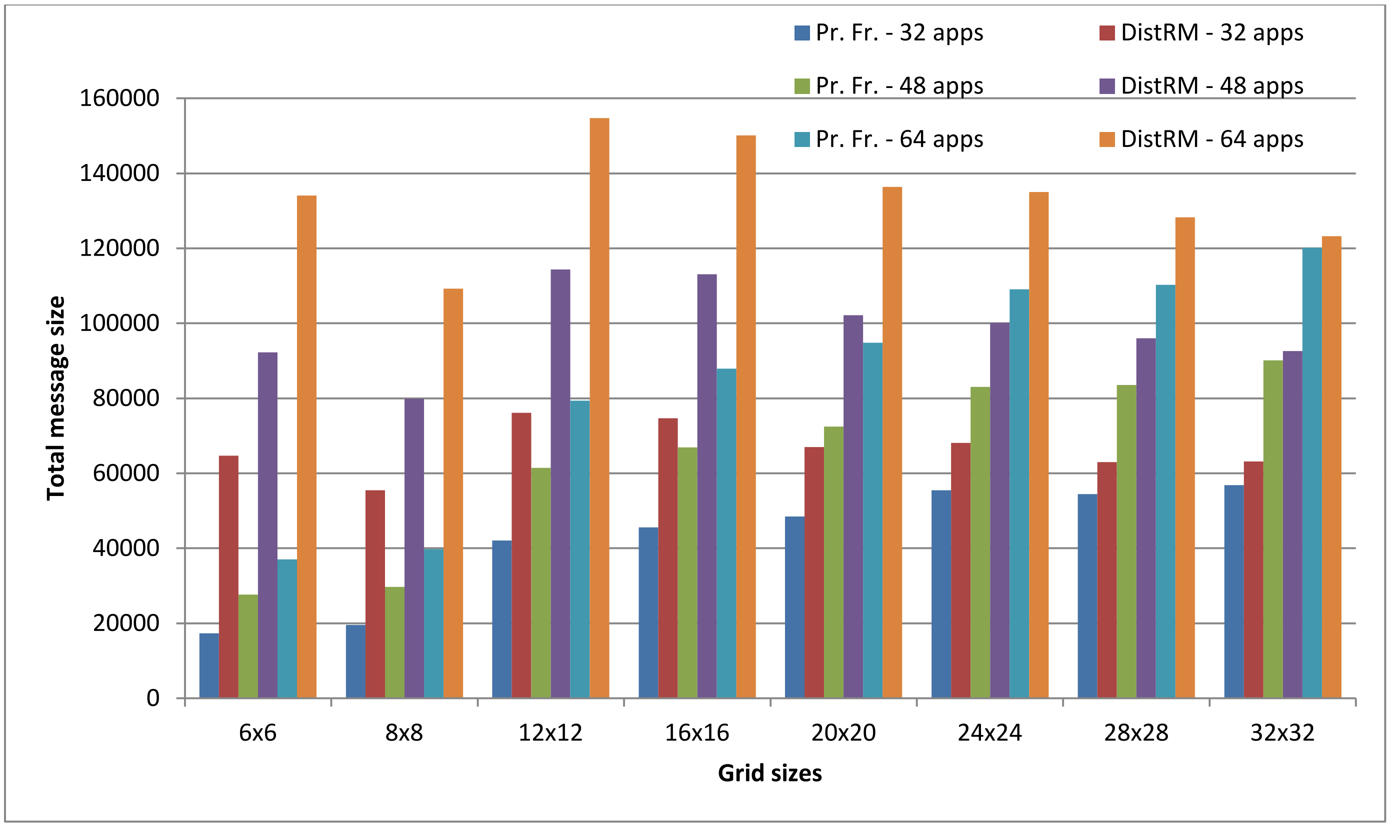


Figure 5.3: Message size in bytes of all the conveyed messages for all grid sizes

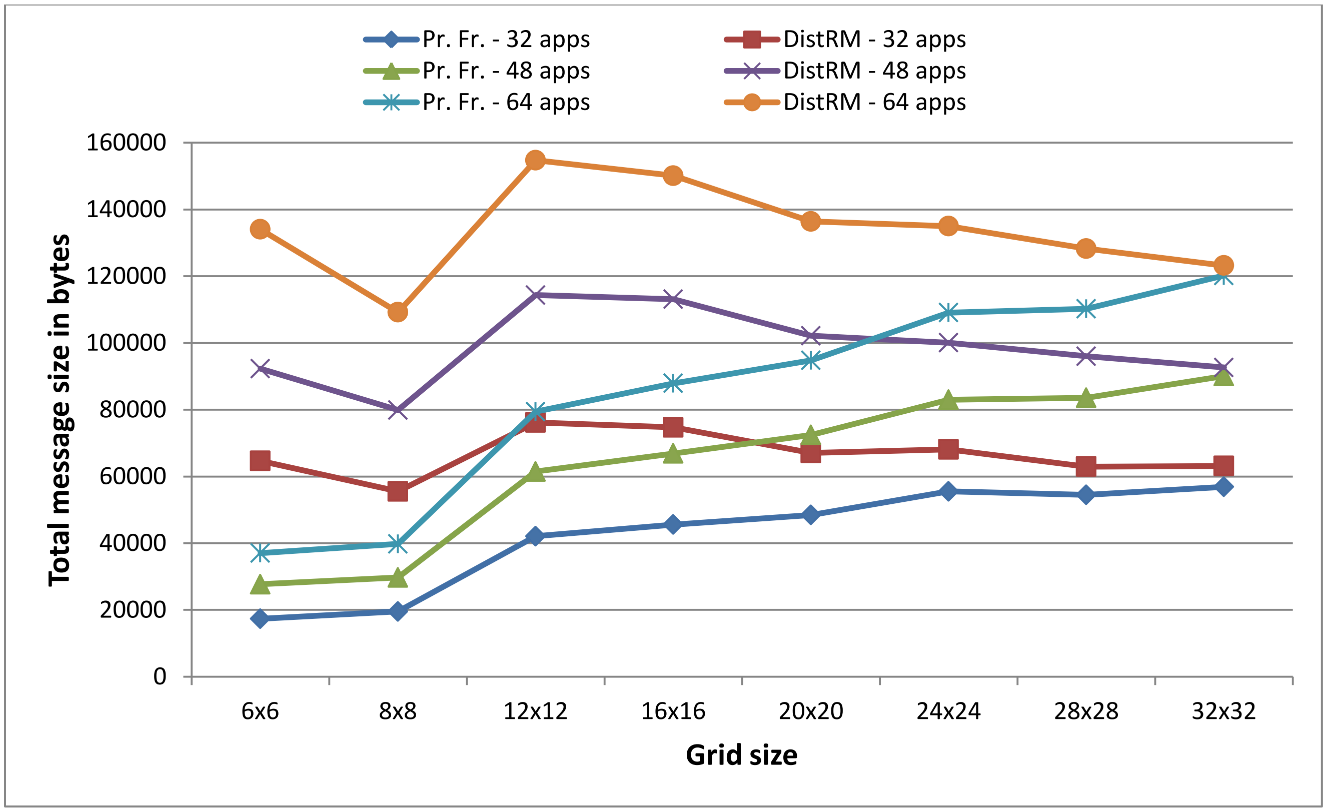


Figure 5.4: Message size in bytes of all the conveyed messages for all grid sizes

We can see that the total size of the exchanged messages is directly related to the number of messages, as it was expected. Even the trend lines of figure 5.2 and figure 5.4 are almost the same. However message count and message size are not directly proportional since a message may vary in size. For example, a message may involve an integer, ergo 4 bytes or a floating point number ergo 8 bytes.

Table 5.3 presents the measured values for average speedup for all applications and all grid sizes. The column "differ." is the difference of the proposed framework compared to DistRM:

Average Speedup									
	32 applications			48 applications			64 applications		
	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.
6x6	3.47	3.34	3.7%	4.97	4.78	3.8%	4.85	4.18	13.8%
8x8	3.71	3.56	4%	5.6	5.22	6.8%	5.31	4.82	9.2%
12x12	3.57	3.63	-1.7%	5.19	5.13	1.2%	5.01	4.93	1.6%
16x16	3.78	3.77	0.3%	5.74	5.52	3.8%	5.48	5.31	3.1%
20x20	3.9	3.78	3.1%	6.07	5.77	4.9%	5.67	5.52	2.6%
24x24	3.89	3.81	2.1%	6.22	5.95	4.3%	5.75	5.61	2.4%
28x28	3.89	3.83	1.5%	6.28	5.95	5.3%	5.78	5.62	2.8%
32x32	3.9	3.83	1.8%	6.3	6.02	4.4%	5.79	5.63	2.8%

Table 5.3: Average application Speedup achieved by the compared frameworks for all grid sizes

Figures 5.5 and 5.6 illustrate the measured values:

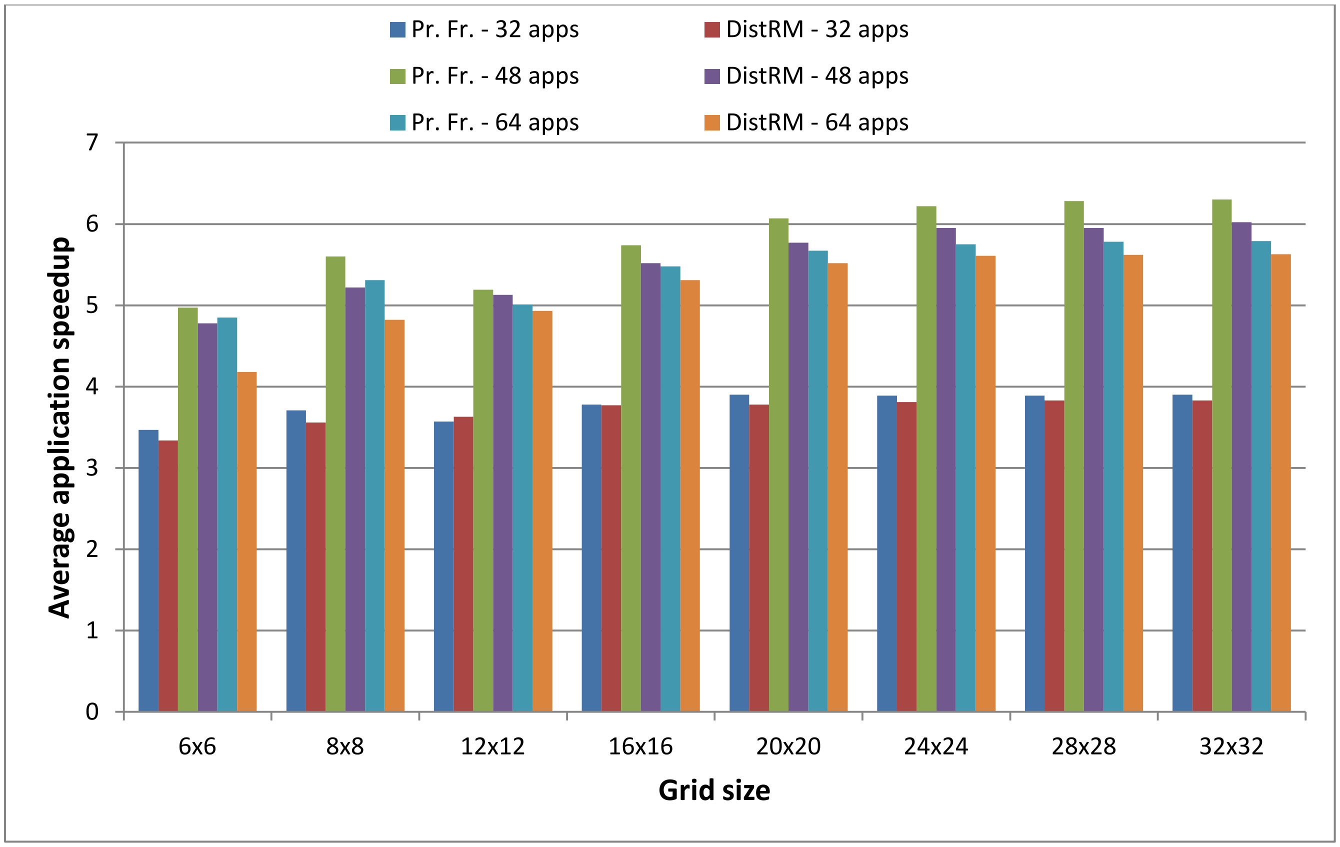


Figure 5.5: Average application Speedup achieved by the compared frameworks for all grid sizes

