Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Υλοποίηση Συστήματος Δυναμικής Διαχείρισης Μνήμης σε **FPGA** μέσω τεχνικών Υψηλού Επιπέδου Σύνθεσης

Διπλωματικη Εργασια

του

Στέφανου Κόφφα

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

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Μάιος 2016

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Υλοποίηση Συστήματος Δυναμικής Διαχείρισης Μνήμης σε **FPGA** μέσω τεχνικών Υψηλού Επιπέδου Σύνθεσης

Διπλωματικη Εργασια

του

Στέφανου Κόφφα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Μαΐου 2016.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Οικονομάκος
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Μάιος 2016

(Υπογραφή)

.....

Στεφανος Κοφφας

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

© 2016 – All rights reserved

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Copyright ©–All rights reserved Στέφανος Κόφφας, 2016.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Acknowledgements

First, I would like to express my gratitude to professor Mr. Dimitrios Sountris for giving me the opportunity to work on this challenging subject. His teaching approach and his insightful and extensive research have strongly increased my motivation and commitment to work on this project.

I also feel obliged to thank Dionisis Diamantopoulos. Acting not only as a scientist but also as a friend, he helped me, overcome every obstacle and difficutly I encountered during this process.

I would like to personally thank my friends Dimitris, Dimitris, Ilias and Thanasis. Their achievements and the time we spent together have contributed to the successfull completion of my studies.

Last but not least, I would like to thank my familly that supported me in the pursuit of my dreams and accomplishment of my personal goals.

Stefanos G. Koffas

Περίληψη

Αυτήν τη χρονική περίοδο οι μηχανικοί του κλάδου προσπαθούν να κατασκευάσουν υπερυπολογιστές που θα έχουν απόδοση της τάξης των 10^{18} πράξεων κινητής υποδιαστολής ανα δευτερόλεπτο (ExaFLOPS). Αρκετές σχετικές μελέτες έχουν δείξει ότι για να γίνει αυτό θα πρέπει να υιοθετήσουμε ένα αρχιτεκτονικό μοντέλο που εκμεταλλεύεται τα πλεονεκτήματα της συνύπαρξης υλικού (hardware) και λογισμικού (software). Για αυτόν τον σκοπό έχουν προταθεί οι ετερογενείς αρχιτεκτονικές πολλών επιταχυντών υλικού. Τα εργαλεία υψηλού επιπέδου σύνθεσης διευκολύνουν την δημιουργία συστημάτων με πολλούς επιταχυντές υλικού και για αυτό αναμένεται να διαδραματίσουν καθοριστικό ρόλο στην επίτευξη αυτού του σκοπού.

Τα FPGA αποτελούν μία ελκυστική πλατφόρμα ανάπτυξης αρχιτεκτονικών πολλαπλών επιταχυντών υλικού, μέσω της εγγενούς ευελιξίας επαναπρογραμματισμού τους καθώς και της ενεργειακής τους απόδοσης. Ωστόσο, η οργάνωση της μνήμης αποτελεί τον κυριότερο περιοριστικό παράγοντα στις αρχιτεκτονικές με πολλούς επιταχυντές. Προηγούμενες μελέτες έχουν δείξει ότι η στατική δέσμευση μνήμης - ο de facto μηχανισμός δέσμευσης μνήμης που υποστηρίζεται από τα σύγχρονα εργαλεία - είναι η κύρια αιτία της υποχρησιμοποίησης πόρων. Μία πρόσφατη προσέγγιση επεκτείνει τις σύγχρονες μεθόδους Υψηλού Επιπέδου Σύνθεσης μέσω ενός συστήματος δυναμικής διαχείρισης μνήμης που μπορεί να ενσωματωθεί στην σύνθεση συστημάτων με πολλαπλούς επιταχυντές.

Η παρούσα διπλωματική εργασία α) επεκτείνει τους μηχανισμούς δέσμευσης και αποδέσμευσης μνήμης για να βελτιστοποιήσει την αποδοση τους σύμφωνα με τις απαιτήσεις κατά την εκτέλεση μίας εφαρμογής β) αναπτύσει μία νέα αρχιτεκτονική για την λίστα με τα ελεύθερα μπλοκ μνήμης και γ) υλοποιεί δύο εναλλακτικούς αλγορίθμους δεσμευσης μνήμης σε συνθέσιμο C κώδικα (Next Fit, Best Fit). Το προτεινόμενο σύστημα αξιολογήθηκε με την βοήθεια του Vivado HLS με μία σειρά από πειράματα με υψηλές απαιτήσεις σε μνήμη. Η πειραματική ανάλυση έδειξε ότι η νέα αυτή αρχιτεκτονική αυξάνει κατά πολύ την ταχύτητα του συστήματος (μέχρι και 40χ) ενώ παράλληλα μειώνει και την χρησιμοποίηση των πόρων του FPGA (-21% φλιπ-φλοπ, -10% LUTs, -10% block-RAMs).

Λέξεις Κλειδιά

FPGA, Δυναμική Διαχείριση Μνήμης, HLS, Αλγόριθμος Πρώτης Τοποθέτησης, Αλγόριθμος Καλύτερης Τοποθέτησης, Αλγόριθμος Επόμενης Τοποθέτησης

Abstract

Breaking the exascale barrier has been recently identified as the next big challenge in computing systems. Several studies, showed that reaching this goal requires a design paradigm shift towards more aggressive hardware/software co-design architecture solutions. Recently, many-accelerator heterogeneous architectures have been proposed to overcome the utilization/power wall.

FPGAs form an intresting solution for many-accelerator architectures. Their flexibility and programmability enables the implementation of several types of hardware accelerators compared to traditional ASICs. However, their memory organization forms a significant bottleneck in the performance of many-accelerator architectures. Previous studies showed that static memory allocation - the de-facto mechanism supported by modern design techniques and synthesis tools - forms the main source of "resource under-utilization" problems. A recent approach extends conventional High Level Synthesis (HLS) with dynamic memory allocation/deallocation mechanisms to be incorporated during many-accelerator synthesis.

This diploma thesis a) extends the allocation/deallocation mechanisms in order to further optimize the efficiency of the memory reservation to the application runtime memory requirements, b)develops a new architectural approach of the free-list organization and c) implements two alternative allocation algorithms in synthesizable C code (Next Fit, Best Fit). The proposed framework is seamlessly integrated with the industrial strength Vivado-HLS tool and its effectiveness is evaluated with a set of memory intensive application scenarios. The analysis showed that the proposed architectural approach delivers significant speedup over the previous implementation (up to 40x) in addition to lower FPGA resource utilization (-21% flip-flops, -10% LUTs, -10% block-RAMs).

Keywords

FPGA, High Level Synthesis (HLS), Dynamic Memory Management (DMM), DMM-HLS, Vivado HLS, First Fit, Next Fit, Best Fit, Memluv

Περιεχόμενα

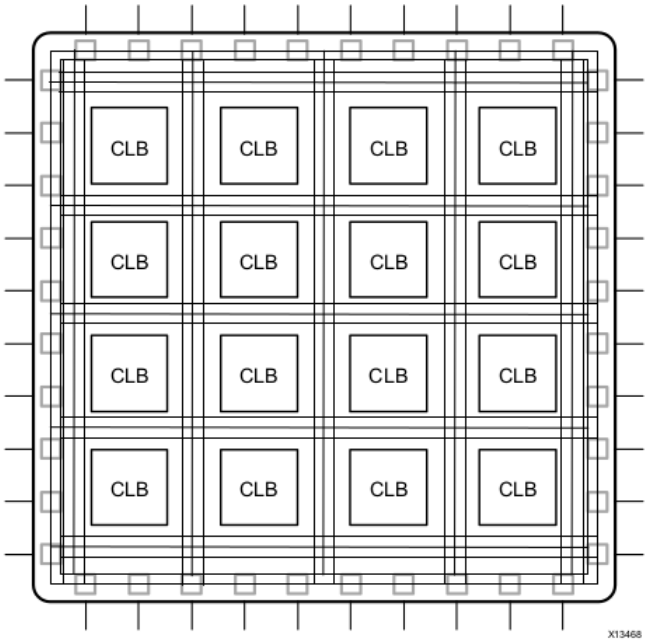
Κεφάλαιο 1

Εισαγωγή

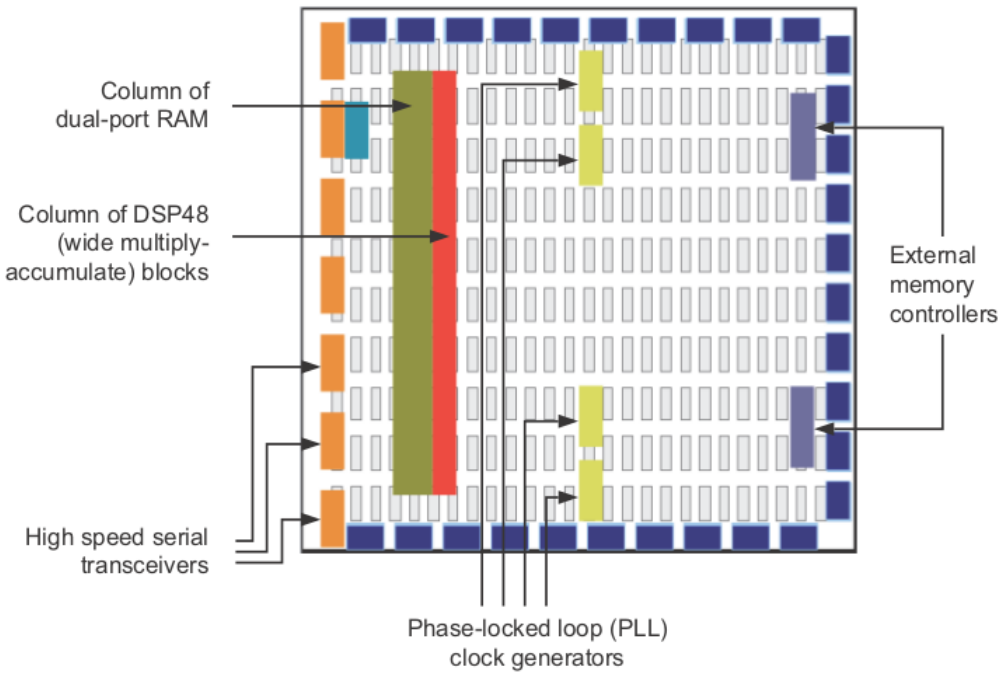
Η συστοιχία προγραμματιζόμενων πυλών (Field Programmable Gate Array-FPGA) είναι ένα κύκλωμα VLSI το οποίο μπορεί να προγραμματιστεί από τον χρήστη αρκετές φορές μετά την κατακευή του. Τα FPGA είναι κατά βάση συσκευές παράλληλης επεξεργασίας και μπορούν να εκτελέσουν ταυτόχρονα πολλούς υπολογισμούς. Για παράδειγμα, κατά τον προγραμματισμό τους είναι να δυνατόν να δημιουργηθεί παραπάνω από μία Αριθμητική και Λογική Μονάδα (Arithmetic Logic Unit-ALU). Αυτό έχει ως αποτέλεσμα, σε αντίθεση με τους επεξεργαστές κοινού σκοπού, να μπορούν να εκτελούν παραπάνω από μία εντολές ταυτόχρονα. Ένα ακόμα σημαντικό πλεονέκτημα των FPGA είναι η χαμηλή κατανάλωση ισχύος. Αυτοί είναι και οι κύριοι λόγοι που τα FPGA είναι ευρέως χρησιμοποιούμενα σε τομείς της τεχνολογίας όπως η αεροδιαστημική και η αμυντική βιομηχανία, η επεξεργασία ψηφιακών σημάτων (εικόνα και ήχος), η αυτοκινητοβιομηχανία, οι κινητές και σταθερές επικοινωνίες, τα κέντρα δεδομένων (Datacenter), η πληροφορική υψηλής απόδοσης (High Performance Computing-HPC) κ.ά.

Το δομικό στοιχείο ενός FPGA είναι το επαναδιαμορφούμενο μπλοκ (Configurable Logic Block-CLB). Το κάθε CLB αποτελείται από πίνακες αναφοράς (Lookup Tables-LUT) οι οποίοι υλοποιούν τον πίνακα αλήθειας διαφορετικής συνάρτησης κάθε φορά, από φλιπ-φλοπ (flip flop-FF) τα οποία αποθηκεύουν το αποτέλεσμα ενός LUT και από θύρες εισόδου/εξόδου (I/O ports). Τα CLB συνδέονται μεταξύ τους με μία προγραμματιζόμενη υποδομή καλωδίωσης, η οποία χρησιμοποιείται για τη διαμόρφωση του FPGA (σχήμα 1.1). Κατά τα τελευταία χρόνια, η αρχιτεκτονική του FPGA έχει ενισχυθεί με επιπλέον στοιχεία. Τα σημαντικότερα από αυτά είναι τα DSP48 και τα μπλοκ μνήμης RAM (BRAM) που αποτελούνται από 18 Kbit το καθένα (σχήμα 1.2).

