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

Hardware/Software Design Exploration Towards Resource Isolation and Performance Improvement in SoC FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξης Χατζηστυλιανός

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αν. Καθηγητής ΕΜΠ

Αθήνα
Απρίλιος 2019

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

Hardware/Software Design Exploration Towards Resource Isolation and Performance Improvement in SoC FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξης Χατζηστυλιανός

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αν. Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Απριλίου 2019.

.....	.....	.....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Γκούμας
Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα
Απρίλιος 2019

.....
Αλέξης Χατζηστυλιανός

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στις μέρες μας, ένας μεγάλος αριθμός εφαρμογών πρέπει να διαχειρίζεται μία συνεχή ροή δεδομένων και να την επεξεργάζεται κατάλληλα ώστε να ικανοποιούνται οι επιθυμητές απαιτήσεις. Είτε η ροή αυτή αντιστοιχεί σε πληροφορία που μεταδίδεται μέσω μίας τηλεπικοινωνιακής υποδομής, σε μετρήσεις που παράγονται από έναν αισθητήρα ή σε δεδομένα που δημιουργούνται από οποιαδήποτε άλλη πηγή, απαιτείται συχνά να την επεξεργαζόμαστε αποτελεσματικά και αδιάλειπτα, ενώ πολλαπλές κρίσιμες εργασίες πραγματικού χρόνου εκτελούνται ανεξάρτητα μεταξύ τους.

Όταν το πρόβλημα που περιγράφεται παραπάνω αντιμετωπίζεται με χρήση παραδοσιακών συστημάτων επεξεργασίας, εμφανίζονται διάφοροι περιορισμοί ως προς την απομόνωση των εργασιών και την επίδοση του συστήματος. Σε περίπτωση που εμπλέκεται ένα ενσωματωμένο σύστημα, ενδέχεται να εμφανιστούν και θέματα όπως η κατανάλωση ισχύος και η φορητότητα. Κατά συνέπεια, άλλες τεχνολογίες θα πρέπει να εξεταστούν για την υλοποίηση τέτοιων εφαρμογών. Τα SoC FPGAs φαίνονται ως μία ιδανική λύση, δεδομένου ότι παρέχουν ισχυρή ψηφιακή επεξεργασία σήματος στο υλισμικό, σε συνδυασμό με την ευελιξία και τη δυνατότητα πολλαπλής επεξεργασίας ενός πολυπύρηνου επεξεργαστή, καθώς και ένα ευρύ σύνολο περιφερειακών για λειτουργίες υψηλού επιπέδου. Ωστόσο, προκειμένου να καταλήξουμε σε μία ικανοποιητική λύση, πρέπει σαφώς να λάβουμε υπόψη την απομόνωση των πόρων και τη βελτιστοποίηση της απόδοσης.

Σε αυτή τη διπλωματική εργασία, σχεδιάζεται και υλοποιείται σε ένα διπύρηνο SoC FPGA μία δοκιμαστική υποδομή κατάλληλη για μία τηλεπικοινωνιακή εφαρμογή. Αποτελείται από έναν πομπό και έναν δέκτη δεδομένων, καθένας εκ των οποίων τρέχει σε ξεχωριστή CPU. Μία ροή δεδομένων υπόκειται σε επεξεργασία στο FPGA και μεταφέρεται από τη μία πλευρά στην άλλη, ενώ κάθε πυρήνας είναι υπεύθυνος για την εκτέλεση των δικών του ανεξάρτητων εργασιών στα δεδομένα. Το σύστημα αναπτύσσεται στην αναπτυξιακή πλακέτα Zybo, η οποία βασίζεται στο Zynq-7000 All Programmable SoC. Επιχειρούνται βελτιστοποιήσεις τόσο από την πλευρά του υλισμικού όσο και του λογισμικού, ώστε να ικανοποιούνται οι απαιτήσεις μίας πιθανής πραγματικής εφαρμογής. Πρώτον, διερευνάται η σχεδίαση του υλισμικού, με στόχο τη μεγιστοποίηση του ρυθμού μεταφοράς των δεδομένων. Στη συνέχεια, υλοποιούνται δύο διαφορετικά σχήματα, από άποψη λειτουργικών συστημάτων: ένα ενιαίο περιβάλλον Linux που εφαρμόζεται και στους δύο πυρήνες και μία αρχιτεκτονική Linux/FreeRTOS, όπου κάθε πυρήνας τρέχει το δικό του ξεχωριστό λειτουργικό σύστημα.

Τέλος, και τα δύο σχήματα δοκιμάζονται στην αναπτυγμένη εφαρμογή και συγκρίνονται σχετικά με την απομόνωση και την απόδοση. Πραγματοποιείται μία σειρά από πειράματα που τοποθετούν το σύστημα σε διαφορετικά σενάρια. Οι μετρήσεις και τα συμπεράσματα που προκύπτουν παρουσιάζονται για την αξιολόγηση του συστήματος και στις δύο περιπτώσεις.

Λέξεις Κλειδιά

SoC FPGA, απομόνωση, επίδοση, συμμετρική και ασύμμετρη πολυεπεξεργασία, ροή δεδομένων, σχεδίαση υλισμικού/λογισμικού, επεξεργασία πραγματικού χρόνου.

Abstract

Nowadays, a vast number of applications need to manage a continuous stream of data and process it appropriately to meet the desired requirements. Whether this stream corresponds to information transmitted through a telecom infrastructure, measurements generated by a sensor or data created by any other source, it is often required to process it efficiently and uninterruptedly, while multiple critical real-time tasks are executed independently.

When dealing with the problem described above in traditional processing systems, engineers are faced with various limitations in terms of task isolation and performance. In case an embedded system is involved, issues like power consumption and portability may also appear. Consequently, other technologies should be considered as target platforms for such applications. Multicore SoC FPGAs, in particular, seem to be an ideal solution, since they provide powerful Digital Signal Processing (DSP) in hardware, combined with the flexibility and multiprocessing features of a multicore processor, as well as a rich set of peripherals for high-level functions. However, in order to end up with a satisfying solution, resource isolation and performance optimization still need to be taken into consideration.

In this thesis, a testbed suitable for a telecom application is designed and implemented on a dual-core SoC FPGA. It consists of a transmitter and a receiver of data, each one running on a separate CPU core. A stream of data is processed in the FPGA and transferred from one side to another, while each core is responsible for executing its own independent tasks on data. The system is deployed on the Zybo development board, which is built around the Zynq-7000 All Programmable SoC device. Optimizations are attempted both from hardware and software perspectives, to satisfy the requirements of a potential real-life application. First, an exploration is done on the hardware design, aiming to maximize the throughput of the data transfers. Then, two different configurations are implemented, in terms of operating systems: a single Linux environment running on both cores and a Linux/FreeRTOS architecture, where each core runs its own separate operating system.

Finally, both configurations are tested on the developed application and compared, with respect to isolation and performance. A number of experiments are conducted to place the system in different scenarios. The resulting measurements and conclusions are presented in order to evaluate the system in both cases.

Keywords

SoC FPGA, isolation, performance, symmetric and asymmetric multiprocessing, data streaming, hardware/software design, real-time processing.

Ευχαριστίες

Καταρχάς, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Δημήτριο Σούντρη, ο οποίος με εμπιστεύτηκε για αυτή τη διπλωματική εργασία και μου έδωσε την ευκαιρία να την διεκπεραιώσω στο Εργαστήριο Μικροεπεξεργαστών και Ψηφιακών Συστημάτων.

Θα ήθελα επίσης να εκφράσω τις ευχαριστίες μου σε όλα τα μέλη του Εργαστηρίου για το φιλικό εργασιακό περιβάλλον που μου προσέφεραν κατά τη διάρκεια εκπόνησης αυτής της εργασίας. Θα ήθελα ιδιαίτερα να ευχαριστήσω τους Κωνσταντίνο Μαραγκό, Γεώργιο Λεντάρη και Ιωάννη Στρατάκο για την πολύτιμη βοήθεια και καθοδήγησή τους καθ' όλη τη διάρκεια της εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους μου για τη στήριξή τους όλα αυτά τα χρόνια.

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Dimitrios Soudris, who trusted me for this diploma thesis and gave me the opportunity to accomplish it at the Microprocessors and Digitals Systems Lab.

I would also like to express my thanks to all the members of the Lab, for the friendly working environment they have offered me during the elaboration of this thesis. I would especially like to thank Konstantinos Maragos, George Lentaris and Ioannis Stratakos for their valuable help and guidance throughout this thesis.

Finally, I would like to thank my family and my friends for their support all these years.

Contents

List of Figures and Tables

Εκτεταμένη Περίληψη

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Εκτεταμένη Περίληψη

Εισαγωγή

Τα τελευταία χρόνια, παρατηρείται μία στροφή προς τη χρήση των SoC FPGAs για την υλοποίηση ποικίλων εφαρμογών. Πρόκειται για μία υβριδική τεχνολογία που ενσωματώνει μικροεπεξεργαστή και FPGA πάνω στην ίδια ψηφίδα.

Το FPGA (Field Programmable Gate Array) είναι ένα ολοκληρωμένο κύκλωμα που αποτελείται από μία συστοιχία ρυθμιζόμενων λογικών μονάδων που συνδέονται μεταξύ τους μέσω προγραμματιζόμενων διασυνδέσεων. Μέσω γλωσσών περιγραφής υλικού, τα FPGAs μπορούν να επαναπρογραμματιστούν ώστε να υλοποιούν διαφορετική λειτουργία, ανάλογα με τις απαιτήσεις της εκάστοτε εφαρμογής. Εκτός από την ευελιξία, τα FPGAs παρουσιάζουν και άλλα πλεονεκτήματα, όπως η επιτάχυνση μέσω παράλληλης επεξεργασίας στο υλισμικό και μειωμένο χρόνο προς την αγορά, συγκριτικά με τα ολοκληρωμένα κυκλώματα ειδικού σκοπού.

Έτσι, τα SoC FPGAs συνδυάζουν τα οφέλη του FPGA με την ευελιξία ενός επεξεργαστή, ενώ παρέχουν επίσης ένα σύνολο από περιφερειακά και διεπαφές για την επίτευξη ειδικών λειτουργιών και επικοινωνίας με εξωτερικές συσκευές. Εισάγεται λοιπόν μία αρχή σχεδίασης υλισμικού/λογισμικού, όπου μία εργασία μπορεί να κατανεμηθεί παράλληλα στον επεξεργαστή και στη λογική του FPGA, βελτιώνοντας τη συνολική απόδοση του συστήματος.

Όσον αφορά τον επεξεργαστή, θεμελιώδης θεωρείται πλέον η χρήση πολυπύρηνων αρχιτεκτονικών. Ένα πολυεπεξεργαστικό σύστημα αποτελείται από πολλαπλές μονάδες επεξεργασίας (ή πυρήνες) που εκτελούν παράλληλα τις δικές τους λειτουργίες. Οι λειτουργίες αυτές μπορεί να ανήκουν σε ανεξάρτητες διεργασίες ή να αποτελούν διαφορετικά νήματα της ίδιας διεργασίας. Ανάλογα με την περίπτωση, οι πολλαπλοί πυρήνες μοιράζονται ίδιο χώρο διευθύνσεων και κοινούς πόρους, όπως κρυφές μνήμες και διαύλους Ε/Ε. Η επικοινωνία και η ανταλλαγή δεδομένων μεταξύ των πυρήνων πραγματοποιείται μέσω κοινής μνήμης, ανταλλαγής μηνυμάτων ή άλλων μηχανισμών επικοινωνίας. Σε μία τέτοια δομή, πρέπει ασφαλώς να ληφθούν υπόψη ζητήματα όπως ο συγχρονισμός και η συνέπεια της μνήμης ανάμεσα στους επεξεργαστές.

Ιδιαίτερο ενδιαφέρον παρουσιάζουν οι περιπτώσεις που περιλαμβάνουν ετερογενείς πυρήνες στο ίδιο σύστημα. Αυτό δεν συνεπάγεται αποκλειστικά διαφορετικές αρχιτεκτονικές επεξεργαστών, αλλά και διαφορετικά λειτουργικά συστήματα ή ειδικευμένες επεξεργαστικές δυνατότητες. Έτσι, κάθε πυρήνας μπορεί να σχεδιαστεί κατάλληλα ώστε να χειρίζεται αποδοτικά εργασίες ειδικού σκοπού.

Σε κάθε περίπτωση, όταν σε ένα πολυεπεξεργαστικό σύστημα εκτελούνται ανεξάρτητες εργασίες, είναι σημαντικό να υπάρχει απομόνωση μεταξύ των διαφορετικών πυρήνων, ώστε να εγγυάται η ασφάλεια, να ικανοποιούνται τυχόν περιορισμοί πραγματικού χρόνου και να επιτυγχάνεται η μέγιστη δυνατή απόδοση του συστήματος. Πιθανές παρεμβολές μεταξύ των εργασιών δύνανται να οδηγήσουν σε κινδύνους ή χρονικές καθυστερήσεις που μπορεί να είναι κρίσιμες για την απαιτούμενη εφαρμογή.

Σκοπός

Σκοπός της παρούσας εργασίας είναι η υλοποίηση σε διπύρηνο SoC FPGA και αξιολόγηση μίας δοκιμαστικής υποδομής για τηλεπικοινωνιακές εφαρμογές. Κατά την υλοποίηση του συστήματος αξιοποιείται τόσο η προγραμματιζόμενη λογική για αποδοτική επεξεργασία όσο και οι δυνατότητες και τα περιφερειακά που παρέχει ο διπύρηνος επεξεργαστής. Επιχειρείται η βελτιστοποίηση του συστήματος από πλευρά υλισμικού και λογισμικού, ως προς την απομόνωση και την απόδοση. Προκειμένου να αξιολογηθεί το σύστημα, αναπτύσσεται μία δοκιμαστική τηλεπικοινωνιακή εφαρμογή που αποτελείται από έναν πομπό και έναν δέκτη δεδομένων, οι οποίοι πρέπει να εκτελούν απομονωμένα και αποτελεσματικά τις αντίστοιχες λειτουργίες τους. Καθώς η ροή δεδομένων

μεταδίδεται από τον πομπό προς τον δέκτη, οι οποίοι εκτελούνται σε ξεχωριστούς πυρήνες του επεξεργαστή, υπόκειται και σε ενδιάμεση επεξεργασία στο FPGA.

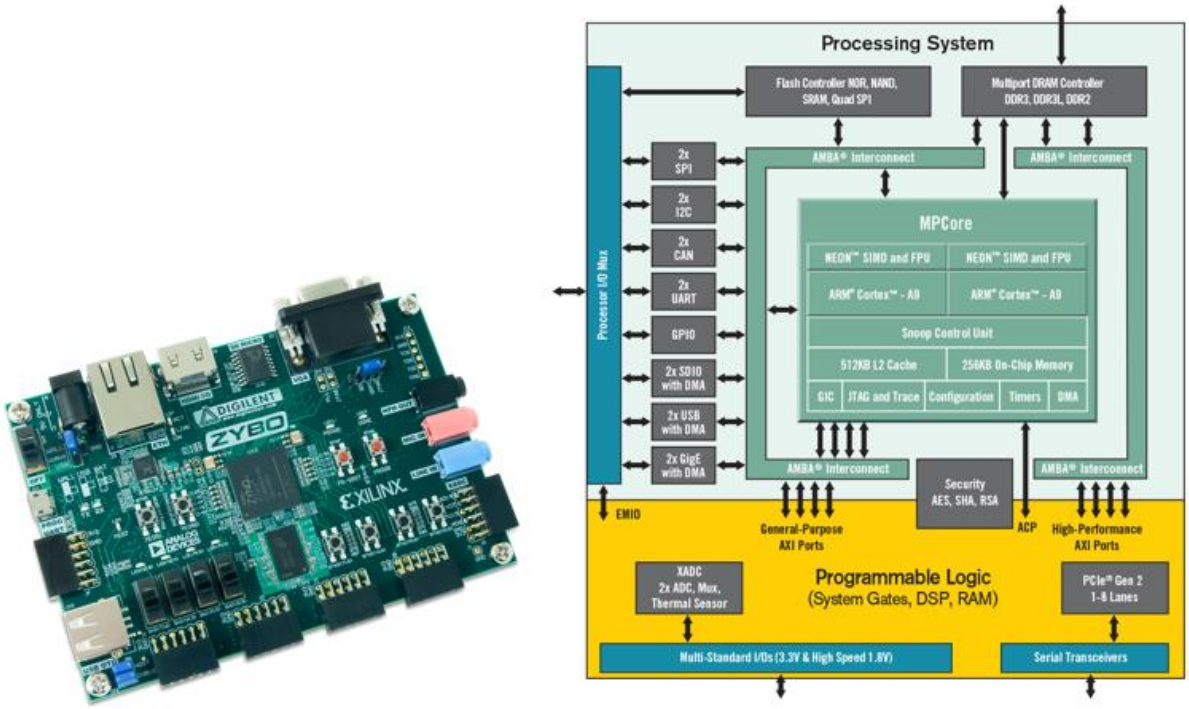
Το κίνητρο για τη δημιουργία και εξέταση ενός δοκιμαστικού συστήματος όπως το παραπάνω, χωρίς τον καθορισμό συγκεκριμένου αλγορίθμου για την επεξεργασία των δεδομένων, είναι ότι έτσι αυτό διατηρείται προσαρμόσιμο για την υλοποίηση διαφόρων εφαρμογών. Σε μία πραγματική τηλεπικοινωνιακή εφαρμογή, η ροή δεδομένων αντιστοιχεί στη μετάδοση σημάτων και η ενδιάμεση επεξεργασία σε διαδικασίες όπως η διαμόρφωση και η πολυπλεξία των σημάτων. Μία άλλη ενδεικτική εφαρμογή είναι η λήψη μίας συνεχούς ακολουθίας μετρήσεων από έναν αισθητήρα, για παράδειγμα ένα LIDAR, η οποία πρέπει να επεξεργαστεί κατάλληλα και σε πραγματικό χρόνο. Παράλληλα, το σύστημα θα πρέπει να είναι ικανό να παρέχει μία διαδραστική διεπαφή προς τον χρήστη, χωρίς αυτή να επηρεάζει τη λήψη και επεξεργασία των δεδομένων.

Βασικές Έννοιες

Η συσκευή

Ως πλατφόρμα ανάπτυξης του συστήματος χρησιμοποιήθηκε η πλακέτα Zybo της Digilent. Αυτή βασίζεται στο Zynq Z-7010 AP SoC της Xilinx, το οποίο ενσωματώνει έναν διπύρηνο ARM Cortex-A9 επεξεργαστή με την προγραμματιζόμενη λογική του FPGA. Η πλακέτα περιλαμβάνει, μεταξύ άλλων, 512 MB DDR3 μνήμης καθώς και πλήθος περιφερειακών (Ethernet, USB, UART/JTAG SDIO, HDMI, GPIO). Η πλακέτα Zybo και η αρχιτεκτονική του Zynq AP SoC φαίνονται στην Εικόνα 1.

Το Zynq SoC χωρίζεται σε δύο υποσυστήματα, το Σύστημα Επεξεργασίας (ΣΕ) και την Προγραμματιζόμενη Λογική (ΠΛ). Στο ΣΕ περιέχονται ο διπύρηνος επεξεργαστής ARM, μαζί με τις αντίστοιχες υπολογιστικές μονάδες και κρυφές μνήμες, καθώς και οι διεπαφές προς τη μνήμη και οι διαχειριστές των διαφόρων περιφερειακών. Η ΠΛ αποτελείται από FPGA λογική γενικού σκοπού, αλλά και ειδικούς πόρους όπως μπλοκ μνήμης RAM και μονάδες ψηφιακής επεξεργασίας σήματος.



Εικόνα 1: Η πλακέτα Zybo και η αρχιτεκτονική του Zynq AP SoC [9][11]

Τα διάφορα τμήματα στο ΣΕ και τη ΠΛ επικοινωνούν μεταξύ τους μέσω AXI διασυνδέσεων, σύμφωνα με το πρότυπο AMBA της ARM. Υπάρχουν τρία διαφορετικά είδη AXI διεπαφών μεταξύ ΣΕ και ΠΛ: τέσσερις θύρες γενικού σκοπού (GP), μία θύρα ACP που διασφαλίζει τη συνέπεια μεταξύ της ΠΛ και των κρυφών μνημών του ΣΕ και τέσσερις θύρες υψηλής απόδοσης (HP), οι οποίες υποστηρίζουν αποδοτική επικοινωνία μεταξύ ΠΛ και μνήμης.

Το AXI4-Stream πρωτόκολλο

Η πιο πρόσφατη έκδοση του AMBA AXI πρωτοκόλλου επικοινωνίας είναι το AXI4 και εμφανίζεται σε τρεις παραλλαγές: AXI4-Full, AXI4-Lite και AXI4-Stream. Εδώ επικεντρωνόμαστε στο AXI4-Stream πρωτόκολλο, το οποίο επιτρέπει την μεταφορά μίας συνεχούς ροής μεγάλου όγκου δεδομένων από έναν αφέντη (master) προς έναν σκλάβο (slave). Το AXI4-Stream είναι λόγω της φύσης του το πλέον κατάλληλο για μία τηλεπικοινωνιακή εφαρμογή, όπου ένας πομπός μεταδίδει μια ροή πληροφορίας προς έναν δέκτη.

Μία AXI4-Stream διεπαφή αποτελείται από μία σειρά σημάτων που επιτυγχάνουν την επικοινωνία και τον συγχρονισμό μεταξύ αφέντη και σκλάβου. Τα κυριότερα σήματα παρουσιάζονται στον πίνακα 1. Για να πραγματοποιηθεί μία μεταφορά δεδομένων, απαιτείται μία διαδικασία χειραψίας μεταξύ των δύο πλευρών, η οποία απαιτεί να είναι ενεργοποιημένα και τα δύο σήματα **TVALID** και **TREADY**.

Σήμα	Πηγή	Περιγραφή
TVALID	Αφέντης	Υποδεικνύει ότι ο αφέντης άγει μία έγκυρη μεταφορά δεδομένων.
TREADY	Σκλάβος	Υποδεικνύει ότι ο σκλάβος μπορεί να δεχθεί μία μεταφορά δεδομένων στον τρέχοντα κύκλο.
TDATA	Αφέντης	Παρέχει τα δεδομένα που μεταφέρονται δια μέσου της διεπαφής.
TLAST	Αφέντης	Υποδεικνύει το όριο ενός πακέτου δεδομένων.

Πίνακας 1: Τα κυριότερα σήματα του AXI4-Stream πρωτοκόλλου

Άμεση Προσπέλαση Μνήμης

Η Άμεση Προσπέλαση Μνήμης (ΑΠΜ) είναι ένας μηχανισμός που επιτρέπει την πρόσβαση στη μνήμη ενός συστήματος για μεταφορά δεδομένων, χωρίς τη μεσολάβηση του επεξεργαστή. Η μεταφορά δεδομένων από και προς συσκευές Ε/Ε είναι γενικά μία αργή διαδικασία. Μέσω της ΑΠΜ, η εργασία αυτή ανατίθεται σε ειδικευμένες για αυτόν τον σκοπό συσκευές, επιτρέποντας στον επεξεργαστή να εστιάσει σε άλλες χρήσιμες δραστηριότητες, όσο η μεταφορά των δεδομένων βρίσκεται σε εξέλιξη.

Η διαδικασία της μεταφοράς δεδομένων μέσω ΑΠΜ γίνεται ως εξής:

1. Ο επεξεργαστής ρυθμίζει κατάλληλα τη συσκευή ΑΠΜ για την επιθυμητή μεταφορά.
2. Η συσκευή δεσμεύει χώρο στη μνήμη για την αποθήκευση των δεδομένων και ξεκινά τη μεταφορά.
3. Τα δεδομένα μεταφέρονται προς τον προορισμό μέσω ΑΠΜ.
4. Η συσκευή λαμβάνει μία επιβεβαίωση μόλις ολοκληρωθεί η μεταφορά.
5. Η συσκευή ενημερώνει τον επεξεργαστή ότι η μεταφορά ολοκληρώθηκε και ότι μία νέα μεταφορά μπορεί πλέον να δρομολογηθεί.

Κατά τη χρήση ΑΠΜ, ιδιαίτερη προσοχή θα πρέπει να δίνεται στη διαχείριση των δεσμευμένων χώρων στη μνήμη και στη διατήρηση της συνέπειας μεταξύ μνήμης και κρυφών μνημών.

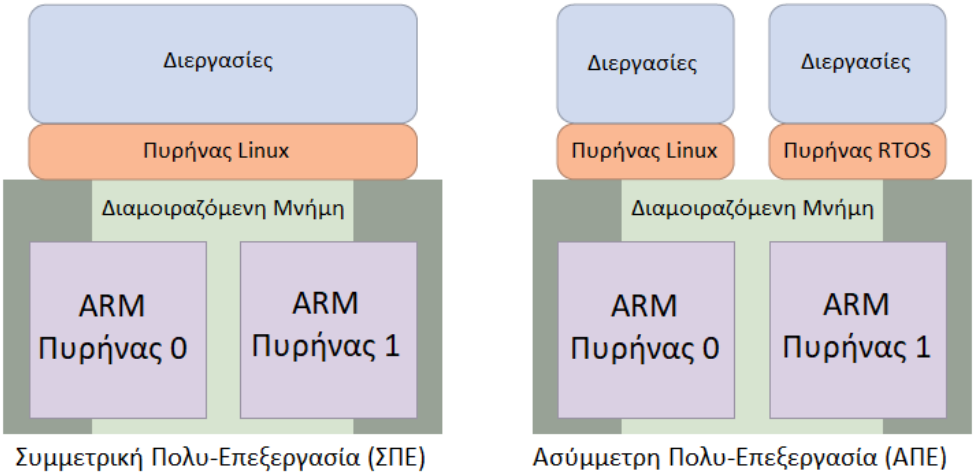
Συμμετρική και Ασύμμετρη Πολυεπεξεργασία

Η προσθήκη Λειτουργικών Συστημάτων (ΛΣ) σε ενσωματωμένα συστήματα γίνεται όλο και συχνότερη, καθώς παρουσιάζουν αρκετά οφέλη. Ωστόσο, τίθεται το ερώτημα για το ποιο είναι το καταλληλότερο είδος ΛΣ για κάθε εφαρμογή. Για παράδειγμα, μπορεί να χρησιμοποιηθεί ένα ΛΣ γενικού σκοπού, όπως το Linux, που προσφέρει ευελιξία και παραγωγικότητα στην εκτέλεση πολλαπλών εργασιών. Εάν πάλι απαιτείται προβλεψιμότητα στον χρόνο απόκρισης μίας εργασίας, προτιμάται ένα ΛΣ πραγματικού χρόνου (RTOS). Στο πλαίσιο αυτό, σε ένα πολυεπεξεργαστικό σύστημα, είναι εφικτό να χρησιμοποιηθεί είτε ένα ενιαίο ΛΣ για όλους τους επεξεργαστές είτε ξεχωριστά ΛΣ σε κάθε επεξεργαστή. Οι τεχνικές αυτές είναι γνωστές ως Συμμετρική Πολυ-Επεξεργασία (ΣΠΕ) και Ασύμμετρη Πολυ-Επεξεργασία (ΑΠΕ).