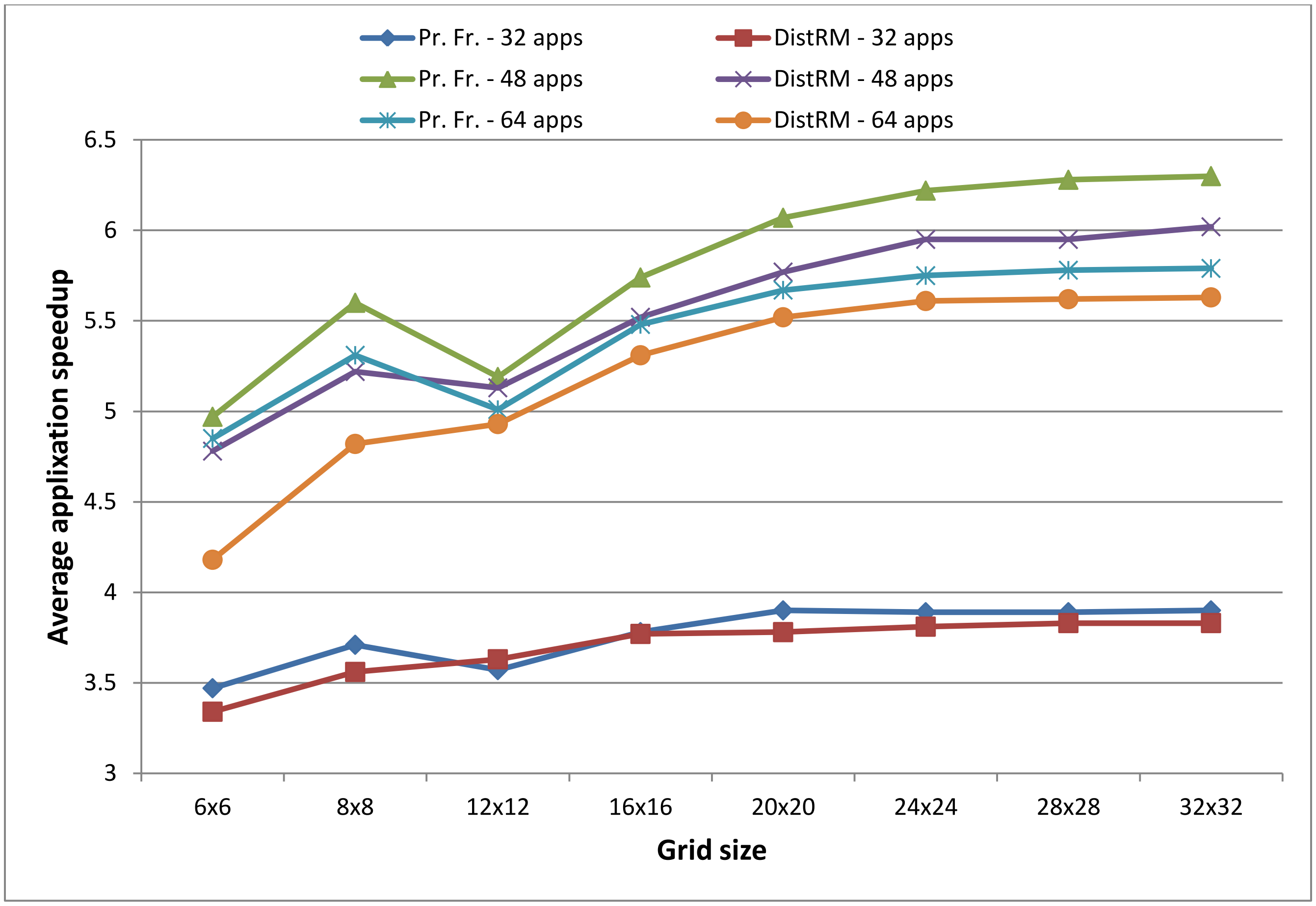


Figure 5.5: Average application Speedup achieved by the compared frameworks for all grid sizes

Both frameworks produce results that are close in value. Our approach produces 3.7% greater application speedup in average. This is once again attributed to the better choice of regions for searching cores and especially the first cores of a newly arrived application. Our approach covers a wider region thus distributing more effectively the resources. However, in contrast to the trend line in message count, here both approaches are affected by the position of initial cores, leading to our approach providing better speedup to the application for an 8x8 grid size compared to that of 12x12. In platform sizes bigger than 12x12, speedup is an increasing function of grid size for both approaches. As expected, the price paid for this fact is an increase in the number of exchanged messages.

Table 5.4 presents the measured values for computational effort for all applications and all grid sizes. The column “differ.” is the difference of the proposed framework compared to DistRM:

Computational effort									
	32 applications			48 applications			64 applications		
	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.
6x6	154	1164	86.8%	247	1696	85.4%	329	2464	86.6%
8x8	150	899	83.3%	225	1273	82.3%	299	1761	83%
12x12	394	1159	66%	569	1739	67.3%	741	2340	68.3%
16x16	397	1027	61.3%	565	1550	63.5%	737	2101	64.9%
20x20	388	832	53.4%	551	1259	56.2%	718	1702	57.8%
24x24	409	820	50.1%	577	1154	50%	760	1575	51.7%
28x28	402	702	42.7%	551	1049	47.5%	731	1426	48.7%
32x32	389	669	41.9%	555	966	42.5%	745	1294	42.4%

Table 5.4: Computational effort that the compared frameworks used for all grid sizes

Figures 5.7 and 5.8 illustrate the measured values:

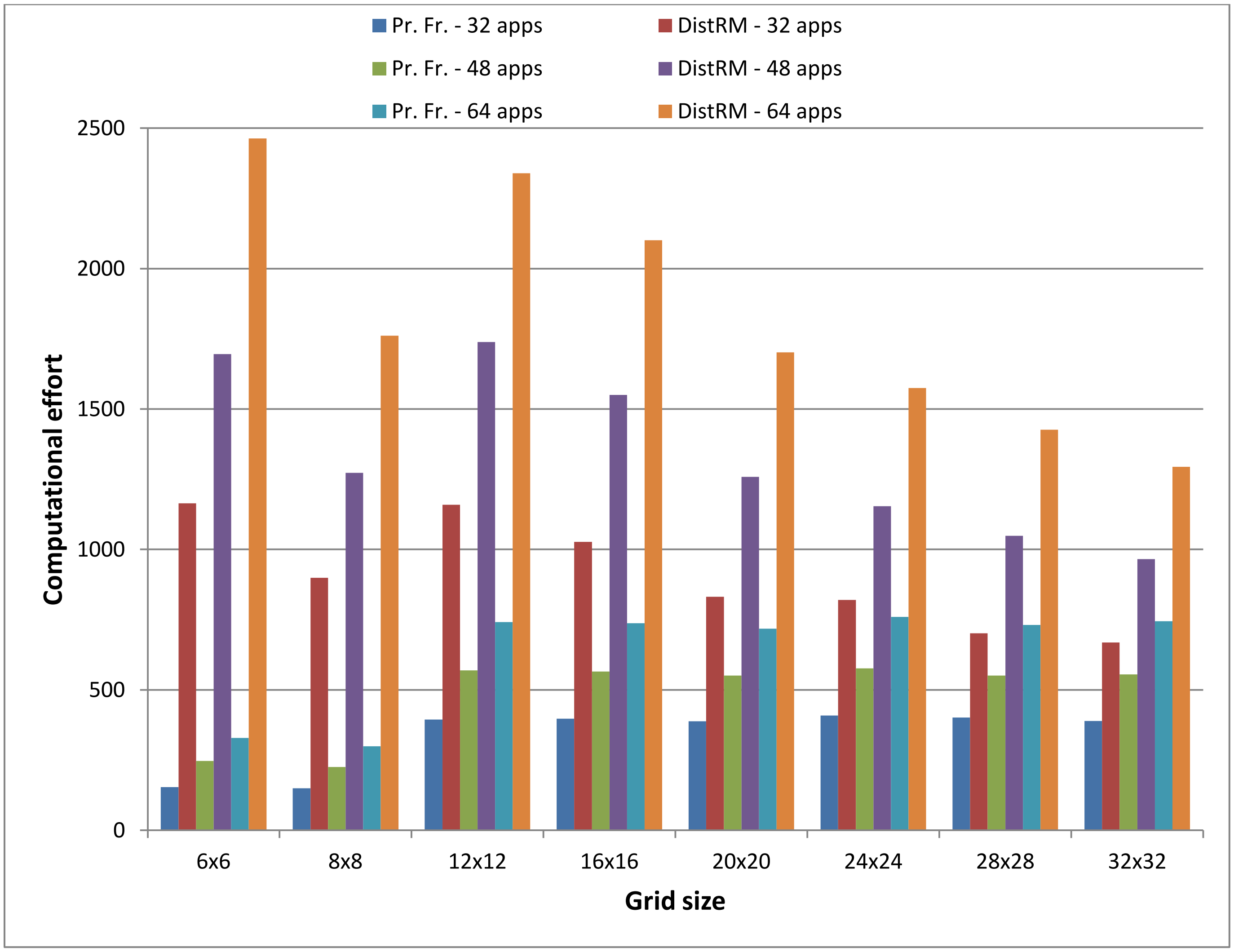


Figure 5.7: Computational effort that the compared frameworks used for all grid sizes