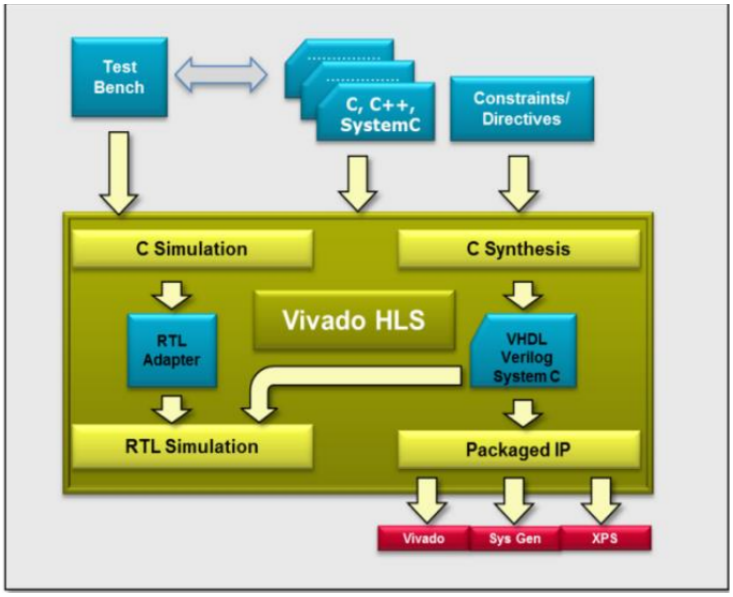
Παραδοσιακά, ο προγραμματισμός των FPGA γίνεται μέσα από τη χρήση γλωσσών περιγραφής υλικού (Hardware Definition Language-HDL). Η ολοκλήρωση όμως αυτής της διαδικασίας είναι πολύ χρονοβόρα και αποτρεπτική για έναν μηχανικό που ασχολείται κυρίως με γλώσσες υψηλού επιπέδου όπως η C/C++. Για αυτόν τον λόγο, τα τελευταία χρόνια έχουν αναπτυχθεί εργαλεία υψηλού επιπέδου σύνθεσης (High Level Synthesis-HLS). Τα εργαλεία αυτά μετατρέπουν κώδικα από μία γλώσσα υψηλού επιπέδου, όπως είναι η C, σε γλώσσα περιγραφής υλικού (HDL). Αυτά τα εργαλεία παρέχουν στον χρήστη τη δυνατότητα να επηρεάσει την παραχθείσα αρχιτεκτονική μέσα από ένα σύστημα οδηγιών βελτιστοποίησης (optimization



Σχήμα 1.1: Αρχιτεκτονική ενός FPGA



Σχήμα 1.2: Αρχιτεκτονική ενός σύγχρονου FPGA



Σχήμα 1.3: Διαδικασία υψηλού επιπέδου σύνθεσης με το Vivado HLS

directives). Σε αυτή τη διπλωματική για την υψηλού επιπέδου σύνθεση θα χρησιμοποιηθεί το Vivado HLS. Όπως φαίνεται και στο σχήμα 1.3 πέρα από τις προδιαγραφές του συστήματος και τις οδηγίες βελτιστοποίησης, μπορεί να δημιουργηθεί και το σενάριο ελέγχου (test bench) σε γλώσσα υψηλού επιπέδου. Αυτό το αρχείο χρησιμοποιείται για την προσομοίωση του συστήματος και στο υψηλό επίπεδο για την διόρθωση λογικών λαθών (C simulation) αλλά και στο επίπεδο υλοποίησης (RTL simulation) για τη διόρθωση λαθών που προέκυψαν από τη διαδικασία σύνθεσης του εργαλείου.

Τα εργαλεία υψηλού επιπέδου σύνθεσης φέρνουν πιο κοντά τους κλάδους του λογισμικού (software) και του υλικού (hardware) προσφέροντας πλεονεκτήματα και στους δύο. Οι μηχανικοί λογισμικού μπορούν με ευκολία να χρησιμοποιήσουν τα FPGA σαν επιταχυντές υλικού (hardware accelerators) για τη γρηγορότερη εκτέλεση υπολογιστικά απαιτητικών εργασιών, ενώ οι μηχανικοί υλικού μπορούν να αυξήσουν την παραγωγικότητά τους μέσα από τη χρήση μιας γλώσσας υψηλού επιπέδου. Πιο συγκεκριμένα, τα πλεονεκτήματα από τη σύνθεση υψηλού επιπέδου είναι τα εξής:

- Η δημιουργία μίας εφαρμογής με τη χρήση μιας γλώσσας υψηλού επιπέδου απαιτεί πολύ λιγότερο χρόνο.
- Η επαλήθευση της λειτουργίας μιας εφαρμογής μπορεί να πραγματοποιηθεί στο υψηλό επίπεδο και κάθε λογικό λάθος μπορεί να αναγνωριστεί και να αντιμετωπιστεί πολύ πιο εύκολα συγκριτικά με μία γλώσσα περιγραφής υλικού (HDL).
- Αχιτεκτονικές με υψηλή απόδοση μπορούν να δημιουργηθούν χωρίς να απαιτείται ιδιαίτερη προσπάθεια από τον χρήστη, απλά και μόνο με τη χρήση των οδηγιών βελτιστοποίησης (optimization directives).
- Μέσα από τις οδηγίες βελτιστοποίησης μπορούν να συγκριθούν εύκολα διαφορετικές

αρχιτεκτονικές της ίδιας εφαρμογής και να βρεθεί η αποδοτικότερη.

- Μέσα από την σύνθεση υψηλού επιπέδου ένας μηχανικός μπορεί να φτιάξει μεταφέρσιμο κώδικα ο οποίος μπορεί να χρησιμοποιηθεί για τον προγραμματισμό διαφορετικών FPGA μόνο με την αλλαγή κάποιων παραμέτρων στο εργαλείο σύνθεσης υψηλού επιπέδου.

Εξαιτίας της φύσης των FPGA, το Vivado HLS δεν μπορεί να μεταγλωττίσει σε HDL οποιαδήποτε δομή ή διαδικασία υποστηρίζεται από την C. Υπάρχουν δύο κατηγορίες τέτοιων δομών: αυτές που δεν υποστηρίζονται καθόλου και αυτές που υποστηρίζονται μερικώς. Αυτές που δεν υποστηρίζονται είναι:

- *Κλήσεις Συστήματος*: Στα FPGA δεν υπάρχει λειτουργικό σύστημα. Το Vivado HLS αγνοεί αυτόματα τις πιο συχνές κλήσεις συστήματος (`abort()`, `atexit()`, `exit()`, `fprintf()`, `printf()`, `perror()`, `putchar()` και `puts()`) χωρίς να παράγει κάποιο σφάλμα

- *Δυναμικά αντικείμενα*: Οτιδήποτε πρόκειται να μεταγλωττιστεί σε HDL πρέπει να είναι γνωστού μεγέθους τη στιγμή της μεταγλώττισης. Αυτό έχει σαν αποτέλεσμα κλήσεις όπως η `malloc()`, η `alloc()`, η `free()`, η `new` και η `delete` να μην υποστηρίζονται.

Αυτές που υποστηρίζονται μερικώς είναι οι εξής:

- *Δείκτες*: Δεν υποστηριζονται πίνακες από δείκτες και πολλές πράξεις διευθύνσεων (pointer arithmetic) δεν είναι συνθέσιμες.

- *Συναρτήσεις Μνήμης*: Η συναρτήσεις `memcpy()` και `memset()` υποστηρίζονται αλλά μόνο όταν χρησιμοποιούν σταθερές τιμές σαν ορίσματά τους.

Η απόδοση των σημερινών υπερυπολογιστών είναι της τάξης των 10^{15} πράξεων κινητής υποδιαστολής ανά δευτερόλεπτο (PetaFLOPS), οπότε αυτή τη στιγμή οι μηχανικοί του κλάδου προσπαθούν να κατασκευάσουν υπερ-υπολογιστές που θα έχουν απόδοση της τάξης των 10^{18} πράξεων κινητής υποδιαστολής ανα δευτερόλεπτο (ExaFLOPS). Αρκετές σχετικές μελέτες έχουν δείξει ότι για να γίνει αυτό θα πρέπει να υιοθετήσουμε ένα αρχιτεκτονικό μοντέλο που εκμεταλλεύεται τα πλεονεκτήματα της συνύπαρξης υλικού (hardware) και λογισμικού (software). Για αυτόν τον σκοπό έχουν προταθεί οι ετερογενείς αρχιτεκτονικές πολλών επιταχυντών υλικού. Τα εργαλεία υψηλού επιπέδου σύνθεσης διευκολύνουν την δημιουργία συστημάτων με πολλούς επιταχυντές υλικού και για αυτό αναμένεται να διαδραματίσουν καθοριστικό ρόλο στην επίτευξη αυτού του σκοπού. Μέσω της σύνθεσης υψηλού επιπέδου, η σχεδίαση ενός συστήματος γίνεται σε κάποια γλώσσα υψηλού επιπέδου, όπως C/C++, με αποτέλεσμα να απαιτείται πολύ λιγότερος χρόνος από την σχεδίαση ενός συστήματος σε HDL. Ακόμα, μέσα από αυτά τα εργαλεία η απόδοση, η κατανάλωση ισχύος, το εμβαδόν του παραγόμενου κυκλώματος και το κόστος διαφορετικών επιταχυντών μπορεί να υπολογιστεί πολύ εύκολα[2].

Κεφάλαιο 2

Σύστημα Δυναμικής Διαχείρισης Μνήμης για ΥΕΣ

2.1 Εισαγωγή

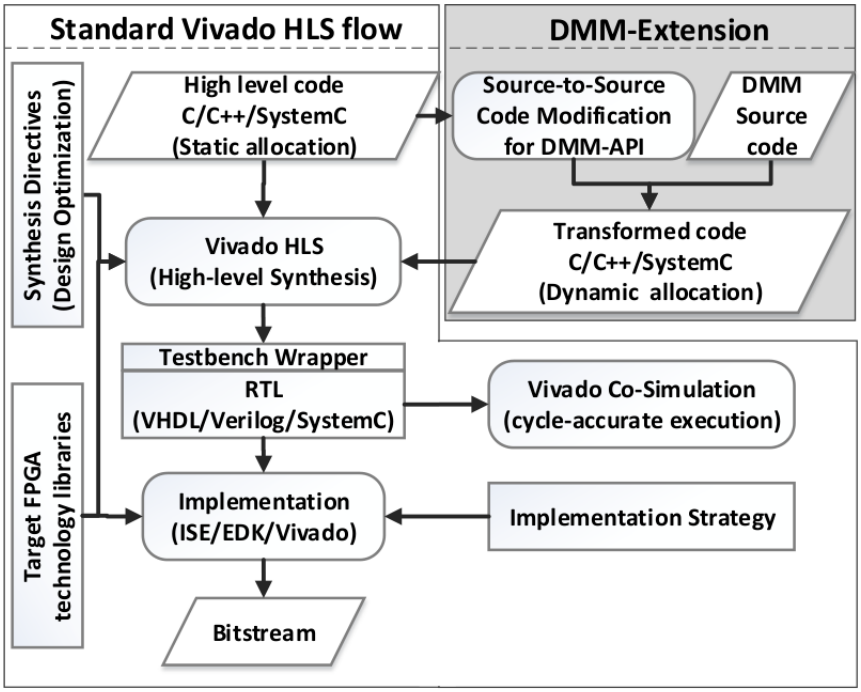
Τα FPGA αποτελούν μία ελκυστική πλατφόρμα ανάπτυξης αρχιτεκτονικών πολλαπλών επιταχυντών υλικού, μέσω της εγγενούς ευελιξίας επαναπρογραμματισμού τους καθώς και της ενεργειακής τους απόδοσης. Ωστόσο, η οργάνωση της μνήμης αποτελεί τον κυριότερο περιοριστικό παράγοντα στις αρχιτεκτονικές με πολλούς επιταχυντές. Ο αριθμός των επιταχυντών που μπορούν να προγραμματιστούν ταυτόχρονα σε ένα FPGA εξαρτάται άμεσα από τους διαθέσιμους πόρους υλικού. Πρόσφατες έρευνες σε συστήματα πολλαπλών επιταχυντών υλικού έχουν δείξει ότι η ενσωματωμένη μνήμη του FPGA δεσμεύεται σε μεγαλύτερο βαθμό έναντι των υπολοίπων πόρων υλικού, οπως τα φλιπ-φλοπ, τα LUTs και τα DSPs. Αυτό συμβαίνει γιατί οι εφαρμογές που υλοποιούνται σε ένα FPGA πρέπει να δεσμεύουν μνήμη στατικά ανάλογα με την χειρότερη περίπτωση σε απαιτήσεις [2].

Μία προτεινόμενη προσέγγιση για την αντιμετώπιση αυτού του περιορισμού είναι η δημιουργία συστήματος δυναμικής διαχείρισης μνήμης [2]. Στην παρούσα διπλωματική εργασία προτάθηκε μία καινοτόμα αρχιτεκτονική δυναμικής διαχείρισης μνήμης σε FPGA. Συγκεκριμένα οι συνεισφορές της παρούσας εργασίας αναφέρονται ως εξής:

- Αξιολογήθηκε και επεκτάθηκε μία βιβλιοθήκη δυναμικής διαχείρησης μνήμης της πρόσφατης βιβλιογραφίας [2]. Η λειτουργία αυτής της βιβλιοθήκης μελετήθηκε ενδελεχώς προκειμένου να προταθούν πιθανές επεκτάσεις.

- Διερευνήθηκαν αρχιτεκτονικές βελτιστοποιήσεις μέσω του εργαλείου Vivado HLS, λ. χ. pipeline, dataflow και array partition, σε αλγόριθμους αυξημένης υπολογιστικής πολυπλοκότητας, π.χ. ιστόγραμμα ψηφιακών εικόνων. Σκοπός αυτού του βήματος ήταν η εξοικείωση με το εργαλείο Vivado HLS και η καλύτερη κατανόηση της μεθοδολογίας υψηλού επιπέδου σύνθεσης.