Στη ΣΠΕ, ένα μοναδικό ΛΣ δρομολογεί και συντονίζει την εκτέλεση των εργασιών ανάμεσα στους πολλαπλούς πυρήνες του πολυεπεξεργαστή. Η επικοινωνία μεταξύ αυτών επιτυγχάνεται με απλό τρόπο μέσω κοινής μνήμης. Όλοι οι μοιραζόμενοι πόροι του συστήματος διαχειρίζονται από το ΛΣ και κατανέμονται στους διάφορους πυρήνες. Σημειώνεται ότι ένα περιβάλλον ΣΠΕ συνήθως δίνει τη δυνατότητα στον σχεδιαστή να περιορίσει μία εργασία σε συγκεκριμένο πυρήνα του επεξεργαστή, εξαναγκάζοντας την εκτέλεσή της αποκλειστικά στον πυρήνα αυτόν. Η μέθοδος αυτή θα αναφέρεται στη συνέχεια ως Δεσμευμένη Πολυ-Επεξεργασία (ΔΠΕ).

Αντίθετα, σε ένα περιβάλλον ΑΠΕ, κάθε πυρήνας τρέχει το δικό του ξεχωριστό ΛΣ, που μπορεί να είναι ίδιο ή διαφορετικό από τα υπόλοιπα. Ο σχεδιαστής του συστήματος ελέγχει λοιπόν άμεσα ποιες εργασίες θα εκτελούνται σε κάθε πυρήνα και με ποιον τρόπο. Η επικοινωνία μεταξύ των πυρήνων απαιτεί σε αυτήν την περίπτωση ολοκληρωμένες δικτυακές υποδομές ή ειδικούς μηχανισμούς επικοινωνίας. Απόφαση του σχεδιαστή είναι επίσης και η κατανομή των πόρων του συστήματος μεταξύ των διαφορετικών ΛΣ. Ο καταμερισμός της μνήμης, ο έλεγχος των περιφερειακών και ο χειρισμός των διακοπών πρέπει να καθοριστεί με ακρίβεια για κάθε ΛΣ.

Η Εικόνα 2 απεικονίζει τα σχήματα της ΣΠΕ και της ΑΠΕ για τα λειτουργικά Linux και RTOS, στην περίπτωση του διπύρηνου ARM επεξεργαστή του Zynq.



Εικόνα 2: Συμμετρική και Ασύμμετρη Πολυεπεξεργασία [12]

Περιγραφή του Συστήματος

Το σύστημα αναπτύχθηκε στην πλακέτα Zybo, αξιοποιώντας τόσο το ΣΕ όσο και την ΠΛ του Zynq SoC. Αποτελείται από δύο κύριες εφαρμογές, τον πομπό και τον δέκτη, που εκτελούνται σε ξεχωριστούς πυρήνες του επεξεργαστή (ο πομπός στη CPU0 και ο δέκτης στη CPU1). Επίσης το Zybo συνδέεται με έναν υπολογιστή μέσω Ethernet, για την είσοδο και έξοδο των δεδομένων.

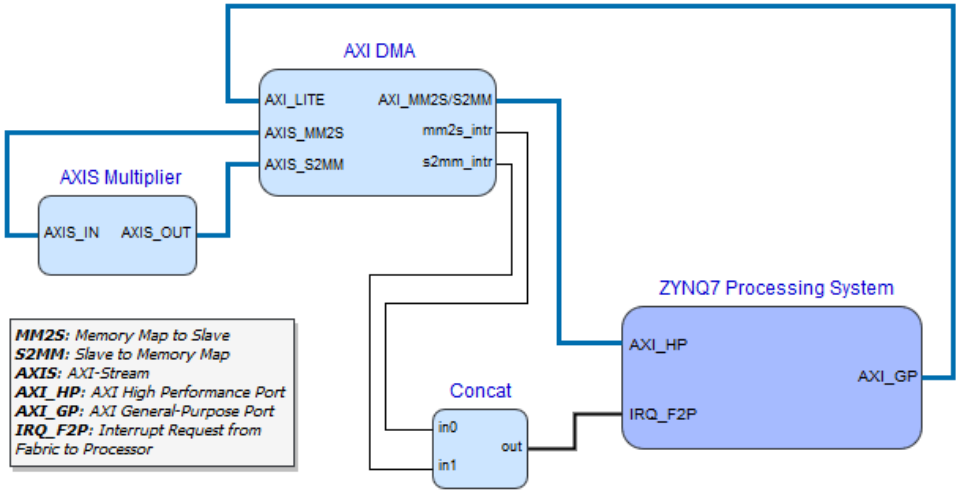
Ο πομπός δέχεται τα δεδομένα εισόδου από τον υπολογιστή μέσω Ethernet και τα αποθηκεύει στη μνήμη. Στη συνέχεια, τα δεδομένα μεταφέρονται στην ΠΛ, μέσω ΑΠΜ και με χρήση του AXI4-Stream πρωτοκόλλου, όπου υπόκεινται σε επεξεργασία στο υλισμικό. Παράλληλα, ο δέκτης από την πλευρά του μεταφέρει τα δεδομένα πίσω στη μνήμη με τον ίδιο τρόπο. Μόλις ληφθούν όλα τα δεδομένα, ο δέκτης τα προσπελάζει ώστε να τα επεξεργαστεί περαιτέρω, αυτή τη φορά με τη χρήση του επεξεργαστή. Τελικά, τα δεδομένα αποστέλλονται πίσω στον υπολογιστή μέσω Ethernet, ώστε να ελεγχθεί η εγκυρότητά τους.

Σχετικά με τα λειτουργικά συστήματα, επιχειρείται η υλοποίηση δύο διαφορετικών σχημάτων, ως διαδοχικά βήματα. Πρώτα υλοποιείται ένα σχήμα Linux ΣΠΕ και ακολούθως ένα σχήμα Linux/FreeRTOS ΑΠΕ. Λεπτομέρειες για την υλοποίηση των δύο σχημάτων, από άποψη υλισμικού και λογισμικού, δίνονται στη συνέχεια.

Σχεδίαση Υλισμικού

Ένα απλοποιημένο μπλοκ διάγραμμα του υλισμικού που χρησιμοποιήθηκε φαίνεται στην Εικόνα 3. Αποτελείται από τα εξής στοιχεία:

- Το ΣΕ του Zynq (ZYNQ7 Processing System), που περιλαμβάνει τον διπύρηνο ARM επεξεργαστή, τη μνήμη και τα περιφερειακά.
- Έναν πολλαπλασιαστή (AXIS Multiplier), υλοποιημένο μέσα στην ΠΛ, ο οποίος δέχεται και παράγει δεδομένα μέσω AXI4-Stream διεπαφών. Ο πολλαπλασιασμός αντιστοιχεί στην επεξεργασία που επιβάλλεται στα δεδομένα μεταξύ πομπού και δέκτη.
- Το AXI DMA, που βρίσκεται επίσης στην ΠΛ και είναι υπεύθυνο για την αποδοτική μεταφορά δεδομένων μέσω ΑΠΜ από τη μνήμη προς τον πολλαπλασιαστή και αντίστροφα.
- Ένα μπλοκ (Concat) το οποίο συνενώνει τα σήματα διακοπών που παράγονται από το AXI DMA σε ένα ενιαίο σήμα, ώστε να δρομολογηθεί προς την ειδική θύρα διακοπών του ΣΕ.



Εικόνα 3: Απλοποιημένο μπλοκ διάγραμμα του συστήματος

Με βάση το υλισμικό που περιγράφηκε παραπάνω, πραγματοποιήθηκε μία διερεύνηση με σκοπό τη μεγιστοποίηση του εύρους ζώνης μεταξύ μνήμης και πολλαπλασιαστή. Τροποποιώντας διάφορες παραμέτρους, μετρήθηκε κάθε φορά ο ρυθμός μεταφοράς των δεδομένων, ξεκινώντας από τα 381.2 MB/s της αρχικής σχεδίασης.

Καταρχάς, επεκτείνοντας το εύρος των διαύλων μεταφοράς δεδομένων από τα 32 bits στα 64 bits, διπλασιάστηκε ο ρυθμός. Περαιτέρω αύξηση του εύρους δεν προσφέρει βελτίωση, καθώς οι θύρα HP που χρησιμοποιεί το AXI DMA για την πρόσβαση στη μνήμη υποστηρίζει διαύλους μέγιστου

εύρους 64 bits. Μεγάλη βελτίωση επιτεύχθηκε φυσικά και με την αύξηση της συχνότητας ρολογιού, από τα 100 MHz στη μέγιστη τιμή των 250 MHz. Στη συνέχεια, χρησιμοποιήθηκαν δύο ξεχωριστά AXI DMAs για καθένα από τα κανάλια ανάγνωσης και εγγραφής της μνήμης, χωρίς ωστόσο να παρατηρηθεί κάποια βελτίωση. Ακολούθως, χρησιμοποιήθηκαν δύο ζεύγη AXI DMA-πολλαπλασιαστών, όπου κάθε ζεύγος είναι ταυτόχρονα υπεύθυνο μόνο για τα μισά δεδομένα. Έτσι, προστέθηκε παραλληλία στο σύστημα η οποία οδήγησε σε σημαντική αύξηση του ρυθμού μεταφοράς των δεδομένων. Ωστόσο, παρατηρήθηκε ότι συμφέρει η χρήση των θυρών HP0-HP2 ή HP1-HP3, λόγω του τρόπου με τον οποίον πολυπλέκονται τα μονοπάτια των τεσσάρων HP θυρών προς τη μνήμη. Τέλος, η συσχέτιση της απόδοσης με τη χρήση των πόρων οδήγησε στο συμπέρασμα ότι η χρήση περισσότερων από δύο ζευγών AXI DMA-πολλαπλασιαστών δεν είναι συμφέρουσα.

Επομένως, η τελική σχεδίαση περιλαμβάνει δύο ζεύγη AXI DMA-πολλαπλασιαστών και χρησιμοποιεί τις θύρες HP0-HP2, 64-bit εύρος διαύλων και συχνότητα 250 MHz. Το εύρος ζώνης του συστήματος σε αυτήν την περίπτωση μετρήθηκε στα 1682.8 MB/s.

Υλοποίηση Σχήματος Linux ΣΠΕ

Για την υλοποίηση του σχήματος ΣΠΕ, χρησιμοποιήθηκε ως λειτουργικό σύστημα το PetaLinux, μία έκδοση Linux της Xilinx προσαρμοσμένη για ενσωματωμένα συστήματα σε Zynq πλατφόρμες.

Το κυριότερο ζήτημα που προκύπτει αφορά το λογισμικό που χρειάζεται για τη διαχείριση του AXI DMA μέσα από το ΛΣ, δηλαδή οι απαιτούμενοι οδηγοί. Πρόκειται για ένα πρόβλημα σύνθετο, καθώς πρέπει να ληφθούν υπόψη αρκετά θέματα, όπως η χρήση ή όχι των κρυφών μνημών κατά την ΑΠΜ, η διατήρηση της συνοχής τους και η επιλογή ανάπτυξης των οδηγών στον χώρο πυρήνα ή χρήστη. Για τον λόγο αυτόν, χρησιμοποιήθηκε μία ιεραρχική δομή διαδοχικών οδηγών, αποτελούμενη από το DMA Engine framework του Linux, τον AXI DMA οδηγό της Xilinx και τη DMA Proxy προτεινόμενη σχεδίαση της Xilinx. Τα δύο πρώτα στοιχεία βρίσκονται εξ' ολοκλήρου στον χώρο πυρήνα και παρέχουν την απαιτούμενη υποδομή για τον έλεγχο του AXI DMA. Το DMA Proxy αποτελείται από έναν οδηγό στον χώρο πυρήνα που συνδέει την παραπάνω υποδομή με μία εφαρμογή στον χώρο χρήστη, δίνοντας έτσι τελικά τη δυνατότητα ελέγχου του AXI DMA από τον χώρο χρήστη. Η δεσμευμένη από τον οδηγό μνήμη για την αποθήκευση των δεδομένων μέσω ΑΠΜ αντιστοιχίζεται στον χώρο χρήστη από την εφαρμογή μέσω της συνάρτησης *mmap()*, αποφεύγοντας έτσι την αντιγραφή δεδομένων μεταξύ χώρου πυρήνα και χρήστη. Η επικοινωνία μεταξύ οδηγού και εφαρμογής πραγματοποιείται μέσω κλήσεων *ioctl()*. Επίσης, κάθε κανάλι ΑΠΜ αναπαρίσταται ως μία συσκευή που μπορεί εύκολα να χειριστεί από την εφαρμογή στον χώρο χρήστη.

Η παραπάνω εφαρμογή προσαρμόστηκε στις απαιτήσεις του επιθυμητού συστήματος, δημιουργώντας δύο ξεχωριστές αντίστοιχες εφαρμογές, τον πομπό και τον δέκτη. Ο πομπός χειρίζεται τα δύο κανάλια ΑΠΜ για την αποστολή δεδομένων και εκτελεί παράλληλα τις δύο μεταφορές από τη μνήμη προς την ΠΛ (μία για κάθε AXI DMA). Αντίστοιχα, ο δέκτης χειρίζεται τα δύο κανάλια ΑΠΜ για τη λήψη των δεδομένων και εκτελεί παράλληλα τις δύο μεταφορές από την ΠΛ προς τη μνήμη. Η εκτέλεση του πομπού και του δέκτη μπορεί να περιοριστεί αποκλειστικά στις CPU0 και CPU1 αντίστοιχα με τη χρήση του εργαλείου taskset, το οποίο δίνει τη δυνατότητα να καθορίσουμε σε ποιον επεξεργαστή θα εκτελεστεί μία δοσμένη διεργασία. Συνεπώς, εκτός του πομπού και του δέκτη, προστέθηκε ως εφαρμογή στο PetaLinux και το taskset εκτελέσιμο.

Όπως προαναφέρθηκε, το Zybo συνδέεται με έναν υπολογιστή μέσω Ethernet. Τα δεδομένα εισόδου βρίσκονται σε ένα αρχείο του υπολογιστή. Εφόσον η επικοινωνία γίνεται μέσω TCP sockets, χρειάζεται να ορίσουμε μία διεύθυνση IP για το Zybo, 192.168.1.11, και μία TCP θύρα για τη σύνδεση, 3000. Έτσι, ο πομπός δημιουργεί αρχικά το αντίστοιχο socket και λαμβάνει τα δεδομένα εισόδου, προτού τα αποθηκεύσει στη μνήμη και ξεκινήσει τις μεταφορές προς την ΠΛ μέσω ΑΠΜ. Όταν ο πομπός ολοκληρώσει τις αντίστοιχες λειτουργίες του, ανοίγει με τη σειρά του ένα socket και

αποστέλλει τα αποτελέσματα πίσω στον υπολογιστή μέσω Ethernet, όπου εγγράφονται σε ένα νέο αρχείο, το οποίο μπορεί να ελεγχθεί ώστε να επιβεβαιωθεί η ορθότητα όλης της διαδικασίας.

Υλοποίηση Σχήματος Linux/RTOS ΑΠΕ

Στο σχήμα ΑΠΕ, εφαρμόζονται τα λειτουργικά συστήματα PetaLinux στη CPU0 και FreeRTOS στη CPU1. Για την υλοποίηση του σχήματος χρησιμοποιήθηκε το OpenAMP framework, τα βασικά συστατικά του οποίου είναι το remoteproc και το RPMsg. Το remoteproc επιτρέπει σε έναν αφέντη επεξεργαστή να ελέγχει τον κύκλο ζωής ενός άλλου απομακρυσμένου επεξεργαστή σε ένα περιβάλλον ΑΠΕ (εκκίνηση, φόρτωση υλικολογισμικού, τερματισμός λειτουργίας του επεξεργαστή). Η επικοινωνία μεταξύ των επεξεργαστών επιτυγχάνεται μέσω του RPMsg. Τα συστατικά αυτά υπάρχουν έτοιμα στον πυρήνα του Linux, όμως το OpenAMP παρέχει την υλοποίησή τους και σε διαφορετικά περιβάλλοντα, όπως το FreeRTOS.

Παρακάτω δίνονται συνοπτικά τα βήματα για τη δημιουργία και εκτέλεση μίας εφαρμογής με το OpenAMP, για ένα σχήμα Linux/FreeRTOS ΑΠΕ.

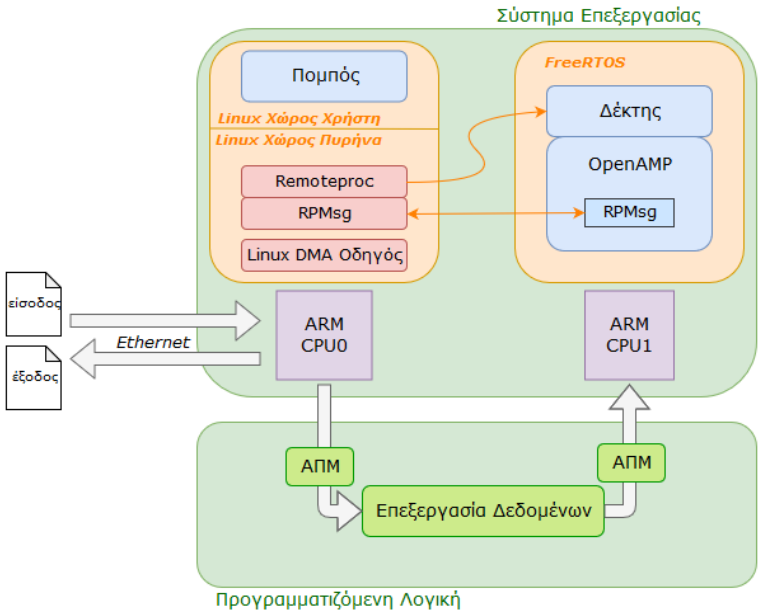
1. Καθορισμός του πίνακα πόρων. Ο πίνακας αυτός απαριθμεί τους πόρους του συστήματος που χρειάζεται ο απομακρυσμένος επεξεργαστής, ώστε αυτοί να κατανεμηθούν κατάλληλα από τον αφέντη.
2. Συγγραφή της απομακρυσμένης εφαρμογής, με χρήση των remoteproc/RPMsg εργαλείων που παρέχονται από τη βιβλιοθήκη του OpenAMP για το FreeRTOS.
3. Δημιουργία του απομακρυσμένου εκτελέσιμου, συνδέοντας την εφαρμογή με τη βιβλιοθήκη OpenAMP και τον πίνακα πόρων.
4. Τοποθέτηση του εκτελέσιμου στο σύστημα αρχείων του Linux, ώστε να είναι προσβάσιμο από τον αφέντη επεξεργαστή.
5. Φόρτωση και εκκίνηση του εκτελέσιμου στον απομακρυσμένο επεξεργαστή από τον αφέντη, με χρήση των remoteproc οδηγών. Μετά την εκκίνηση της εφαρμογής, εγκαθίσταται και ένα κανάλι RPMsg για την επικοινωνία μεταξύ των επεξεργαστών.
6. Τερματισμός λειτουργίας του απομακρυσμένου επεξεργαστή από τον αφέντη, όταν ολοκληρωθούν οι εργασίες του.

Η γενική δομή του σχήματος Linux/FreeRTOS ΑΠΕ για την εφαρμογή πομπού-δέκτη που αναπτύχθηκε νωρίτερα, πάνω στο σύστημα Zynq του Zybo, φαίνεται στην Εικόνα 4. Ο πομπός εκτελείται στη CPU0 ως αφέντης, χρησιμοποιώντας τα remoteproc/RPMsg συστατικά στον χώρο πυρήνα του Linux, ενώ ο δέκτης ως απομακρυσμένη εφαρμογή στη CPU1, με χρήση του OpenAMP framework σε περιβάλλον FreeRTOS.

Ο πομπός αρχικά ανοίγει το κανάλι RPMsg προς τον δέκτη και εγκαθιδρύει μία σύνδεση Ethernet με τον υπολογιστή, όπου βρίσκονται τα δεδομένα εισόδου. Αφού λάβει τα δεδομένα μέσω ενός socket, τα αποθηκεύει στη μνήμη και στέλνει στον δέκτη το μέγεθος των δεδομένων μέσα από το κανάλι RPMsg. Στη συνέχεια, ξεκινά τις μεταφορές των δεδομένων από τη μνήμη προς την ΠΛ μέσω ΑΠΜ, όπου γίνεται η επεξεργασία από τον πολλαπλασιαστή.

Ο δέκτης πρώτα αρχικοποιεί τα συστατικά του OpenAMP και μόλις λάβει το μέγεθος των δεδομένων από τον πομπό, ξεκινά την εκτέλεση των αντίστοιχων μεταφορών δεδομένων από την ΠΛ στη μνήμη μέσω ΑΠΜ. Μπορεί έπειτα να προσπελάσει τα ληφθέντα δεδομένα στη μνήμη για να τα επεξεργαστεί εκ νέου μέσω του επεξεργαστή. Η CPU1 στη συνέχεια παραμένει άεργη ώσπου να δεχθεί ένα μήνυμα τερματισμού από τον αφέντη, για να απενεργοποιήσει τα συστατικά του OpenAMP και να απελευθερώσει τους δεσμευμένους πόρους.

Τέλος, επειδή ο έλεγχος του Ethernet έχει αποδοθεί εξ' ολοκλήρου στο Linux, ο πομπός προσπελάζει την καθορισμένη περιοχή μνήμης, όπου έχουν αποθηκευτεί τα αποτελέσματα, και στέλνει ο ίδιος τα δεδομένα εξόδου στον υπολογιστή, χρησιμοποιώντας το ίδιο socket που δημιουργήθηκε στην αρχή.



Εικόνα 4: Η γενική εικόνα του συστήματος σε σχήμα ΑΠΕ

Κατά την υλοποίηση του συστήματος, αντιμετωπίστηκαν ορισμένα ζητήματα σχετικά με την ΑΠΕ των δύο επεξεργαστών. Το πρώτο αφορά τον συγχρονισμό μεταξύ των επεξεργαστών κατά τη ροή εκτέλεσης της εφαρμογής πομπού-δέκτη. Ο μηχανισμός που χρησιμοποιήθηκε για τον σκοπό αυτόν βασίζεται στην ανταλλαγή απλών μηνυμάτων μέσω του καναλιού RPMsg. Ειδικότερα, αυτό απαιτείται σε δύο περιπτώσεις. Πρώτον, ο πομπός πρέπει να βεβαιωθεί ότι ο δέκτης έχει ήδη αρχικοποιήσει τα AXI DMAs και είναι έτοιμος να λάβει τα δεδομένα, προτού αρχίσει να τα μεταδίδει. Εάν αυτό παραμεληθεί, παρατηρείται μία απώλεια των αρχικών δεδομένων στην πλευρά του δέκτη. Έτσι, ο δέκτης στέλνει ένα μήνυμα *'READY'* στον πομπό μόλις είναι έτοιμος και μόνο τότε μπορεί ο τελευταίος να εκκινήσει τη μεταφορά των δεδομένων. Η δεύτερη περίπτωση εμφανίζεται όταν ο πομπός πρέπει να προσπελάσει τα τελικά αποτελέσματα στη μνήμη ώστε να τα στείλει πίσω στον υπολογιστή. Ο δέκτης στέλνει ένα μήνυμα *'OK'* στον πομπό μόλις ολοκληρώσει την επεξεργασία των δεδομένων, ώστε εκείνος να γνωρίζει πότε είναι διαθέσιμα στη μνήμη τα σωστά δεδομένα.

Ένα δεύτερο ζήτημα αφορά τη συνοχή της κρυφής μνήμης, η οποία μπορεί να παραβιαστεί στο σχήμα ΑΠΕ όταν οι δύο επεξεργαστές έχουν πρόσβαση στην ίδια περιοχή της κύριας μνήμης. Στο ΣΕ του Zynq, κάθε πυρήνας έχει τη δική του κρυφή μνήμη επιπέδου L1, ενώ κανονικά μοιράζονται μία κοινή κρυφή μνήμη επιπέδου L2. Για να αποφευχθούν τυχόν προβλήματα διαμοιραζόμενων πόρων σε περιβάλλον ΑΠΕ, το OpenAMP framework απενεργοποιεί τη χρήση της μνήμης L2 από τη CPU1. Ωστόσο, καθώς ο πομπός διαβάζει τα τελικά δεδομένα από τη μνήμη για να τα στείλει στον υπολογιστή, παρατηρήθηκε ότι κάποιες τιμές δεν γίνονταν ορατές από τη CPU0. Αυτό οφείλεται στο γεγονός ότι η CPU1 γράφει τα δεδομένα απευθείας στην κύρια μνήμη, αλλά η CPU0 δεν το γνωρίζει και διαβάζει τις παλιές τιμές από τη μνήμη L2. Επομένως, απαιτείται η ανανέωση της μνήμης L2 μέσα από το λογισμικό του πομπού, για την καθορισμένη περιοχή μνήμης, ακριβώς πριν εκείνος διαβάσει τα δεδομένα εξόδου.