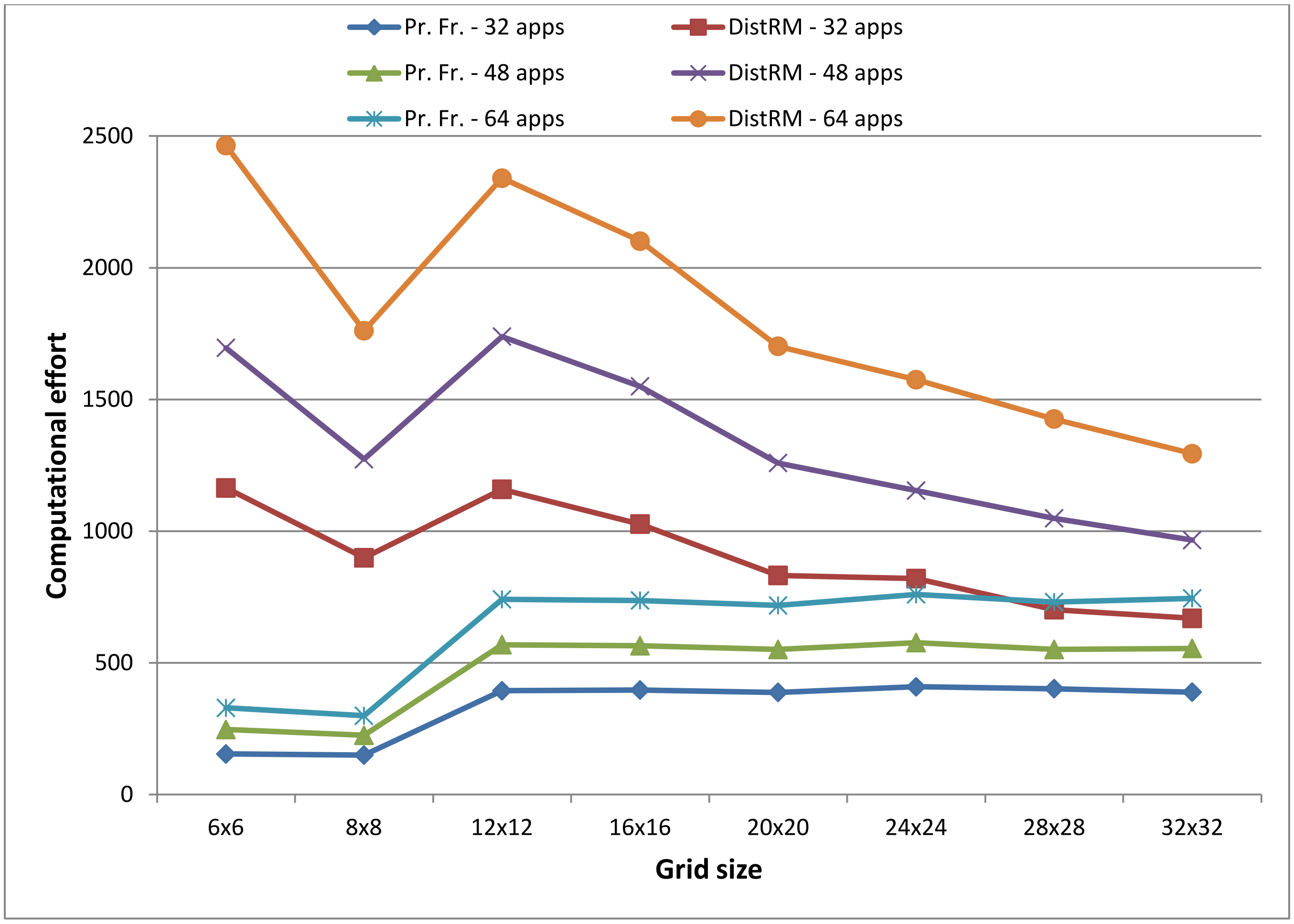


Figure 5.8: Computational effort that the compared frameworks used for all grid sizes

As far as computational effort is concerned, our framework is much more computationally inexpensive compared to DistRM framework. The reason for this is twofold. First, due to small areas of search, an agent will be asked for cores for a lot of areas thus initiating the algorithm for each one of them. Secondly, many self-optimization processes are not initiated in our proposed framework thus leading to less use of computational resources of other cores. In overall, DistRM asks many times for cores and a great number of these requests result in negative offers. As a consequence, our framework produces the same or slightly better resource management by using far less computational resources.

5.3 Results in Intel SCC platform

The evaluation methodology uses in SCC platform is identical to one used in the C-simulator. Considering that we have only 48 cores available, we work in a 6x6 grid of processors using only 36 of them.

Table 5.5 presents the measured values for all metrics for all number of applications. The column "Differ." is the difference of the proposed framework compared to DistRM:

	Message count			Message size			Average speedup			Computational effort		
	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.	Pr. Fr.	DistRM	Differ.
8 apps	671	2337	71.3%	4500	15615	71.2%	2.76	2.39	13.5%	41	289	85.8%
16 apps	1220	3797	67.9%	8512	25409	66.5%	2.03	1.87	7.6%	77	463	83.4%
24 apps	1728	5555	68.9%	12168	37146	67.2%	2.55	2.3	9.8%	108	684	84.2%
32 apps	2290	7212	68.2%	16336	47778	65.8%	3.47	2.92	16%	136	899	84.9%
48 apps	3483	10662	67.3%	25093	70553	64.4%	5.03	3.86	23.3%	202	1330	84.8%
64 apps	4635	14743	68.6%	33442	96880	65.5%	4.59	2.72	40.8%	272	1861	85.4%

Table 5.5: Measured values for all metrics for all number of applications

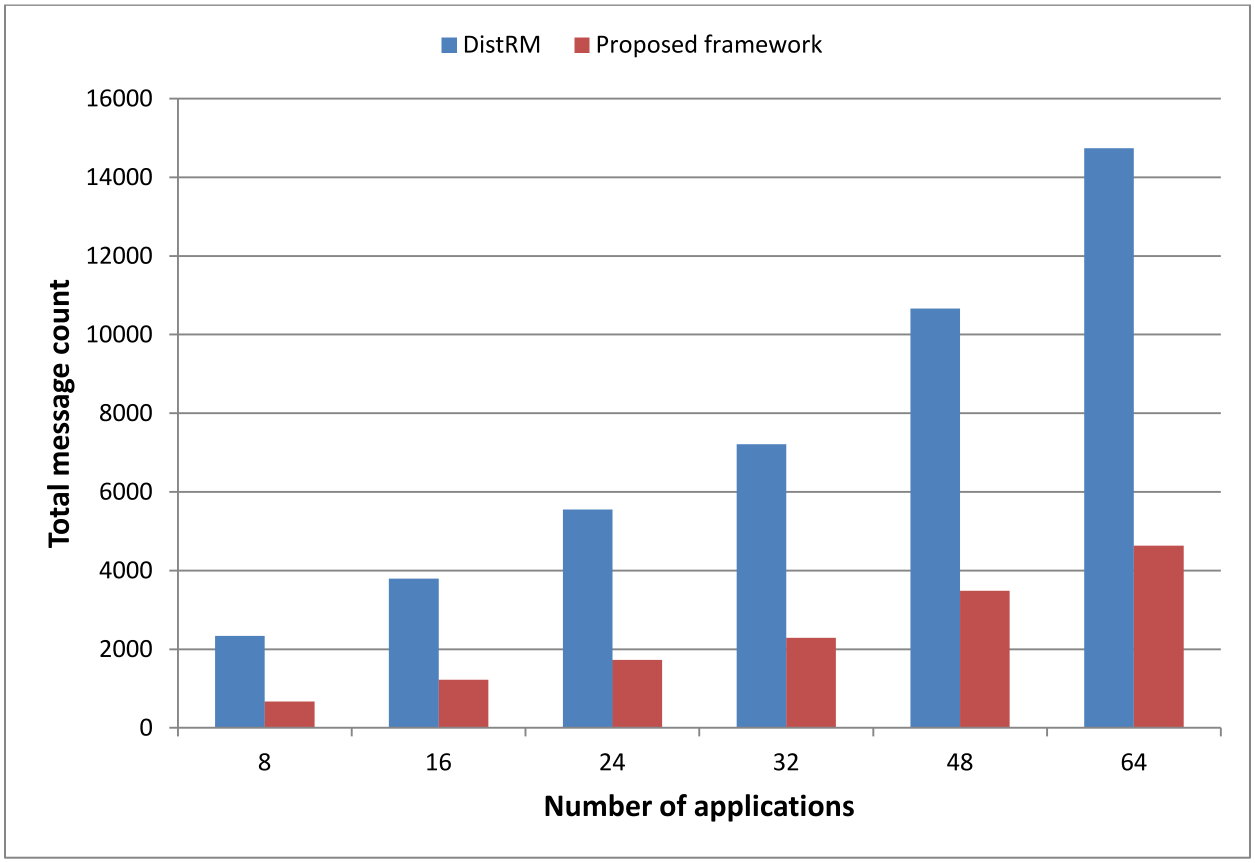


Figure 5.9: Total message count needed by the two resource management frameworks for all number of applications on the SCC platform

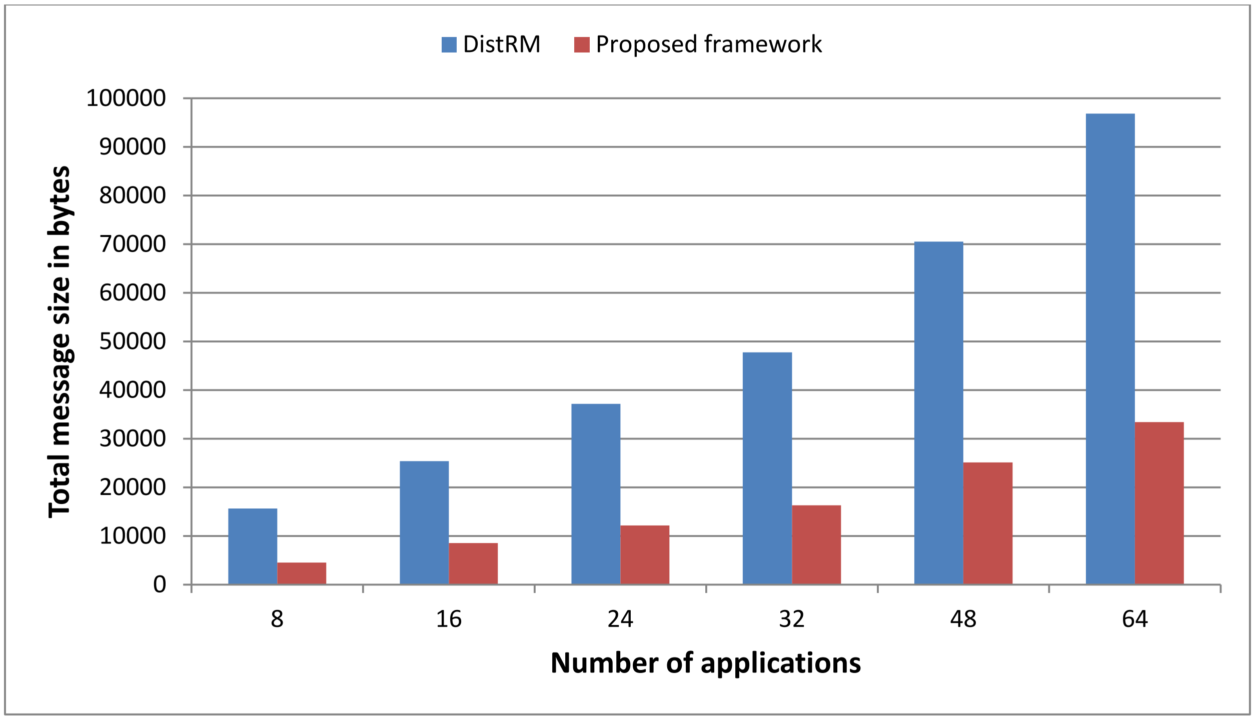


Figure 5.10: Total message size measured in bytes, needed by the two resource management frameworks for all number of applications on the SCC platform