- Προστέθηκε στο σύστημα ο αλγόριθμος επόμενου ταιριάσματος (Next Fit). Αναπτύχθηκαν δύο εναλλακτικές υλοποιήσεις του αλγορίθμου αυτού.



Σχήμα 2.1: Τροποποιημένη διαδικασία χρήσης του Vivado HLS σύμφωνα με το Σύστημα Δυναμικής Διαχείρισης μνήμης[2]

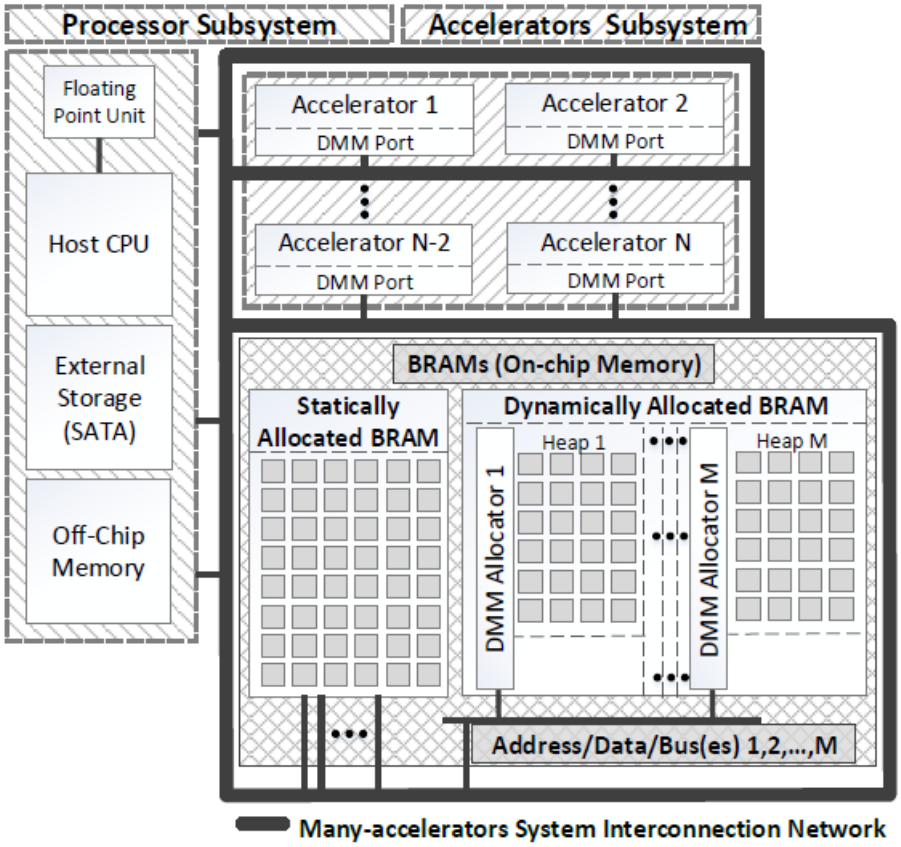
- Προστέθηκε στο σύστημα ο αλγόριθμος καλύτερου ταιριάσματος (Best Fit). Για την υλοποίηση του αλγορίθμου, αναπτύχθηκε μία εναλλακτική υλοποίηση του πυρήνα της υπάρχουσας αρχιτεκτονικής του συστήματος διαχείρισης μνήμης.

- Η προτεινόμενη προσέγγιση αξιολογήθηκε με πολλαπλά σενάρια λειτουργίας προκειμένου να επαληθευτεί η λειτουργικότητα της και να συγκριθεί με την υπάρχουσα αρχιτεκτονική.

2.2 Πρώτη υλοποίηση του συστήματος Δυναμικής διαχείρισης μνήμης

Όπως φαίνεται και στο σχήμα 2.1 η διαδικασία της σύνθεσης υψηλού επιπέδου δεν μεταβάλλεται ιδιαίτερα από τη βιβλιοθήκη δυναμικής διαχείρισης μνήμης. Οι μόνες αλλαγές που πρέπει να κάνει ο χρήστης είναι να συμπεριλάβει στην εφαρμογή τον κώδικα του διαχειριστή μνήμης και να μετασχηματίσει κάθε στατική δέσμευση μνήμης από το δυναμικό της ισοδύναμο. Τα υπόλοιπα στάδια (σύνθεση, προσομοίωση και υλοποίηση), είναι ακριβώς τα ίδια.

Το αρχιτεκτονικό πρότυπο ενός συστήματος πολλών επιταχυντών υλικού που χρησιμοποιεί τον δυναμικό διαχειριστή μνήμης φαίνεται στο σχήμα 2.2. Οι διαθέσιμες BRAM του FPGA χωρίζονται σε ομάδες δημιουργώντας διαφορετικούς σωρούς (heaps). Ο κάθε σωρός διαχειρίζεται από έναν διαχειριστή μνήμης (allocator). Ο κάθε επιταχυντής υλικού μπορεί να ζητήσει μνήμη από οποιονδήποτε σωρό δηλαδή από οποιονδήποτε διαχειριστή. Όταν ζητείται

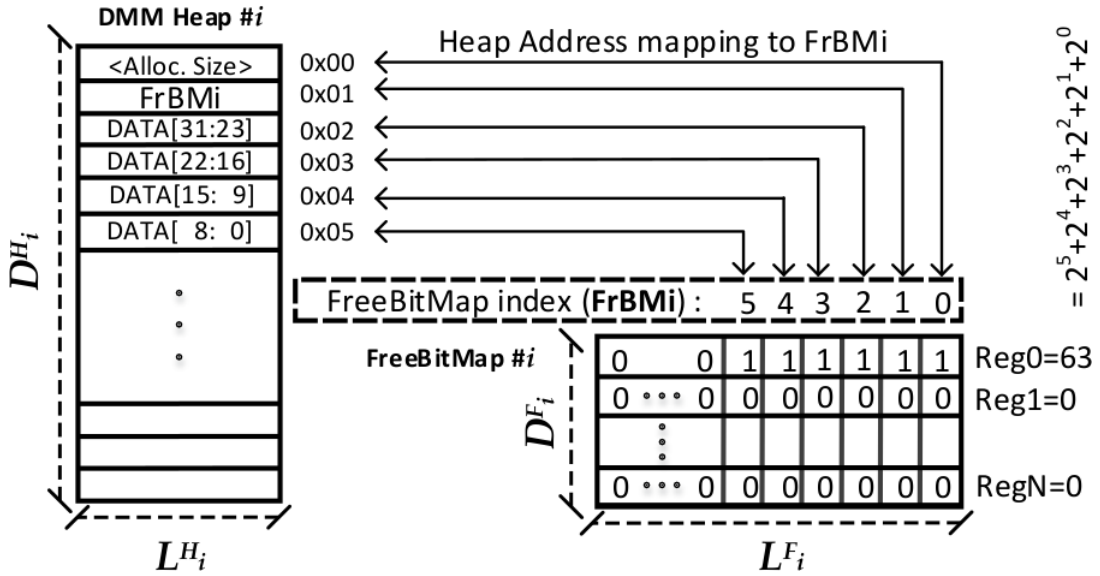


Σχήμα 2.2: Αρχιτεκτονικό πρότυπο για συστήματα πολλών επιταχυντών υλικού μετά από την εφαρμογή του συστήματος δυναμικής διαχείρισης μνήμης[2]

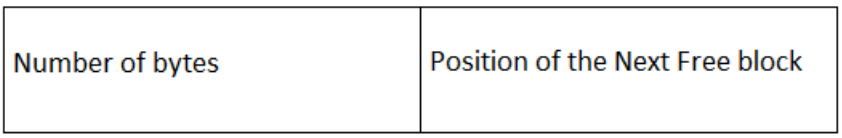
μνήμη από έναν διαχειριστή αυτός βρίσκει, ανάλογα με τον αλγόριθμο δέσμευσης που χρησιμοποιεί, μία ελεύθερη περιοχή της μνήμης του και επιστρέφει έναν δείκτη στην πρώτη λέξη του.

Ο κάθε allocator διαχειρίζεται ένα `C struct`. Στο κάθε `struct` υπάρχει και το αντίστοιχο heap που είναι ένας πίνακας. Στην πρώτη υλοποίηση χρησιμοποιείται ακόμα ένας πίνακας ο οποίος είναι ένας χαρτης bit (bit map) του οποίου το κάθε bit αντιστοιχεί σε ένα byte του heap. Η τιμή του κάθε bit θα είναι 1 αν το αντίσοιχο byte στο heap είναι δεσμευμένο ενώ θα είναι 0 σε αντίθετη περίπτωση. Στο σχήμα 2.3 παρουσιάζεται ο τρόπος λειτουργίας του allocator όταν έχει ζητηθεί η δέσμευση ενός ακεραίου. Ο ακέραιος έχει μέγεθος 4 byte, και ο allocator δεσμεύει δύο λέξεις ακόμα για να αποθηκεύσει μεταδεδομένα σχετικά με αυτή την διαδικασία. Τα μεταδεδομένα που αποθηκεύονται σαν επικεφαλίδα στην αρχή της μνήμης που δεσμεύτηκε, περιέχουν το μέγεθος της δέσμευσης αυτής και τη θέση της στον bit map πίνακα. Αφού βρεθούν λοιπόν 6 bit ίσα με το 0 στον bit map πίνακα η αναζήτηση ελεύθερης μνήμης ολοκληρώνεται, τα αντίστοιχα bit μαρκάρονται (γίνονται ίσα με 1), τα μεταδεδομένα εγγράφονται και ο allocator επιστρέφει έναν δείκτη στην αρχή της περιοχής που δεσμεύτηκε.

```
1. int *A = HlsMalloc ( 1 * sizeof ( int ) , i ) ;
2. A [ 0 ] = DATA ;
```



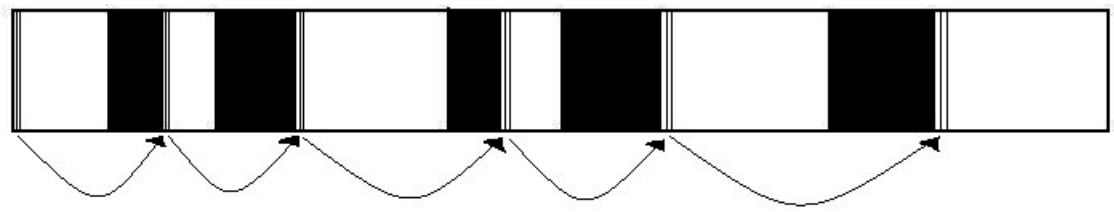
Σχήμα 2.3: Υλοποίηση 1: bit map πίνακας για τον έλεγχο των ελεύθερων και δεσμευμένων byte του σωρού [2]



Σχήμα 2.4: Δομή επικεφαλίδας των ελεύθερων μπλοκ μνήμης για την δεύτερη υλοποίηση

Σχήμα 2.5: Στιγμιότυπο μνήμης πριν την δημιουργία της λίστας των ελεύθερων μπλοκ (λευκά:ελεύθερα, μαύρα:δεσμευμένα)

Αυτή την υλοποίηση χρησιμοποιούν οι αλγόριθμοι πρώτου ταιριάσματος (First Fit) και επόμενου ταιριασματος (Next Fit). Ο αλγόριθμος First Fit ξεκινάει κάθε φορά να αναζητάει ελεύθερη μνήμη από την αρχή του bit map πίνακα ενώ ο αλγόριθμος Next Fit ξεκινάει από το σημείο που σταμάτησε η τελευταία του εκτέλεση.