Αξιολόγηση της Απομόνωσης του Συστήματος

Ένα σημαντικό χαρακτηριστικό του ανεπτυγμένου συστήματος που χρειάζεται να αξιολογηθεί είναι το επίπεδο απομόνωσης που επιτυγχάνεται μεταξύ των δύο πυρήνων ARM του ΣΕ του Zynq. Ο πομπός και ο δέκτης αναπαριστούν δύο ανεξάρτητες οντότητες που πρέπει να εκτελούν τις λειτουργίες τους ανεπηρέαστα η μία από την άλλη. Εξετάζουμε λοιπόν στη συνέχεια πώς οι εργασίες που εκτελεί

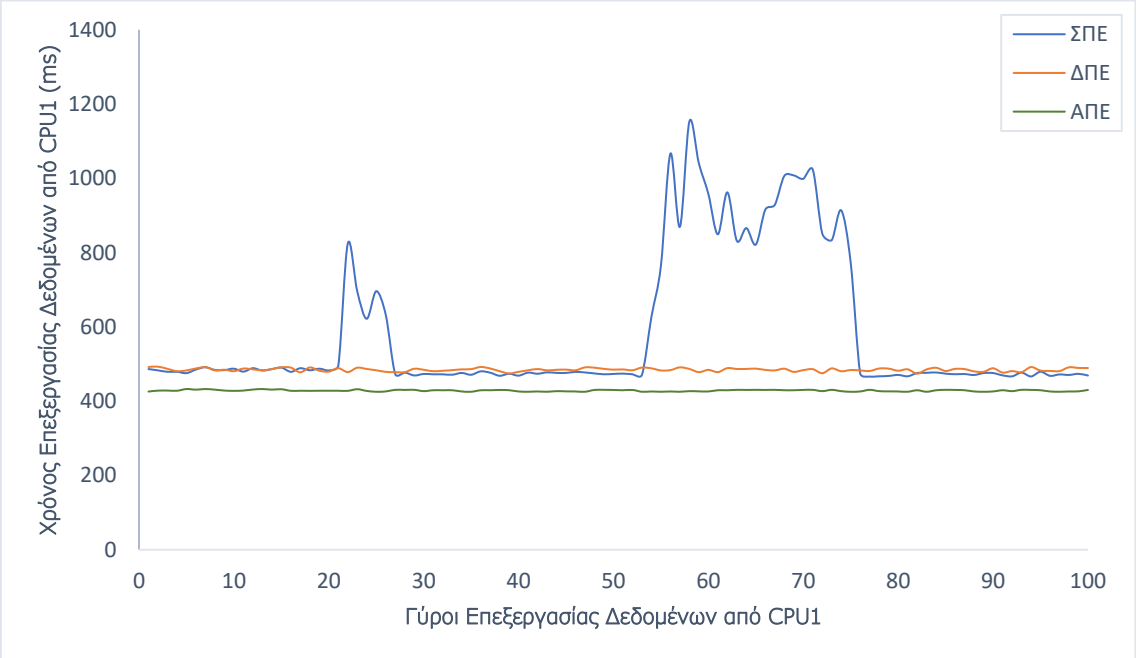
ο πομπός επηρεάζουν τη συμπεριφορά του δέκτη, ο οποίος εκτελεί ταυτόχρονα μία κρίσιμη εργασία, για καθένα από τα σχήματα ΣΠΕ (κλασική ΣΠΕ και ΔΠΕ) και ΑΠΕ.

Παρεμβολή μεταξύ διεργασιών

Σε πολυπύρηνες αρχιτεκτονικές, η απομόνωση κρίσιμων εργασιών είναι επιτακτική, ειδικά όταν εμπλέκεται συμπεριφορά πραγματικού χρόνου. Ένας βαρύς φόρτος εργασίας μπορεί να προκαλέσει παρεμβολές μεταξύ εντατικών ως προς τον επεξεργαστή διεργασιών και χρονικά κρίσιμων διεργασιών, σε περίπτωση μη καλής απομόνωσης του συστήματος. Επιπλέον, οι διάφορες διεργασίες ενδέχεται να υποστούν παρεμβολές όταν χρησιμοποιούν ταυτόχρονα κοινούς πόρους του συστήματος. Στο πλαίσιο αυτό λοιπόν διεξήχθη ένα πείραμα που διερευνά δύο διαφορετικά σενάρια παρεμβολών στο σύστημά μας, τις οποίες παράγει μία διεργασία εντατική είτε ως προς τον επεξεργαστή είτε ως προς τη μνήμη.

Στο πρώτο σενάριο, ένα πακέτο μεγέθους 4 MB μεταφέρεται από τον πομπό στον δέκτη. Ο δέκτης επεξεργάζεται τα ληφθέντα δεδομένα, προσπελάζοντάς τα σε διαδοχικούς γύρους (1-100) και εκτελώντας διάφορες συγκρίσεις και μαθηματικές πράξεις (πολλαπλασιασμούς και διαιρέσεις). Η διαδικασία αυτή αντιστοιχεί στη χρονικά κρίσιμη εργασία του συστήματος. Μετά την παρέλευση ενός αρχικού χρονικού διαστήματος, ο πομπός αρχίζει παράλληλα να εκτελεί μία παρόμοια εργασία, εντατική ως προς τον επεξεργαστή, πάνω σε ανεξάρτητα δεδομένα. Σε πρώτη φάση δημιουργεί δύο νήματα που το καθένα εκτελεί την παραπάνω εργασία και αργότερα επαναλαμβάνει το ίδιο με πέντε νήματα. Ο χρόνος που χρειάστηκε ο δέκτης για την ολοκλήρωση κάθε γύρου επεξεργασίας μετρήθηκε ώστε να ποσοτικοποιηθεί η παρεμβολή και να αξιολογηθεί η απομόνωση.

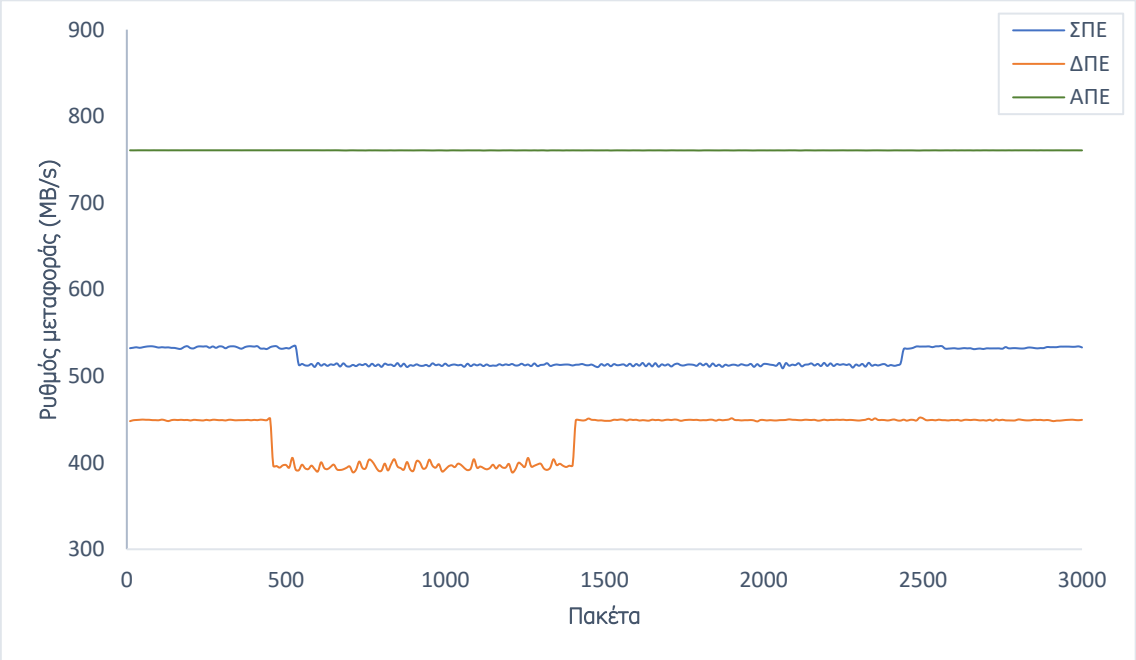
Η Εικόνα 5 απεικονίζει τα αποτελέσματα από την πραγματοποίηση του παραπάνω σεναρίου, για όλα τα σχήματα πολυεπεξεργασίας. Είναι εμφανές ότι υπάρχει πράγματι παρεμβολή στην περίπτωση της ΣΠΕ, η οποία είναι μάλιστα ανάλογη του βάρους του επεξεργαστικού φόρτου που επιβάλλεται στο σύστημα. Αυτό οφείλεται στην ελεύθερη μετακίνηση των νημάτων του φόρτου από τη CPU0 στη CPU1, με αποτέλεσμα να αναστέλλεται η λειτουργία του δέκτη και να αυξάνεται ο χρόνος εκτέλεσης της κρίσιμης εργασίας. Αντίθετα, σε περιβάλλον ΔΠΕ ή ΑΠΕ δεν παρατηρούνται παρεμβολές και η χρονική συμπεριφορά της κρίσιμης εργασίας είναι ικανοποιητική.



Εικόνα 5: Παρεμβολή κρίσιμου χρόνου επεξεργασίας από εργασία εντατική ως προς τον επεξεργαστή

Στο δεύτερο σενάριο, ένα σύνολο από 3000 πακέτα μεταφέρεται μεταξύ πομπού και δέκτη. Με αυτόν τον τρόπο, εισάγεται μεγάλη κίνηση δεδομένων μεταξύ ΣΕ και κύριας μνήμης. Κάποια στιγμή, ο πομπός ξεκινά παράλληλα την εκτέλεση μίας εργασίας εντατικής ως προς τη μνήμη, πραγματοποιώντας πολλαπλές εγγραφές και αναγνώσεις από τη μνήμη. Μετρώντας τον χρόνο μεταφοράς για κάθε 10 πακέτα, υπολογίστηκε η μεταβολή του ρυθμού μεταφοράς των δεδομένων κατά τη διάρκεια του πειράματος, όπως φαίνεται στην Εικόνα 6.

Παρατηρούμε ότι οι περιπτώσεις ΣΠΕ και ΔΠΕ υποφέρουν από παρεμβολή, αφού ο ρυθμός μειώνεται σημαντικά κατά τη διάρκεια εκτέλεσης της παρεμβληθείσας εργασίας, σε αντίθεση με την ΑΠΕ όπου ο ρυθμός παραμένει πρακτικά αναλλοίωτος. Αυτό μας οδηγεί στην υπόθεση ότι ο πόρος που τελικά ευθύνεται για την παρεμβολή είναι η κρυφή μνήμη L2, η οποία είναι διαμοιραζόμενη μεταξύ των δύο πυρήνων όταν τρέχουν στο ίδιο ΛΣ, με αποτέλεσμα να δημιουργούνται συγκρούσεις μεταξύ των διεργασιών. Επιπλέον, είναι απαραίτητα και πρωτόκολλα διατήρησης της συνοχής της κρυφής μνήμης, τα οποία ενδέχεται να παράγουν επιπρόσθετη κίνηση δεδομένων και να ενισχύουν τις παρεμβολές. Στην ΑΠΕ, αντιθέτως, η κρυφή μνήμη L2 δεν χρησιμοποιείται από τη CPU1, επιτρέποντας στον δέκτη να προσπελάζει τη μνήμη ανεπηρέαστα και με σταθερό ρυθμό.



Εικόνα 6: Παρεμβολή ρυθμού μεταφοράς από εργασία εντατική ως προς τη μνήμη

Ασφάλεια και ακεραιότητα δεδομένων

Εκτός από την αποφυγή παρεμβολών στη χρονική απόκριση των διεργασιών, η απομόνωση είναι εξίσου σημαντική ώστε να διασφαλίζεται ότι ένα τυχόν λάθος ή πρόβλημα σε μία εργασία δεν επηρεάζει τη σωστή λειτουργία άλλων εργασιών. Σε γενικές γραμμές, η ΑΠΕ παρέχει ένα επιπλέον στρώμα προστασίας από τέτοια συμβάντα, διατηρώντας τα λειτουργικά συστήματα διαχωρισμένα σε ξεχωριστούς επεξεργαστές.

Η απομόνωση πρέπει επίσης να εγγυάται την ακεραιότητα των δεδομένων. Διεξήγαμε ένα σχετικό πείραμα στο σύστημά μας, με το εξής σενάριο: ο πομπός εξαναγκάστηκε να προσπελάσει στη μνήμη τα ληφθέντα από τον δέκτη δεδομένα και να μηδενίσει όλες τις τιμές, ενόσω ο δέκτης τις επεξεργαζόταν. Παρατηρήθηκε ότι αυτό κατέστη εφικτό ανεξαρτήτως εφαρμοσμένου σχήματος πολυεπεξεργασίας. Το ενδιαφέρον συμπέρασμα εδώ είναι ότι ακόμη και η ΑΠΕ δεν μπορεί να αποτρέψει τέτοιου είδους αλλοίωση της μνήμης μεταξύ διεργασιών, και συνεπώς απαιτείται ένας ισχυρότερος προστατευτικός μηχανισμός.

Η ακεραιότητα των δεδομένων σχετίζεται επίσης με τη χρήση των κρυφών μνημών. Τα πρωτόκολλα συνοχής και διόρθωσης λαθών εγγυώνται τη διατήρηση της ακεραιότητας, όμως ενδέχεται παράλληλα να επιβαρύνουν τη χρονική απόδοση του συστήματος. Στη ΣΠΕ, αυτά είναι αναγκαία για την ορθή χρήση της κρυφής μνήμης L2 από τους δύο πυρήνες. Ωστόσο, στην ΑΠΕ η CPU1 απομονώνεται από τα θέματα ασφαλείας της κρυφής μνήμης L2. Παρ' όλα αυτά, εξακολουθούν να υπάρχουν κίνδυνοι όταν οι δύο πυρήνες έχουν πρόσβαση στην ίδια περιοχή της κύριας μνήμης, όπως παρατηρήθηκε νωρίτερα κατά την υλοποίηση του σχήματος ΑΠΕ. Επομένως, ιδιαίτερη προσοχή χρειάζεται όταν σε ένα σύστημα διαμοιραζόμενης μνήμης απομονώνεται μία ή περισσότερες κρυφές μνήμες από κάποιον επεξεργαστή.

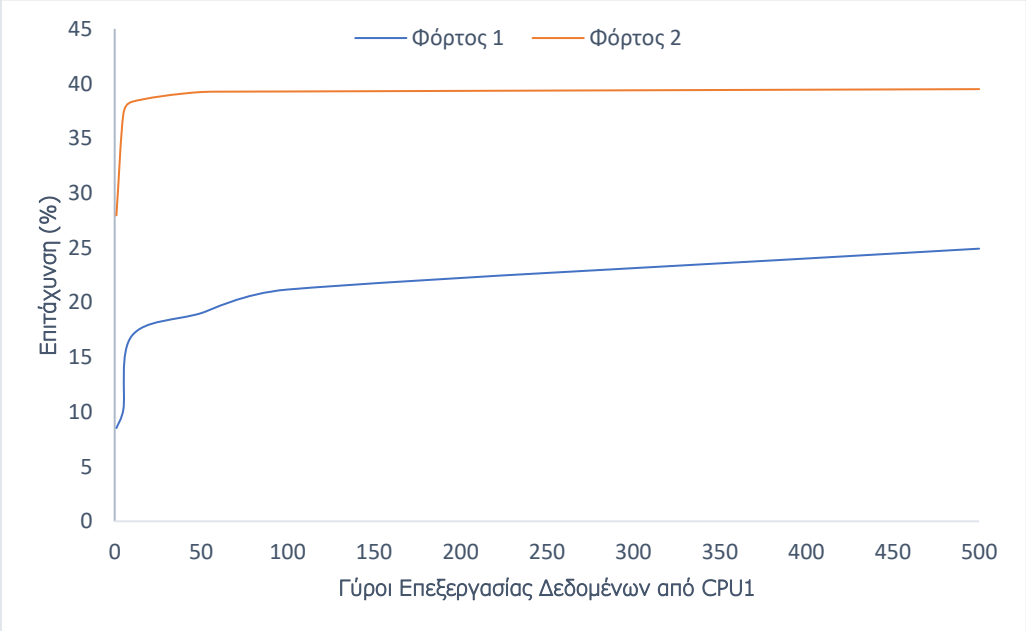
Αξιολόγηση της Επίδοσης του Συστήματος

Τα σχήματα Linux ΣΠΕ και Linux/RTOS ΑΠΕ παρουσιάζουν ορισμένες διαφοροποιήσεις αναφορικά με την επίδοση του συστήματος, οι οποίες πηγάζουν από την ίδια τη φύση των ΛΣ που χρησιμοποιούνται σε κάθε περίπτωση. Τα ακόλουθα πειράματα αξιολογούν ορισμένα χαρακτηριστικά του συστήματος σε διάφορα σενάρια, συγκρίνοντας την επίδοση μεταξύ PetaLinux και FreeRTOS.

Ταχύτητα επεξεργασίας

Ένα πακέτο 4 MB μεταφέρεται από τον πομπό στον δέκτη, ο οποίος στη συνέχεια επεξεργάζεται εντατικά τα ληφθέντα δεδομένα. Αυτό γίνεται μέσω ενός επεξεργαστικού φόρτου που περιλαμβάνει κυρίως πολλαπλασιασμούς και διαιρέσεις πάνω στα δεδομένα και επαναλαμβάνεται για έναν μέγιστο αριθμό διαδοχικών γύρων. Μετρήθηκε ο χρόνος που χρειάζεται ο δέκτης για να ολοκληρώσει την επεξεργασία, αυξάνοντας κάθε φορά το μέγιστο όριο των γύρων επεξεργασίας.

Τα συγκριτικά αποτελέσματα απεικονίζονται στην Εικόνα 7 (Φόρτος 1), για τα Linux και RTOS λειτουργικά. Παρατηρούμε ότι η ταχύτητα επεξεργασίας δεδομένων από τη CPU1 σε RTOS είναι σημαντικά αυξημένη συγκριτικά με το Linux, αγγίζοντας σταδιακά μία επιτάχυνση της τάξης του 25%. Αυτό μπορεί να εξηγηθεί αν αναλογιστούμε ότι μία διεργασία σε RTOS επηρεάζεται λιγότερο από τη λειτουργία του ΛΣ σε σχέση με το Linux, ενώ δίνεται επιπλέον πλήρης προτεραιότητα στην εκτέλεσή της έναντι άλλων διεργασιών.



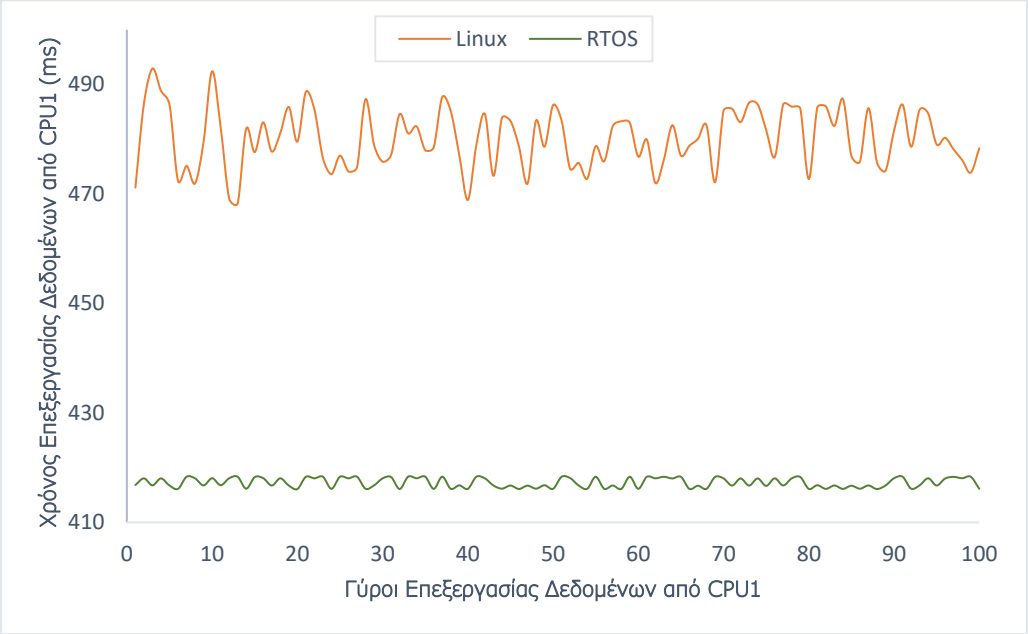
Εικόνα 7: Επιτάχυνση στην επεξεργασία δεδομένων σε RTOS έναντι Linux

Το προηγούμενο σενάριο επαναλήφθηκε μία ακόμη φορά, αφού τροποποιήθηκε ελαφρώς ο ίδιος ο φόρτος εργασίας: αντί για πολλαπλασιασμούς και διαιρέσεις, ο δέκτης πραγματοποιεί τώρα προσθέσεις και αφαιρέσεις στα ληφθέντα δεδομένα. Η επιτάχυνση που επιτυγχάνει το RTOS έναντι του Linux φαίνεται στην Εικόνα 7 (Φόρτος 2), όπου πλέον το ποσοστό φθάνει στη μέγιστη τιμή 40%. Επομένως, το είδος της επεξεργασίας επιδρά σημαντικά στη βέλτιστη χρήση του συστήματος και κατ' επέκταση στη βελτίωση της απόδοσης.

Διακύμανση χρόνου εκτέλεσης

Το επόμενο πείραμα εξετάζει τη διακύμανση του χρόνου εκτέλεσης της επεξεργασίας των δεδομένων από τη CPU1, ανάλογα με το υπάρχον ΛΣ. Ένα πακέτο 4 MB μεταδίδεται από τον πομπό και επεξεργάζεται από τον δέκτη, για 100 διαδοχικούς γύρους. Οι απαιτούμενοι χρόνοι που μετρήθηκαν για την ολοκλήρωση κάθε γύρου επεξεργασίας φαίνονται στην Εικόνα 8.

Είναι εμφανές ότι ο χρόνος εκτέλεσης κάθε γύρου από τη CPU1 εμφανίζει πολύ μεγαλύτερη διακύμανση σε περιβάλλον Linux απ' ότι σε RTOS. Το γεγονός αυτό οφείλεται στη ντετερμινιστική συμπεριφορά του RTOS που ελαχιστοποιεί τη διαφορά μεταξύ ελάχιστου και μέγιστου χρόνου απόκρισης μίας διεργασίας. Αυτό είναι βέβαια εξαιρετικά σημαντικό σε εφαρμογές που χαρακτηρίζονται από αυστηρές απαιτήσεις πραγματικού χρόνου.



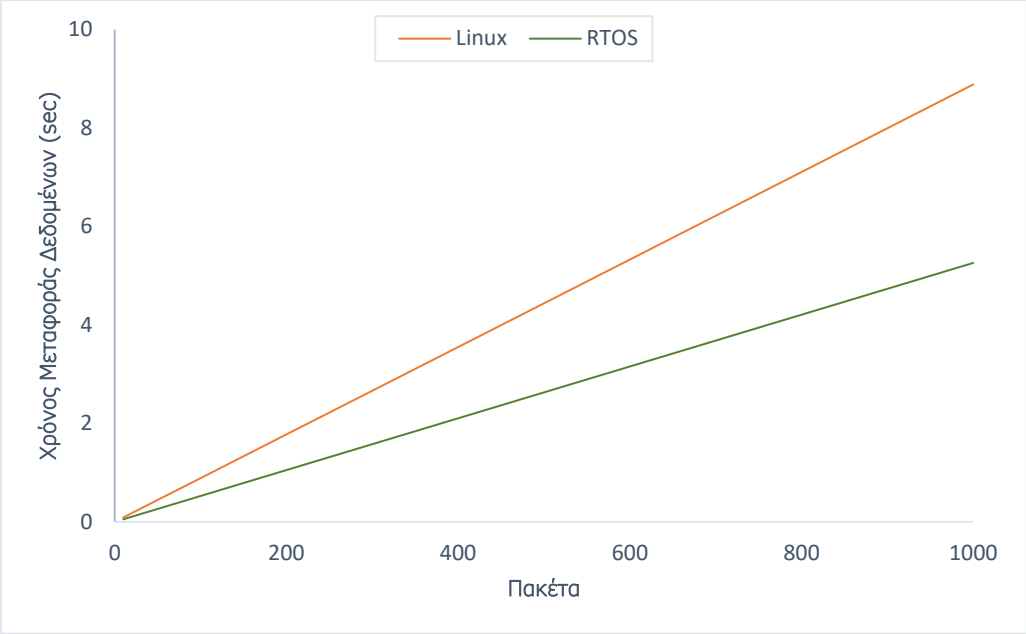
Εικόνα 8: Διακύμανση χρόνου επεξεργασίας από CPU1 σε Linux και RTOS

Ταχύτητα μεταφοράς δεδομένων

Το τελευταίο σενάριο δημιουργεί εντατική κίνηση δεδομένων από τον πομπό προς τον δέκτη. Ένας σταδιακά αυξανόμενος αριθμός πακέτων, καθένα μεγέθους 4 MB, μεταφέρεται μέσω ΑΠΜ από τη μνήμη στην ΠΛ και αντίστροφα. Μετρήθηκε ο απαιτούμενος χρόνος για την ολοκλήρωση της μεταφοράς όλων των πακέτων, ξεκινώντας από τα 10 πακέτα και φθάνοντας μέχρι τα 1000 πακέτα.