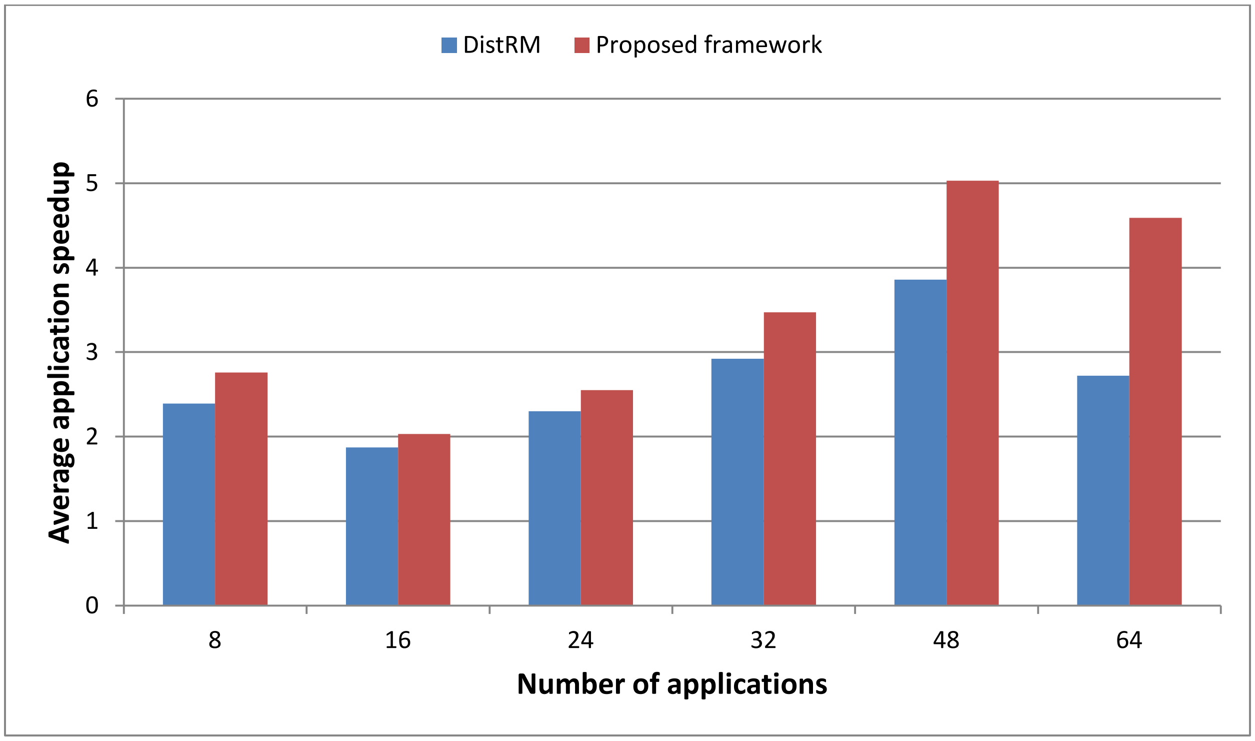


Figure 5.11: Average application speedup produced by the two resource management frameworks for all number of applications on the SCC platform

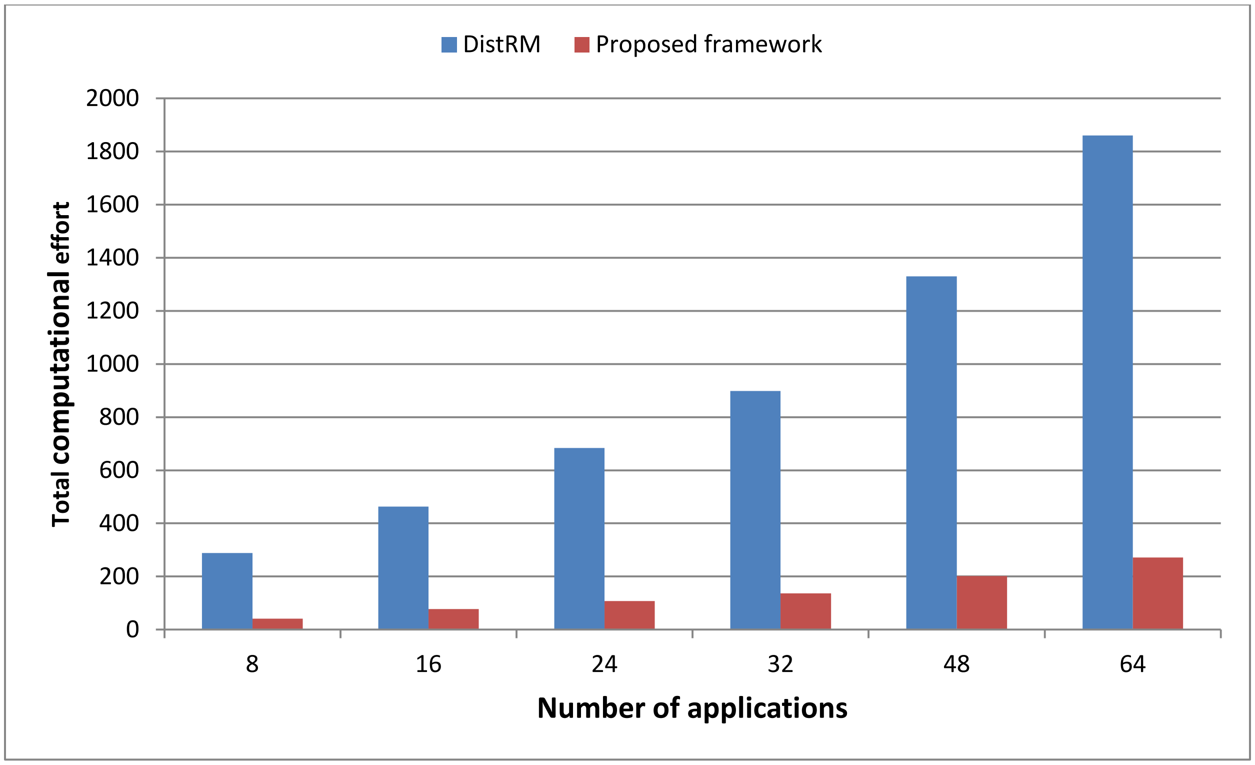


Figure 5.12: Total computational effort needed by the two resource management frameworks for all number of applications on the SCC platform

By inspecting the table and the figures we can observe that our proposed methodology performs much better in all metrics. An interesting observation is that in all cases, for both resource management algorithms, the measured results are lower in all metrics compared to those of the simulation platform. That is to say, fewer messages are exchanged but also worse application speedup is achieved.

The explanation for this is that the inter-node signaling mechanism that we developed does not respond as fast as the signaling mechanism of the Linux operating system. Thus, in a sense the vast majority of most operations take place “slower” compared to the C-simulator. As a consequence, many messages arrive after a timer has expired and their enclosed offers are rejected. For example in the case of a self-optimization process, where a timer is set while the manager core waits for offers, if an offer arrives after the timer has expired, it is rejected. As a result, the resource management frameworks cannot achieve the same speedup for their applications compared to the C-simulation. Since DistRM sends far more messages than our proposed methodology, it is much more prone to this anomaly. This is why in the actual platform our proposed methodology achieves significantly better results in average speedup compared to DistRM.

5.4 Results in simulator for heterogeneous platforms

In the case of heterogeneous platform simulation, we use as input only 32 and 64 applications. Grid sizes are the same as in homogeneous simulation and we test how the MF factor explained in section 4.1.7 influences the final results. The DistRM framework is not simulated, since its authors do not suggest a way for their framework to work on a heterogeneous platform. We consider that our four available different processing elements are equal in number and evenly distributed amongst the processing elements of the platform. An example in a 6x6 grid is the following:

```
0, 1, 2, 3, 0, 1,
2, 3, 0, 1, 2, 3,
0, 1, 2, 3, 0, 1,
2, 3, 0, 1, 2, 3,
0, 1, 2, 3, 0, 1,
2, 3, 0, 1, 2, 3
```

This pattern of PE distribution is extended in all grid sizes. The metrics used are the same as in the homogenous platform simulations.

The measured values are presented in the following tables:

	Platform size = 6x6							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	2811	2783	2739	2719	6013	5980	6118	6138
Message size	29608	29048	28868	28750	64248	63160	64818	64616
Computational effort	197	193	187	191	446	432	446	448
Average Speedup	2.981	2.973	2.618	2.417	3.538	3.653	3.308	3.228

Table 5.6: Measured values for all matching factor values for an 6x6 NoC platform

	Platform size = 8x8							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	2524	2532	2785	3007	5541	5520	6077	6133
Message size	26092	26436	28732	30856	57118	56660	61898	63736
Computational effort	162	162	165	184	375	359	389	404
Average Speedup	3.344	3.407	2.606	2.658	4.2249	4.22	3.692	3.578

Table 5.7: Measured values for all matching factor values for an 8x8 NoC platform

	Platform size = 12x12							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	5442	5461	5587	5601	10985	10974	10731	11122
Message size	63136	63546	66072	66258	128042	128502	126706	131234
Computational effort	437	433	446	440	892	881	841	865
Average Speedup	3.084	3.071	2.57	2.52	3.743	3.851	3.723	3.552

Table 5.8: Measured values for all matching factor values for an 12x12 NoC platform

	Platform size = 16x16							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	5640	5640	5868	6220	10937	10973	11637	11943
Message size	64782	65374	67514	71190	128130	128822	135554	139782
Computational effort	429	444	427	454	849	838	854	867
Average Speedup	3.332	3.35	2.572	1.984	4.392	4.26	3.814	3.159

Table 5.9: Measured values for all matching factor values for an 16x16 NoC platform

	Platform size = 20x20							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	5924	5859	5982	5994	11288	11206	11380	11713
Message size	64748	64654	66852	67064	128628	128428	132332	135366
Computational effort	411	403	397	396	800	787	783	796
Average Speedup	3.368	3.334	3.181	2.896	4.713	4.701	4.517	4.191

Table 5.10: Measured values for all matching factor values for an 20x20 NoC platform

	Platform size = 24x24							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	6285	6364	6499	6516	11650	11667	12268	12434
Message size	66920	68186	69970	70492	131794	132678	139446	141514
Computational effort	386	386	389	389	760	746	756	762
Average Speedup	3.793	3.783	3.646	3.565	5.261	5.274	4.994	4.807

Table 5.11: Measured values for all matching factor values for an 24x24 NoC platform

	Platform size = 28x28							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	6981	7044	7597	7444	12211	12497	13473	13535
Message size	69386	70448	76360	75022	132322	135738	146346	147552
Computational effort	379	383	403	405	719	725	746	737
Average Speedup	3.855	3.856	3.164	3.136	5.483	5.414	4.7	4.55

Table 5.12: Measured values for all matching factor values for an 28x28 NoC platform

	Platform size = 32x32							
	32 application				64 applications			
Matching factor	1	0.75	0.5	0.25	1	0.75	0.5	0.25
Message count	7542	7510	7723	7785	13326	13543	14473	14646
Message size	72036	71892	74398	74922	142356	144828	155036	157368
Computational effort	400	381	382	395	752	754	761	760
Average Speedup	3.802	3.736	3.621	3.519	5.554	5.14	4.683	4.16

Table 5.13: Measured values for all matching factor values for an 24x24 NoC platform

The following figures illustrate the relationship between average application speedup and MF variations:

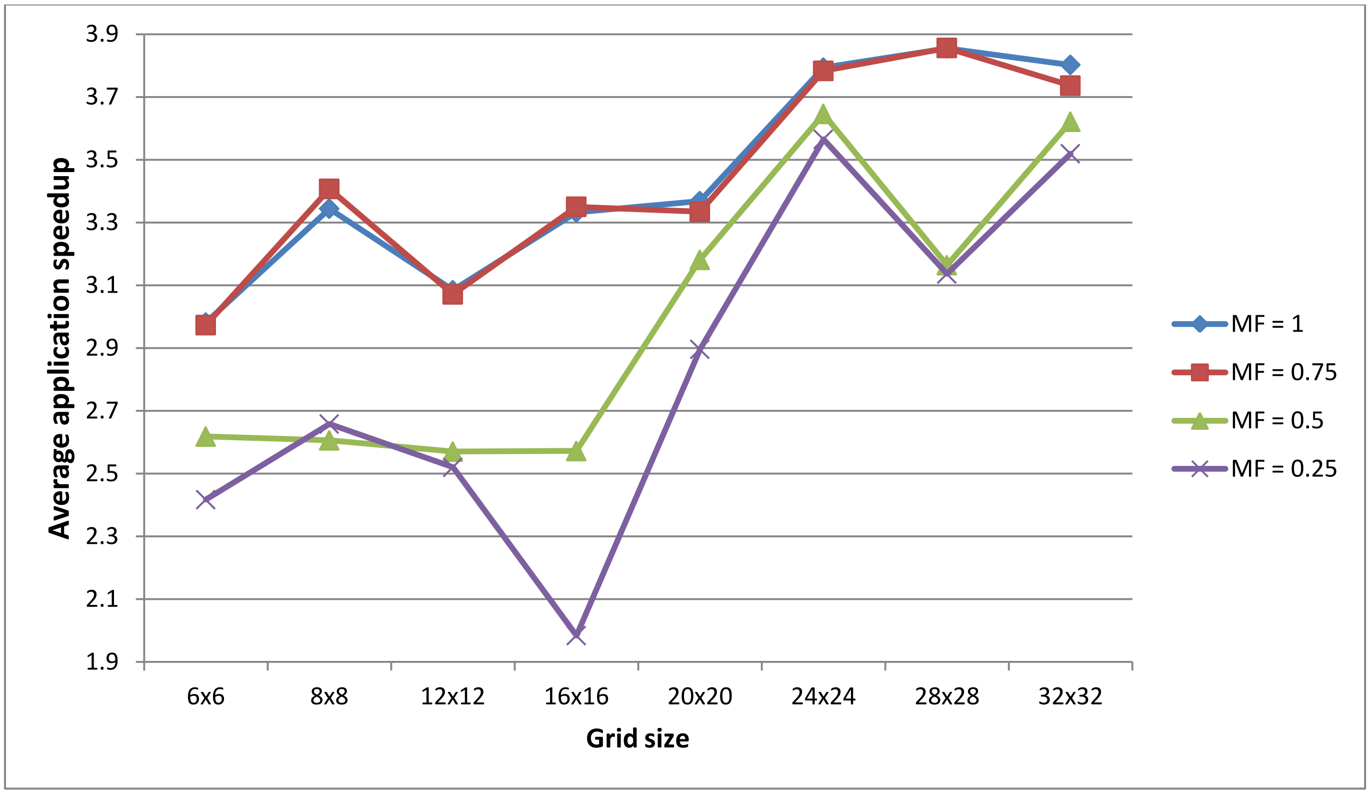


Figure 5.13: Average application speedup for 32 applications for all grid sizes and all MF values

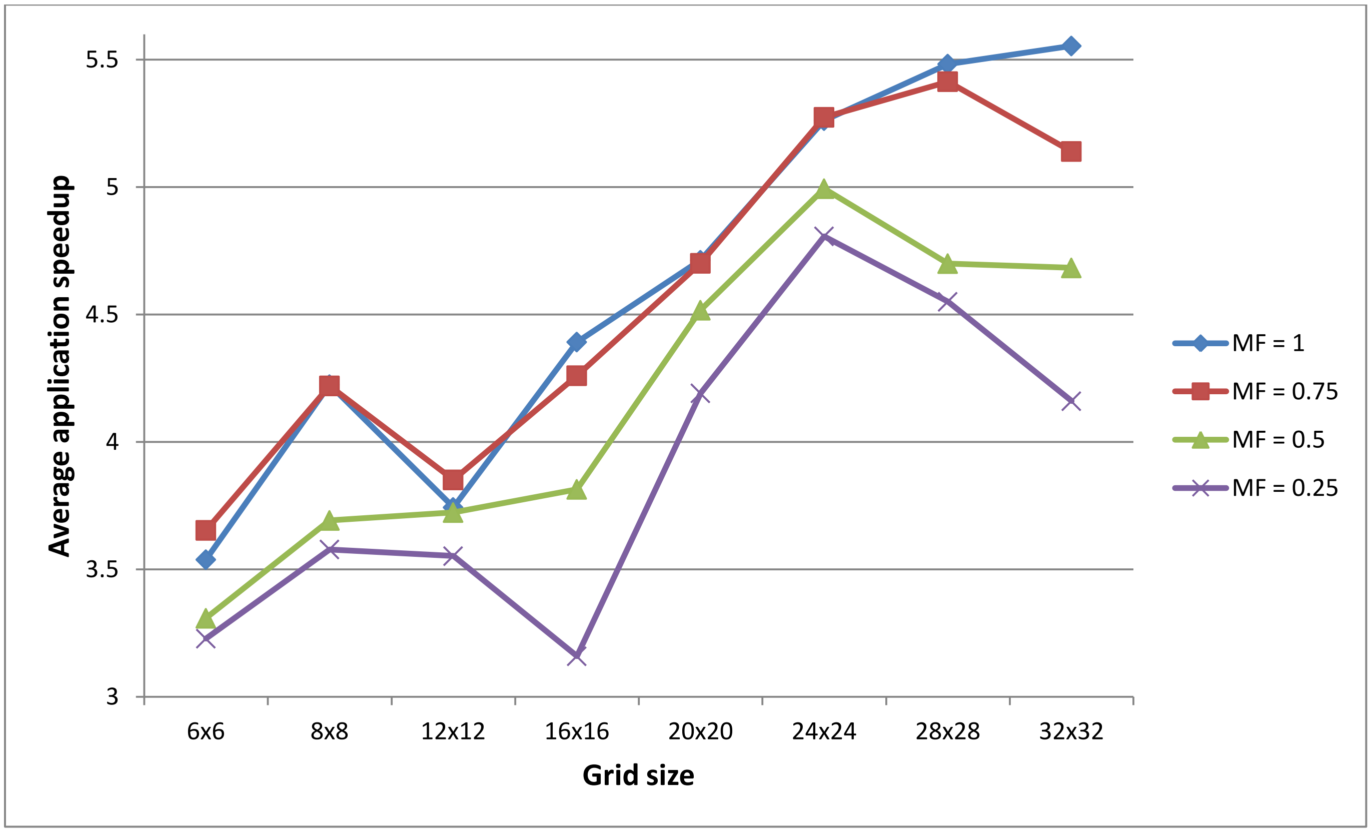


Figure 5.13: Average application speedup for 64 applications for all grid sizes and all MF values

We can see that in most cases, a decrement in the MF factor results to reduction of the average speedup. Considering that the MF factor determines how strict an application is as far allocation of cores of preferred processing elements is considered, this is to be expected. In the case of MF equal to 1, an application can accept only its most preferred processing element. As a result, its speedup will be the highest possible. On the other hand with the smallest MF in value, the application is flexible in receiving any of its acceptable processing elements. Some of the processing elements it will receive will not be its most preferable, thus leading to a decrease in its speedup. This trend can be observed clearly in 64 applications, in a 32x32 grid where both the number of applications and the number of available processing elements is high enough for the average speedup to decrease significantly due to the reduction of the MF factor.

In the following charts, the message count and computational effort for 64 applications for all grid sizes and mf values to evaluate the relationship of measured values and the MF factor.

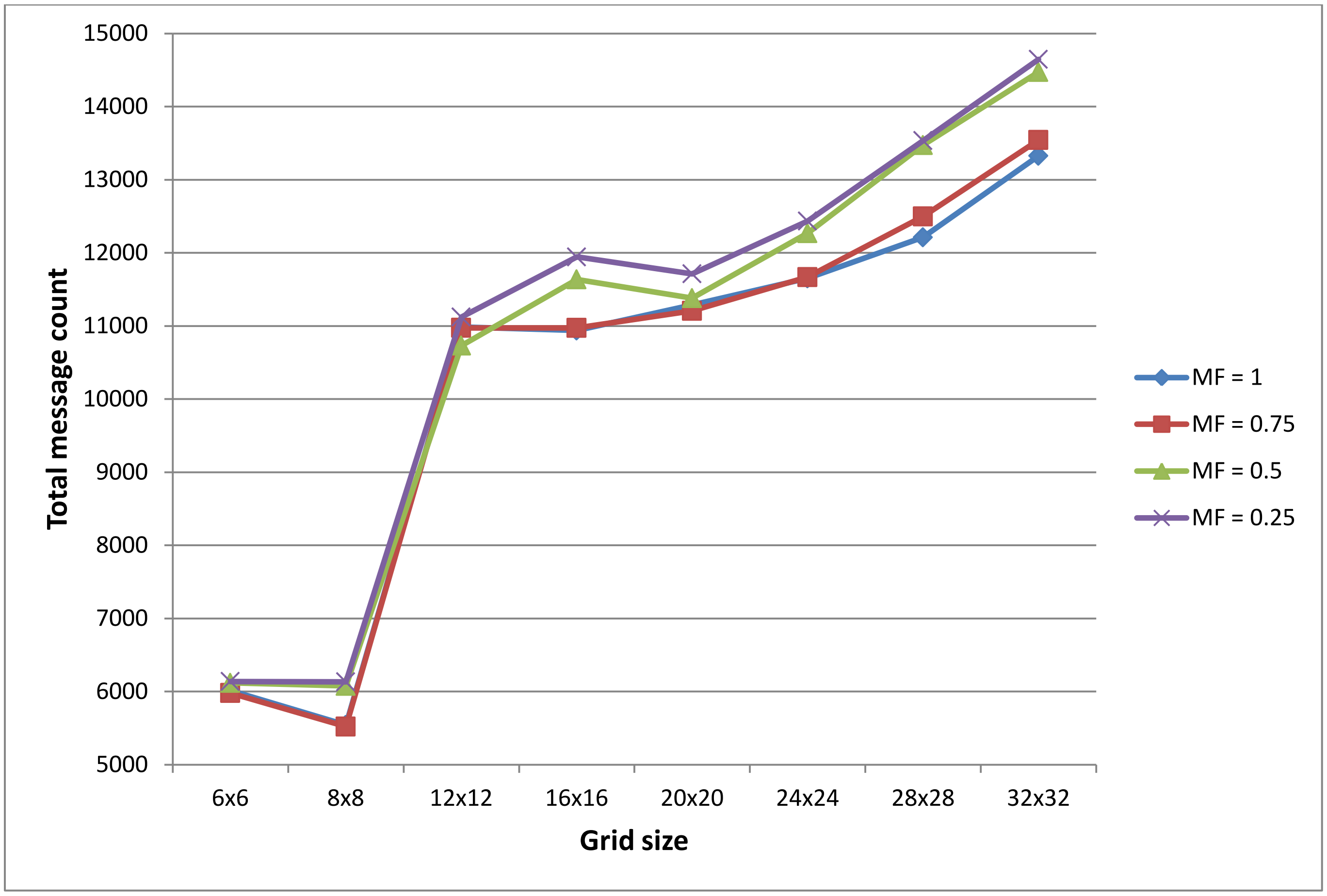


Figure 5.14: Total message count for 64 applications for all grid sizes and all MF values