Σχήμα 2.6: Στιγμιότυπο μνήμης μετά την δημιουργία της λίστας των ελεύθερων μπλοκ (λευκά:ελεύθερα, μαύρα:δεσμευμένα)

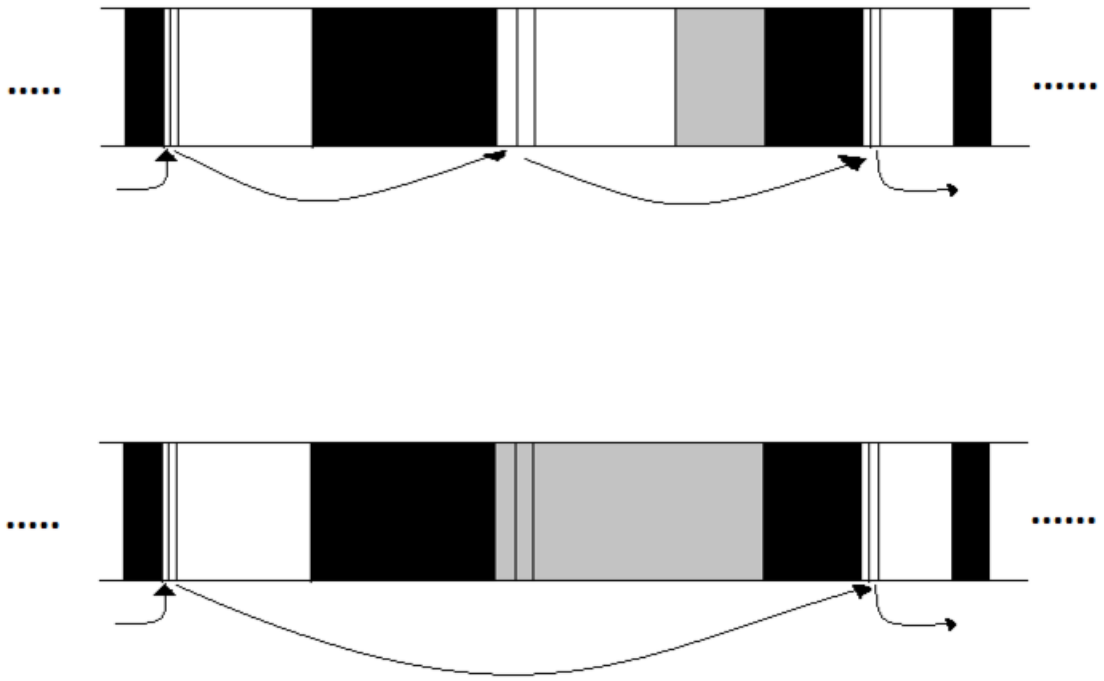
2.3 Δεύτερη υλοποίηση του συστήματος δυναμικής διαχείρισης μνήμης

Στη δεύτερη υλοποίηση καταργείται η έννοια του bit map πίνακα και χρησιμοποιείται το ίδιο το heap για να δημιουργήθει η λίστα με τα ελεύθερα μπλοκ μνήμης. Για αυτόν το λόγο, χρησιμοποιείται μία επικεφαλίδα από δύο λέξεις μνήμης σε κάθε ελεύθερο μπλοκ. Στην επικεφαλίδα (σχήμα 2.4), αποθηκεύεται το μέγεθος του εκάστοτε ελεύθερου μπλοκ αλλά και η θέση του επόμενου ελεύθερου μπλοκ. Στο σχήμα 2.5 φαίνεται ένα στιγμιότυπο της μνήμης χωρίς την εφαρμογή αυτής της επικεφαλίδας. Με μαύρο χρώμα φαίνονται τα δεσμευμένα byte ενώ με λευκό είναι τα ελεύθερα μπλοκ. Η χρήση αυτής της επικεφαλίδας μετασχηματίζει το στιγμιότυπο αυτό όπως φαίνεται στο σχήμα 2.6.

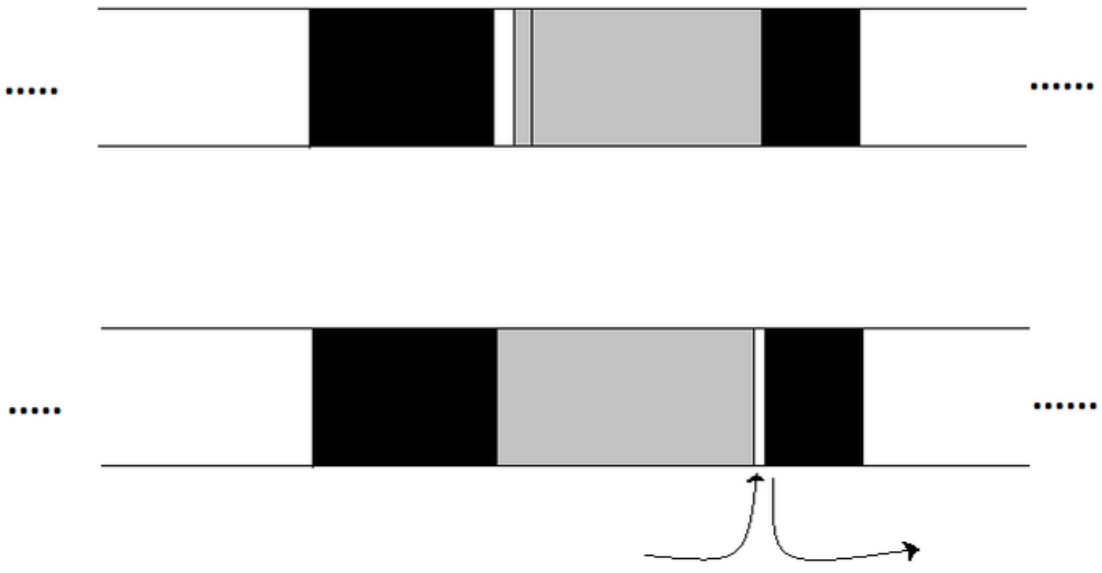
Στη δεύτερη υλοποίηση καταργείται ο bit map πίνακας και κάθε διαδικασία που σχετίζεται με αυτόν, αλλά δημιουργούνται νέες τεχνικές δέσμευσης και αποδέσμευσης της μνήμης. Οι τεχνικές αυτές είναι άρρηκτα συνδεδεμένες με την επικεφαλίδα που χρησιμοποιείται στα ελεύθερα μπλοκ. Τα σημαντικά σημεία αυτών των τεχνικών είναι τα εξής:

- *Δέσμευση μνήμης*: Στο σχήμα 2.7 φαίνονται οι δύο περιπτώσεις που μπορεί να προκύψουν κατά την δέσμευση μνήμης. Στην πρώτη περίπτωση η δέσμευση γίνεται (γκρι μέρος) από ένα μπλοκ το οποίο έχει μεγαλύτερο μέγεθος από αυτό που έχει ζητηθεί. Σε αυτή την περίπτωση, το μόνο που χρειάζεται από τον allocator είναι να ενημερώσει την επικεφαλίδα με το νέο μέγεθος του μπλοκ. Στη δεύτερη περίπτωση η δέσμευση που πρόκειται να πραγματοποιηθεί (γκρι μέρος) καταλαμβάνει ολόκληρο το μπλοκ οπότε η επικεφαλίδα του προηγούμενου ελεύθερου μπλοκ πρέπει να ενημερωθεί κατάλληλα και να δείχνει στο επόμενο ελεύθερο μπλοκ. Σε αυτό το σημείο αξίζει να σημειωθεί ότι λόγω της επικεφαλίδας που υπάρχει στην αρχή κάθε ελεύθερου μπλοκ, οι δεσμεύσεις γίνονται από το τέλος προς την αρχή του κάθε μπλοκ.

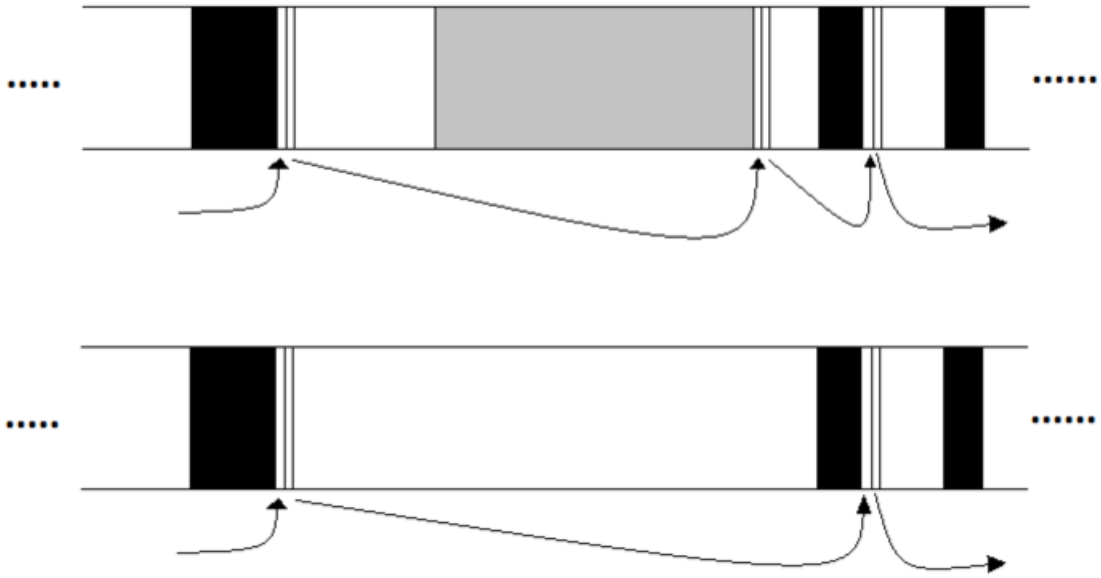
- *Ελάχιστο μέγεθος ελεύθερου μπλοκ*: Επειδή σε κάθε ελεύθερο μπλοκ χρησιμοποιείται επικεφαλίδα δύο λέξεων το ελάχιστο μέγεθος ενός ελεύθερου μπλοκ είναι δύο λέξεις. Όταν λοιπόν μία δέσμευση μνήμης είναι κατά μία λέξη μικρότερη από το μπλοκ (πρώτο μέρος του σχήματος 2.8) θα πρέπει να δεσμευτεί ολόκληρο το μπλοκ. Όμως αυτή η πληροφορία είναι γνωστή μόνο στον allocator και πρέπει να διατηρηθεί με τη βοήθεια



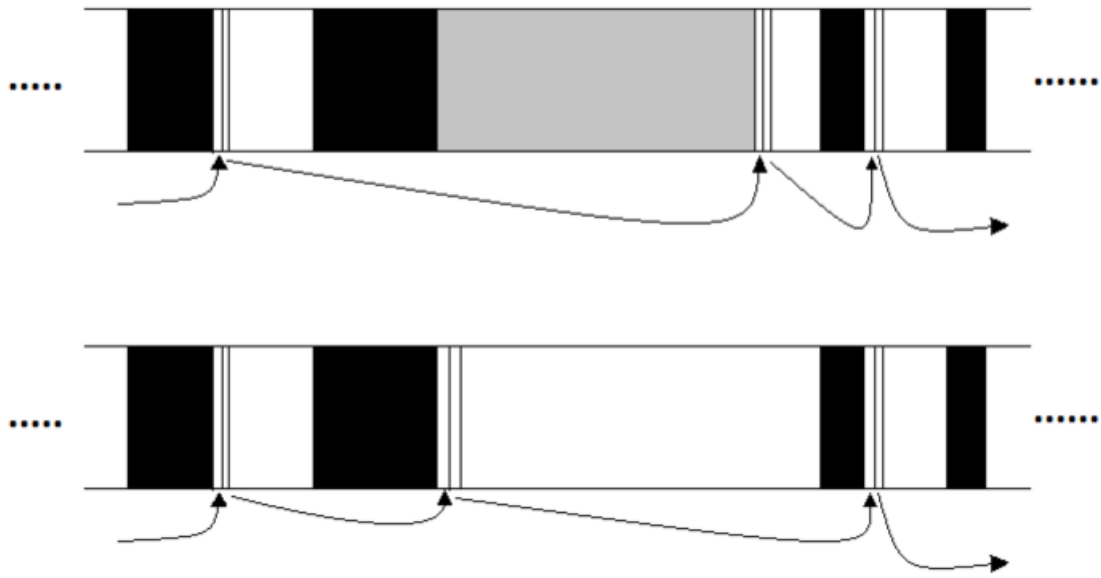
Σχήμα 2.7: Διαδικασία δέσμευσης μνήμης



Σχήμα 2.8: Όταν η δέσμευση που πρόκειται να γίνει είναι κατά μία λέξη μικρότερη από το μπλοκ θα πρέπει να δεσμευθεί και η λέξη που περισεύει. Αν δεν χρησιμοποιείται επικεφαλίδα δέσμευσης τότε δημιουργείται μία λίστα με αυτές τις λέξεις

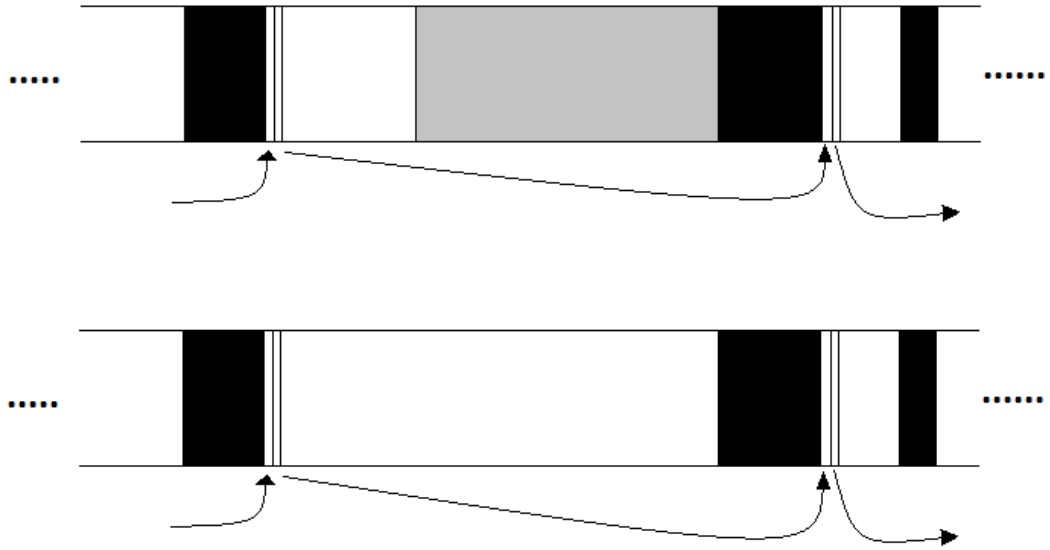


Σχήμα 2.9: 1η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συγχωνεύεται και με το προηγούμενο αλλά και με το επόμενο ελεύθερο μπλοκ