Η απεικόνιση των μετρήσεων γίνεται στην Εικόνα 9, για τις περιπτώσεις Linux και RTOS. Παρατηρούμε ότι η μεταφορά των δεδομένων με χρήση του RTOS είναι εμφανέστατα ταχύτερη συγκριτικά με την αποκλειστική χρήση Linux, ανεξαρτήτως του συνολικού αριθμού πακέτων. Η αιτία πρέπει να αναζητηθεί στην πλευρά του δέκτη και στον τρόπο με το οποίο μεταφέρονται τα δεδομένα μέσω ΑΠΜ. Ειδικότερα, στο Linux η εφαρμογή του δέκτη χειρίζεται τα AXI DMAs από τον χώρο χρήστη μέσω μίας ιεραρχίας ειδικών οδηγών στον χώρο πυρήνα (σχεδίαση DMA Proxy). Οι οδηγοί

αυτοί μαζί με το ίδιο το ΛΣ εισάγουν καθυστερήσεις στη διαχείριση των AXI DMAs. Στο RTOS, όμως, η επιβάρυνση αυτή είναι πολύ χαμηλότερη αφού ο δέκτης προγραμματίζει απευθείας τα AXI DMAs, χωρίς ενδιάμεσα στρώματα λογισμικού, με αποτέλεσμα ο χρόνος μεταφοράς των δεδομένων να είναι μικρότερος.



Εικόνα 9: Χρόνος μεταφοράς δεδομένων σε Linux και RTOS

Συμπεράσματα

Τα SoC FPGAs είναι ιδανικά για την ανάπτυξη και δοκιμή εφαρμογών, όπως αυτή που υλοποιήθηκε στην παρούσα εργασία. Εφαρμόζοντας μία σχεδίαση υλισμικού/λογισμικού στο Zynq SoC και αξιοποιώντας τον διπύρηνο ARM επεξεργαστή, υλοποιήσαμε μία υποδομή κατάλληλη για τηλεπικοινωνιακές εφαρμογές.

Αποδείχθηκε ότι μία διερεύνηση του υλισμικού και των δυνατοτήτων του μπορεί να βελτιώσει σε σημαντικό βαθμό το εύρος ζώνης του συστήματος. Επιπλέον, από άποψη λειτουργικών συστημάτων, μελετήθηκαν και αξιολογήθηκαν τα σχήματα ΣΠΕ (Linux) και ΑΠΕ (Linux/RTOS), ως προς την απομόνωση και την επίδοση του συστήματος.

Συμπερασματικά, η ΑΠΕ προσφέρει καλύτερη απομόνωση, περιορίζοντας τις παρεμβολές μεταξύ διεργασιών και διασφαλίζοντας υπό ορισμένες συνθήκες την αξιοπιστία και την ακεραιότητα των δεδομένων. Επίσης, η χρήση RTOS ως λειτουργικό σύστημα είναι αποδοτικότερη συγκριτικά με το Linux, ως προς την ταχύτητα επεξεργασίας και μεταφοράς δεδομένων, καθώς και την προβλεψιμότητα εκτέλεσης των διεργασιών.

Μελλοντική Εργασία

Ως επέκταση της παρούσας εργασίας, προτείνεται η διαμόρφωση του ανεπτυγμένου συστήματος προς την κατεύθυνση μίας εφαρμογής του πραγματικού κόσμου. Ο πολλαπλασιαστής στην ΠΛ μπορεί εύκολα να αντικατασταθεί από ένα άλλο στοιχείο, συμβατό με το AXI4-Stream πρωτόκολλο, που θα επεξεργάζεται τα δεδομένα σύμφωνα με έναν πραγματικό αλγόριθμο. Η επιπρόσθετη

επεξεργασία από τον επεξεργαστή μπορεί επίσης να προσαρμοστεί στις απαιτήσεις της εφαρμογής, ενώ και τα ίδια τα δεδομένα εισόδου θα μπορούσαν να προέρχονται από μία πραγματική πηγή.

Επιπλέον, μία ιδέα για μελλοντική εργασία είναι η υλοποίηση της ίδιας εφαρμογής σε άλλη πλατφόρμα με διαφορετικά χαρακτηριστικά από το Zynq. Συγκεκριμένα, η Zynq UltraScale+ MPSoC συσκευή της Xilinx θα μπορούσε να εξεταστεί, καθώς συνδυάζει επεξεργαστές γενικού σκοπού (ARM Cortex-A53) με επεξεργαστές πραγματικού χρόνου (ARM Cortex-R5), επιτρέποντας πραγματική ετερογενή πολυεπεξεργασία.

Τέλος, ενδιαφέρον θα είχε η μελέτη τεχνολογιών εικονικοποίησης ως εναλλακτική τεχνική απομόνωσης της ΑΠΕ. Αντί για τον διαχωρισμό των ΛΣ στους δύο πυρήνες του επεξεργαστή, θα μπορούσε να χρησιμοποιηθεί ένας επόπτης (hypervisor) που θα διαχειρίζεται πολλαπλά εικονικά περιβάλλοντα στο ίδιο σύστημα. Η αξιολόγηση ενός τέτοιου σχήματος, αναφορικά με την απομόνωση και την επίδοση του συστήματος, και η σύγκριση με τα προαναφερθέντα σχήματα θα ήταν ιδιαίτερα χρήσιμη.

Chapter 1

Introduction

1.1 SoC FPGAs

Recently, there has been a significant technology trend towards a hardware-based approach to digital design. In contrast to traditional general-purpose processors and microcontrollers, where functionality is provided by sequential instructions programmed in software, FPGAs (Field Programmable Gate Arrays) have been increasingly used to provide solutions in a wide range of applications.

An FPGA is a semiconductor device containing an array of configurable logic blocks connected through programmable interconnects [1]. These elements can be programmed to perform merely simple logic gates, complex functions or even blocks of memory that can be interconnected to form configurations with different functionalities. The key aspect about FPGAs is that they can be reconfigured after manufacturing, providing a flexible way to meet any desired application requirements. The following figure presents the basic FPGA architecture, outlining the fundamental components of the device [2]:

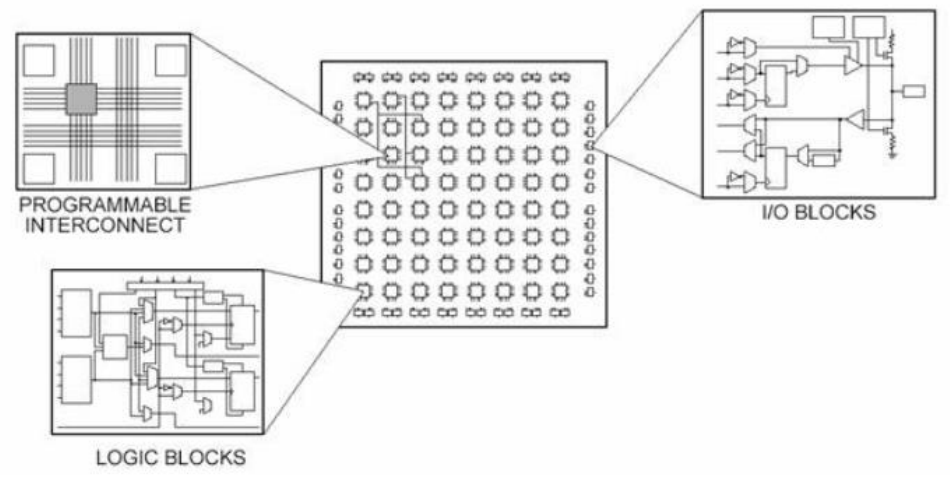


Figure 1.1: The main components of the FPGA architecture [3]

- Configurable Logic Blocks (CLBs), which consist of Look-Up Tables (LUTs), multiplexers and storage elements. The logic blocks are laid out in an array-type architecture and they implement most of the logic in the FPGA, including Boolean, arithmetic and data-storage operations.

- Input-Output Blocks (IOBs), which provide the interface between internal logic and external package pins. They consist of components such as pull-up/pull-down resistors, buffers and inverters.

- Programmable Interconnects, which implement internal connections between CLBs as well as I/O and global routing. They are composed of metal segments with programmable switching matrices to achieve desired routing.

An FPGA is programmed using a Hardware Description Language (HDL), such as Verilog or VHDL. Design automation tools are responsible for mapping a hardware design created with HDL code to a specific device and generating the bitstream that configures appropriately the FPGA programmable logic. Of course, the design flow is simplified and significantly accelerated with the use of predefined complex functions and circuits, available from FPGA vendors and third-party suppliers, known as IP (Intellectual Property) cores.

FPGAs present various benefits, both for designers and customers [4]:

- Flexibility. FPGA functionality can be reconfigured after manufacturing, to meet different application requirements.

- Acceleration. FPGAs offer a high level of parallelism and can complete tasks faster in hardware, speeding up the system performance.

- Simple design cycle. FPGAs can be easily programmed, while software tools handle complex placing, routing and timing procedures.

- Shorter time to market. Compared to ASICs (Application-Specific Integrated Circuits), FPGAs can be available faster to the market, since application development is faster on FPGAs rather than ASICs.

On the other hand, there are still some drawbacks involved in FPGA design, that should certainly be considered when such a solution is examined. Specifically, FPGA programming is generally considered more difficult than CPU programming, since it requires knowledge of HDL and simulation tools, making development time higher. FPGAs also consume more power than ASICs and provide limited resources depending on the device being used [4].

The application areas where FPGAs can provide effective solutions are numerous, including high performance computing, digital signal processing, hardware acceleration, video and image processing, automotive, telecommunications, aerospace, cloud computing, data centers, machine learning, consumer electronics and many others.

A recent trend has been to integrate the programmable logic of the FPGA with embedded microprocessors and related peripherals on the same device, to form a powerful computing platform, known as SoC (System-on-Chip) FPGA. This hybrid technology combines the high-level management functionality of processors and the hardware programmability of an FPGA, making SoC FPGAs an ideal fit to various applications. In addition, functionality is further extended by a rich set of dedicated peripherals, such as ADCs/DACs, high-speed transceivers, on-chip memory and various interfaces [5].

Apart from the benefits mentioned before for traditional FPGAs, SoC FPGAs present some additional advantages. These include lower communication overhead between different units, improved power consumption and smaller size. Also, SoC FPGAs introduce an efficient hardware/software co-design principle, where different operations of a task can be allocated either to CPU processing or dedicated FPGA logic, leading to increased overall performance.

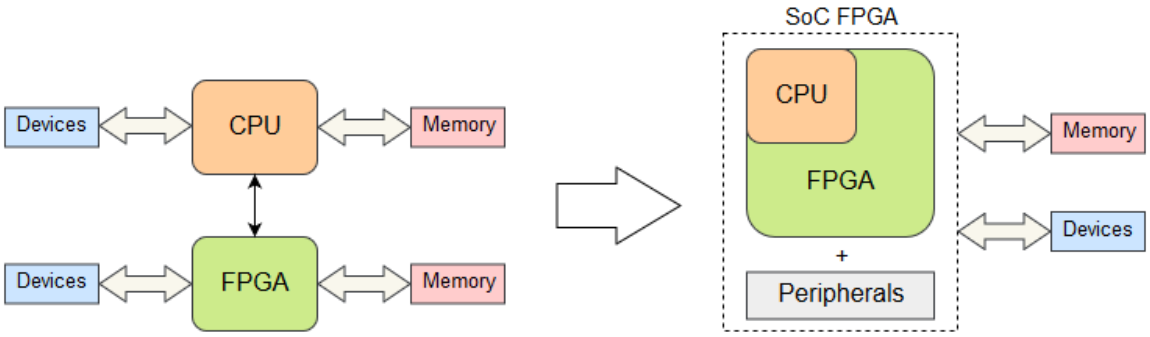


Figure 1.2: Transition from standalone CPU and FPGA to SoC FPGA

1.2 Multicore Architectures

Multicore architectures have become a fundamental concept in processor design. In an attempt to increase performance, the processor industry switched from traditional single-core processors to multicore processors, where more than one independent processing units (or cores) are integrated onto a single chip. This significant shift was mainly motivated by the power constraints that arised when trying to make single-core processors more powerful, which eventually became intolerable. According to the processor performance equation:

$$P = \frac{1}{T} = \frac{IPC \times f}{IC}$$

where P is the performance of the machine, T is the execution time for a specific task, IC is the number of instructions executed, IPC is the instructions per clock cycle ratio and f is the clock frequency. Increasing the clock frequency is related to the power constraints mentioned above, leading to higher power consumption and heat generation. The instruction count, on the other side, is determined by the program, the compiler and the processor's ISA (Instruction Set Architecture). This leaves the IPC factor to be optimized with changes to the architecture. Indeed, multicore processors can run multiple instructions on separate cores in parallel, increasing the IPC ratio and consequently performance as well. It is important to note that multiprocessing can be combined with parallelization techniques applied to single-core systems, such as simultaneous multithreading and superscalar pipelining, leading to enhanced overall performance [6].

In a typical multiprocessor structure, each core executes its own instruction stream. The instructions may belong to different processes, where processors run independent tasks. However, it is possible for multiple cores of a multiprocessor to run separate threads of a single program. In that case, they share code and most of their address space. The multiple cores may also share some resources, such as cache memories and I/O buses, depending on the level of memory centralization and the interconnection strategy between nodes. Communication and data exchange among processors can be achieved through a shared address space, message passing or any other communication mechanism [7].

Obviously, parallel programming is essential in order to benefit from the parallelism offered by multicore processors. Software programmers need to make their code parallel and tune it appropriately to fit well with the targeted architecture. In particular, they should examine the nature of the processing and allocate the work efficiently among all available cores. This often introduces subtle issues that should be definitely taken into consideration, like synchronization and memory consistency. Synchronization is defined as a mechanism which ensures that concurrent processes do not access simultaneously a critical section or that they perform actions in a specific desired

order. Memory consistency enforces rules on memory operations to maintain memory coherence and data integrity among multiple processors.

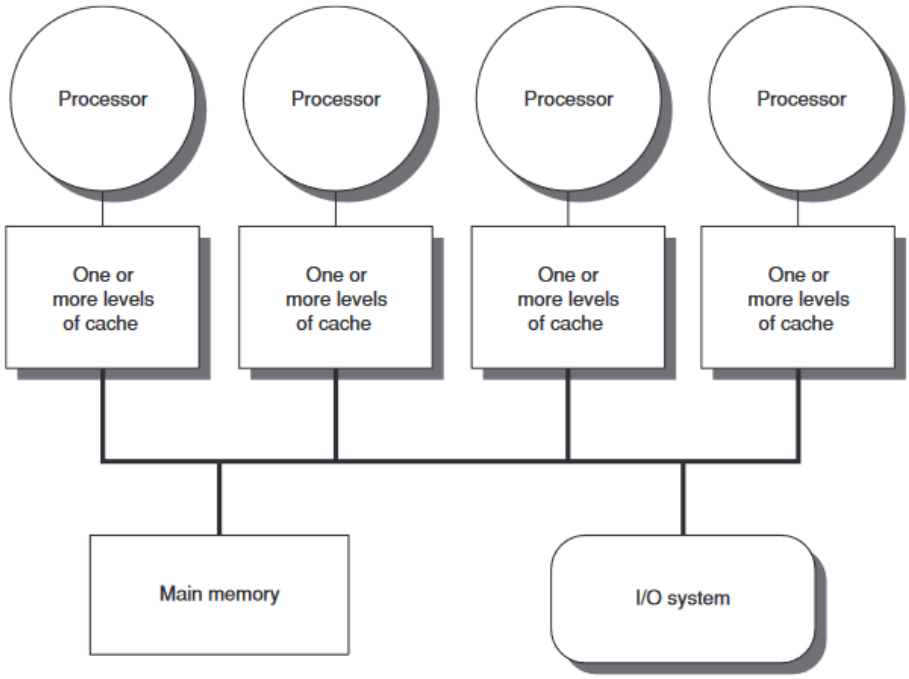


Figure 1.3: A typical structure of a centralized shared-memory multiprocessor [7]

A recent trend in multicore processor development has been to include heterogeneous cores in the same system. These cores are not identical, but they incorporate specialized processing capabilities that make them suitable to handle special-purpose applications. In such a system, it is common to offload a specific task to a dedicated core, which is optimized for that particular type of processing, while another one is responsible for other general-purpose operations. Heterogeneous multicore processing can be extremely beneficial for a number of reasons, including performance optimization, power consumption reduction and system reliability improvement [8].

Whether heterogeneous or not, multicore processors have proven to be a good solution for many applications, such as multimedia and digital signal processing, networking, telecommunications and various embedded systems. In many cases, more than one Operating System (OS) run on a multicore architecture, a technique known as Asymmetric Multi-Processing (AMP). A typical example is when one core requires hard real-time constraints, so it runs a Real-Time Operating System (RTOS). This individual core performs real-time processing to meet tight time requirements, while other cores can run a general-purpose OS to handle other tasks more loosely.

It is worth mentioning that a critical issue involved in multicore processors is isolation between separate cores. In case these are responsible for independent tasks, it should be ensured that they are isolated as much as possible, to meet real-time constraints, ensure security and achieve maximum performance. Potential interference between different cores may lead to security vulnerabilities or delays that can be of great importance. AMP is a technique that can increase the isolation level between cores that run separate operating systems.

1.3 Thesis Purpose and Outline

The purpose of this thesis is to design and evaluate a general testbed for a telecom application requiring efficient data streaming and processing, on a dual-core SoC FPGA. The processor is used to provide both high-level functions and dedicated processing on data. The programmable logic of the FPGA is also exploited, to perform efficient data processing in hardware. We attempt to optimize the system both from hardware and software perspectives, in terms of isolation and performance, and test it using an application that emulates the basic behavior of a real-life application.

The application consists of two main entities, the data transmitter and the data receiver, that should run on separate cores of the multiprocessor. The transmitter initiates the transmission of input data, which are first sent to the FPGA where they are subject to an initial processing phase. They are then retrieved by the receiver which is responsible for a final special-purpose processing function, for example a real-time operation.

The motivation for this thesis is that creating a testbed as described above, where we do not implement specific DSP algorithms or define strict specifications for the type of processing, makes it adaptable and very useful for testing various applications. By keeping a high abstraction level, the testbed can be easily modified appropriately to match the requirements of each case. In a telecom application, the streaming of data corresponds to the transmission of signals and the data processing to various processes done between the transmitter and receiver sides, like modulation, multiplexing and filtering. Another similar application is when a sensor generates a continuous sequence of measurements, which need to be processed appropriately before the results are stored for later analysis. A LIDAR system, where reflected laser pulses are measured to scan a target surface and make a 3D representation, can be mentioned as an example. If this is applied in automotive, for instance, critical real-time processing on data is required as well. In addition, it is useful that the system also provides an interactive interface to users, without interrupting the data processing and affecting the response time of the system.

Some issues that are considered throughout this thesis are how to exploit available hardware resources, how to achieve efficient communication between hardware and software, which operating systems to use and in what configuration, how to process data in software and how to isolate the two processing cores. Optimizing these parameters aims to improve isolation between the executed tasks and increase the performance of the critical real-time processing task.

A brief outline of the following chapters is given below:

- In chapter 2, we present basic concepts about the device and some of the techniques that are used, as well as a system-level description of the testbed.

- In chapter 3, the implementation of the system is described in detail, both from a hardware and software point of view.

- In chapter 4, an evaluation of the system is done, based on the results acquired from different experiments.

- Finally, chapter 5 includes the thesis summary and provides some ideas for potential extensions and future work.

Chapter 2

Basic Concepts and System Description

2.1 The Device

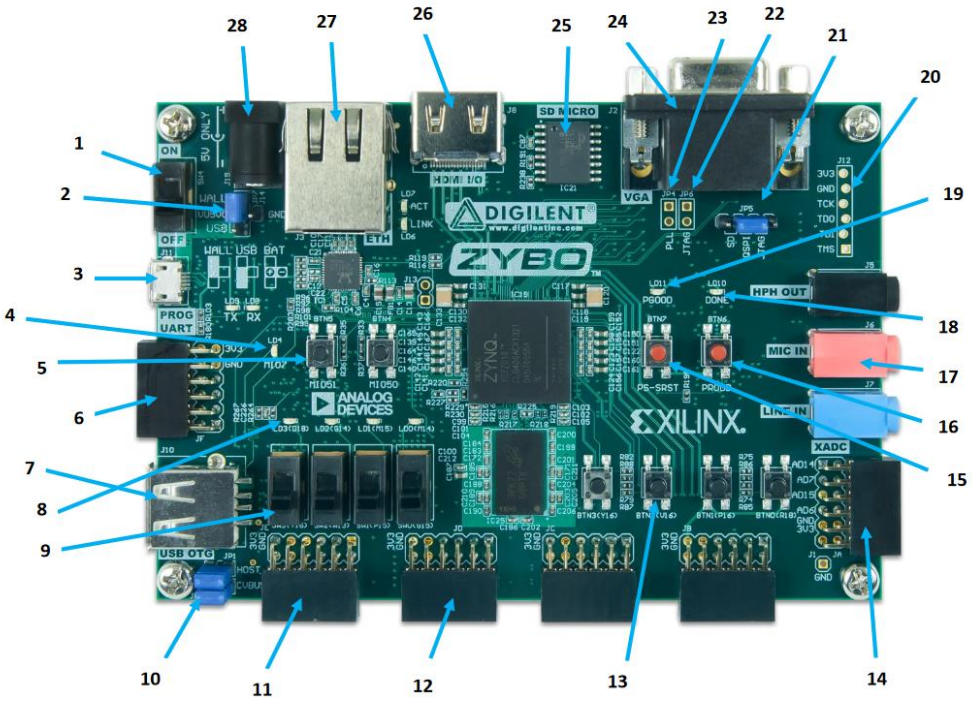
2.1.1 The Zynq Board (Zybo)

The device that was used in this thesis is the Zynq Board [9], usually referenced as Zybo, a SoC FPGA development board manufactured by Digilent. The Zybo is built around the Z-7010 AP SoC (All Programmable System-on-Chip) of the Xilinx Zynq-7000 family. The Z-7010 integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series FPGA logic. Additionally, a rich set of multimedia and connectivity peripherals are available on the board to provide support for a wide range of applications.

Some of the board features are listed below:

- Xilinx Zynq Z-7010 AP SoC
  - 650 MHz dual-core Cortex-A9 processor
  - DDR3 memory controller with 8 DMA channels
  - High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
  - Low-bandwidth peripheral controllers: SPI, UART, CAN, I^2C
  - Programmable logic equivalent to Artix-7 FPGA
  - On-chip Analog to Digital Converter (XADC)
- 512 MB DDR3 memory with 1050 Mbps bandwidth
- 128 Mb Serial FLASH memory with QSPI interface
- External EEPROM
- 10Mbit/100Mbit/1Gbit Ethernet PHY
- OTG USB 2.0 PHY
- Dual-role HDMI port
- 16-bit VGA source port
- Audio codec with headphone, microphone and line in jacks
- microSD slot
- On-board JTAG programming
- GPIO: 6 pushbuttons, 4 slide switches, 5 LEDs
- 6 Pmod ports

The Zybo can be powered from the UART/JTAG USB port or from an external power supply. It supports three different boot modes: booting from a microSD card, from the onboard QSPI Flash memory or from software loaded by a host computer via JTAG. A USB-UART bridge enables PC applications to communicate with the board using serial port commands.



Callout	Component Description	Callout	Component Description
1	Power Switch	15	Processor Reset Pushbutton
2	Power Select Jumper and battery header	16	Logic configuration reset Pushbutton
3	Shared UART/JTAG USB port	17	Audio Codec Connectors
4	MIO LED	18	Logic Configuration Done LED
5	MIO Pushbuttons (2)	19	Board Power Good LED
6	MIO Pmod	20	JTAG Port for optional external cable
7	USB OTG Connectors	21	Programming Mode Jumper
8	Logic LEDs (4)	22	Independent JTAG Mode Enable Jumper
9	Logic Slide switches (4)	23	PLL Bypass Jumper
10	USB OTG Host/Device Select Jumpers	24	VGA connector
11	Standard Pmod	25	microSD connector (Reverse side)
12	High-speed Pmods (3)	26	HDMI Sink/Source Connector
13	Logic Pushbuttons (4)	27	Ethernet RJ45 Connector
14	XADC Pmod	28	Power Jack

Table 2.1: The Zybo and its components [9]

2.1.2 The Zynq-7000 AP SoC

The Zynq-7000 AP SoC [10] is divided into two distinct subsystems, the Processing System (PS) and the Programmable Logic (PL). Figure 2.2 shows an overview of the SoC architecture, where the PS and PL components can be seen, together with various interconnections between them.

The heart of the Zynq PS is the Application Processor Unit (APU), which includes two ARM Cortex-A9 CPUs, each with associated computational units: a NEON Media Processing Engine and Floating Point Unit (FPU), a Memory Management Unit (MMU) and 32 KB L1 Instruction and Data caches. The APU also contains a shared 512 KB L2 cache memory, a 256 KB SRAM

On-Chip-Memory (OCM), a Snoop Control Unit (SCU), a DMA controller, a General Interrupt Controller (GIC), System-Level Control Registers (SLCRs) and Timers.

Besides the APU, the PS includes memory interfaces (DDR and FLASH memory controllers) and I/O Peripherals for external interfaces, including USB, SDIO, Gigabit Ethernet, SPI, CAN, UART, I^2C and GPIO. Communication between the PS and external interfaces is achieved primarily via 54 dedicated pins called Multiplexed I/O (MIO), which can be flexibly mapped to peripherals as required. Peripheral connections are also possible via the Extended MIO (EMIO) interface, which routes the I/O path through the PL.

The PL is based on the Artix-7 FPGA fabric and is composed of general-purpose FPGA logic as well as special resources, such as Block RAMs (BRAM) for memory requirements and DSP48E1 slices for high-speed arithmetic. Additionally, an Analog to Digital Converter (XADC) is integrated into the logic fabric. Note that the PCI Express block and Serial Transceivers, shown in Figure 2.1, are not available on the Zynq-7010 device.