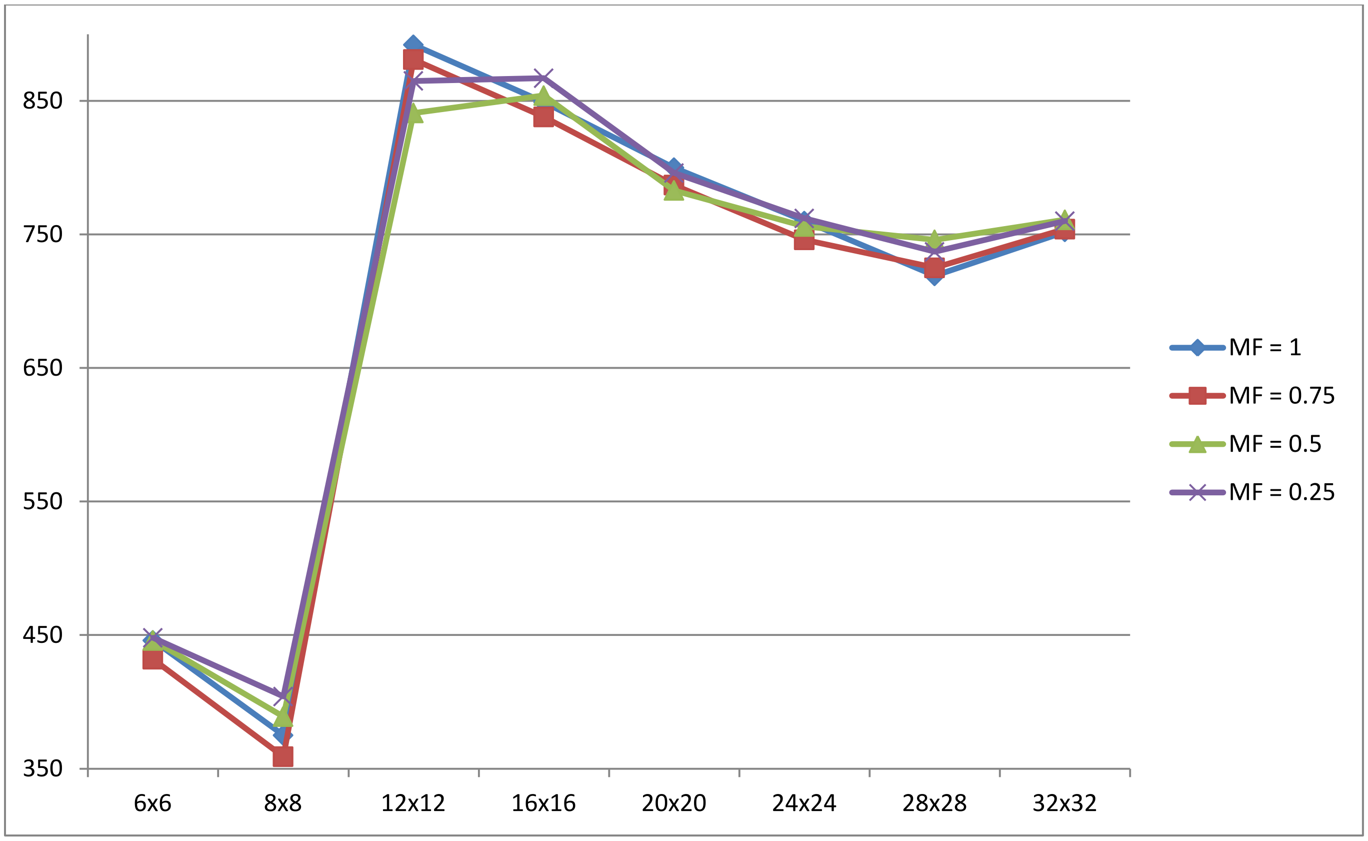


Figure 5.15: Total computational effort for 64 applications for all grid sizes and all MF values

First, we can see that the computational effort values are closer together compared to message count, as far at the variation due to different MF value is concerned. This means that, approximately, for a given platform size, the same in number requests for cores take place. What is different is the offers in response of these requests since for a high MF factor, many of these responses will be for zero cores. This is what creates the difference in the average speedup of the applications as explained in the previous paragraph.

As far as the message count is concerned, it is clearer that smaller MF values lead to smaller message count. This can be attributed to the fact that despite resource allocation is faster and probably results in more processing elements to be provided to an application, the average application speedup is smaller thus leading to more inter-communication of the cores of an application in order to execute their workload. In addition, this may lead to increased self-optimization activity leading once again to increased message count.

5.5 Experimental results conclusion

Taking into consideration that our proposed recourse manager framework produces better results in all metrics compared to the DistRM we conclude following:

i. When initiating a search for cores in a near region, it is much better practice to ask in one big region compared to searching to a lot small one. This is because the small ones are overlapping leading to an increase in messages that is in fact unnecessary.
ii. Self-optimization without specific criteria is useless since it only burdens the platform with extra communication without providing better resource allocation
iii. The random choice of initial cores can severely affect the outcome of the resource management algorithms.
iv. The implementation of the resource management algorithms in an actual platform, can severely affect the ability of the algorithm to distribute resources effectively. This is the case especially in platforms like Intel SCC which use a message passing interface.
v. In heterogeneous platforms, the matching factor dictating how strict the resource allocation will be as far as processing element preferences of the application are taken into account, indeed affects the results of our proposed framework. A decrement in the MF value, meaning a less strict resource allocation, can lead to decreased average application speedup and increases message count.

Chapter 6: Conclusion and future work

6.1 Summary

In the current thesis, we were involved with the problem of distributed run-time resource management for applications on Network-on-Chip Multi-Processor System-on-Chip. Initially an introduction is presented concerning the technology of NoC platforms and the essentials of the mapping problem in such platforms. After studying a set of similar works in the field, we introduced a resource management framework specializing for a type of parallel applications called the malleable one.

The proposed methodology targets both homogeneous and heterogeneous platforms and was tested on simulator, simulating homogeneous and heterogeneous platforms and an actual NoC platform. It was compared to another state-of-the-art run-time resource management framework called DistRM, which also targets malleable applications. The comparison favors our proposed methodology in all metrics used, with highest values at the actual NoC platform where for the resource management of the same number of applications our framework needs 70% less messages, reduces overall message size up to 64% and produces up to 20% better average application speedup using at the same time 85% less computational resources.

It is important to point out that parts of the presented work were published as [19] in Design Automation Conference at Austin, Texas at 2013. The publication included the description of the framework, the analysis of the gain calculation algorithm and the results of the experiments of the simulation of the homogeneous platform and Intel SCC platform.

6.2 Future work

The following chapters present ways to enhance the proposed methodology either by incorporating extra features or testing it in cases where new, interesting conclusions will occur.

6.2.1 Task migration

Task migration is the process of transferring a running application from a core to another. It is a very important field of study since it can severely affect any resource management algorithm. In our case, we constantly instruct resizing of applications as if it is a penalty free operation. Of course in real life this is **not** the case. A resizing operation can be a very expensive operation, especially in applications that are memory bound. In the extreme case where a careless task migration algorithm is employed it can prove to be a bottleneck for the entire platform if we try to migrate memory-bound applications in great distances on the platform.

In [15] the authors present a task-migration algorithm for multi core architectures. They state that their motivations for a task migration mechanism are:

- **Load balancing**. In this way the workload to be executed is more evenly and appropriately distributed over the platform leading on the one hand to better resource utilization and on the other hand avoiding overheating or faulty cores of the platform.

- **Reliability**. As a real-life system proceeds its working life, run-time error checking and on-chip communication fault statistics may dictate to avoid the excessive use of some cores.
- **Adaptivity**. There are technologies that enable us to have hardware on demand or reconfigurable hardware. In other words, the available resources of the platform may vary both in number and in type and thus some tasks may have to be migrated from one processing element to another.
- **Variability**. In sub-22 nm CMOS VLSI fabrication technologies we have a decreased predictability of the exact functional, electrical and thermal behavior of a system in the pre-silicon phase. Runtime adaptivity with task migration can address these issues.
- **Power consumption**. It is important to allocate out resources in a way that power consumption is minimized. For example, some cores can be shut down in case they consume too much power and for this to take place, an appropriate task migration mechanism is required.

In the following figure, we can see the important steps of a task migration process and the time metrics the authors of [15] try to control.

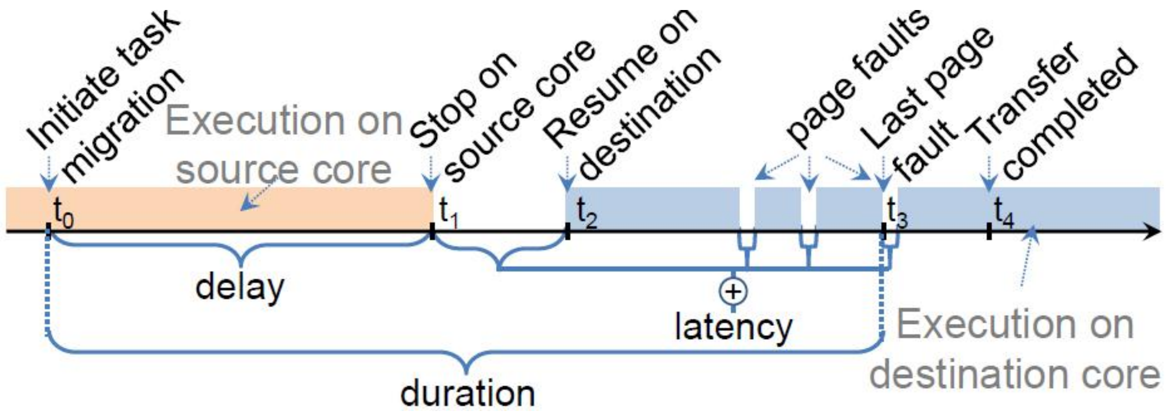


Figure 6.1: Latency, duration and delay of task migration. [15]

The authors propose Context-Aware Run-time adaptive Task Migration scheme. Its main characteristics are:

i. It leads to large reduction of migration latency.
ii. It provides a way to control the task migration delay and the tradeoff between the latency and the task migration bandwidth requirements in runtime-adaptive manner.
iii. A set of applications including state-of-the-art multimedia, compression and embedded applications have been profiled and the proposed mechanism is based on the interpretation of their memory access behavior.

The key point of the mechanism is that the memory pages that are migrated from one core to the other are chosen in an intelligent way. The ultimate goal is the order of the migration to match the order of the accesses on the destination core as closely as possible. A parameter α is introduced to control the tradeoff between latency and bandwidth and provide run-time adaptivity. A high value of α leads to an increased performance of the task migration mechanism while a smaller value reduces the memory pages transferred to the destination node prior to a page fault, thus leading to less power consumption. The following figure illustrates the use of the parameter α.