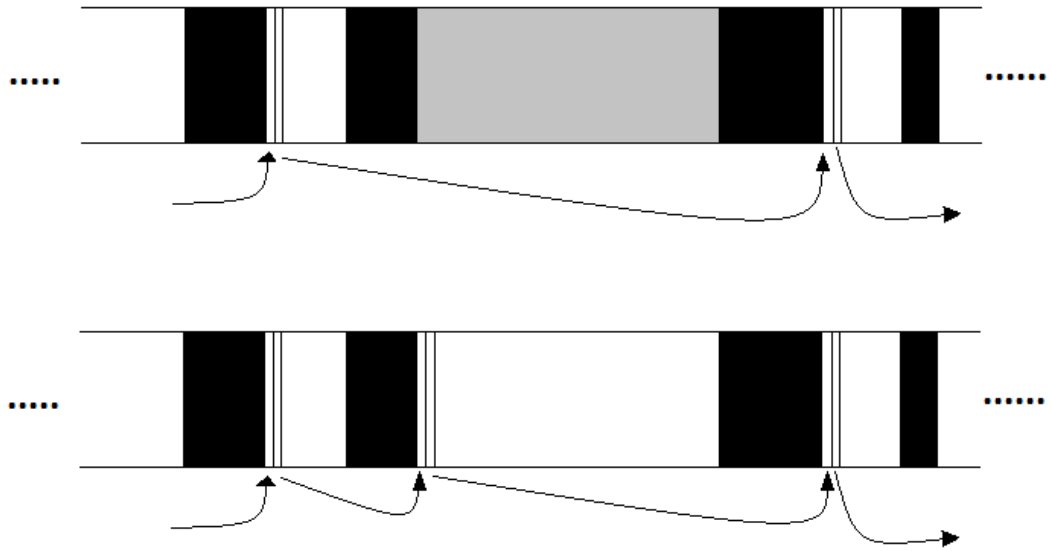


Σχήμα 2.10: 2η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συχγωνεύεται μόνο με το επόμενο ελεύθερο μπλοκ

ενός εσωτερικού μηχανισμού. Αν χρησιμοποιείται επικεφαλίδα για τα δεσμευμένα μπλοκ τότε απλά ο σωστός αριθμός των byte αποθηκεύεται εκεί. Σε αντίθετη περίπτωση (δεύ-



Σχήμα 2.11: 3η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συχγωνεύεται μόνο με το προηγούμενο ελεύθερο μπλοκ



Σχήμα 2.12: 2η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) δεν συχγωνεύεται με κάποιο γειτονικό ελεύθερο μπλοκ

τερο μέρος του σχήματος 2.8) δημιουργείται μία λίστα από αυτές τις «αχρησιμοποίητες» λέξεις. Για τη δημιουργία της λίστας αυτής, σε κάθε τέτοια λέξη αποθηκεύεται η θέση της επόμενης «αχρησιμοποίητης» λέξης στη λίστα αυτή.

- *Αποδέσμευση μνήμης*: Κατά την αποδέσμευση μνήμης θα πρέπει να ελεγχθεί αν το μπλοκ που ελευθερώνεται πρέπει να συγχωνευτεί με κάποιο γειτονικό του. Έτσι προ-

κύπτουν 4 περιπτώσεις. Στην πρώτη (σχήμα 2.9) το μπλοκ που πρόκειται να ελευθερωθεί πρέπει να συγχωνευτεί με το προηγούμενο και το επόμενο ελεύθερο μπλοκ. Όπως φαίνεται και στο σχήμα 2.9 τα τρία μπλοκ θα γίνουν ένα. Στην επικεφαλίδα του νέου μπλοκ θα αποθηκευτεί το αθροισμα των μεγεθών των τριών μπλοκ που συγχωνεύτηκαν και η θέση του επόμενου ελεύθερου μπλοκ στη λίστα. Οι υπόλοιπες περιπτώσεις είναι οι εξής: το μπλοκ που ελευθερώνεται πρέπει να συγχωνευτεί με το επόμενο μπλοκ της λίστας (σχήμα 2.10), το μπλοκ που ελευθερώνεται πρέπει να συγχωνευτεί με το προηγούμενο της λίστας (σχήμα 2.11) και το μπλοκ που ελευθερώνεται απλά προστίθεται στη λίστα χωρίς να συγχωνευτεί με κάποιο άλλο (σχήμα 2.12).

Κεφάλαιο 3

Πειράματα και Αποτελέσματα

Σε αυτό το κεφάλαιο περιγράφεται η διαδικασία μέσα από την οποία αξιολογήθηκε το σύστημα δυναμικής διαχείρισης μνήμης. Αρχικά παρουσιάζεται το test bench που χρησιμοποιήθηκε από το Vivado® HLS. Στη συνέχεια αναλύεται κάθε πείραμα που εκτελέστηκε. Αξίζει να σημειωθεί εδώ ότι το πιο ακριβές και το πιο κοντινό σε κάποιο αληθινό σενάριο είναι το πείραμα Larson.

3.1 Πειραματικό περιβάλλον

Το εργαλείο Vivado® HLS χρησιμοποιεί ένα test bench για την αξιολόγηση ενός σχεδίου. Γιαυτό δημιουργήθηκε ένα test bench το οποίο χρησιμοποιήθηκε για την σύγκριση των τριών διαφορετικών υλοποιήσεων (listing 3.1). Το μόνο μέρος της συνάρτησης αυτής που είναι συνθέσιμο είναι η συνάρτηση `yadmm()!` (γραμμή 17) και θα αναλυθέι παρακάτω. Τα υπόλοιπα βήματα της συνάρτησης αυτή είναι: η αρχικοποίηση κάθε δομής δεδομένων που χρησιμοποιείται από το σύστημα (γραμμή 5), η εκτύπωση χρήσιμων πληροφοριών που σχετίζονται με τους σωρούς μνήμης (memory heaps) που χρησιμοποιούνται, όπως το συνολίκό μέγεθος, η πρώτη και η τελευταία διεύθυνση, το μέγεθος της freelist κ.α. (γραμμή 7), η εκτύπωση των περιεχομένων της freelist (γραμμή 24) αν χρησιμοποιείται και τέλος η εκτύπωση των περιεχομένων ολόκληρης της μνήμης (γραμμή 25)

```
1  int main(void){
2    uint_t i=4, returned, val;
3    MemLuvStats mlvstats;
4    MemLuvCore *mlvcore;
5    MemluvInit();
6    mlvcore = ReturnMemLuvCore();
7    MemluvInfo(NULL, stdout, ALL);
8
9    for (i=205;i>204;i--) {
10     printf("try i=%u\n", i);
11
12 #if RANDOM_SEED==1
13     val=RandMinMax(1,i);
```

```
#else
    val=i;
#endif
    returned = yadmm(val
               #if HW_DEBUG_MEMLUV==1
                  , &mlvstats
               #endif
               );
  }
#if SIM_WITH_GLIBC_MALLOC==0
  MemluvDumpFreeList(mlvcore, ALL);
  MemluvDumpCore(mlvcore, ALL);
#endif
  MemluvEnd();
  return 0;
}
```

Listing 3.1: Test Bench[2]

3.2 Πειραματική ανάλυση

3.2.1 Πρώτη Ομάδα Πειραμάτων

```
#if TEST==1
  char *data1,*data2,*data3,*data4;
  uint_t i;
  data1 = (char *)MemluvAlloc(2048,0);
  data2 = (char *)MemluvAlloc(2048,0);
  MemluvFree(data1,2048,0);
  data3 = (char *)MemluvAlloc(4096,0);
  data4 = (char *)MemluvAlloc(2040,0);
  MemluvFree(data2,2048,0);
  MemluvFree(data3,4096,0);
  MemluvFree(data4,2040,0);
#endif
```

Listing 3.2: Test 1

```
#if TEST==2
  char *data1,*data2,*data3,*data4;
  char *data5,*data6,*data7,*data8;
  char *data9,*data10,*data11,*data12;

  data1 = (char *)MemluvAlloc(300, 0);
  data2 = (char *)MemluvAlloc(150, 0);
  data3 = (char *)MemluvAlloc(29, 0);
  data4 = (char *)MemluvAlloc(55, 0);
  data5 = (char *)MemluvAlloc(653, 0);
  data6 = (char *)MemluvAlloc(323, 0);
```

```
12    data7 = (char *)MemluvAlloc(63, 0);
13    data8 = (char *)MemluvAlloc(89, 0);
14    data9 = (char *)MemluvAlloc(76, 0);
15
16    MemluvFree((void *)data5, 653, 0);
17    MemluvFree((void *)data3, 29, 0);
18    MemluvFree((void *)data1, 300, 0);
19    data5 = (char *)MemluvAlloc(50, 0);
20  #endif
```

Listing 3.3: Test 2

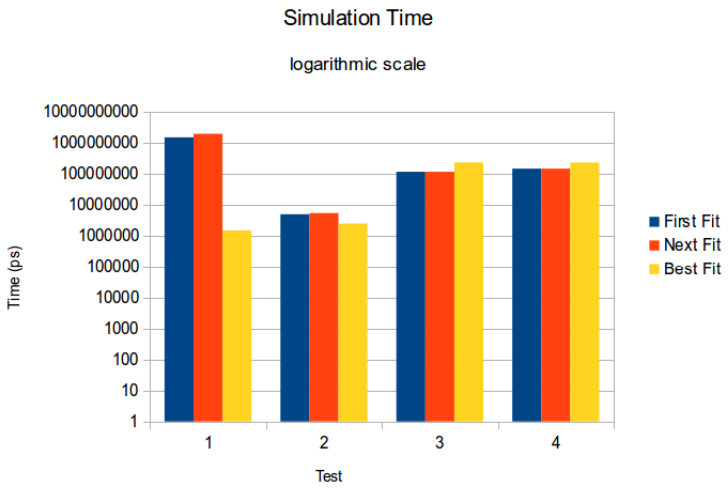
```
1   #if TEST==3
2     int i;
3
4     char *p1, *p2;
5
6     for(i=0; i<256; i++){
7       p1 = (char *)MemluvAlloc(32, 0);
8       p2 = (char *)MemluvAlloc(32, 0);
9
10      MemluvFree((void *)p1, 32, 0);
11      MemluvFree((void *)p2, 32, 0);
12    }
13  #endif
```

Listing 3.4: Test 3

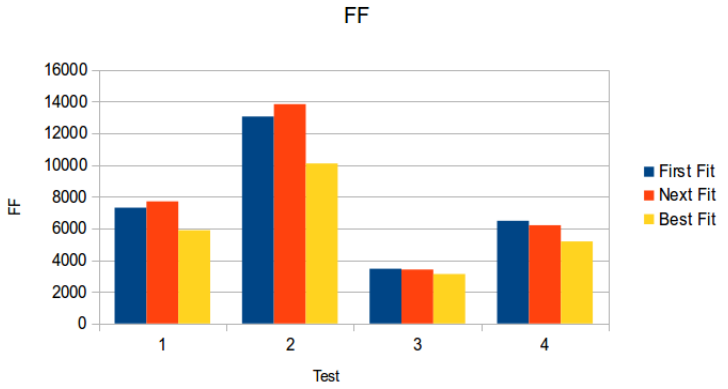
```
1   #if TEST==4
2     int i;
3
4     char *p1, *p2;
5     char *p3, *p4;
6
7     for(i=0; i<128; i++){
8       p1 = (char *)MemluvAlloc(32, 0);
9       p2 = (char *)MemluvAlloc(32, 0);
10      p3 = (char *)MemluvAlloc(32, 0);
11      p4 = (char *)MemluvAlloc(32, 0);
12
13      MemluvFree((void *)p1, 32, 0);
14      MemluvFree((void *)p2, 32, 0);
15      MemluvFree((void *)p3, 32, 0);
16      MemluvFree((void *)p4, 32, 0);
17    }
18  #endif
```

Listing 3.5: Test 4

Τα σενάρια που δημιουργήθηκαν για την πειραματική αξιολόγηση της βιβλιοθήκης χωρίζονται σε 4 κατηγορίες σύμφωνα με κάποια κοινά τους χαρακτηριστικά.



Σχήμα 3.1: 1η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα

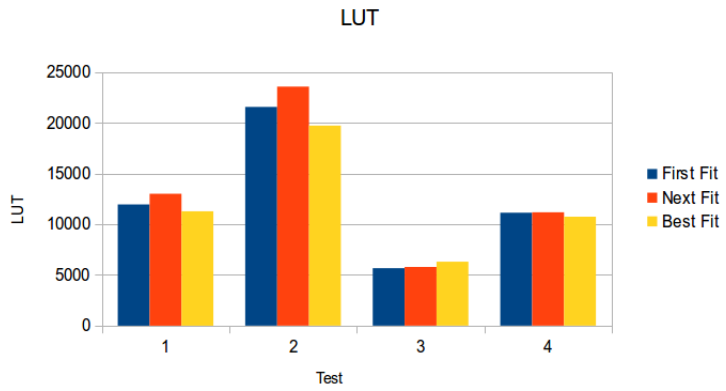


Σχήμα 3.2: 1η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ

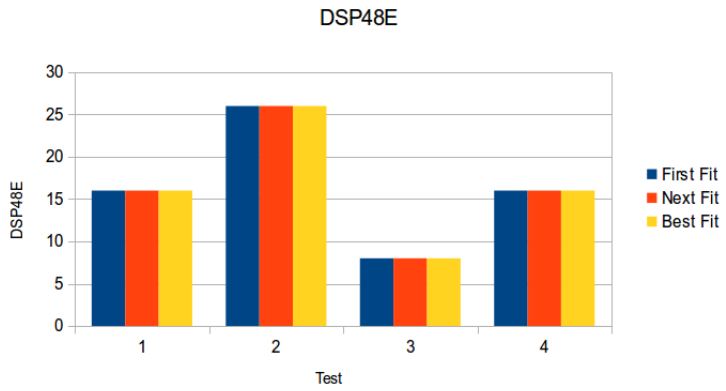
Στην πρώτη κατηγορία πειραμάτων ανήκουν τα πιο απλά σενάρια που χρησιμοποιήθηκαν. Τα 4 πειράματα της κατηγορίας αυτής είναι τα εξής:

- *Πείραμα 1*: Σενάριο με 4 δεσμεύσεις και 4 αποδεσμεύσεις (listing 3.2).
- *Πείραμα 2*: Σενάριο με 10 δεσμεύσεις και 4 αποδεσμεύσεις (listing 3.3).
- *Πείραμα 3*: Σενάριο με 256 επαναλήψεις από 2 δεσμεύσεις σταθερού μεγέθους και 2 αποδεσμεύσεις(listing 3.4).
- *Πείραμα 4*: Σενάριο με 128 επαναλήψεις από 4 δεσμεύσεις σταθερού μεγέθους και 4 αποδεσμεύσεις(listing 3.5).