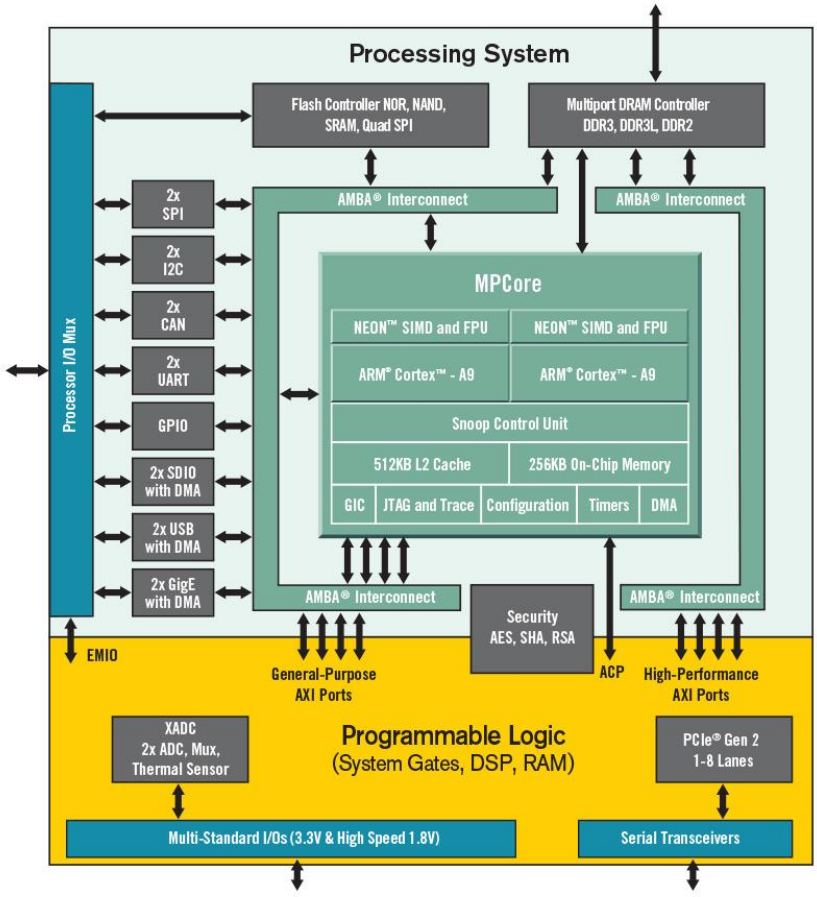


Figure 2.1: The Zynq-7000 AP SoC architecture [11]

The APU, memory controllers and I/O Peripherals of the PS are connected to each other and to the PL through an AMBA AXI interconnect. AXI stands for Advanced eXtensible Interface and is part of the AMBA (Advanced Microcontroller Bus Architecture) standard developed by ARM. A set of AXI interconnects and interfaces forms a network that supports master-slave transactions between the different components of the SoC. There are three different types of PS-PL AXI interfaces [12]: four General-Purpose ports (AXI_GP), one Accelerator Coherency Port (AXI_ACP) and four High Performance ports (AXI_HP). The AXI_GP is a 32-bit data bus suitable for low and medium rate PS-PL communications. The AXI_ACP provides a 64-bit wide asynchronous connection between the PL and the SCU, to achieve coherency between the APU caches and the PL. Finally, the AXI_HP ports support high rate communications between the PL

and memory elements in the PS. A more detailed description of the AXI_HP interfaces will be given below, since they have been useful for the implementation of our system.

The AXI_HP interfaces [10] provide high bandwidth datapaths from the PL to the DDR and OCM memories. The PL is the master of all four AXI interfaces. The data width of the High Performance ports can be independently programmed to either 32 or 64 bits. Each interface includes two FIFO buffers for read and write traffic, in order to accommodate large bursts of data. An interconnect routes the HP ports to two DDR memory ports or the OCM. The AXI_HP interfaces are also referred to as AFI (AXI FIFO Interface), in reference to their buffering capabilities. A diagram showing the AXI_HP connectivity is given in Figure 2.3.

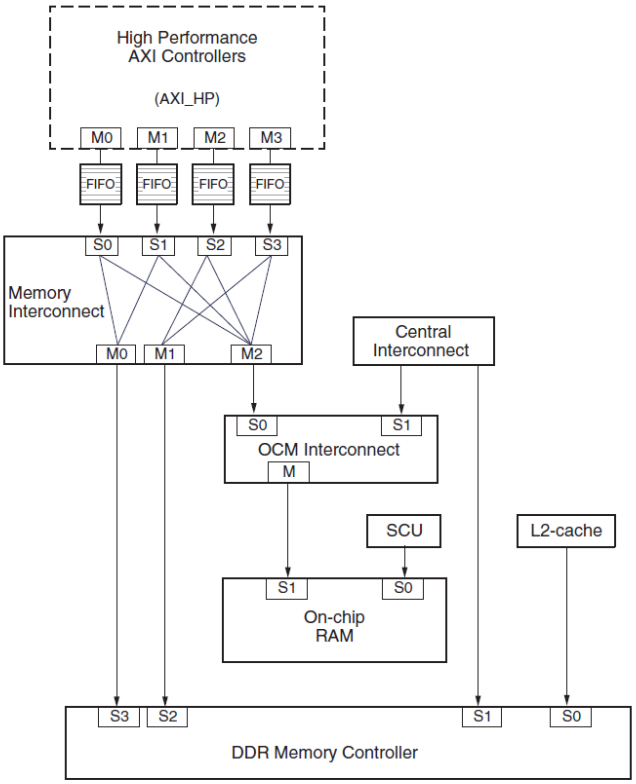


Figure 2.2: AXI High Performance ports connectivity [10]

It has been made clear that the AXI_HP ports are optimized for high throughput applications. Therefore, it is not recommended to use them for general purpose AXI transfers, where the AXI_GP interfaces are preferred instead. A common design for an IP residing in the PL is to consist of a low-speed control interface through the AXI_GP port and a higher performance data burst interface through the AXI_HP port. Finally, the HP interfaces provide bandwidth management functions, through specific signals and registers, to assist priority and queue (FIFO occupancy) management [10].

2.2 The AMBA AXI4-Stream Protocol

The latest version of the AMBA AXI communication protocol is AXI4. There are three different variations of the AXI4 interface, each suited to a different nature of application [13]:

- AXI4-Full, a memory-mapped interface allowing bursts of up to 256 data transfer cycles with a single address phase.

- AXI4-Lite, a light-weight variant of the interface for memory-mapped communication, which allows just a single data transfer per transaction.

- AXI4-Stream, which allows unlimited data burst size, being suitable for high-speed transmission of streaming data.

In this section, we examine the AXI4-Stream interface, since it was used for the implementation of our system, as it will be explained later. The AXI4-Stream protocol defines a single channel for transmission of streaming data. It allows unidirectional transfers from a master, that generates data, to a slave, that receives data. AXI4-Stream interfaces do not require an address phase and are therefore not considered to be memory-mapped. The streaming nature of AXI4-Stream makes it best suited for applications requiring a constant stream of data, such as communications/networking and audio or image processing.

The AXI4-Stream interface signals are listed in Table 2.1, along with the source that generates them and a brief description.

Signal	Source	Description
TVALID	Master	Indicates that the master is driving a valid transfer.
TREADY	Slave	Indicates that the slave can accept a transfer in the current cycle.
TDATA	Master	The payload used to provide the data that is passing across the interface.
TSTRB	Master	Indicates whether the associated byte of TDATA is a data byte or a position byte.
TKEEP	Master	Indicates whether the associated byte of TDATA is a null byte or not.
TLAST	Master	Indicates the boundary of a packet.
TID	Master	The data stream identifier.
TDEST	Master	Provides routing information for the data stream.
TUSER	Master	User defined information that can be transmitted alongside the data stream.

Table 2.2: The AXI4-Stream signals [14]

Most of the above signals are optional in the AXI4-Stream interface. In fact, we are only interested in **TVALID**, **TREADY**, **TDATA** and **TLAST**. A handshake process is necessary between the master and the slave for information to be passed across the interface. This is a two-way flow control mechanism that enables both sides to control the rate at which the data are transmitted. For a transfer to occur, both the **TVALID** and **TREADY** signals must be asserted. This indicates that the master has placed valid data on the bus and that the slave is ready to accept that data. Either **TVALID** or **TREADY** can be asserted first, for the handshake to complete successfully [14].

In Figure 2.4, two examples of the handshake sequence are presented in form of timing diagrams. In the first example, **TVALID** is asserted before **TREADY**. The valid information from the master remains unchanged until **TREADY** is also asserted by the slave, at which point the transfer can occur. In the second example, the slave asserts **TREADY** first, before information is valid. The transfer will now take place once the master asserts **TVALID**. In both cases, the exact time at which the transfer occurs is indicated by the arrow at the bottom of the diagram.

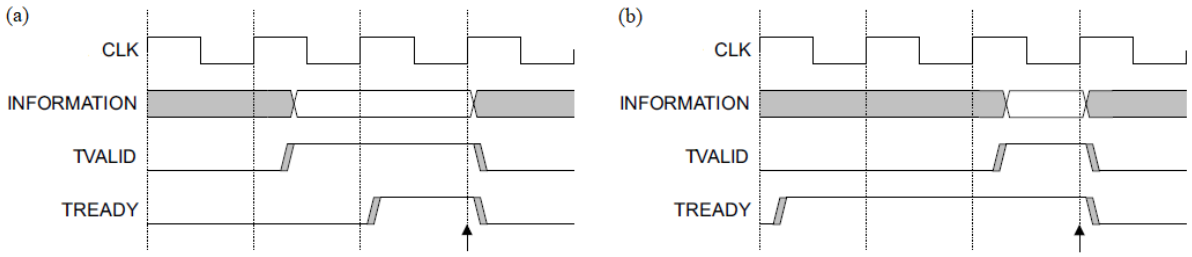


Figure 2.3: The TVALID-TREADY handshake [14]
(a) *TVALID before TREADY*
(b) *TREADY before TVALID*

The information that is transmitted across the interface is represented by the **TDATA** signal. The width of **TDATA** is an integer number of bytes, corresponding to the data bus width. The transmitted bytes may be grouped together in multiple packets. Dealing with packet transfers is typically more efficient for infrastructure components than processing continuous data streams [14]. The **TLAST** signal can be used in such cases to indicate the boundary of each transmitted packet.

2.3 Direct Memory Access

Direct Memory Access (DMA) is a mechanism that allows computer hardware to access system memory for data movement without CPU intervention. Moving data to and from I/O devices is a slow process that requires cycle-intensive memory transfers. In programmed I/O methods – polling or interrupts – such tasks require a significant portion of the CPU time, especially when bulk data movement is needed. With DMA, these tasks are assigned to dedicated data transfer devices, allowing the CPU to focus on other useful computing activity while the transfer is in progress.

Using DMA for data transfers in computer systems has several advantages. First of all, it off-loads the processor by moving the data transfer overhead to dedicated hardware, leading to improved system performance. In addition, the dedicated hardware is capable of moving data from one computer location to another faster than the CPU, reducing the transfer time. High-speed data acquisition devices can benefit from the improved data transfer speed and respond time, since latency in servicing these devices is minimized. Finally, when the CPU operates out of its cache, DMA data transfers are literally done in parallel, thus increasing the system utilization [15].

The hardware that is responsible for DMA operations is called DMA controller. A DMA controller can directly access memory and is used to transfer data on behalf of the CPU. The direction of transfer can be from an I/O device to system memory and vice versa, or from one memory location to another. The controller manages several DMA channels, each of which can be programmed to perform a DMA transfer upon request. The CPU and DMA controller usually share the same memory and I/O bus. The DMA controller can either become the bus master to perform a transfer or have the transfer set up by the CPU, as a bus slave. In the second case, the CPU programs some internal registers of the controller, which contain configurations like source and destination addresses, burst size, as well as control and status information [15].

In typical computer systems, the DMA controller is usually a device peripheral to the CPU. In a SoC FPGA, however, the device can either be a module embedded in the processing system or implemented in the programmable logic of the FPGA. In Zynq, for example, Xilinx provides several DMA IP cores, compliant with the AXI standard, that can be added to the PL [16].

The whole procedure of a DMA transfer involves the following steps [16]:

1. The CPU programs the DMA controller, by allocating a DMA channel and setting the DMA capacity and transfer configurations.
2. The DMA controller requests a transfer, by allocating first buffers in memory where the data will be saved. A transfer may also be initiated by the CPU, without a request from the DMA controller.
3. The data are transferred from source to destination using DMA.
4. An acknowledgment is received by the DMA controller, once the transfer is complete.
5. The controller informs the CPU about the accomplishment and that a new transfer can now be scheduled. This is done through an interrupt or with the CPU polling the DMA controller until the transfer is complete.

There are some software-related issues that should be considered when using DMA. First, the multiple channels of the DMA controller are often used by independent applications. Therefore, the controller, as a shared system resource, should be properly managed to avoid applications affecting each other. Moreover, the DMA allocated buffers should be managed as data are collected or generated, according to the application requirements. In particular, DMA buffer management complexity increases in virtual memory systems, where allocation of physically contiguous buffers and memory mappings must be taken into account. Another major concern has to do with the amount of overhead that is introduced when setting up the DMA controller. If the data volume to be transferred is small or the DMA buffering capacity is poor, that overhead may result to a slower transfer than just assigning the task to the CPU. In order to fully take advantage of the DMA high throughput, the setup time should be compensated by the gains in the DMA data transfer time. Finally, special care should be taken to maintain cache coherence when using DMA, since external memory can be directly accessed by DMA devices and thus become inconsistent with the CPU cache. Non-coherent systems should handle this in software, by flushing or invalidating cache lines before starting DMA transfers that affect related memory ranges [15].

2.4 Embedded Operating Systems and Multiprocessing

In earlier years, it was quite common to design embedded systems by writing firmware that would run directly on hardware, without any underlying software abstraction layers. This is known as bare metal programming and is still used today for the development of some embedded applications. However, as microprocessors grew gradually more powerful, use of Operating Systems (OS) in embedded systems became more and more frequent. Nowadays, the vast majority of embedded systems include an OS that handles the interaction between the software application and the hardware platform.

There are a number of advantages when including an operating system for the development of an embedded system. Firstly, an OS offers standardized system interfaces, APIs and driver support for a wide range of devices. The use of these – and many other – existing features can significantly reduce development time. In addition, targeting software at an OS rather than a specific device offers great flexibility, as the developed product could potentially be easily moved to a new platform. Finally, the developer does not need to worry about resource management or task scheduling because the OS takes care of these functions [12].

An important question that a designer is faced with when developing an embedded system is to determine which type of OS to use. There are several available options to choose from, each one offering its own benefits. The simplest type of embedded operating system is a standalone OS, which provides low-level software modules to directly access processor features, like cache configuration and interrupt setup, and hardware-specific functions. A standalone OS provides limited functionality but enables close control over code execution and hardware. When there is need for real-time behavior, a Real-Time Operating System (RTOS) is the best choice. A RTOS guarantees short and predictable response time for given tasks, thus being suitable for applications with critical timing requirements. In contrast, if flexibility and productivity are preferred over time determinism, general-purpose operating systems should be used, such as Linux or Android OS [12].

The Zynq processing system supports all of the environments that were mentioned above. Xilinx provides standalone platforms for their products, several embedded Linux solutions, such as PetaLinux, as well as a version of FreeRTOS ported to Zynq.

In multi-processor systems, a decision must also be made as to whether a single OS will run across all processing cores or multiple OS instances will be used for individual cores. These techniques are described with the terms Symmetric Multiprocessing and Asymmetric Multiprocessing respectively, which are introduced in the following sections.

2.4.1 Symmetric Multiprocessing

In Symmetric Multi-Processing (SMP) [17], all cores run a single OS instance, which coordinates the execution of tasks between them. Applications can exploit the full compute power of the multiprocessor, as all threads of execution are allowed to run concurrently on any core. The OS has thread preemption and prioritization features, providing efficient scheduling of tasks and optimal allocation of work on the multiple cores. All CPUs in the system must be of matching architectures to support SMP.

Unfortunately, SMP cannot offer the expected linear boost of performance as the number of cores increases. This is because SMP is designed to exploit software parallelism, but all applications are not natively parallel (single-threaded programs, I/O bound applications). Another reason is that internal kernel mechanisms are required to manage dynamic load balancing, which bring an overhead that increases geometrically with the number of cores.

In a SMP environment, communication between cores is achieved through the use of shared memory. There is no need for any special Inter-Process Communication (IPC) protocol, which simplifies significantly the programming model for applications.

All system resources, including memory, which are shared between the multiple cores, are managed by the OS and dynamically allocated among them. In fact, the OS can allocate resources to specific applications rather than CPU cores, offering greater utilization of available hardware. Nevertheless, protective measures must be taken to deal with concurrency issues, preventing simultaneous access to shared resources by different threads. To achieve this, locking mechanisms, such as mutexes and semaphores, are inserted into the application code. These elements, though, may ultimately lead to decreased CPU utilization and that is why SMP performance does not scale ideally for processors with larger number of cores.

It is worth mentioning that SMP usually gives designers the ability to lock any application to a specific core of the multiprocessor. All related threads are then bound to the specified core and forced to execute exclusively on it. This approach is sometimes referred to as Bound Multi-Processing (BMP). It offers several advantages, compared to full SMP, such as cache thrashing elimination and easier migration of uniprocessor applications to multicore environments [18].

2.4.2 Asymmetric Multiprocessing

In Asymmetric Multi-Processing (AMP) [17], each core runs its own instance of OS, which can be the same or different from other instances. The designer has the ability to directly control how every core is used and partition them appropriately to achieve maximum CPU utilization, even in the case of I/O bound processing. Of course, the AMP model allows applications running on separate cores to communicate with each other. AMP can be used in heterogeneous systems, consisting of multiple CPUs with different architectures. The SMP and AMP designs for the Linux and RTOS operating systems, with reference to the dual-core ARM processor of the Zynq platform, are depicted in Figure 2.5.

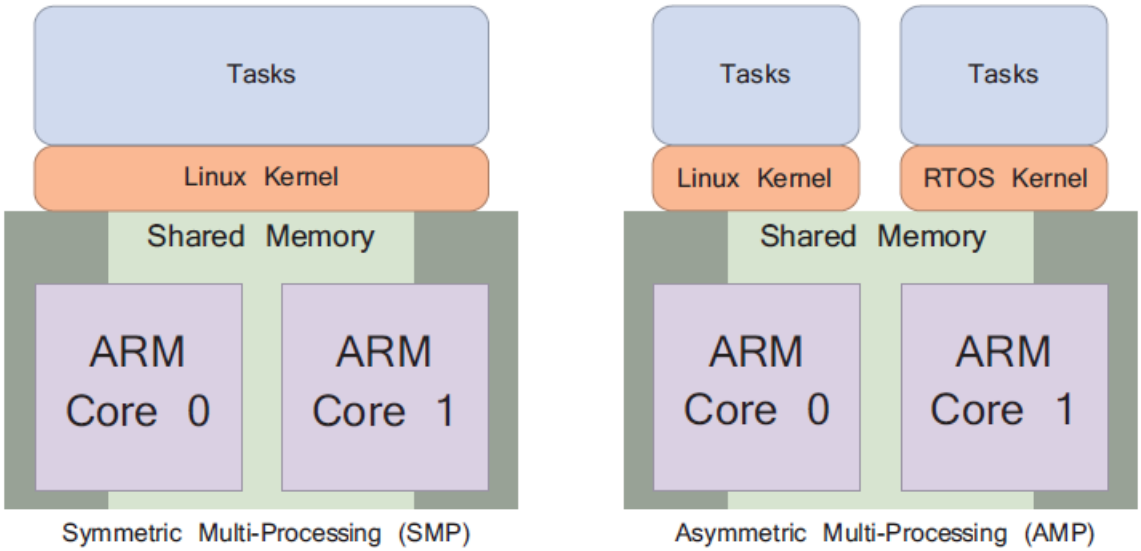


Figure 2.4: Symmetric vs. Asymmetric Multiprocessing [12]

Since multiple operating systems coexist in an AMP system, a complete networking infrastructure or dedicated communication framework is required to support IPC. There are several such implementations, including I/O peripherals, shared memory/interrupt-based schemes and hardware-assisted messaging mechanisms.

With AMP, the designer determines how to divide the shared hardware resources between the multiple cores. Memory allocation, peripheral usage and interrupt handling are usually defined statically during boot time, so that each OS is aware of the resources it has been allocated. Concurrency hazards may only appear here between independent applications running on different cores, which is usually easier to handle than in SMP mode.

A very interesting concept is lately explored by the industry, which involves a hybrid approach that combines both the SMP and AMP models on the same multicore device. If more than two homogeneous cores are available, some of them can be configured to run in SMP and form one processing domain. Then, this domain can run in AMP with other cores, either homogeneous or heterogeneous. The benefits from such a multiprocessing model include improved power consumption and higher performance.

2.5 System Description

A system-level description of the testbed will now be presented. All the necessary concepts that will be referenced have already been described in the previous sections of this chapter.

The testbed is implemented on the Zybo, using both the PS and PL features of the Zynq-7000 AP SoC. The system consists of two main software applications, called transmitter and receiver. Each application is configured to run on a different core of the ARM Cortex-A9 dual-core processor of the Zynq PS. The transmitter runs on CPU0, while the receiver runs on CPU1. In addition, an Ethernet connection is established between the Zybo and a computer, where a file containing input data values is located.

The transmitter receives the input data via Ethernet and stores the values in memory. A custom IP is implemented in the PL, the role of which is to process the input data. The data are transferred from memory to the IP using DMA. The hardware responsible for the DMA transfer is also implemented in the PL and the data movement from memory to the IP is achieved with the AXI4-Stream protocol.

The processed data are stored back in memory with a new DMA transfer. Once again, the data are moved from the IP to the memory through AXI4-Stream. The receiver waits for the DMA transfer to complete and then accesses the results in memory to perform further processing on them. This processing phase is executed by the processor (CPU1), in contrast to the first phase executed in hardware. In the end, the final data are sent back to the computer via Ethernet, where they are written to an output file to check their validity.

On the subject of operating systems, two different implementations are realized as successive steps. First, a Linux SMP configuration is tested. In other words, both CPUs run the same Linux OS. However, it is ensured that the transmitter and receiver applications execute solely on CPU0 and CPU1 respectively, to provide a level of isolation between them. The second implementation involves a Linux/RTOS AMP configuration, where CPU0 runs Linux and CPU1 runs FreeRTOS. This aims to increase the isolation level between the transmitter and the receiver, provide real-time features to CPU1 and potentially increase the system performance.

More specific details about the implementation procedure are given in the next chapter. Both the SMP and AMP cases are examined, with regard to hardware and software design options.

Chapter 3

System Implementation

3.1 Hardware Design

First, we will present the components that were used to create the hardware design of the system. Then, we will describe some modifications and measurements done to the design, in an attempt to explore performance optimization.

3.1.1 Hardware Components

The block design was created using the Xilinx Vivado Design Suite. It consists of the following elements:

- The **ZYNQ7 Processing System**. This is the PS subsystem of the Zynq-7000 SoC, including the dual-core ARM processor, memory and peripheral interfaces.

- The **AXIS Multiplier**, a custom IP core implemented in the PL, receiving and generating data through AXI4-Stream interfaces. It consists of a simple parallel multiplier with input and output buffers, implemented as two AXI4-Stream Data FIFOs. The width of the input and output data buses is customizable. The multiplication corresponds to the first processing phase of data, which is executed in the PL.

- The **AXI Direct Memory Access** (AXI DMA), residing in the PL. The AXI DMA provides high-bandwidth Direct Memory Access between the DDR memory (AXI4 Memory-Mapped interface) and the AXI4-Stream Multiplier. It is responsible for moving data from memory to the multiplier and vice versa, without the processor's interaction. An AXI4-Lite interface enables initialization and management from the PS, while access to the DDR memory is achieved through one of the available High Performance (AXI_HP) ports [19]. The AXI DMA parameters that were customized are described below.

    - Width of Buffer Length Register: This parameter determines the maximum number of bytes that can be transferred with a DMA transfer. It was set to the highest possible value of 23 bits, to allow transfers of 2^{23} bytes (8 MB).
    - Max Burst Size: This specifies the maximum size of the burst cycles on the memory-mapped interface and was set to 256, both for read and write channels, to achieve maximum throughput.
    - Memory Map and Stream Data Widths: These settings define the widths of the corresponding data buses. All values are initially set to 32 bits, but they are later modified during performance exploration.

- The **Concat IP**. The AXI DMA can be used either in polled mode or interrupt mode, to inform the processor that a DMA transfer is complete. In case interrupts are used, AXI DMA generates two interrupts, one for the read channel and one for the write channel. The Concat IP is used to concatenate those interrupt signals into a single bus, which can be routed to the ZYNQ7 PS port for PL→PS interrupts (IRQ_F2P).

- Other blocks that are automatically added to the design by Vivado, including the Processor System Reset module and various AXI Interconnects.

A simplified block design of the system is shown in Figure 3.1.

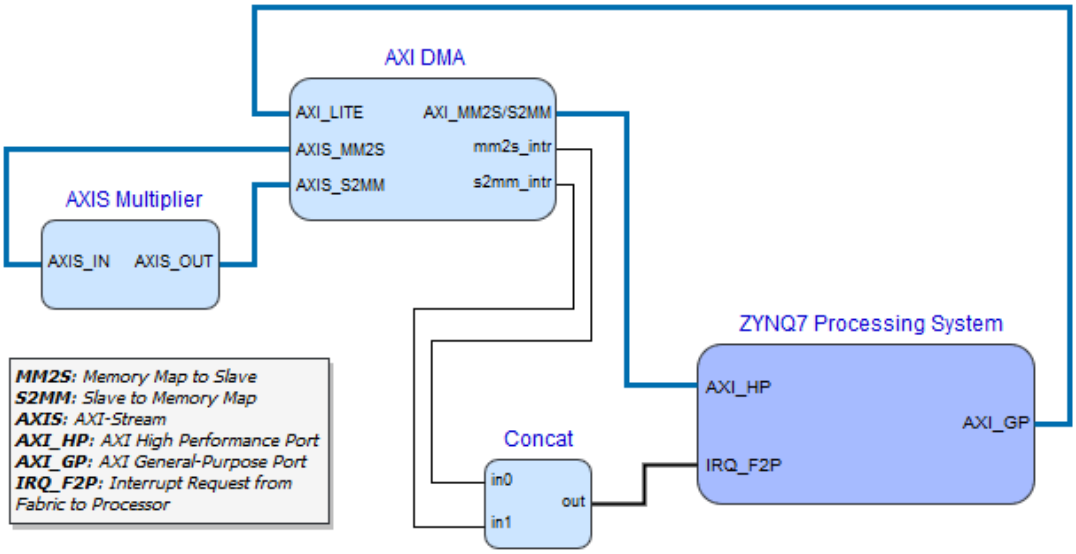


Figure 3.1: A simplified block design of the system

3.1.2 Design Optimization

The optimization process aims to maximize the bandwidth that the system can support when moving data between the DDR memory and the multiplier, using the AXI DMA. The experiment consists of writing 1 MB (2^{20} bytes) of data on memory and sending it to the multiplier by performing one DMA transfer. The results of the multiplier are written back to memory with a second DMA transfer. The two distinct transfers are performed concurrently. The total time of the first transfer is measured in order to find the bandwidth and evaluate the design.