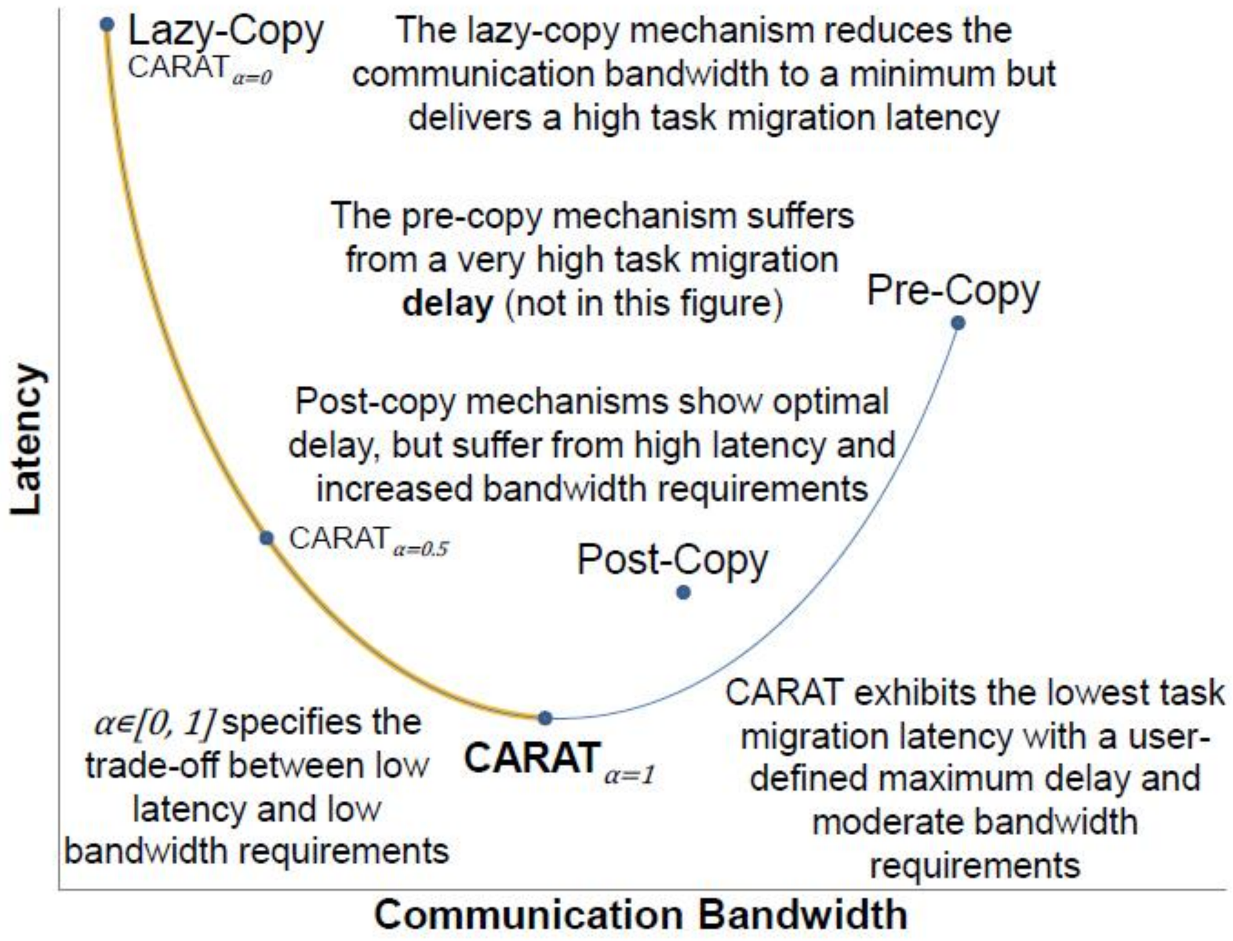

Lazy-Copy
CARAT$_{\alpha=0}$
The lazy-copy mechanism reduces the communication bandwidth to a minimum but delivers a high task migration latency
The pre-copy mechanism suffers from a very high task migration **delay** (not in this figure)
Pre-Copy
Post-copy mechanisms show optimal delay, but suffer from high latency and increased bandwidth requirements
CARAT$_{\alpha=0.5}$
Post-Copy
$\alpha\in[0, 1]$ specifies the trade-off between low latency and low bandwidth requirements
CARAT$_{\alpha=1}$
CARAT exhibits the lowest task migration latency with a user-defined maximum delay and moderate bandwidth requirements
Latency
Communication Bandwidth

Figure 6.2: Latency and bandwidth tradeoff through parameter α. [15]

Figure 6.3 illustrates the overall flow of the proposed task migration mechanism. In the beginning, the mechanism sends memory pages to the destination node in parallel to the execution of the migrating task, until the maximum task migration delay is reached (delay threshold) or a maximum amount of pages is transferred (bandwidth threshold). The order of which pages are sent is the following:

a. Send N x α pages of the B most recently read memory blocks.

b. Send small buffers that have been read frequently.
c. Send α x 100% of the remaining pages that have not yet been accessed for read access in blocks where other pages have recently been read.
d. Send α of the blocks that have predominantly read after having been written.

N and B are design parameters and their values are chosen by experimentation.

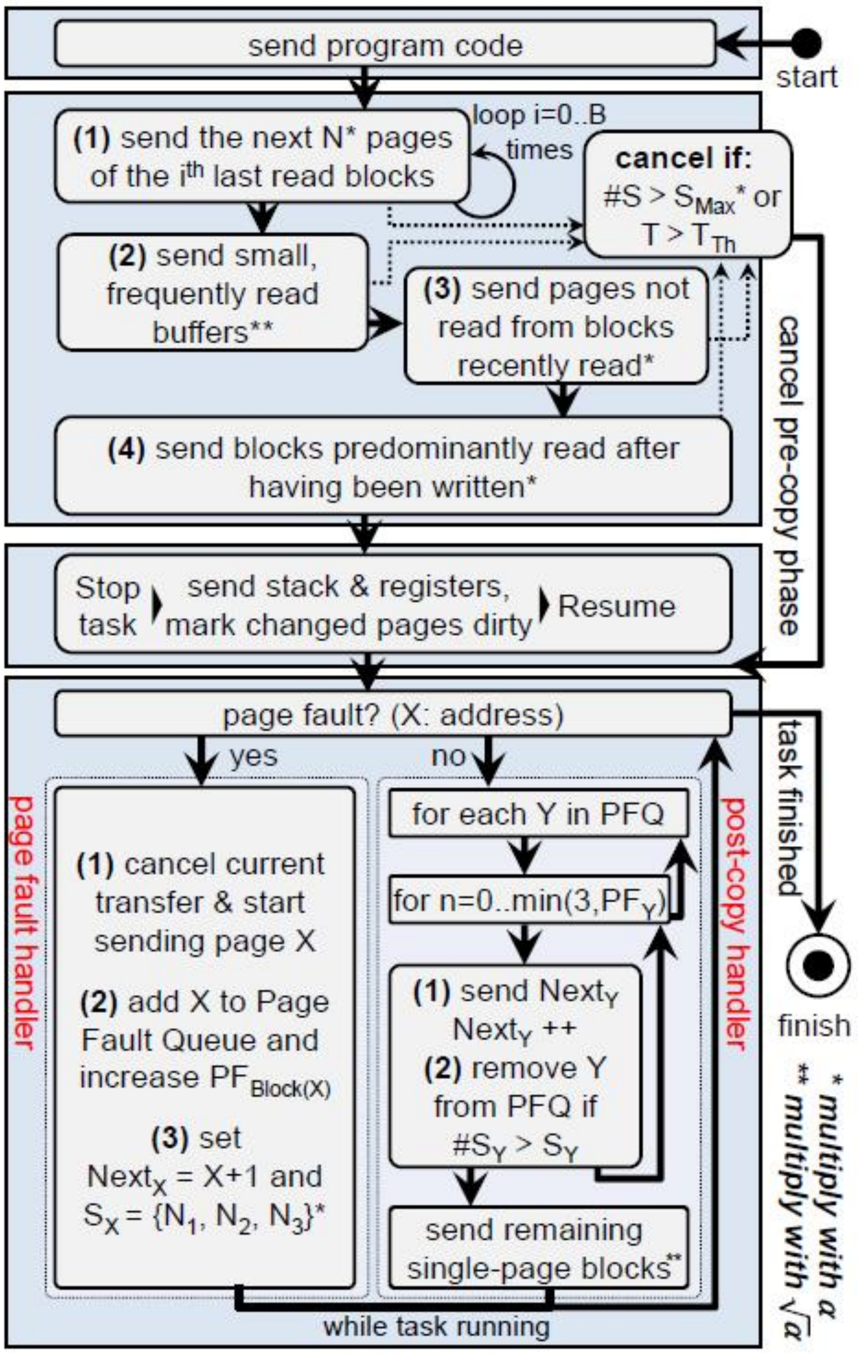

send program code
start
(1) send the next N* pages of the ith last read blocks
loop i=0..B times
cancel if: #S > SMax* or T > TTh
(2) send small, frequently read buffers**
(3) send pages not read from blocks recently read*
(4) send blocks predominantly read after having been written*
cancel pre-copy phase
Stop task
send stack & registers, mark changed pages dirty
Resume
page fault? (X: address)
yes
no
task finished
page fault handler
(1) cancel current transfer & start sending page X
(2) add X to Page Fault Queue and increase PFBlock(X)
(3) set NextX = X+1 and SX = {N1, N2, N3}*
for each Y in PFQ
for n=0..min(3,PFY)
(1) send NextY NextY ++
(2) remove Y from PFQ if #SY > SY
send remaining single-page blocks**
post-copy handler
finish
* multiply with α
** multiply with √α
while task running

Figure 6.3: Flow of the CARAT algorithm [15]

After the aforementioned pages have been sent, the task being migrated is paused on the source node and its stack and register contents are transferred to the destination node where its execution is resumed. When a page fault occurs, the current transfer is canceled and ant the missing page along with some succeeding pages are transferred immediately from the source to the destination node.

6.2.2 Evaluation on Nostrum platform

The Application Platform is a complex and full Multi-core NoC experimental platform presented in [18] (fig.6.4). It uses the LEON3 as the processor in each Processor-Memory node and uses the Nostrum NoC as the onchip network. Each PM node has a LEON3 processor (complete with I-Cache and D-Cache), a Data Management Engine (DME) [18] (fig. 6.5) as the network interface, plus a local memory. The LEON3 processor core is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The Nostrum NoC is a 2D mesh packet-switched network with configurable size that uses the XY-routing protocol. It serves as a customizable platform.

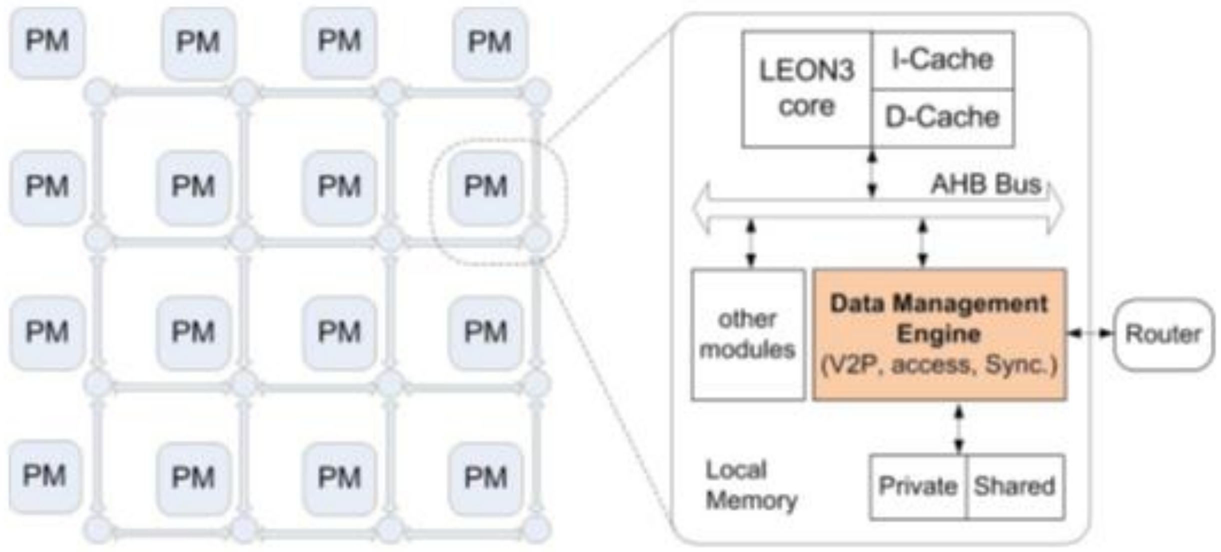


Figure 6.4: The application platform. [18]