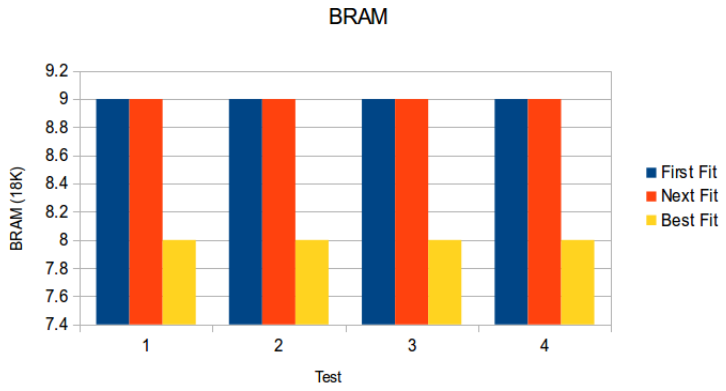
Από το σχήμα 3.1 βλέπουμε ότι στο πρώτο πείραμα ο Best Fit παρουσιάζει σημαντικά βελτιωμένη απόδοση σε σχέση με τους άλλους δύο αλγορίθμους (1000 φορές πιο γρήγορος από τον First Fit και 1300 φορές πιο γρήγορος από τον Next Fit). Οι δεσμεύσεις μνήμης που γίνονται σε αυτό το πείραμα είναι λίγες στο πλήθος, αλλά σχετικά μεγάλες, οπότε αυτή η μεγάλη διαφορά στην απόδοση των δύο υλοποιήσεων οφείλεται στις χρονοβόρες διαδικασίες



Σχήμα 3.3: 1η ομάδα πειραμάτων: δέσμευση LUT



Σχήμα 3.4: 1η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 3.5: 1η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

ενημέρωσης των bit του bit map πίνακα. Στο δεύτερο πείραμα η διαφορά της απόδοσης των δύο υλοποιήσεων είναι μικρή γιατί τα μεγέθη των δεσμεύσεων είναι πιο μικρά και η ενημέρωση των bit του bit map πίνακα πραγματοποιείται σε πολύ λιγότερους κύκλους. Τα άλλα δύο πειράματα χρησιμοποιούν μικρές δεσμεύσεις μνήμης μικρού και σταθερού μεγέθους και όπως είναι φυσικό οι αλγόριθμοι First Fit και Next Fit υπερισχύουν έναντι του Best Fit. Ο απόλυτος αριθμός των φλιπ-φλοπ (σχήμα 3.2) και των LUT (σχήμα 3.3) που χρησιμοποιούνται

εξαρτάται σε μεγάλο βαθμό από το πείραμα που εκτελείται. Έν γένει η υλοποίηση του Best Fit δεσμεύει κατά 18% λιγότερα φλιπ-φλοπ, 5% λιγότερα LUT από τον First Fit, και 13% λιγότερα LUT από τον Next Fit. Οι μπλοκ RAM (σχήμα 3.5) που χρησιμοποιούνται από τους First Fit και Next Fit είναι κατά μία περισσότερες γιατί χρειάζονται τον bit map πίνακα ο οποίος έχει μέγεθος ίσο με το $\frac{1}{8}$ του heap. Τα DSP48E που απαιτούνται είναι τα ίδια και για τις τρεις υλοποιήσεις (σχήμα 3.4).

3.2.2 Δεύτερη ομάδα πειραμάτων

```
1  //50% chance of freeing the pointer
2  #if TEST==5
3    int i;
4
5    char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
6    char *p8,*p9,*p10,*p11,*p12,*p13,*p14,*p15;
7
8    unsigned short lfsr_ptr = 0xACE1u;
9
10   for(i=0; i<10; i++){
11     p1 = (char *)MemluvAlloc(32, 0);
12     p2 = (char *)MemluvAlloc(32, 0);
13     p3 = (char *)MemluvAlloc(32, 0);
14     p4 = (char *)MemluvAlloc(32, 0);
15     p5 = (char *)MemluvAlloc(32, 0);
16     p6 = (char *)MemluvAlloc(32, 0);
17     p7 = (char *)MemluvAlloc(32, 0);
18     p8 = (char *)MemluvAlloc(32, 0);
19     p9 = (char *)MemluvAlloc(32, 0);
20     p10 = (char *)MemluvAlloc(32, 0);
21     p11 = (char *)MemluvAlloc(32, 0);
22     p12 = (char *)MemluvAlloc(32, 0);
23     p13 = (char *)MemluvAlloc(32, 0);
24     p14 = (char *)MemluvAlloc(32, 0);
25     p15 = (char *)MemluvAlloc(32, 0);
26
27     if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
28       MemluvFree((void *)p1, 32, 0);
29     }
30     if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
31       MemluvFree((void *)p2, 32, 0);
32     }
33     if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
34       MemluvFree((void *)p3, 32, 0);
35     }
36     if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
37       MemluvFree((void *)p4, 32, 0);
38     }
39     if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
40       MemluvFree((void *)p5, 32, 0);
```

```
41      }
42      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
43        MemluvFree((void *)p6, 32, 0);
44      }
45      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
46        MemluvFree((void *)p7, 32, 0);
47      }
48      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
49        MemluvFree((void *)p8, 32, 0);
50      }
51      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
52        MemluvFree((void *)p9, 32, 0);
53      }
54      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
55        MemluvFree((void *)p10, 32, 0);
56      }
57      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
58        MemluvFree((void *)p11, 32, 0);
59      }
60      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
61        MemluvFree((void *)p12, 32, 0);
62      }
63      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
64        MemluvFree((void *)p13, 32, 0);
65      }
66      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
67        MemluvFree((void *)p14, 32, 0);
68      }
69      if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
70        MemluvFree((void *)p15, 32, 0);
71      }
72
73    }
74  #endif
```

Listing 3.6: Test 5

```
1  //10% chance of freeing each pointer
2  #if TEST==6
3    int i;
4
5    char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
6    char *p8,*p9,*p10,*p11,*p12,*p13,*p14,*p15;
7
8    unsigned short lfsr_ptr = 0xACE1u;
9
10   for(i=0; i<30; i++){
11     p1 = (char *)MemluvAlloc(32, 0);
12     p2 = (char *)MemluvAlloc(32, 0);
13     p3 = (char *)MemluvAlloc(32, 0);
14     p4 = (char *)MemluvAlloc(32, 0);
```

```
15      p5 = (char *)MemluvAlloc(32, 0);
16      p6 = (char *)MemluvAlloc(32, 0);
17      p7 = (char *)MemluvAlloc(32, 0);
18      p8 = (char *)MemluvAlloc(32, 0);
19
20      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
21        MemluvFree((void *)p1, 32, 0);
22      }
23      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
24        MemluvFree((void *)p2, 32, 0);
25      }
26      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
27        MemluvFree((void *)p3, 32, 0);
28      }
29      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
30        MemluvFree((void *)p4, 32, 0);
31      }
32      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
33        MemluvFree((void *)p5, 32, 0);
34      }
35      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
36        MemluvFree((void *)p6, 32, 0);
37      }
38      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
39        MemluvFree((void *)p7, 32, 0);
40      }
41      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
42        MemluvFree((void *)p8, 32, 0);
43      }
44    }
45  #endif
```

Listing 3.7: Test 6

```
1   //30% chance of freeing each pointer
2   #if TEST==7
3     int i;
4
5     char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
6     char *p8,*p9,*p10,*p11,*p12,*p13,*p14,*p15;
7
8     unsigned short lfsr_ptr = 0xACE1u;
9
10    for(i=0; i<50; i++){
11      p1 = (char *)MemluvAlloc(32, 0);
12      p2 = (char *)MemluvAlloc(32, 0);
13      p3 = (char *)MemluvAlloc(32, 0);
14      p4 = (char *)MemluvAlloc(32, 0);
15      p5 = (char *)MemluvAlloc(32, 0);
16      p6 = (char *)MemluvAlloc(32, 0);
17      p7 = (char *)MemluvAlloc(32, 0);
```

```
18      p8 = (char *)MemluvAlloc(32, 0);
19
20      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
21        MemluvFree((void *)p1, 32, 0);
22      }
23      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
24        MemluvFree((void *)p2, 32, 0);
25      }
26      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
27        MemluvFree((void *)p3, 32, 0);
28      }
29      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
30        MemluvFree((void *)p4, 32, 0);
31      }
32      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
33        MemluvFree((void *)p5, 32, 0);
34      }
35      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
36        MemluvFree((void *)p6, 32, 0);
37      }
38      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
39        MemluvFree((void *)p7, 32, 0);
40      }
41      if(RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3){
42        MemluvFree((void *)p8, 32, 0);
43      }
44    }
45  #endif
```

Listing 3.8: Test 7

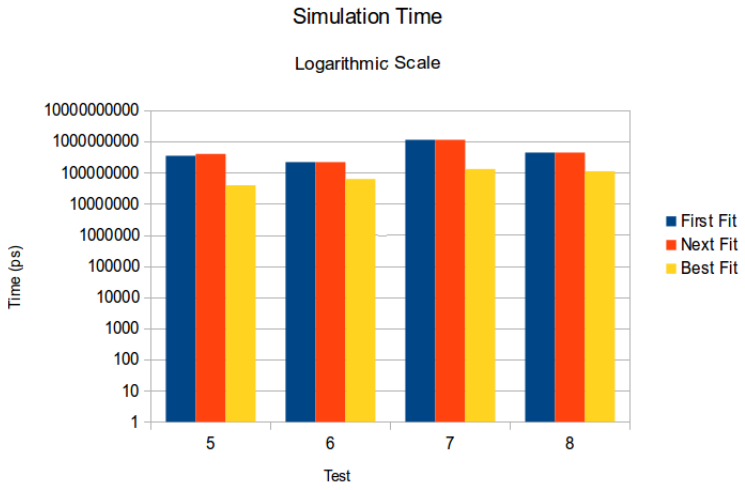
```
1  //10% chance of freeing the blocks
2  #if TEST==8
3    int i;
4
5    char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
6    char *p8,*p9,*p10,*p11,*p12,*p13,*p14,*p15;
7
8    unsigned short lfsr_ptr = 0xACE1u;
9
10   for(i=0; i<50; i++){
11     p1 = (char *)MemluvAlloc(32, 0);
12     p1[31] = 'a';
13     p2 = (char *)MemluvAlloc(32, 0);
14     p2[31] = p1[31]+1;
15     p3 = (char *)MemluvAlloc(32, 0);
16     p3[31] = p2[31]+2;
17     p4 = (char *)MemluvAlloc(32, 0);
18     p4[31] = p3[31]+3;
19     p5 = (char *)MemluvAlloc(32, 0);
20     p5[31] = p4[31]+4;
```

```
21      p6 = (char *)MemluvAlloc(32, 0);
22      p6[31] = p5[31]+5;
23      p7 = (char *)MemluvAlloc(32, 0);
24      p7[31] = p6[31]+6;
25      p8 = (char *)MemluvAlloc(32, 0);
26      p8[31] = p7[31]+7;
27
28      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
29        MemluvFree((void *)p1, 32, 0);
30      }
31      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
32        MemluvFree((void *)p2, 32, 0);
33      }
34      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
35        MemluvFree((void *)p3, 32, 0);
36      }
37      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
38        MemluvFree((void *)p4, 32, 0);
39      }
40      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
41        MemluvFree((void *)p5, 32, 0);
42      }
43      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
44        MemluvFree((void *)p6, 32, 0);
45      }
46      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
47        MemluvFree((void *)p7, 32, 0);
48      }
49      if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
50        MemluvFree((void *)p8, 32, 0);
51      }
52    }
53  #endif
```

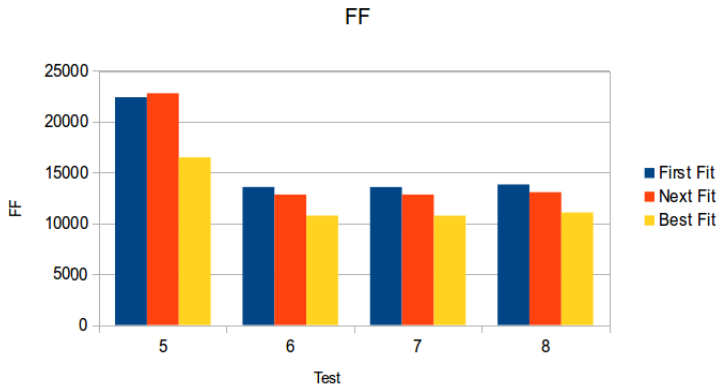
Listing 3.9: Test 9

Η δεύτερη κατηγορία πειραμάτων περιλαμβάνει πειράματα τα οποία εκτελούν πλήθος δεσμεύσεων μνήμης σταθερού μεγέθους. Οι αποδεσμεύσεις σε αυτήν την κατηγορία γίνονται με κάποια πιθανότητα αποτυχίας. Τα πειράματα αυτής της κατηγορίας είναι τα εξής:

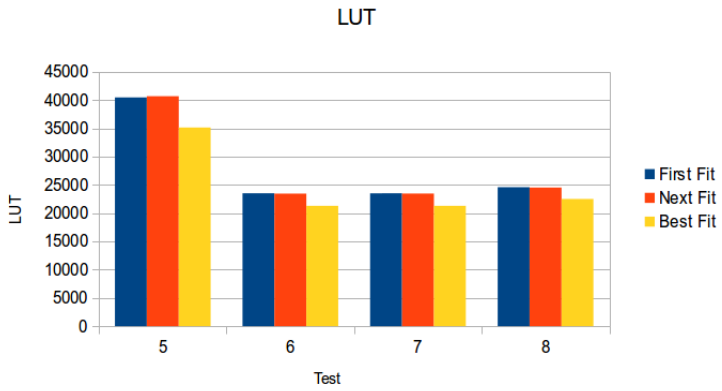
- *Πείραμα 5*: Σενάριο με 10 επαναλήψεις από 16 δεσμεύσεις σταθερού μεγέθους και 16 αποδεσμεύσεις με πιθανότητα 50% (listing 3.6).
- *Πείραμα 6*: Σενάριο με 30 επαναλήψεις από 8 δεσμεύσεις σταθερού μεγέθους και 8 αποδεσμεύσεις με πιθανότητα 10%(listing 3.7).
- *Πείραμα 7*: Σενάριο με 50 επαναλήψεις από 8 δεσμεύσεις σταθερού μεγέθους και 8 αποδεσμεύσεις με 30% πιθανότητα(listing 3.8).
- *Πείραμα 8*: Σενάριο με 50 επαναλήψεις από 8 δεσμεύσεις σταθερού μεγέθους και 8 αποδεσμεύσεις με 10% πιθανότητα(listing 3.9).



Σχήμα 3.6: 2η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα

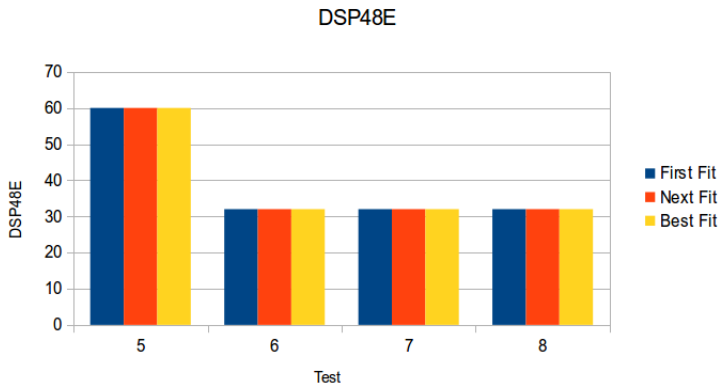


Σχήμα 3.7: 2η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ

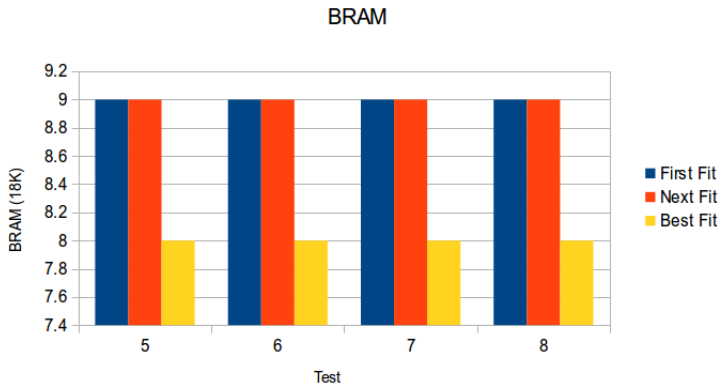


Σχήμα 3.8: 2η ομάδα πειραμάτων: δέσμευση LUT

Σε αυτή την κατηγορία πειραμάτων ο Best Fit παρουσιάζεται πέντε φορές πιο γρήγορος από τους άλλους δύο αλγορίθμους (σχήμα 3.6). Αυτό συμβαίνει γιατί ο τρόπος που αναζητείται στον Best Fit η κατάλληλη ελεύθερη περιοχή μνήμης είναι πολύ πιο αποδοτικός και



Σχήμα 3.9: 2η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 3.10: 2η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

αυξάνεται η ταχύτητά του όσο τα δεδομένα στη μνήμη αυξάνονται. Ο χρόνος εκτέλεσης των αλγορίθμων First Fit και Next Fit είναι ο ίδιος για τα πειράματα 6, 7, και 8 γιατί τα ελεύθερα κομμάτια μνήμης που δημιουργούνται με την πάροδο του χρόνου έχουν το ίδιο μέγεθος. Έτσι η αναζήτηση ελεύθερου μπλοκ με το απαιτούμενο μέγεθος ολοκληρώνεται και στις δύο περιπτώσεις αρκετά γρήγορα. Ο μόνος παράγοντας καθυστέρησης των αλγορίθμων αυτών σε αυτήν την περίπτωση είναι η διαδικασίες ενημέρωσης του πίνακα bit map που είναι κοινές. Ο απόλυτος αριθμός των φλιπ-φλοπ (σχήμα 3.7) και των LUT (σχήμα 3.8) που χρησιμοποιούνται εξαρτάται σε μεγάλο βαθμό από το πείραμα που εκτελείται. Για αυτόν τον λόγο στο πείραμα 5, που το σώμα της for είναι μεγαλύτερο σε σχέση με τα υπόλοιπα, παρατηρείται μία σημαντική αύξηση των πόρων που δεσμεύονται. Εν γένει όμως, η υλοποίηση του Best Fit δεσμεύει κατά 21% λιγότερα φλιπ-φλοπ από τον First Fit, κατά 18% λιγότερα φλιπ-φλοπ από τον Next Fit και 10% λιγότερα LUT από τον First Fit, και τον Next Fit. Οι μπλοκ RAM (σχήμα 3.10) που χρησιμοποιούνται από τους First Fit και Next Fit είναι κατά μία περισσότερες γιατί χρειάζονται τον bit map πίνακα ο οποίος έχει μέγεθος ίσο με το $\frac{1}{8}$ του heap. Τα DSP48E που απαιτούνται είναι τα ίδια και για τις τρεις υλοποιήσεις (σχήμα 3.9).

3.2.3 Τρίτη Ομάδα Πειραμάτων

```
1  //random size and 25% chance of freeing each block
2  #if TEST==9
3    uint_t rand1, rand2, rand3;
4    int i;
5    char *p1, *p2, *p3;
6    unsigned short lfsr_ptr = 0xACE1u;
7
8    for(i=0; i<140; i++){
9      rand1 = RandMinMaxSyn(0, 256, &lfsr_ptr, 0);
10     p1 = (char *)MemluvAlloc(rand1, 0);
11
12     rand2 = RandMinMaxSyn(0, 256, &lfsr_ptr, 0);
13     p2 = (char *)MemluvAlloc(rand2, 0);
14
15     rand3 = RandMinMaxSyn(0, 256, &lfsr_ptr, 0);
16     p3 = (char *)MemluvAlloc(rand3, 0);
17
18     if(!RandMinMaxSyn(0, 3, &lfsr_ptr, 0)){
19       MemluvFree((void *)p1, rand1, 0);
20     }
21     if(!RandMinMaxSyn(0, 3, &lfsr_ptr, 0)){
22       MemluvFree((void *)p2, rand2, 0);
23     }
24     if(!RandMinMaxSyn(0, 3, &lfsr_ptr, 0)){
25       MemluvFree((void *)p3, rand3, 0);
26     }
27   }
28 #endif
```

Listing 3.10: Test 9

```
1  #if TEST==10
2    uint_t rand1, rand2, rand3, rand4, rand5;
3    int i;
4    char *p1, *p2, *p3, *p4, *p5;
5    unsigned short lfsr_ptr = 0xACE1u;
6
7    //i<84
8    for(i=0; i<84; i++){
9      rand1 = lookup_allocate[i*5];
10     p1 = (char *)MemluvAlloc(rand1, 0);
11
12     rand2 = lookup_allocate[i*5+1];
13     p2 = (char *)MemluvAlloc(rand2, 0);
14
15     rand3 = lookup_allocate[i*5+2];
16     p3 = (char *)MemluvAlloc(rand3, 0);
17
18     rand4 = lookup_allocate[i*5+3];
19     p4 = (char *)MemluvAlloc(rand4, 0);
```

```
20
21      rand5 = lookup_allocate[i*5+4];
22      p5 = (char *)MemluvAlloc(rand5, 0);
23
24      if(!lookup_free[i]){
25        MemluvFree((void *)p1, rand1, 0);
26      }
27      if(!lookup_free[i+1]){
28        MemluvFree((void *)p2, rand2, 0);
29      }
30      if(!lookup_free[i+2]){
31        MemluvFree((void *)p3, rand3, 0);
32      }
33      if(!lookup_free[i+3]){
34        MemluvFree((void *)p4, rand4, 0);
35      }
36      if(!lookup_free[i+4]){
37        MemluvFree((void *)p5, rand5, 0);
38      }
39    }
40  #endif
```

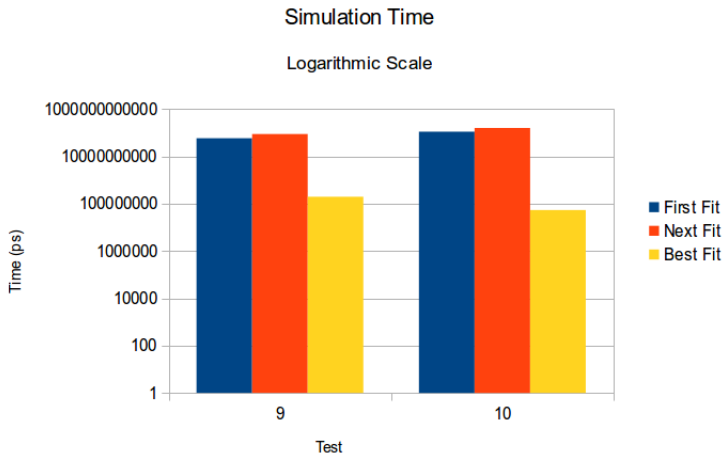
Listing 3.11: Test 10

Η τρίτη κατηγορία πειραμάτων περιλαμβάνει πειράματα τα οποία εκτελούν πλήθος δεσμεύσεων μνήμης μεταβλητού μεγέθους. Οι αποδεσμεύσεις σε αυτήν την κατηγορία γίνονται με κάποια πιθανότητα αποτυχίας. Τα πειράματα αυτής της κατηγορίας είναι τα εξής:

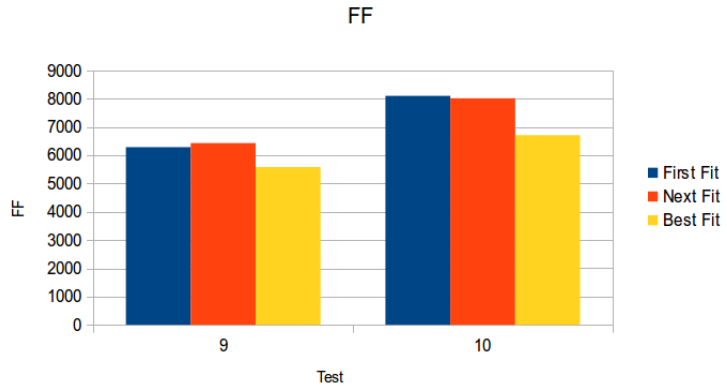
- *Πείραμα 9*: Σενάριο με 140 επαναλήψεις από 2 δεσμεύσεις τυχαίου μεγέθους και 2 αποδεσμεύσεις με 25% πιθανότητα. Το τυχαίο μέγεθος της κάθε δέσμευσης προκύπτει από μία ομοιόμορφη κατανομή στο διάστημα $[5, 256]$(listing 3.10).

- *Πείραμα 10*:Σενάριο με 84 επαναλήψεις από 5 δεσμεύσεις τυχαίου μεγέθους και 5 αποδεσμεύσεις με 10% πιθανότητα. Το τυχαίο μέγεθος της κάθε δέσμευσης προκύπτει από μία ομοιόμορφη κατανομή στο διάστημα $[5, 256]$(listing 3.11).