In the initial design, the width of the Memory Map and Stream data buses is 32 bits. On each clock cycle, a 32-bit value (containing two 16-bit values) is sent to the multiplier, which computes the 32-bit result. With a clock frequency of 100 MHz, the theoretical bandwidth is

$$BW_{th} = \frac{32\ bits}{\frac{1}{100\ MHz}} = 381.5\ MB/s$$

The actual bandwidth achieved after sending 1 MB of data to the multiplier was

$$BW_{act} = 381.2\ MB/s$$

The above value is very close to the theoretical one, which means that our design was successfully configured to achieve the best performance.

An obvious option to increase throughput is to expand the width of the Memory Map and Stream data buses from 32 to 64 bits. This way, the bandwidth is expected to double. Indeed, the actual bandwidth was measured to be

$$BW_{act} = 761.9\ MB/s$$

which is almost twice as much the value that was measured in the case of 32-bit data width.

In an attempt to increase even further the throughput, the data width was once again expanded to 128 bits. However, the bandwidth remained unchanged at 761.9 MB/s, showing that there is a limit to the speed we can gain by data width expansion. The four AXI High Performance (AXI_HP) interfaces provide either 32-bit or 64-bit wide datapaths to the memory [10]. Since the DMA engine uses one of these HP ports to move data to and from memory, wider data values are simply split to fit one of the supported 32-bit or 64-bit modes. Therefore, each transaction must be completed in multiple steps and the throughput does not improve.

The initial clock frequency of 100 MHz can be increased to get better results in terms of throughput. Setting the maximum value of 250 MHz gives a theoretical bandwidth as shown below.

$$BW_{th} = \frac{32\ bits}{\frac{1}{250\ MHz}} = 953.7\ MB/s$$

The actual bandwidth after sending 1 MB of data to the multiplier was measured to be

$$BW_{act} = 952.5\ MB/s$$

Another thought is to use one AXI DMA core for each one of the write and read channels of the transactions separately. In other words, the first core is responsible only for reading data from memory and sending the stream to the multiplier, while the second one only receives the stream of results from the multiplier and writes the data back to memory. However, the throughput was not measured any higher than before, which means that one single DMA core can handle both channels simultaneously, without any interference between them. Therefore, the use of multiple DMA cores as described above does not offer any improvement.

Two DMA cores can efficiently improve performance if they both use their read and write channels simultaneously. By adding a second multiplier to the design and connecting each one of them with a different DMA core, we can execute our task in parallel. Each pair of AXI DMA-multiplier is responsible only for half of the data, while the memory accesses and computations are done simultaneously.

Two DMA cores, though, require two different AXI_HP interfaces. The experiment was held for two different HP port pairs (at 250 MHz). After choosing first the ports HP0 and HP1 and sending 500 KB to each multiplier (1 MB in total) the bandwidth achieved was

$$BW_{01} = 1124.7\ MB/s$$

We do observe an increase but not as high as we expected it to be, compared to the 952.5 MB/s measured when there was not at all parallelism in our design. After repeating the same experiment using the ports HP0 and HP2, the bandwidth was now measured to be

$$BW_{02} = 1601.6\ MB/s$$

This significant improvement of the performance can be explained by examining the connectivity of the High Performance ports in the system hardware (Figure 2.3). The ports HP0 and HP1 share the same path towards the DDR memory controller. This is also true for the ports HP2 and HP3. Therefore, it is preferable to use simultaneously ports connected to different paths (in our case HP0 and HP2) to avoid conflicts and increase throughput. However, the theoretical bandwidth expected when using two interfaces in parallel is

$$BW_{th} = \frac{(32 + 32)\ bits}{\frac{1}{250\ MHz}} = 1907.3\ MB/s$$

So even with the independent HP ports there is an overhead in the system that prevents the bandwidth from going higher. This happens probably because the DDR memory controller provides a unique path towards the memory, so the read/write requests are still bound to transaction scheduling and arbitration [10].

We increased even further the parallelism of our system by using simultaneously three – or even all four – of the available HP interfaces. Each interface is related to one AXI DMA-multiplier pair and to one fraction of the data to be transferred. After sending 1 MB of data in total, using the HP ports HP0-HP1-HP2, the bandwidth was

$$BW_{012} = 1389.9\ MB/s$$

The theoretical bandwidth is now

$$BW_{th} = \frac{(32 + 32 + 32)\ bits}{\frac{1}{250\ MHz}} = 2861\ MB/s$$

Therefore, adding a third HP interface to our system just leads to a bandwidth which is almost half the theoretical one (48.6%). It is obvious that this gain is not profitable, considering the added hardware and software complexity of the extra HP interface. An explanation of the phenomenon can be once again derived from Figure 2.3. We clearly see that the multiplexed HP ports are connected to only two ports of the DDR memory controller, so it is expected that three HP interfaces will not bring a linear increase in the bandwidth.

Combining the results presented above, we repeated the experiment using 64-bit (instead of 32-bit) data width along with a clock frequency of 250 MHz and 3 HP interfaces simultaneously. With these settings, the bandwidth was found to be

$$BW_{012} = 1508.2\ MB/s$$

In this case, the theoretical bandwidth is

$$BW_{th} = \frac{(64 + 64 + 64)\ bits}{\frac{1}{250\ MHz}} = 5722\ MB/s$$

We can notice that the overhead is bigger with 64-bit data width, compared to 32-bit, since we managed to achieve just 26.4% of the theoretical bandwidth.

In conclusion, we expect to achieve the maximum bandwidth if we use 64-bit data width together with two independent HP interfaces, for example HP0 and HP2. This way, we shall

provide parallelism to our design efficiently, without adding too much overhead to the system. Indeed, the bandwidth was measured to be

$$BW_{max} = 1682.8\ MB/s$$

The final block design, created in Vivado with respect to the last settings applied above and optimized in terms of bandwidth, is given in Figure 3.2. There are two AXI DMA blocks, which are connected to the AXI High Performance ports HP0 and HP2. They are controlled by the PS through the General-Purpose ports GP0 and GP1. Each AXI DMA can transmit and receive data from a multiplier in the PL, using the AXI4-Stream protocol. There are in total four interrupts generated by the two AXI DMAs, which are concatenated and routed to the IRQ_F2P port of the PS.

3.2 Linux SMP

We previously designed and optimized the hardware of the Zynq SoC in order to build a system capable of transferring data between the memory and the PL efficiently using the AXI DMA. However, the hardware was only tested using a bare metal application. The next step is to have our system running under an operating system, which in our case is an embedded Linux environment.

We used an embedded Linux distribution created with the *PetaLinux Tools* provided by Xilinx. PetaLinux is a great way to build embedded Linux on the Zynq processing system that is compatible with the Xilinx hardware design flow. This means that we can just export a hardware platform already created with Vivado and easily deploy a Linux OS upon it.

3.2.1 Linux DMA Drivers

The main challenge when moving to the Linux environment is the transition from the bare metal application to the software that is compatible with the operating system. In our case, the hardware component that we need to control is the AXI DMA, so we must write the software that will carry out the DMA transfers under Linux.

There are several issues we should consider when performing DMA operations [20]:

1. Cache coherency. The CPU caches must be coherent with the system memory. Depending on the processing system, cache maintenance functions may be required to ensure that. This is true for the Zynq-7000 processors when using the HP ports.

2. Using cached or non-cached memory. Each option may present better performance based on the size of the data being transferred with DMA and the amount of it being touched by the CPU.

3. Kernel space vs. user space. The DMA driver can be designed to lie in the Linux kernel space, user space or both.

Fortunately, Linux provides a framework, known as DMA Engine, that provides the infrastructure for DMA drivers to plug into and then be accessed from kernel space with another client driver using a standard API. Such a client driver is provided by Xilinx, supporting general purpose DMA, including the AXI DMA. Therefore, we can write our own higher layer Linux driver which will use the Xilinx driver through the DMA Engine subsystem [20].

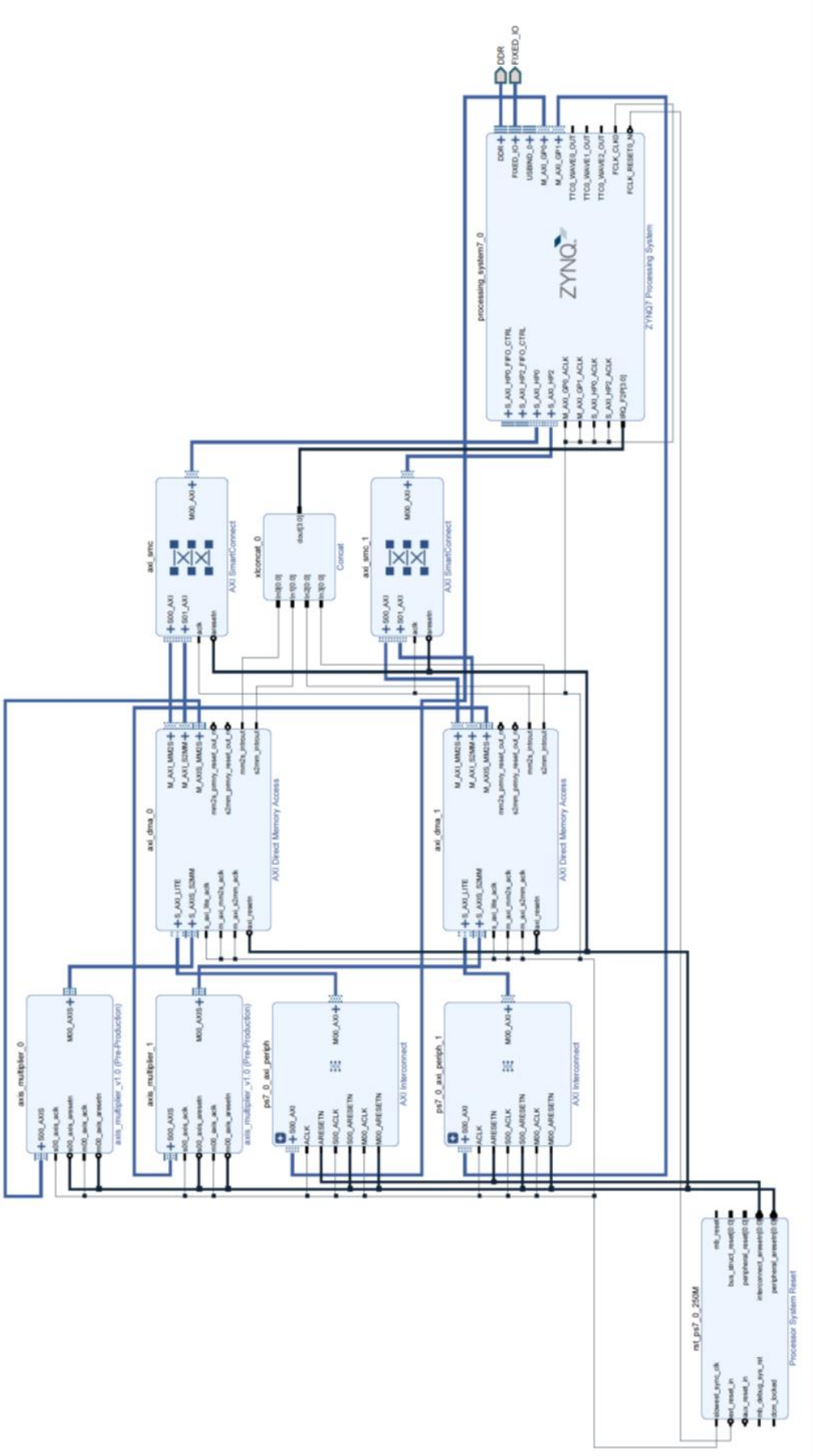


Figure 3.2: The final block design created with Vivado

We used a design suggested by Xilinx, called DMA Proxy [20]. The DMA Proxy design consists of a kernel driver module and a user space application. This hybrid approach gives us the ability to control DMA transfers from user space, while using the Linux DMA Engine framework from kernel space. A detailed design of the DMA Proxy software is presented in Figure 3.3 below.

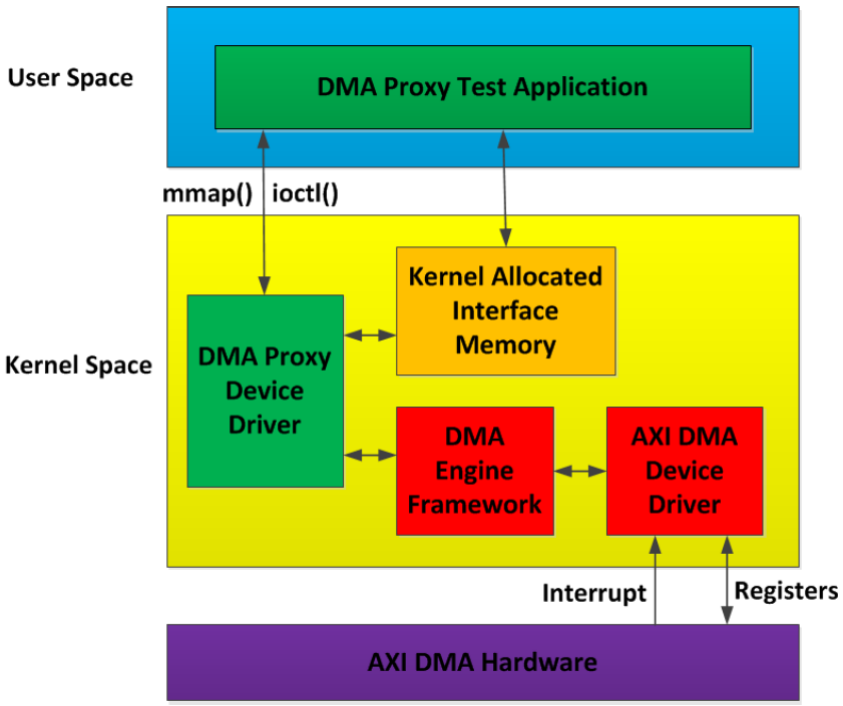


Figure 3.3: The DMA Proxy design [20]

The kernel driver is responsible for allocating non-cached memory for the DMA buffers and controlling the transmit and receive channels of the Xilinx DMA driver through the DMA Engine. It behaves as a character device driver that can be controlled from user space through the *ioctl()* interface. Each DMA channel is represented as a character device node that can be accessed from user space. The user space application maps the kernel allocated memory into user space through the *mmap()* function. This removes the need to copy data between kernel and user space, which would otherwise be inefficient for large DMA transfers, a method known as zero copy design. The application can then access the DMA buffers and manage transfers through the *ioctl()* function.

A PetaLinux project was created and configured in accordance with the hardware platform specification created with Vivado earlier [21]. A custom kernel module and a user application were added to the root file system, containing the source code of the DMA Proxy driver (*dma-proxy.c*) and the DMA Proxy application (*dma-proxy-test.c*) respectively. A shared header file (*dma-proxy.h*) was also included, which defines the shared memory interface between kernel and user space. Finally, an appropriate node was added into the device tree, which defines the device nodes for all the AXI DMA channels. After the system image is generated, we boot it on the Zybo.

First, we need to load the *dma-proxy* module into the kernel. Then we are free to use the *dma-proxy-test* application, giving as arguments the number of DMA transfers to perform and the length of each transfer in bytes. The maximum length of one transfer has been configured to be 8 MB. The data values to be transferred are randomly generated inside the *dma-proxy-test* code. We ran the application successfully with various different configurations. The data were transferred from the memory to the PL for processing and the results were received back to memory without errors. Therefore, we have a functional embedded Linux system for streaming data efficiently with AXI DMA.

3.2.2 Software Applications

When running Linux on the dual-core ARM Zynq processor, by default the kernel is configured to use both CPUs in SMP mode. Normally, the operating system takes care of the task scheduling between the two CPUs of the processing system. However, as it was mentioned in Chapter 2, it is possible for the user to choose a specific CPU for the execution of an application. In our case, where we do DMA transfers between the memory and the PL, we would like to transmit data only from one CPU and receive data from the other one. In other words, we should run two applications, *transmitter* and *receiver*, separately on the two CPU cores. Obviously, the two applications will be executed in parallel. In terms of hardware, we want the *transmitter* to configure both AXI DMAs to fetch data simultaneously from memory through the HP ports and send the values to the respective multipliers. The processed data must be moved back to memory in the same way by the DMA engines, with the *receiver* application being responsible for these transfers.

Based on the *dma-proxy-test* application we already used in PetaLinux, we created two separate applications for the data *transmitter* and *receiver*. The *transmitter* handles the device files for the two transmit channels and executes two DMA transfers from memory to the PL. The *receiver* handles the device files for the two receive channels and executes two DMA transfers from the PL to memory. These two applications work with the DMA Proxy driver, in a similar way with the *dma-proxy-test* application. Since the *ioctl()* calls to the driver are blocking, the transmit or receive transfer for the second channel of each AXI DMA is executed in a thread, because we want all transfers to be done in parallel.

When the user runs either of the above applications, they must also specify the number of transfers and the length (in bytes) of each transfer. This load is equally allocated to the two AXI DMAs. For example, let us say that the user executes the following command:

```
# transmitter 5 1048576
```

A transfer of 1 MB (1.048.576 bytes) should be repeated 5 times. Each AXI DMA is responsible for the transmission of 500 KB (½ MB) for each of the 5 transfers. The *receiver* works in the exact same way.

The *transmitter* and *receiver* applications were added to the PetaLinux project as custom user space applications, in the same way the *dma-proxy-test* was added. The DMA Proxy driver was loaded to PetaLinux as a custom kernel module.

We can also force the applications to run on separate CPU cores. This can be done with the Linux *taskset* utility from the *util-linux* package. *Taskset* gives us the ability to launch a command with a given CPU affinity, thus bonding the related process to the given CPU. By specifying the CPU affinity, the Linux scheduler will not run the process on any other CPUs. PetaLinux does not include the *taskset* utility. Therefore, we cross-compiled the *util-linux* package and added the *taskset* binary to PetaLinux as a prebuilt application. Then *taskset* can be used under PetaLinux with any other application to specify its CPU affinity.

We built and booted on the Zybo the PetaLinux image based upon the hardware platform presented earlier and including the *transmitter*, *receiver* and *taskset* applications. With the following commands, the *receiver* runs on CPU0 and the *transmitter* on CPU1:

```
# taskset -a 1 receiver 5 1048576 &
taskset -a 2 transmitter 5 1048576
```

The `-a` option sets the given CPU affinity for all the threads of the application. This is necessary because our applications use two threads to handle both DMA channels concurrently. The

numbers 1 and 2 are bitmasks representing the CPU affinity. *Receiver* runs on the first CPU (CPU0), while *transmitter* runs on the second CPU (CPU1).

In conclusion, we have successfully created a data *transmitter* and a data *receiver* on the two cores of the Zynq PS, in Linux SMP mode. The data processing is done in the PL. The system is designed to achieve high throughput, with the use of two AXI DMAs in parallel.

3.2.3 Ethernet Communication

Until now we have tested our system by transferring data that has been created by the *transmitter* application itself. However, we should be able to transfer any data given to us by a source independent of our software. For example, let us say we have a file *input* on a computer, which contains data that is supposed to be processed by our system running on the Zybo. In our case, the file will have a list of 64-bit values, as shown below:

input

```
0xf8fd915c2fb37114
0xbe07669093ad7e23
0x8582289bf6f6b915
0x4fe02d3d2ad10a67
…
```

We want to transfer the input data from our computer to the Zybo. This is achieved by connecting them on the same local network, using an Ethernet cable. The development board is of course equipped with an Ethernet interface. From the Zynq Processing System point of view, we enabled one Gigabit Ethernet Controller (ENET 0). This was easily done in our hardware configuration from Vivado. The computer and the Zybo have now a link to communicate with each other.

Regarding the software applications, the *transmitter* and *receiver* communicate with our computer through UNIX Stream (TCP) Sockets. First of all, we have to choose a static IP address for the Zybo and a TCP port number for the connection, for example 192.168.1.11 and 3000.

The *transmitter* creates a socket, binds it to the IP/port pair {192.168.1.11, 3000} and waits to accept an incoming connection. An application *send_data* running on our computer is responsible for creating a socket that will connect to the transmitter. Once the connection is established, *send_data* opens the *input* file, reads the data from it and sends everything through the socket. The *transmitter* reads the data from the socket and stores the 64-bit values in the transmit buffers of the AXI DMAs. Half of the data is stored in the transmit buffer of DMA 0 and the other half in that of DMA 1. The socket connection can now be closed. Then, the *transmitter* is ready to initiate the DMA transfers, where the input data are sent to the PL for processing.

The *receiver* also performs two DMA transfers, where the output data from the PL are stored in the receive buffers of the AXI DMAs. It is now time for the *receiver* to create a socket and listen to {192.168.1.11, 3000} for a new connection. The *send_data* application still running on the computer will repeat the socket creation and will try to connect to the *receiver*. After the connection is established, the *receiver* sends through the socket all the results back to our computer. The *send_data* application opens an *output* file, receives the results from the socket and writes them on the file, before closing the connection once again. In the end, considering the multiplication of the values done in the PL, the *output* file will look like this:

output

```
0x2e6514f7e84cf730
0x6d9f08869ecde5b0
0x80cbbe9df9f757b7
0xd5c00afd5c3958b
…
```

3.2.4 Testing the Testbed

The *input* file contains 1 MB (1.048.576 bytes) of data. The computer and the Zybo are connected with the Ethernet cable. After booting PetaLinux on the Zybo, we execute the following commands:

```
# ifconfig eth0 192.168.1.11 netmask 255.255.255.0
# modprobe dma_proxy
# taskset -a 1 transmitter 1048576 & taskset -a 2 receiver 1048576
```

The first command assigns the IP address 192.168.1.11 to the interface `eth0` of the Zybo. The second one adds the *dma_proxy* module to the Linux kernel. The last command executes the *transmitter* application – and all related threads – on CPU 0 and the *receiver* application – and all related threads – on CPU 1. The two applications run in parallel on the two cores of the Zynq PS. From our computer, we just execute the following command:

```
$ ./send_data 1048576
```

The test completes successfully and we can verify that the results are located in the *output* file on our computer. The messages we receive through the whole process, both on the computer and the Zybo, are presented below:

Computer

```
$ ./send_data 1048576
Data sent to ZYBO.
Waiting for results…
Done! Results are located in output file.
```

ZYBO

```
# taskset -a 1 transmitter 1048576 & taskset -a 2 receiver 1048576
Starting DMA Data Transmitter.
Starting DMA Data Receiver.
Receiver: Receiving data with DMA 0…
Receiver: Receiving data with DMA 1…
Transmitter: Data received from PC and stored in transmit buffers.
Transmitter: Transmitting data with DMA 0…
Transmitter: Transmitting data with DMA 1…
Transmitter: Transmitted all data successfully. Exiting.
Receiver: Received all data successfully.
Receiver: Sending results back to PC…
Receiver: Done! Exiting.
```

3.3 Linux AMP

As an intermediate step between Linux SMP and Linux/RTOS AMP, an effort was made to implement a Linux AMP configuration on the Zynq processing system. Instead of having one single Linux OS shared by the two CPU cores, each one should run its own separate instance of Linux.

AMP in Linux can be achieved with the help of the *remoteproc* and *RPMsg* frameworks. The *remoteproc* (remote processor) framework allows a master processor to manage the life cycle of a remote processor in an AMP configuration. This includes the master booting, loading firmware and shutting down the remote processor. Inter Processor Communication (IPC) between the processors of an AMP system can be achieved through the *RPMsg* (Remote Processor Messaging) framework, which allows kernel drivers to communicate with remote processors using *VirtIO* devices. The *remoteproc/RPMsg* infrastructure and APIs are present in the main Linux kernel.

However, during the implementation, some important issues were encountered that made the task difficult. First of all, there is lack of documentation about the use of the *remoteproc/RPMsg* framework in a Linux AMP configuration. The AMP architectures that are usually required to be designed consist of heterogeneous processing cores running different operating systems. For example, Linux/RTOS or Linux/Bare Metal, where the Linux master executes general purpose operations and the RTOS or Bare Metal remote is dedicated for time critical tasks or specialized functionality.

The *OpenAMP* framework developed by Xilinx and other members of the Multicore Association builds upon the Linux *remoteproc/RPMsg* framework to provide the infrastructure required for FreeRTOS and bare metal environments to communicate with the Linux kernel in AMP systems [22]. Therefore, support is limited to configurations including at least one processor running on FreeRTOS or bare metal environment.

Furthermore, compatibility with PetaLinux may be a problem. PetaLinux builds the embedded Linux OS following predefined steps, thus limiting the possible customizations done by the user. In particular, the *remoteproc* framework requires the remote firmware to include a special section, the resource table. This is a structure describing the system resources required by the remote processor and publishing the supported features. The ELF Linux image created with the PetaLinux flow could not be customized to include the resource table section.

In conclusion, in order to successfully implement a Linux AMP system on the dual-core Zynq processing system, someone should study in depth the Linux *remoteproc/RPMsg* framework. Moreover, an alternative to PetaLinux should perhaps be considered for the deployment of the Linux remote image. A solution that enables to manually create the Linux ELF image and customize its format would possibly be more suitable.