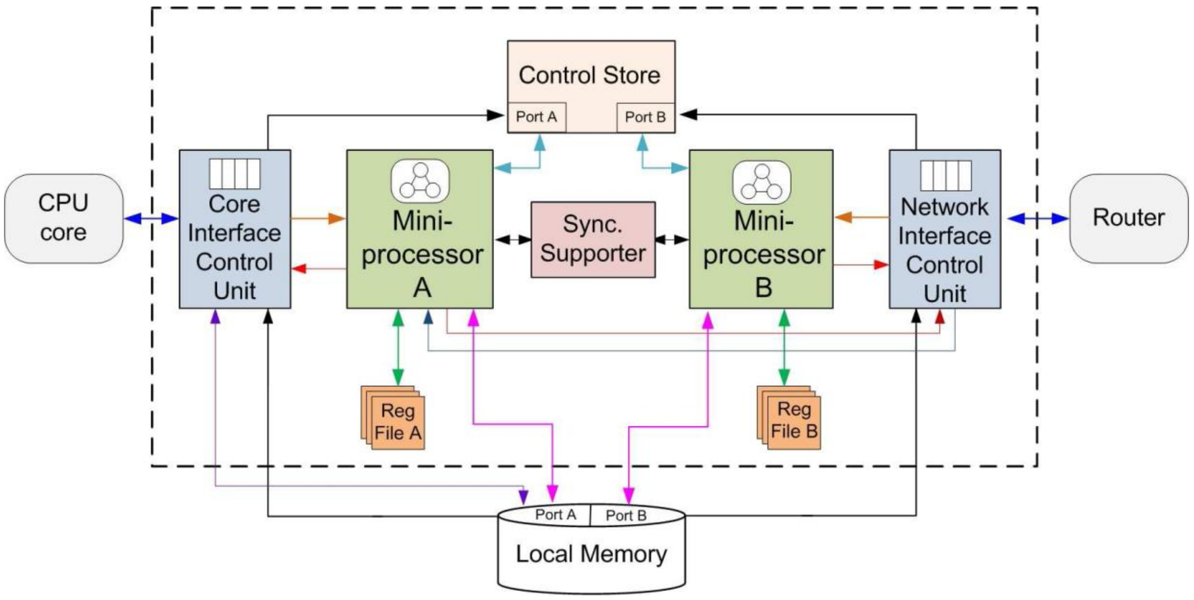


Figure 6.5: Architecture of the Data Management Engine. [18]

This platform has been developed with a different way of memory management compared to Intel SCC platform which uses a clear message passing interface, implemented on a message passing buffer. In this NoC platform, Memories are distributed in each node and tightly integrated with processors. All local memories can logically form a single global memory address space (fig. 6.6).The local memory is partitioned into two parts: private and shared and two addressing schemes are introduced: physical addressing and logic (virtual) addressing.

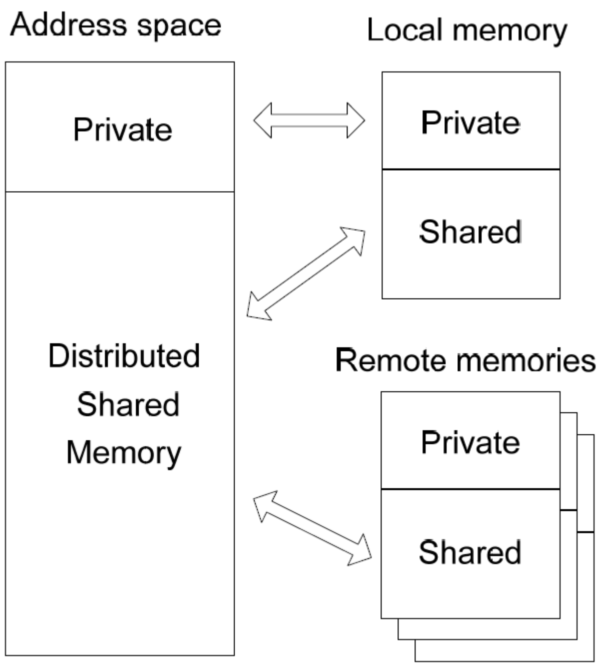


Figure 6.6: The global memory address space of each core. [18]

As a consequence the implementation of the two compared resource management algorithms in our work must be revaluated and redesigned to fit the need of this platform. This is an interesting challenge since it would prove if the design and implementation on the SCC platform severely influenced the measured values or if the compared frameworks are different in nature. An additional characteristic of this platform is that there is no OS present on the cores. Thus, the final results would be deprived of any overhead related to the operating system.

6.2.3 Evaluation as a run-time mapping service in Barrelfish OS

Barrelfish is a new research operating system being built from scratch and released by ETH Zurich in Switzerland, with assistance from Microsoft Research. [16] It is an effort to combine the two new trends in hardware design, the growing number of available processing elements and the diversity and heterogeneity of these processing elements. The existing operating systems mainly focus on scheduling of application while a constant need arises for correct run-time mapping of applications even for general purpose operating systems.

Barrelfish is structured as a set of low-level CPU drivers, each of which manages a processor core in isolation. Across the cores, a distributed system of cooperating processes known as monitors communicate exclusively via message passing. Barrelfish uses scheduler activations, and so each application runs a set of dispatchers, one per core, which collectively implement an application-specific user-level thread scheduler. Dispatchers are an implementation of scheduler activations, thus each dispatcher can run and schedule its own threads.

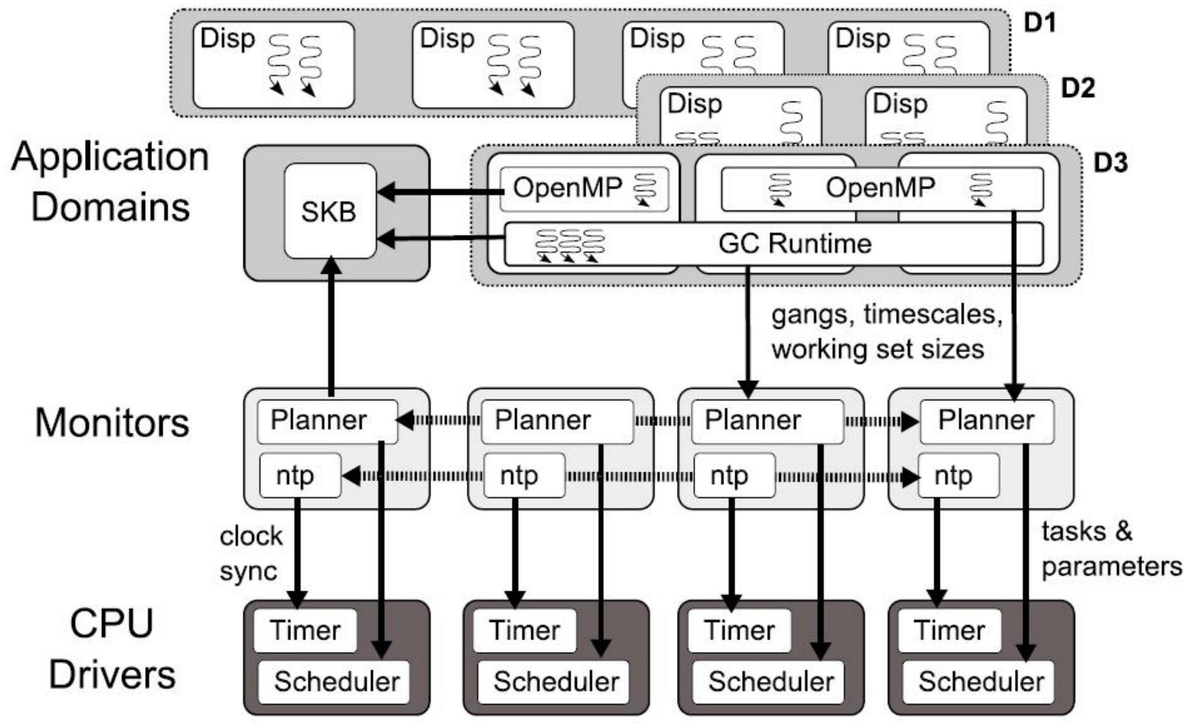


Figure 6.7: The Barrelfish scheduling architecture [17]

Barrelfish also introduces a novel approach concerning the interaction of running applications and the operation system. The allocation of resources to applications requires renegotiation while applications are running. This can occur when a new application starts, but also as its ability to use resources changes and in response to user input or changes in the underlying hardware. Hints from the application to the OS can be used to improve overall scheduling efficiency, but should not adversely impact other applications, violating the OS scheduler's fairness conditions. [17]

Taking all the characteristics of the Barrelfish OS into account, it is very interesting to try to incorporate our proposed resource management within the scheduling and mapping mechanism of Barrelfish. In that way we would be able to compare the resource management abilities of our framework compared to the way the Barrelfish distributes resources. It is very interesting fact that this OS has already been imported into the SCC platform thus making the task of comparison feasible in an actual multi-core NoC platform.

References

[1]: Tobias Bjerregaard and Shankar Mahadevan: *A Survey of Research and Practices of Network-on-Chip*. ACM Computing Survey Vol. 38, 2006

[2]: T. Mattson et al. *The 48-core SCC processor: the programmer's view*. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 13-19 November 2010

[3]: Tim Mattson, Intel Labs and Rob van der Wijngaart, Software and Services Group. *RCCE: A small library for many-core communication.*

[4]: The SCC Programmer's Guide. Revision 1.0. Intel Corporation, 2010.

[5]: Jingcao Hu, Radu Marculescu: *Energy- and Performance- Aware Mapping for Regular NoC Architectures.* IEEE Trans. On CAD of Integrated Circuits and Systems

[6]: Mohammad Abdullah Al Faruque et al.: *Adam: run-time agent-based distributed application mapping for on-chip communication.* DAC 2008

[7]: I. Anagnostopoulos et al. *A divide and conquer based distributed run-time mapping methodology for many-core platforms*. In Proc. of DATE, pages 111-116, 2012.

[8]: D.G. Feitelson and L. Rudolph. *Toward convergence in job schedules for parallel supercomputers*. In Proc. of JSSPP, pages 1-26, Springer – Verlag, 1996.

[9]: A. B. Downey. *A model for speedup of parallel programms*. Technical report, 1997.

[10]: D. Feitelson. Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload.

[11]: S. Kobbe et al. *DistRM: distributed resource management for on-chip many-core systems*. In Proc. of CODES+ISSS, pages 119-128. ACM, 2011.

[12]: V. Nollet et al.: *Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles*, in Proc. of DATE. IEEE Computer Society, 2005

[13]: G. Sabin et al. *Moldable parallel job scheduling using job efficiency: an iterative approach*. In Proc. of JSSPP, pages 94-114. Springer-Verlag, 2007.

[14]: T. Desell et al. *Malleable applications for scalable high performance computing*. Cluster Computing, 10(3): 323-337, 2007.

[15]: Janmartin Jahn et al. *CARAT: Context-aware runtime adaptive task migration for multi core architectures*. DATE 2011: 515-520

[16]: Simon Peter et al. *Design Principles for End-to-End Multicore Schedulers* HotPar'10 Proceedings of the 2nd USENIX conference on Hot topics in parallelism, pages 10-10

[17]: ETH Zurich, Microsoft Research The Barrelfish Operating System. http://www.barrelfish.org/

[18]: Axel Jantsch et al.: *Memory Architecture and Management in a NoC Platform*, In Axel Jantsch and Dimitrios Soudris, editors, *Scalable Multi-core Architectures: Design Methodologies and Tools*. Springer, 2011

[19]: Iraklis Anagnostopoulos et al: *Distributed run-time resource management for malleable applications on many-core platforms*. DAC 2013: 168