3.4 Linux/RTOS AMP

In this chapter, we present the Linux/RTOS AMP implementation of the testbed. The first core runs a Linux kernel (PetaLinux), while the second one runs a separate real-time operating system (FreeRTOS). The steps that were followed to adapt the previously developed Linux SMP system to the AMP model will be described in detail.

3.4.1 The OpenAMP Framework

The *OpenAMP* framework provides software components that enable the development of software applications for AMP systems. The key components provided by *OpenAMP* include *remoteproc* and *RPMsg*. These are already implemented in the Linux kernel, however *OpenAMP* provides their implementations in different environments, such as bare metal, FreeRTOS and Linux user-space. Therefore, in an AMP system consisting of a master processor and one – or more – remote processors, Life Cycle Management (LCM) of remote cores and Inter Processor Communication (IPC) can be achieved with *OpenAMP* regardless of the software context running on each core [22]. Figure 3.4 illustrates how the *OpenAMP* framework can be used with various environments in different master/remote processor configurations.

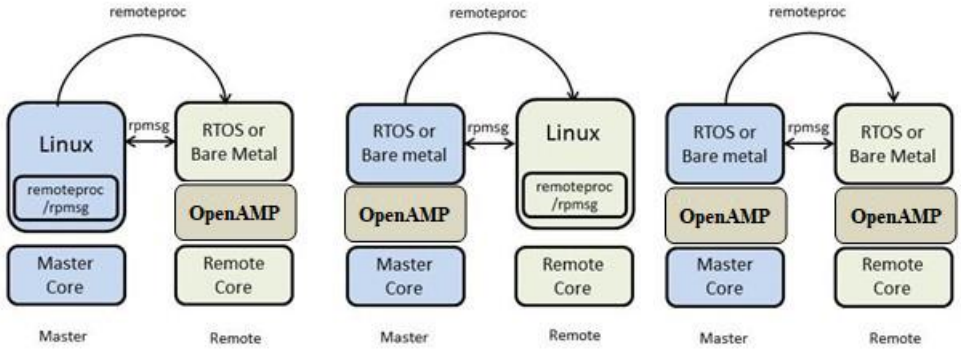


Figure 3.4: The various software configurations supported by OpenAMP [22]

Below is a brief description of the workflow to create and execute an AMP application using *OpenAMP*, for a Linux/FreeRTOS configuration [23].

1. Define the resource table. This is a structure listing the system resources required by the remote processor, such as memory carveout and *VirtIO* device information for IPC. These requirements will be published to the master processor which will allocate appropriate resources.

2. Write the remote application using suitable *remoteproc*/*RPMsg* APIs for the remote software context, provided by the *OpenAMP* library.

3. Create the remote firmware ELF image, by linking the remote application with the *OpenAMP* library and placing the resource table in a specific section of the ELF image.

4. Make the remote firmware accessible to the master processor. In case of a Linux master, the firmware can be placed in the root file system for use by the Linux *remoteproc* drivers.

5. Use the Linux *remoteproc* drivers to load the firmware and boot the image on the remote processor. After the remote application is running, a *RPMsg* channel is established between the master and remote for IPC.

6. Shut down the remote software context and processor when its task is completed. This is also done by the Linux master *remoteproc* drivers.

A conceptual diagram showing the flow of a master processor booting and establishing communication with a remote processor, using the *remoteproc* and *RPMsg* components of the *OpenAMP* framework, is given in Figure 3.5.

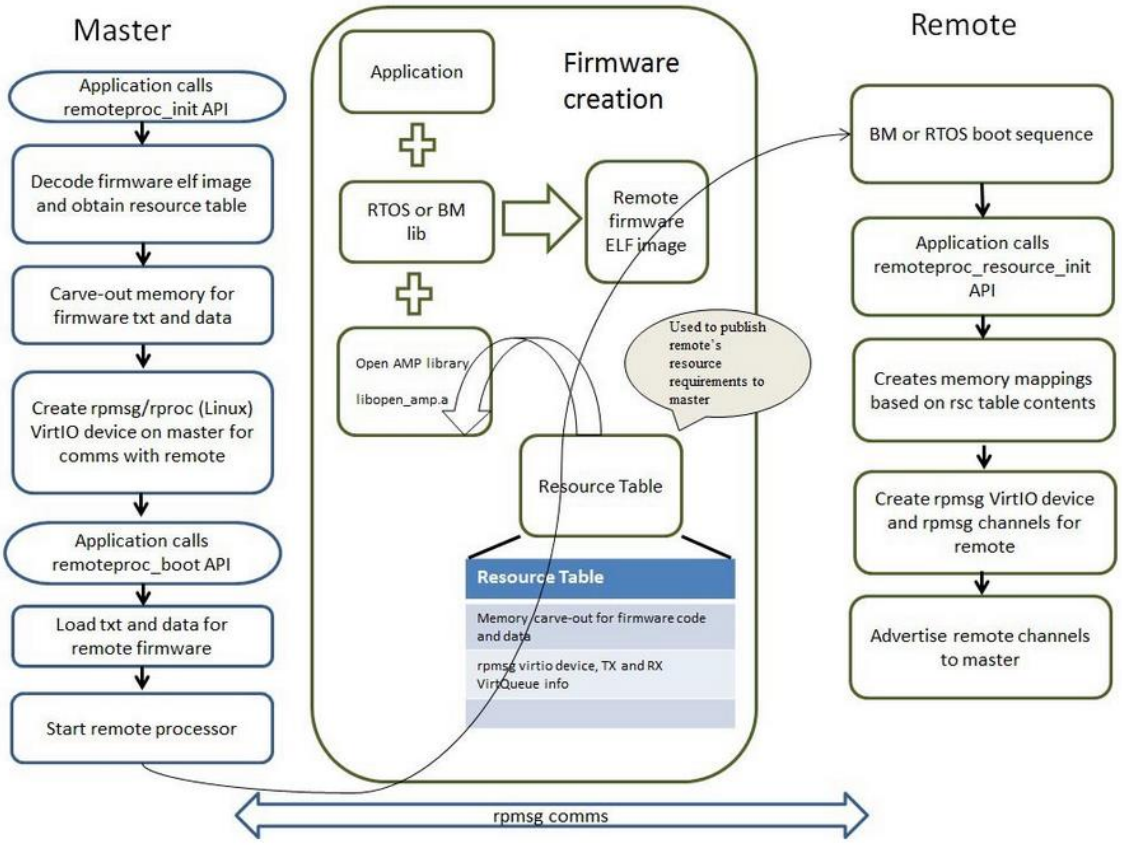


Figure 3.5: The remoteproc/RPMsg flow between the master and remote processor [22]

3.4.2 System Overview

The system consists of a Linux/FreeRTOS AMP configuration on the dual-core Zynq-7000 processing system of the Zybo. Linux runs on CPU0 as master while FreeRTOS runs on CPU1 as the remote processor. We have developed the same *transmitter-receiver* application that was executed in Linux SMP mode, adapting it to the above AMP architecture.

The Linux master first establishes an Ethernet connection with a computer, where the input data are located. Those are a list of 64-bit values written in a file named *input*. The master receives the data from the computer, stores the values in memory and then performs a DMA transfer to transmit them to the PL for processing. Actually, the hardware has been configured to have two DMA engines, so that two transmit DMA transfers can be performed at the same time, each one responsible for only half of the input data. The PL processing is done in hardware by the AXI4-Stream Multiplier, but this can be easily replaced by another component that processes data in a different way.

The FreeRTOS remote performs the two respective receive DMA transfers, to get the processed data from the PL and store them in a specified memory location, accessible to the Linux master. In addition, the remote processor can do some extra processing on the received data, before they are accessed by the master processor, possibly to get advantage of the real-time properties of the FreeRTOS operating system.

Finally, the master acquires the results from the specified memory location and sends them back to the computer through the same Ethernet connection. The results are then written to a file named *output*, which will also eventually contain a list of 64-bit values, modified accordingly to the processing executed by the PL and the remote processor.

The following diagram (Figure 3.6) shows the architecture of the system described above.

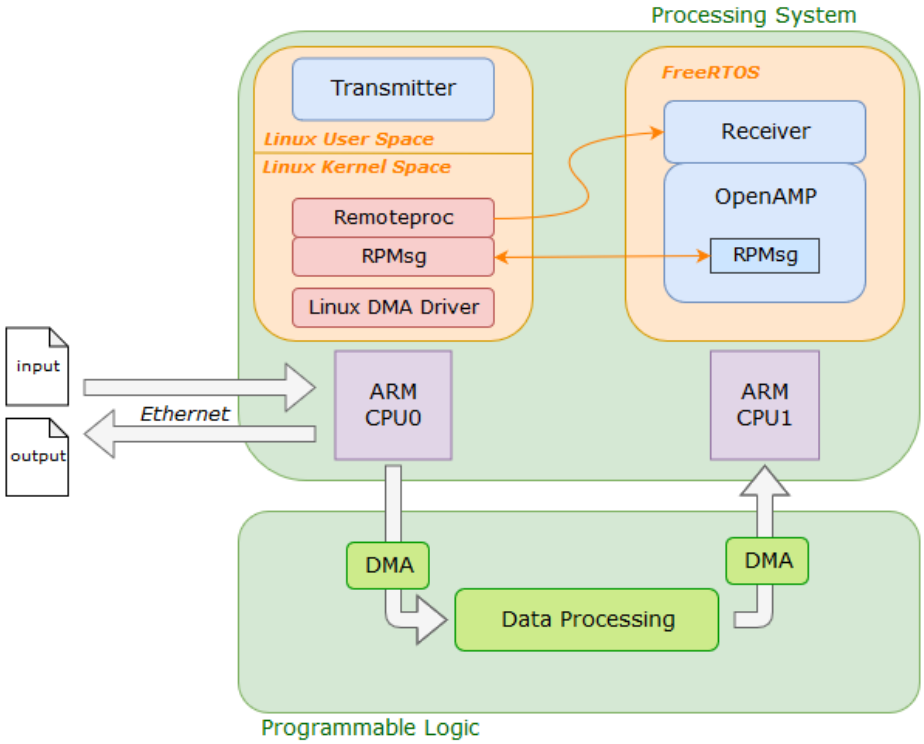


Figure 3.6: An overview of the AMP system

3.4.3 Software Applications

The software applications for the AMP system consist of the *transmitter* (master application), running in PetaLinux on CPU0, and the *receiver* (remote application), running in FreeRTOS on CPU1. Details about the development and functionality of each application are given below.

a) Remote application

The *receiver* application was developed in the Xilinx SDK, where FreeRTOS was specified as the OS platform and Cortex A9 core 1 as the target processor.

The task created initializes *OpenAMP* components, sets up the Interrupt Controller for the *RPMsg* channel interrupts and then waits for the master to send the size of the DMA transfer through the channel. Then, it configures the AXI DMA devices for the transfer, allocates half of the data to each device and initiates the DMA transfers. After completion, the results are accessed in the specified memory location to perform extra processing on them by the CPU. The processor then remains idle until a shutdown message is received from the master to de-initialize *OpenAMP* components and free the application resources.

The application source files, including the resource table where memory and *VirtIO* device resources for *RPMsg* communication are defined, were compiled to create an ELF binary – the remote processor firmware.

b) Master application

The *transmitter* application first opens the *RPMsg* device to enable communication with the remote core and sets up the transmit channels of the AXI DMA devices with the help of the *dma-proxy* driver.

Then it creates a stream socket to receive input data from a computer through an Ethernet connection (IP address 192.168.1.11, port 3000), splits the data in two equal chunks and stores them in the two DMA transmit buffers. After sending the transfer size to the remote core, the DMA transfers from memory to PL are executed through the *dma-proxy* driver. Since this is done through blocking *ioctl()* calls, a thread is created for the second transfer to enable both transmit channels to be operating simultaneously.

Finally, the results are acquired from the specified memory location and sent back to the computer through the socket, before closing the Ethernet connection and shutting down the remote processor.

In PetaLinux, we created a project based upon the same hardware platform specification that was used for the Linux SMP application. The *transmitter* application was added to the root file system as a custom user space application. The remote firmware ELF image was also included in the file system as a prebuilt application. Finally, the *dma-proxy* driver was added to the project as a custom kernel module.

From the PetaLinux kernel configuration menu, we enabled *user space firmware loading support*, as well as *remoteproc* and *RPMsg* drivers [23]. We also configured the system to boot with the static IP address 192.168.1.11 for the Ethernet interface.

In the Device Tree, we added an appropriate entry to declare a reserved memory section for the remote processor and the interrupt numbers for the *RPMsg* channels [23].

With the above changes done, the PetaLinux boot image was successfully generated for the Zynq device.

3.4.4 AMP Issues and Solutions

During the implementation of the system, we encountered some issues relevant to the Asymmetric Multiprocessing of the Zynq CPU cores. It would be useful to describe those issues and the way they were resolved.

a) Synchronization between CPUs

Since both CPUs operate simultaneously in AMP, it is necessary to implement a synchronization mechanism between them, if the tasks assigned to them present dependencies. The *OpenAMP* framework provides an easy way to synchronize the master and remote applications which relies on the *RPMsg* component. In particular, the applications can exchange simple messages through the *RPMsg* channel established between them to synchronize appropriately their execution flow.

For the *transmitter* and *receiver* applications, there are two cases where synchronization is needed. First, before starting transmitting data, the *transmitter* needs to make sure that the *receiver* has already initialized the AXI DMAs and they are ready to receive data from

the PL. If this is neglected, we observe a loss of initial data on the *receiver*'s side, due to an issue of the AXI DMA IP core. More specifically, in the absence of any initial setup time, the AXI DMA will pull the **TREADY** signal low after taking in 4 beats of streaming data, throttling the input data stream. To avoid this, the AXI DMA must be set up to run much before the actual data arrives [19]. For this reason, the *receiver* sends a *'READY'* message to the *transmitter* after setting up the DMAs. The *transmitter* cannot initiate the data transfer until this message has been received.

After the data have been received and processed by the remote processor, a second synchronization message is sent to the master. The *transmitter* obviously needs to know when output data are ready to be sent back to the computer. Therefore, the *receiver* sends an *'OK'* message when its task is accomplished to let the *transmitter* know that it can now access the right data.

b) Cache coherence issues

Cache inconherence can occur in an AMP system, when both processors access the same physical regions of the main memory. In the Zynq-7000 processing system, each CPU has its own L1 cache, but they share a common L2 cache. In the Linux/FreeRTOS AMP system, the Linux master (CPU0) takes control of the Snoop Control Unit (SCU) and manages L2 cache operations. To prevent problems with shared resources, L2 cache has been disabled for CPU1 by the *OpenAMP* framework [23].

However, when running the *transmitter-receiver* application, we observed that a few output data values were not correct. In fact, CPU1 did write the right values to the DDR memory but those were not visible to CPU0. These values always appeared in groups of four (4 × 8 bytes = 32 bytes), which is relevant to the 32-byte cache line size of the L2 cache. The assumption made was that CPU1 writes directly to the DDR memory, but CPU0 is not aware of that so it reads stale data from the L2 cache. Indeed, disabling the *L2x0 Cache Controller* from the PetaLinux kernel resolved the issue. However, the issue was further investigated in order to keep the L2 cache enabled for CPU0 for performance optimization. The final solution was to flush the cache data corresponding to the memory region where output data are stored by CPU1, just before accessing them by CPU0. This way, CPU0 is forced to read the most recent values from the DDR memory.

3.4.5 Testing the Testbed

We connect the Zybo with a computer through an Ethernet cable so that they can exchange data. The *BOOT.BIN* and *image.ub* files generated with PetaLinux are saved on a microSD card which is used to boot Linux on the board. The *input* file contains 4 MB (4.194.304 bytes) of data, as shown below:

input

```
0x1acee1123a5b8f34
0x583057301ceb78d7
0xe2e21d452d1017d2
0xca7eb622de1791db
…
```

After booting Linux, we have to load into the kernel the modules for the RPMsg communication (*rpmsg_user_dev_driver*) and the DMA handling (*dma_proxy*). The next step is to declare to *remoteproc* the name of the remote firmware and boot the remote processor with it. When the remote is up, we can run the *transmitter* application.

From the computer side, we just execute an application that reads data from the *input* file, sends them to the Zybo through Ethernet, waits to receive the results and writes them to the *output* file, as shown below:

output

```
0x061c75871724c60c
0x09f6678a0287b9b4
0x27f00263636d35fe
0xafac86b2d89d117a
…
```

The commands executed during the process described above, as well as the messages we receive through the execution, are presented below:

Computer

```
$ ./send_data_amp 4194304
Data sent to ZYBO.
Waiting for results…
Done! Results are located in output file.
```

ZYBO

```
# modprobe rpmsg_user_dev_driver
# modprobe dma_proxy

# echo receiver-amp.elf > /sys/class/remoteproc/remoteproc0/firmware
# echo start > /sys/class/remoteproc/remoteproc0/state
remoteproc remoteproc0: powering up remoterpoc@0
remoteproc remoteproc0: Booting fw image receiver-amp.elf, size
2608200
remoteproc remoteproc0: registered virtio0 (type 7)
virtio_rpmsg_bus virtio0: rpmsg host is online
CPU1: shutdown
remoteproc remoteproc0: remote processor remoteproc@0 is now up
virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-channel addr
0x1
rpmsg_user_dev_driver virtio0: rpmsg-openamp-channel:
rpmsg_user_dev_rpmsg_drv_probe
rpmsg_user_dev_driver driver virtio0: rpmsg-openamp-channel: new
channel: 0x400 -> 0x1!

# transmitter-amp 4194304
Starting DMA Data Transmitter.
rpmsg_user_dev_driver virtio0: rpmsg-openamp-channel: Sent init_msg
to target 0x1.
Data received from PC and stored in transmit buffers.
```

```
Starting DMA Data Receiver.
Receiving data…
Transmitting data with DMA 0…
Transmitting data with DMA 1…
Transmitted all data successfully.
Received all data successfully.
Completed data processing.
Exiting DMA Data Receiver.
rpmsg_user_dev_driver virtio0: rpmsg-openamp-channel: Sending
shutdown message.
virtio_rpmsg_bus virtio0: destroying channel rpmsg-openamp-channel
addr 0x1.
rpmsg_user_dev_driver virtio0: rpmsg-openamp-channel: Removing rpmsg
user dev.
rpmsg_user_dev_driver rpmsg0: Releasing rpmsg user dev device.
Sending results back to PC…
Done! Exiting DMA Data Transmitter.
```

Chapter 4

System Evaluation

4.1 Isolation Between Cores

An important feature of the implemented testbed that needs to be evaluated is the isolation level between the two ARM cores of the Zynq processing system. The *transmitter* and *receiver* applications represent two independent entities, running on separate CPUs of the device. The *transmitter*, apart from initiating the DMA data transfers, should be able to execute other independent tasks, while the *receiver* executes its own critical task on the received data. We will examine how these tasks affect the *receiver's* behavior in SMP (default SMP and BMP) and AMP environments in order to evaluate the isolation of the system in each case.

4.1.1 Inter-Task Interference

In multicore embedded systems, isolating critical processes is imperative when addressing real-time issues. Heavy processor load may cause inter-task interference between processor-intensive and time-critical tasks, in the absence of isolation in the system [24]. Telecommunication applications, in particular, are typically designed with respect to a planar architecture pattern: software is divided into management, control and user planes, which present different performance requirements. Here, inter-task interference is translated into cross-plane influence, which can have a significant impact on timing performance [25]. The different tasks performed in a multicore processor may also suffer from interferences when accessing shared hardware resources at the same time, like a shared bus, cache or main memory [26]. This is consequently another factor that can affect the timing behavior of critical tasks.

We conducted an experiment to examine inter-task interference in our system and evaluate whether the AMP model offers better isolation than SMP. Two different scenarios were explored depending on the interference source, one for a CPU-intensive task and the other for a memory-intensive one.

In the first scenario, one 4 MB packet is transferred between the *transmitter* and *receiver*. The *receiver* performs intensive processing on received data, which corresponds to the time-critical task. The data processing is performed by accessing the data in successive rounds (1-100) and executing comparisons and math functions (multiplications and divisions) on them. At the same time, after remaining idle for an initial time interval, the *transmitter* starts executing an independent CPU-intensive task, similar to the one mentioned above. Two threads are created for that, each one executing the same task. Later, the *transmitter* repeats the CPU-intensive task, but this time the number of threads is increased to five, so that the response of the system to a heavier workload is made visible. The time that the *receiver* takes to complete each processing round is measured in order to evaluate the interference from the CPU-intensive task.

Figure 4.1 illustrates the results acquired during the realization of the scenario described above, for the SMP, BMP and AMP configurations of the testbed. It is clear that inter-task interference

is indeed observed in the case of SMP, as the execution time of processing rounds increases significantly when the "adversary" task is executed in parallel. In fact, the interference level is proportionate to the weight of the workload inflicted on the system: five threads cause greater slow-down on critical processing than two threads. This is due to thread migration among the cores of the processor. SMP allows all tasks to be assigned on any core, thus some of the workload threads steal processing time from CPU1, suspending the critical task and eventually increasing its execution time. On the other hand, in BMP and AMP configurations, the *receiver* does not seem to be influenced by the CPU-intensive tasks. Therefore, we can assume that allocating a critical task to a different core than other CPU-intensive tasks is an effective isolation technique, in terms of timing performance. This can be achieved either in the same operating system, with CPU-bound processes, or by assigning tasks to separate operating systems, in an AMP design.

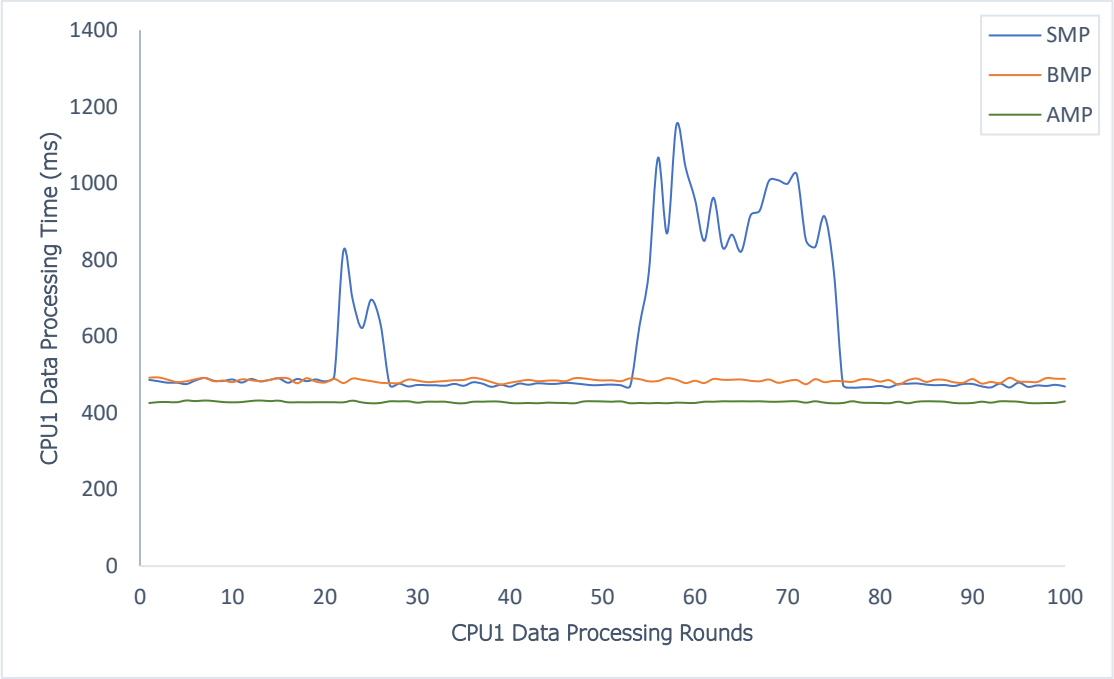


Figure 4.1: Critical processing time interference from CPU-intensive task

The second scenario explores inter-task interference originating from access to shared hardware resources. In our experiment, the shared resource between the two cores is the DDR memory. A total of 3000 packets, each one of size 4 MB, are transferred from memory to PL and vice versa by the *transmitter* and *receiver* applications. The transfers are done through DMA, so they do not consume much of the CPUs processing time, but instead they introduce heavy traffic between the PS and the DDR memory. Meanwhile, the *transmitter* starts at some point a memory-intensive task, which consecutively accesses memory and performs multiple write and read operations. Here, the critical time measured is the time needed for a complete transfer of 10 packets (that is, the time needed for 10 packets to move from memory to the PL and back to memory), from which the throughput can be found.

Figure 4.2 shows how the throughput varies during the transferring of the packets in SMP, BMP and AMP environments. We observe that SMP and BMP suffer from inter-task interference, since the simultaneous execution of the memory-intensive task causes a considerable degradation in throughput. Approximately, a 4% decrease in throughput is observed in SMP, and a 12% decrease in BMP. In contrast, in AMP the throughput remains unchanged at 760 MB/s during the whole experiment, which indicates that the separation of the operating systems provides good

isolation. This scenario certainly creates memory bus congestion for the dual-core processor as a means of interference. However, the fact that the AMP design is not influenced makes us assume that another resource is mainly responsible for the interference observed in SMP and BMP. This is probably the L2 cache, which is a shared resource between cores when they run one single OS and introduces conflicts between the memory-intensive task and the packet transfers. In addition, memory coherence protocols are required, which may also affect performance. Write-through protocols, in particular, generate a large amount of traffic and amplify inter-task interference [26]. In AMP, on the other hand, L2 cache is not used by CPU1, keeping the rate at which the *receiver* accesses data from memory steady. Therefore, when a system involves the execution of a critical task among others, clean separation of resources and allocation to appropriate cores can reduce inter-task interference and improve the performance of the critical task.

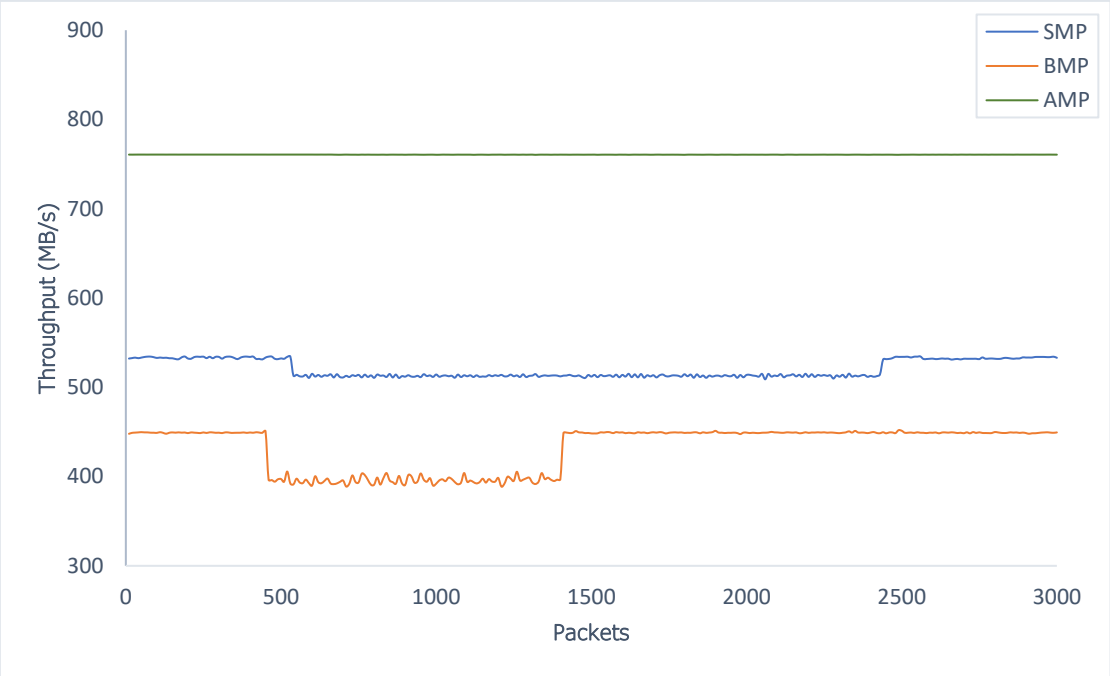


Figure 4.2: Throughput interference from memory-intensive task

4.1.2 Safety and Data Integrity

Apart from timing isolation, which guarantees that there is no interaction between tasks affecting their timing behavior, functional isolation is equally important, such that a bug or misbehavior in a function does not affect other tasks [26]. The *transmitter* and *receiver* applications run independently in our system, both in SMP and AMP environments. This means that if a problem appears during the execution of one of them, it will not generally affect the functionality of the other. Nevertheless, in case issues related to the operating systems are introduced, AMP certainly provides an additional layer of protection, by keeping the RTOS and Linux operating systems separated on different cores.

Another aspect of functional isolation has to do with data integrity. A typical example is when a task is able to access memory belonging to other tasks and corrupt the data they manipulate. We conducted an experiment to test this scenario in our system. The *transmitter* application was forced to access the buffers containing the data processed by the *receiver* and zero out all the values. We observed that it was allowed to do so, no matter which OS configuration was applied to the system. Although this result was expected in SMP, where all processes share the same memory, it was

useful to confirm that even AMP could not prevent this kind of memory corruption between tasks. The remote firmware, along with the FreeRTOS kernel, do run in a dedicated memory space, outside of the context of PetaLinux memory. However, PetaLinux can read and write this address space just like any other device memory [27].

Data integrity issues are also related to the use of cache memories. Cache coherence must be maintained among different cores, as well as potential errors that occur on cached data should be corrected. Although a write-back policy reduces the overhead on the execution time of interfered tasks, it requires more complex consistency protocols (MESI) and is less robust in terms of fault tolerance, requiring implementation of error correction schemes in higher levels [26]. In other words, it is possible to face a trade-off between optimization of timing performance and safety maintenance. When our system runs in SMP mode, the L2 cache is shared between CPU0 and CPU1. Consequently, dealing with the above issues is more complicated than in AMP, where L2 cache is private to CPU0. This design option, as long as it has not an impact on CPU1 performance, gives the AMP architecture a simplified programming model and isolates CPU1 from the L2 cache-related safety issues of CPU0.

However, disabling L2 cache for CPU1 still hides some risks, in case both CPUs have to access the same memory location. We already came across this scenario during the implementation of the AMP system, where CPU1 processes data by writing directly to the DDR memory. It was observed that when the same memory addresses were accessed from CPU0, not all the values read were up to date. A flush of the L2 cache for that specific memory location was necessary to read the correct values from the main memory. This scenario makes it clear that special care should be taken when isolating caches and sharing memory between CPUs, in order to ensure data integrity.

4.2 Performance Comparison

The Linux SMP and Linux/RTOS AMP implementations of the testbed present some performance differences that originate from the nature of the operating systems used in each case. The purpose of the following experiments is to test the behavior of our system in various scenarios that will reveal whether the execution of the *receiver* application in FreeRTOS (AMP) has better performance than in PetaLinux (SMP) or not.

4.2.1 Processing Speed

The first scenario aims to compare the speed of data processing from the *receiver* running on CPU1, under Linux and RTOS operating systems. A 4 MB packet is transferred through DMA between the *transmitter* and the *receiver*. The *receiver* performs intensive processing on received data, by executing comparisons and math functions (multiplications and divisions) on the values. The processing is repeated on the entire data buffer for a maximum number of successive rounds. We measured the time needed by the *receiver* to complete the data processing, setting each time a higher limit to the number of processing rounds.

The results are shown in Figure 4.3, both for Linux and RTOS. We can observe that data processing by CPU1 in RTOS is considerably faster than in Linux. In fact, as the maximum processing rounds increase, the speed up gradually increases and eventually reaches a percentage of about 25%, as illustrated in Figure 4.4. This can be very important when considering the real-time constraints that the processing task may be bound to.

The speed up offered by RTOS can be explained by the fact that the task is less affected by the OS operation than in Linux, thus having higher performance when run in isolation [26]. Moreover, RTOS gives full priority to the task, pushing its responsiveness, whereas in Linux it is not guaranteed that any task will monopolize the services of the processor.

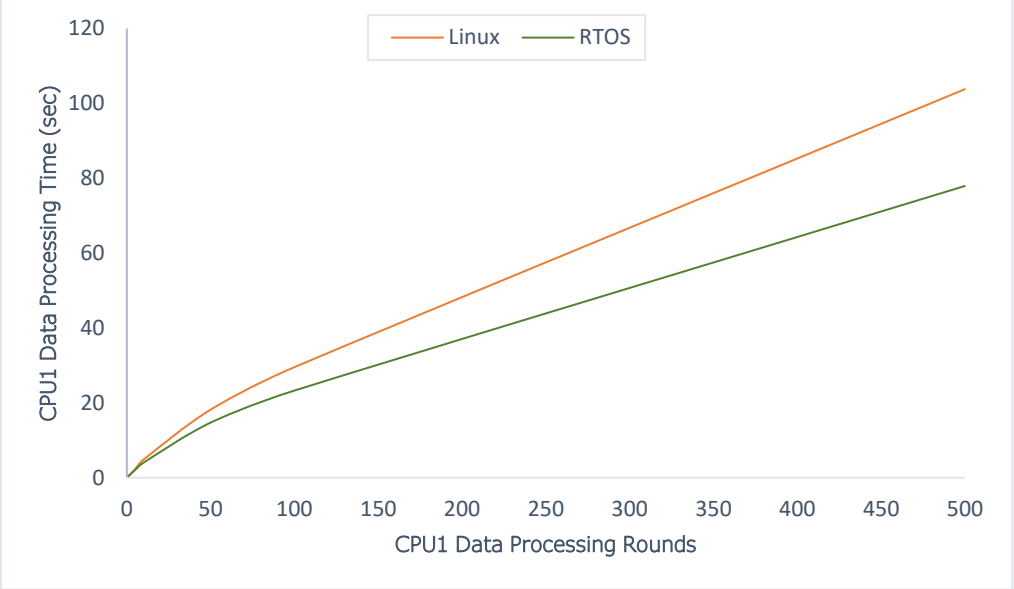


Figure 4.3: CPU1 data processing time in Linux and RTOS

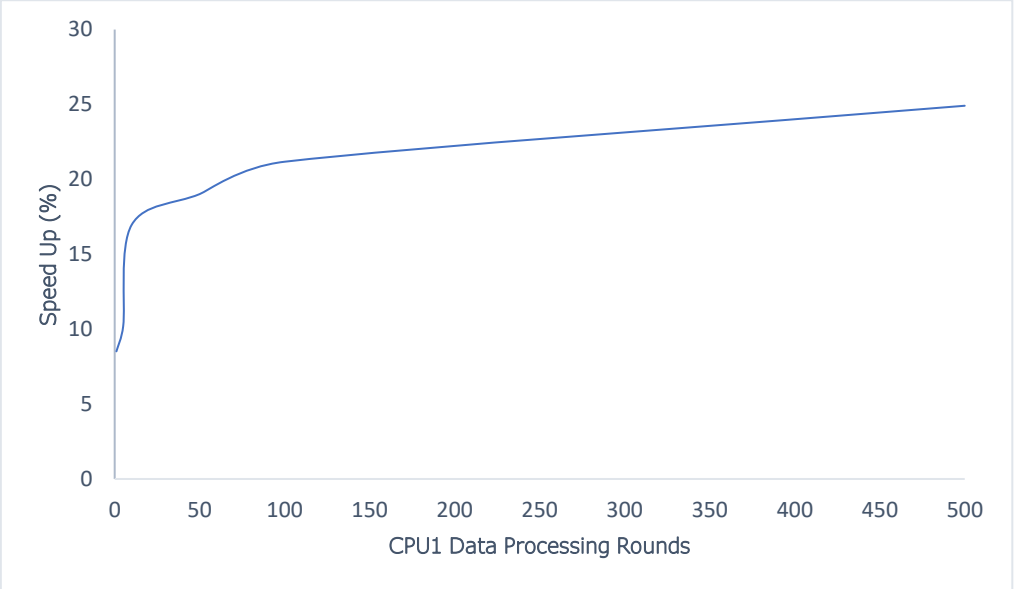


Figure 4.4: RTOS vs. Linux speed up in CPU1 data processing

In order to study the performance dependency from the workload executed by CPU1, the scenario described above was repeated after modifying slightly the way that the *receiver* processes data. Instead of multiplications and divisions, the task was configured to perform additions and subtractions on the received values. Every other parameter remained the same, including how the data buffer is accessed and how many successive processing rounds are executed.

The time measurements for the new CPU1 workload are seen in Figure 4.5. Of course, data processing by CPU1 in RTOS is again faster than in Linux. However, Figure 4.6 shows a significant increase in speed up, compared to the previous workload. The speed up reaches now the maximum value of 40%. That means that the kind of processing done by CPU1 greatly affects the performance difference between RTOS and Linux. It is therefore important to process data in a way that offers optimal CPU utilization.

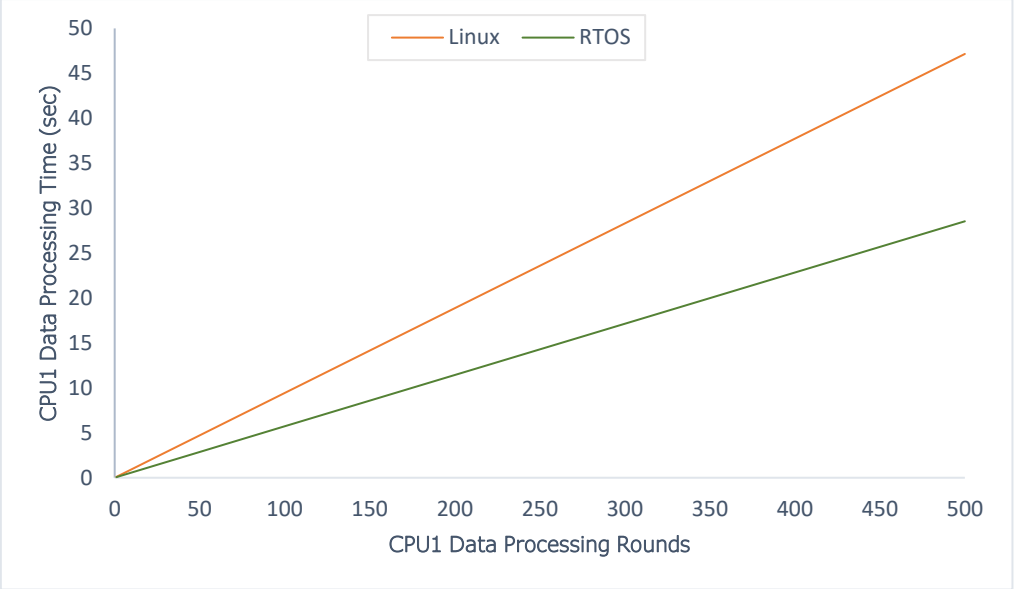


Figure 4.5: CPU1 data processing time in Linux and RTOS (second workload)

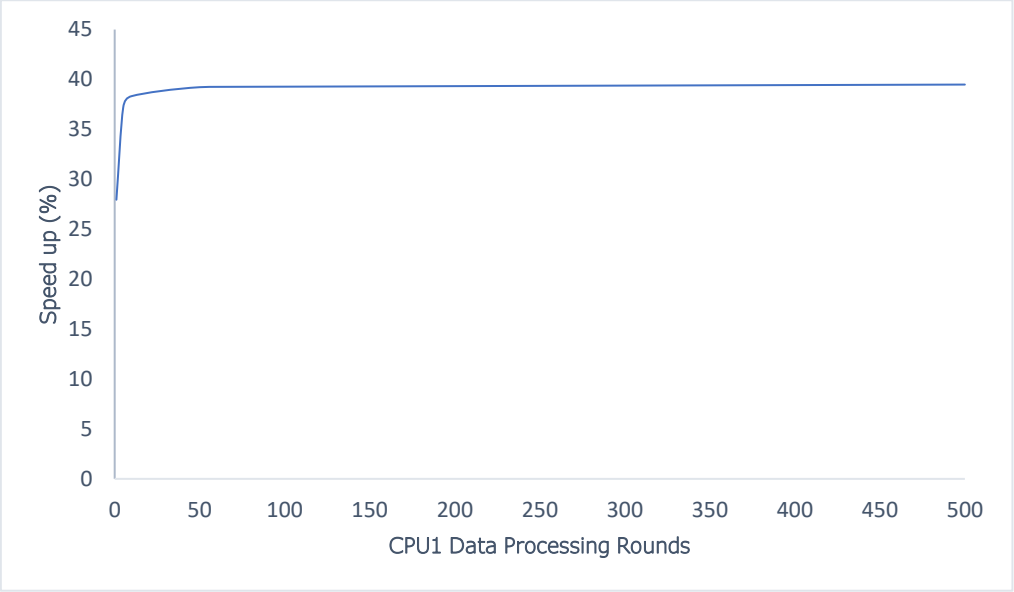


Figure 4.6: RTOS vs. Linux speed up in CPU1 data processing (second workload)

4.2.2 Execution Time Variation

The following experiment evaluates the variation of the CPU1 execution time for the data processing performed by the *receiver* application. Again, a 4 MB packet is transferred and processed like before for a total of 100 successive rounds. For each processing round, the time needed by the *receiver* to process the data is measured, under Linux and RTOS operating systems.

Figure 4.5 shows the results of the experiment. It is clear that the CPU1 execution time presents considerably higher variation in Linux, compared to RTOS. More specifically, a maximum variation of 2.7% is measured for Linux, whereas in RTOS it is just 0.3%. This is due to the deterministic scheduling favored by RTOS, which minimizes the difference between the minimal and maximal response time of the task (jitter) [24]. The absence of high jitter values means

predictable responsiveness and deterministic execution time, which are definitely required in an application with strict real-time constraints.

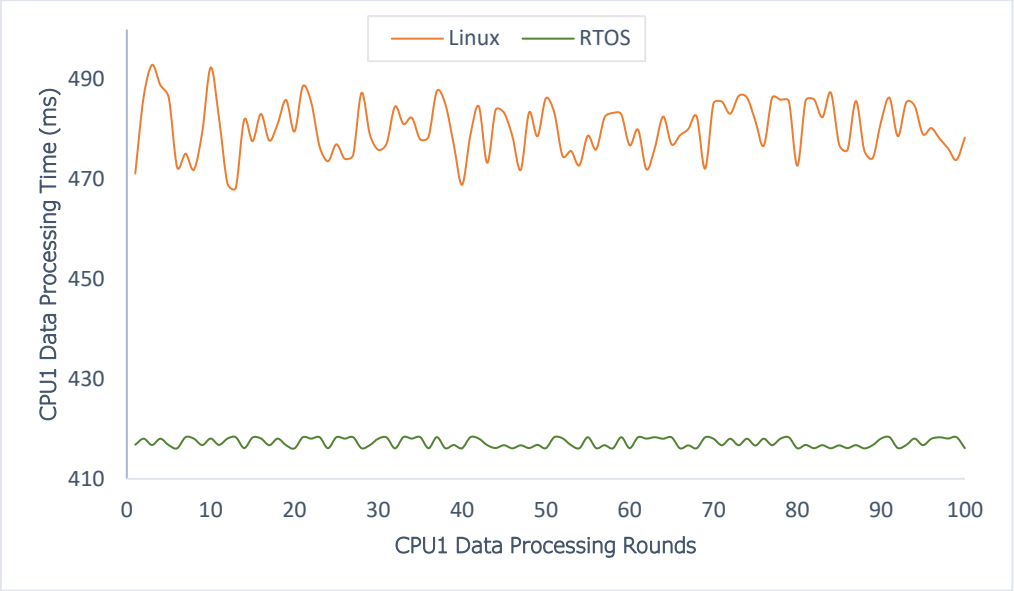


Figure 4.7: Variation of CPU1 processing time in Linux and RTOS

4.2.3 Data Transfer Speed

The last scenario involves intensive transfer of data from the *transmitter* to the *receiver*. It is used to examine how fast we can move data in our system and compare the performance in Linux SMP and Linux/RTOS AMP environments. A gradually increasing number of packets, each one of size 4 MB, are transferred through DMA from memory to PL and vice versa. The elapsed time for the complete transfer of all packets is measured each time, starting from 10 packets and ending at 1000 packets. The comparison of the measured transfer times can be seen in Figure 4.6 below.

Figure 4.8: Data transfer time in Linux and RTOS

Clearly, data movement in Linux/RTOS AMP is achieved much faster than in Linux SMP, regardless of the total amount of packets that need to be transferred. In Table 4.1, the average throughput of the system is given for both cases, as well as the resulting speed up that AMP provides compared to SMP. This was measured to be 69.1%, which is a great improvement in terms of performance.

Since the *transmitter* application runs under Linux in both cases, the cause of the difference should be searched at the *receiver's* side. Indeed, the transfer of packets from PL to memory is performed through the AXI DMAs in a different way under Linux and RTOS. In Linux, the *receiver* application, running in user space, manages the DMA engines through a hierarchy of OS-specific DMA drivers, residing in kernel space (DMA Proxy design). The drivers and software components of the Linux OS introduce latencies to the initial set up operations of the AXI DMAs [28]. This overhead, though, is much lower in RTOS, where the *receiver* programs directly the AXI DMAs through specific function calls, without any intermediate software layer. However, the speed at which data are transferred through the DMA channels should be the same for both operating systems, as this is performed by hardware. It should also be noted that the initialization overhead has a greater impact on performance for small amounts of data, whereas it can ultimately be negligible for larger transfers [28].

	Linux	***RTOS***
Average Throughput	450 MB/s	761 MB/s
Speed Up	69.1%	

Table 4.1: RTOS vs. Linux throughput

Chapter 5

Conclusions

5.1 Thesis Summary

Multicore processors, combined with the benefits of SoC FPGAs, constitute an ideal platform for testing and implementing applications that require isolated processing of streaming data. In this thesis, we demonstrated how to build a testbed for such a telecom application on the Zybo development board, taking advantage of the hardware/software programmability of the Zynq SoC and the different multiprocessing environments supported by the dual-core ARM processor of the device.

From a hardware perspective, an efficient method for streaming fast large amounts of data was applied, using Direct Memory Access and the AXI4-Stream Protocol. The hardware design was optimized in terms of throughput, after an extensive exploration of the system's capacities. The available resources in the FPGA were also exploited for the processing of the transmitted data.

Special emphasis was placed on the possible configurations of operating systems on the testbed. A Linux SMP and a Linux/RTOS AMP design were deployed and eventually compared to each other. A series of conducted experiments applied various scenarios on the system, in order to evaluate the isolation between the cores and the processing performance, in both cases.

In conclusion, we can say that AMP provides better isolation, reducing inter-task interferences and further ensuring reliability and data integrity in the system. Furthermore, the use of a RTOS is also significant, when faster processing and data transfer are desired, as well as improved time predictability of a specific task.

5.2 Future Work

As an extension of this thesis, we would suggest the modification of the functionality of the implemented testbed towards a real-life application. The multiplier used for data processing in the programmable logic could be easily replaced by another AXI4-Stream compliant component, that would process data with respect to a more complex and realistic algorithm. The critical processing performed on received data by CPU1 could also be adapted to the application requirements. Finally, instead of streaming random data from a computer file, the device could be configured to acquire real data from the outside world.

Moreover, future work includes the implementation of the testbed on other platforms, more powerful and feature-rich that the Zynq-7000 SoC of the Zybo. In particular, the Zynq UltraScale+ MPSoC device from Xilinx should definitely be considered as a target platform. Its extended architecture, combining ARM Cortex-A53 application processors and real-time ARM Cortex-R5 processors, would enable true heterogeneous multiprocessing on data. It would be interesting to investigate whether the heterogeneous cores offer any improvement to the AMP design and evaluate the isolation and performance of the system on such a device.

Finally, virtualization technologies could be tried as an alternative technique to native AMP. Instead of clearly separating two operating systems on the dual-core processor of the Zynq processing system, a hypervisor could be used to create a single software layer that will be capable of managing multiple heterogeneous operating systems on the same hardware. This way, the testbed could be extended to support more than two separate OS domains. Again, it would be useful to evaluate the isolation and safety level between different domains, as well as investigate the circumstances under which performance can be optimized in such virtual environments.

References

[1] https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[2] http://mazsola.iit.uni-miskolc.hu/cae/docs/pld1.en.html

[3] http://www.ni.com/en-us/innovations/white-papers/08/fpga-fundamentals.html

[4] http://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-FPGA.html

[5] Altera Corp., "What is a SoC FPGA?", Architecture Brief, July 2014. Available on: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf

[6] P. Gepner and M. F. Kowalik, "Multi-Core Processors: New Way to Achieve High System Performance", *International Symposium on Parallel Computing in Electrical Engineering (PARELEC)*, Bialystok, 2006, pp. 9-13.

[7] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", Fourth Edition, Elsevier, Inc., 2007, pp. 198-202.

[8] https://blog.nxp.com/tech-in-depth/3-reasons-why-embedded-heterogeneous-systems-are-more-efficient

[9] Digilent, Inc., ZYBO FPGA Board Reference Manual, February 27, 2017. Available on: https://reference.digilentinc.com/_media/reference/programmable-logic/zybo/zybo_rm.pdf

[10] Xilinx, Inc., Zynq-7000 All Programmable SoC Technical Reference Manual, UG585 (v1.12.1), December 6, 2017.

[11] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[12] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, "The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC", First Edition, Strathclyde Academic Media, 2014.

[13] Xilinx, Inc., AXI Reference Guide, UG761 (v13.1), March 7, 2011.

[14] ARM, AMBA 4 AXI4-Stream Protocol Specification, Version 1.0, 2010.

[15] A. F. Harvey and Data Acquisition Division Staff, "DMA Fundamentals on Various PC Platforms", National Instruments Application Note, April 1991.

[16] A. Choudhary, J. B. Chavda, A. P. Ganatra and R. J. Nayak, "Performance evaluation PL330 DMA controller for bulk data transfer in Zynq SoC", IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), Bangalore, 2016, pp. 1811-1815.

[17] M. Christofferson, "Fundamentals of Multicore Operating Systems for Telecom/Networking Applications", Enea White Paper, 2010. Available on: http://www.rtcgroup.com/whitepapers/files/enea.pdf

[18] Freescale Semiconductor, Inc., "Running AMP, SMP or BMP Mode for Multicore Embedded Systems", Beyond Bits Magazine, Power Architecture Edition, 2012. Available on: https://cache.freescale.com/files/32bit/doc/brochure/PWRARBYNDBITSRAS.pdf

[19] Xilinx, Inc., AXI DMA v7.1 LogiCORE IP Product Guide, PG021, October 4, 2017.

[20] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842418/Linux+DMA+From+User+Space

[21] Xilinx, Inc., PetaLinux Tools Reference Guide, UG1144 (v2017.3), October 4, 2017.

[22] Mentor Graphics Corp., OpenAMP Framework User Reference, 2010-2014.

[23] Xilinx, Inc., Libmetal and OpenAMP for Zynq Devices User Guide, UG1186 (v2017.3), January 4, 2018.

[24] P. Cordemans, N. De Witte, J. Vankeirsbilck, W. Melis and J. Boydens, "Isolating real-time from processor-intensive processes in embedded multi-core systems", International Scientific and Applied Science Conference (Electronics-ET), Sozopol, 2015.

[25] N. De Witte, R. Vincke, S. Van Landschoot, E. Steegmans and J. Boydens, "Comparing Dual-Core SMP/AMP Performance on a Telecom Architecture", International Scientific Conference Electronics (ET), Sozopol, 2013, pp. 72-75.

[26] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello and F. J. Cazorla, "Assessing the suitability of the NGMP multi-core processor in the space domain", International Conference on Embedded Software (EMSOFT), Tampere, 2012, pp. 175-184.

[27] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841668/Multi-OS+Support+AMP+Hypervisor

[28] M. Fularz, D. Pieczyński and M. Kraft, "The performance comparison of the DMA subsystem of the Zynq SoC in bare metal and Linux applications", Measurement Automation Monitoring, vol. 63, no. 5, pp. 189-191, 2017.