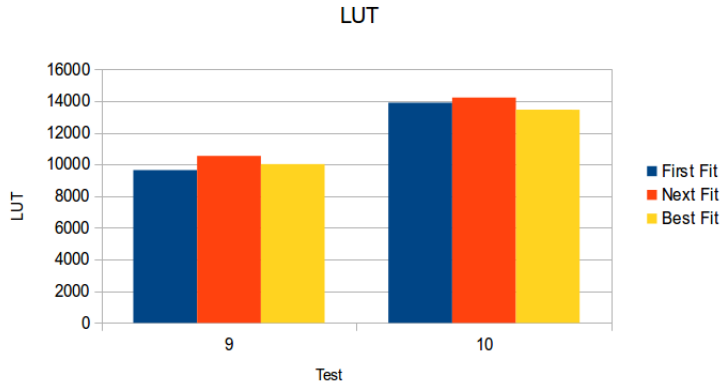
Αυτή η ομάδα πειραμάτων περιέχει πειράματα που μοιάζουν πιο πολύ με μία πραγματική περίπτωση σε σχέση με τις προηγούμενες ομάδες. Η δέσμευση τυχαίου μεγέθους δημιουργεί ελεύθερα κομμάτια διαφορετικού μεγέθους στη μνήμη και η μικρή πιθανότητα ελευθέρωσης των δεικτών οδηγεί σε πολύ υψηλότερα ποσοστά χρησιμοποίησης της μνήμης. Ο Best Fit είναι 446 και 2952 φορές πιο γρήγορος από τον Next Fit, ενώ είναι 258 και 2052 φορές πιο γρήγορος από τον First Fit στα πειράματα 9 και 10 αντίστοιχα (σχήμα 3.11). Το διαφορετικό μέγεθος των μπλοκ αναγκάζουν τους First Fit και Next Fit να πραγματοποιούν όλο και μεγαλύτερες αναζητήσεις στον bit map πίνακα το οποίο καταναλώνει αρκετούς κύκλους. Ο First Fit είναι πιο γρήγορος από τον Next Fit (x1.5) γιατί οι πιο πολλές δεσμεύσεις μνήμης συγκεντρώνονται στο ένα άκρο της με αποτέλεσμα να δημιουργούνται μεγαλύτερα κενά στο άλλο άκρο της. Έτσι υπάρχει μεγαλύτερη πιθανότητα για τον First Fit να πραγματοποιήσει τη δέσμευση μνήμης



Σχήμα 3.11: 3η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα

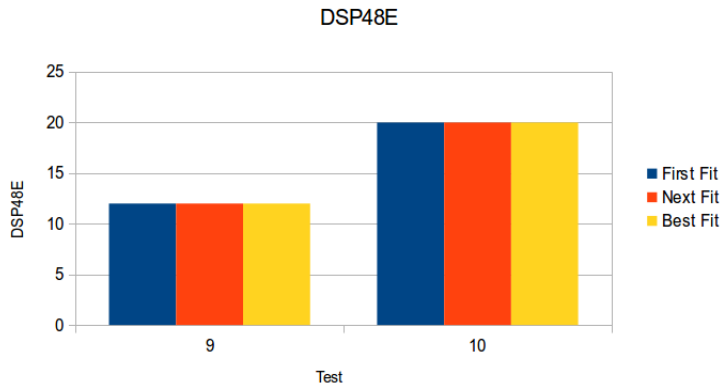


Σχήμα 3.12: 3η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ

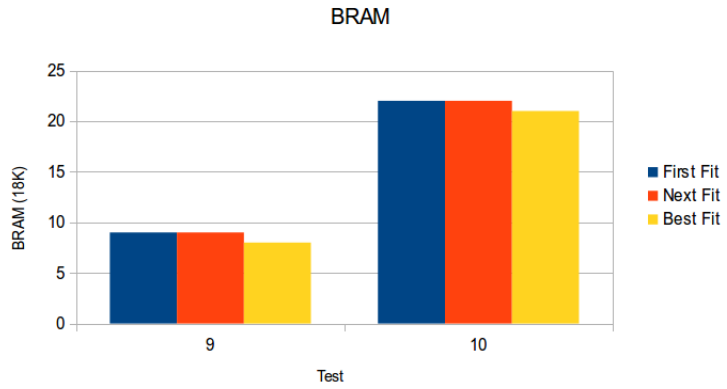


Σχήμα 3.13: 3η ομάδα πειραμάτων: δέσμευση LUT

χωρίς να χρειαστεί να ψάξει σε ολόκληρο τον πίνακα bit map [11]. Ο απόλυτος αριθμός των φλιπ-φλοπ (σχήμα 3.12) και των LUT (σχήμα 3.13) που χρησιμοποιούνται εξαρτάται σε μεγάλο βαθμό από το πείραμα που εκτελείται. Έν γένει όμως, η υλοποίηση του Best Fit δεσμεύει κατά 14% λιγότερα φλιπ-φλοπ από τους άλλους δύο αλγορίθμος, 3% λιγότερα LUT από τον First Fit και 5% λιγότερα LUT από τον Next Fit. Οι μπλοκ RAM (σχήμα 3.15) που



Σχήμα 3.14: 3η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 3.15: 3η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

χρησιμοποιούνται από τους First Fit και Next Fit είναι κατά μία περισσότερες γιατί χρειάζονται τον bit map πίνακα ο οποίος έχει μέγεθος ίσο με το $\frac{1}{8}$ του heap. Η σημαντική αύξηση στον αριθμό των μπλοκ RAM που παρατηρείται και για τους τρεις αλγορίθμους στό πείραμα 10 οφείλεται στη χρήση πινάκων για τις τυχαίες τιμές που χρειάζονται για την δέσμευση και την αποδέσμευση. Τα DSP48E που απαιτούνται είναι τα ίδια και για τις τρεις υλοποιήσεις (σχήμα 3.14).

3.2.4 Τέταρτη Ομάδα πειραμάτων

```
1  ...
2  #define MAX_BLOCKS 1000000
3  char * blkp[MAX_BLOCKS]
4  ...
5  void runloops(long sleep_cnt, int num_chunks ){
6
7    int cblks ;
8    int victim ;
9    int blk_size ;
10
11   long ticks_per_sec ;
```

```
12    long start_cnt, end_cnt ;
13    app_int64 ticks ;
14    double duration ;
15    double reqd_space ;
16    //ULONG used_space ;
17    int sum_allocs=0 ;
18
19    QueryPerformanceFrequency( &ticks_per_sec ) ;
20    QueryPerformanceCounter( &start_cnt) ;
21
22    for( cblks=0; cblks<num_chunks; cblks++){
23      if (max_size == min_size) {
24        blk_size = min_size;
25      } else {
26        blk_size = min_size+lran2(&rgen)%(max_size - min_size) ;
27      }
28
29      blkp[cblks] = (char *) malloc(blk_size) ;
30      blksize[cblks] = blk_size ;
31      assert(blkp[cblks] != NULL) ;
32    }
33
34    while(TRUE){
35      for( cblks=0; cblks<num_chunks; cblks++){
36        victim = lran2(&rgen)%num_chunks ;
37        free(blkp[victim]) ;
38        if (max_size == min_size) {
39          blk_size = min_size;
40        } else {
41          blk_size = min_size+lran2(&rgen)%(max_size - min_size) ;
42        }
43        blkp[victim] = (char *) malloc(blk_size) ;
44        blksize[victim] = blk_size ;
45        assert(blkp[victim] != NULL) ;
46      }
47
48      sum_allocs += num_chunks ;
49      QueryPerformanceCounter( &end_cnt) ;
50      ticks = end_cnt - start_cnt ;
51      duration = (double)ticks/ticks_per_sec ;
52      if( duration >= sleep_cnt) break ;
53    }
54    reqd_space = (0.5*(min_size+max_size)*num_chunks) ;
55  }
```

Listing 3.12: Larson Test [5]

```
1  //larson test
2  #if TEST==11
3    //char *          blkp[MAX_BLOCKS] ;
4    char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7,*p8,*p9;
```

```
5   char *p10,*p11,*p12,*p13,*p14,*p15,*p16,*p17,*p18,*p19;
6
7   uint32_t  blksize[MAX_BLOCKS] ;
8   uint32_t  min_size=10, max_size=500 ;
9
10  //number of seconds that the test lasts
11  long sleep_cnt = 10;
12
13  //number of memory chunks
14  int num_chunks = 1000;
15
16  int       cblks ;
17  int       victim ;
18  uint32_t  blk_size ;
19  uint32_t  free_size;
20  int       sum_allocs=0 ;
21  int       i=0;
22
23  blk_size = min_size+lookup_allocate[0];
24  blksize[0]=blk_size;
25  p0=(char *)MemluvAlloc(blk_size, 0);
26
27  blk_size = min_size+lookup_allocate[1];
28  blksize[1]=blk_size;
29  p1=(char *)MemluvAlloc(blk_size, 0);
30
31  blk_size = min_size+lookup_allocate[2];
32  blksize[2]=blk_size;
33  p2=(char *)MemluvAlloc(blk_size, 0);
34
35  blk_size = min_size+lookup_allocate[3];
36  blksize[3]=blk_size;
37  p3=(char *)MemluvAlloc(blk_size, 0);
38
39  blk_size = min_size+lookup_allocate[4];
40  blksize[4]=blk_size;
41  p4=(char *)MemluvAlloc(blk_size, 0);
42
43  blk_size = min_size+lookup_allocate[5];
44  blksize[5]=blk_size;
45  p5=(char *)MemluvAlloc(blk_size, 0);
46
47  blk_size = min_size+lookup_allocate[6];
48  blksize[6]=blk_size;
49  p6=(char *)MemluvAlloc(blk_size, 0);
50
51  blk_size = min_size+lookup_allocate[7];
52  blksize[7]=blk_size;
53  p7=(char *)MemluvAlloc(blk_size, 0);
54
```

```
55    blk_size = min_size+lookup_allocate[8];
56    blksize[8]=blk_size;
57    p8=(char *)MemluvAlloc(blk_size, 0);
58
59    blk_size = min_size+lookup_allocate[9];
60    blksize[9]=blk_size;
61    p9=(char *)MemluvAlloc(blk_size, 0);
62
63    blk_size = min_size+lookup_allocate[10];
64    blksize[10]=blk_size;
65    p10=(char *)MemluvAlloc(blk_size, 0);
66
67    blk_size = min_size+lookup_allocate[11];
68    blksize[11]=blk_size;
69    p11=(char *)MemluvAlloc(blk_size, 0);
70
71    blk_size = min_size+lookup_allocate[12];
72    blksize[12]=blk_size;
73    p12=(char *)MemluvAlloc(blk_size, 0);
74
75    blk_size = min_size+lookup_allocate[13];
76    blksize[13]=blk_size;
77    p13=(char *)MemluvAlloc(blk_size, 0);
78
79    blk_size = min_size+lookup_allocate[14];
80    blksize[14]=blk_size;
81    p14=(char *)MemluvAlloc(blk_size, 0);
82
83    blk_size = min_size+lookup_allocate[15];
84    blksize[15]=blk_size;
85    p15=(char *)MemluvAlloc(blk_size, 0);
86
87    blk_size = min_size+lookup_allocate[16];
88    blksize[16]=blk_size;
89    p16=(char *)MemluvAlloc(blk_size, 0);
90
91    blk_size = min_size+lookup_allocate[17];
92    blksize[17]=blk_size;
93    p17=(char *)MemluvAlloc(blk_size, 0);
94
95    blk_size = min_size+lookup_allocate[18];
96    blksize[18]=blk_size;
97    p18=(char *)MemluvAlloc(blk_size, 0);
98
99    blk_size = min_size+lookup_allocate[19];
100   blksize[19]=blk_size;
101   p19=(char *)MemluvAlloc(blk_size, 0);
102
103   for(i=0; i<20; i++){
104     victim = lookup_20_ints[i];
```

```
105     free_size = blksize[victim];
106     blk_size = min_size+lookup_allocate[i];
107
108     switch(victim){
109       case(0):
110         MemluvFree((void *)p0, free_size, 0);
111         p0=(char *)MemluvAlloc(blk_size, 0);
112         blksize[victim] = blk_size;
113         break;
114       case(1):
115         MemluvFree((void *)p1, free_size, 0);
116         p1=(char *)MemluvAlloc(blk_size, 0);
117         blksize[victim] = blk_size;
118         break;
119       case(2):
120         MemluvFree((void *)p2, free_size, 0);
121         p2=(char *)MemluvAlloc(blk_size, 0);
122         blksize[victim] = blk_size;
123         break;
124       case(3):
125         MemluvFree((void *)p3, free_size, 0);
126         p3=(char *)MemluvAlloc(blk_size, 0);
127         blksize[victim] = blk_size;
128         break;
129       case(4):
130         MemluvFree((void *)p4, free_size, 0);
131         p4=(char *)MemluvAlloc(blk_size, 0);
132         blksize[victim] = blk_size;
133         break;
134       case(5):
135         MemluvFree((void *)p5, free_size, 0);
136         p5=(char *)MemluvAlloc(blk_size, 0);
137         blksize[victim] = blk_size;
138         break;
139       case(6):
140         MemluvFree((void *)p6, free_size, 0);
141         p6=(char *)MemluvAlloc(blk_size, 0);
142         blksize[victim] = blk_size;
143         break;
144       case(7):
145         MemluvFree((void *)p7, free_size, 0);
146         p7=(char *)MemluvAlloc(blk_size, 0);
147         blksize[victim] = blk_size;
148         break;
149       case(8):
150         MemluvFree((void *)p8, free_size, 0);
151         p8=(char *)MemluvAlloc(blk_size, 0);
152         blksize[victim] = blk_size;
153         break;
154       case(9):
```

```
155           MemluvFree((void *)p9, free_size, 0);
156           p9=(char *)MemluvAlloc(blk_size, 0);
157           blksize[victim] = blk_size;
158           break;
159         case(10):
160           MemluvFree((void *)p10, free_size, 0);
161           p10=(char *)MemluvAlloc(blk_size, 0);
162           blksize[victim] = blk_size;
163           break;
164         case(11):
165           MemluvFree((void *)p11, free_size, 0);
166           p11=(char *)MemluvAlloc(blk_size, 0);
167           blksize[victim] = blk_size;
168           break;
169         case(12):
170           MemluvFree((void *)p12, free_size, 0);
171           p12=(char *)MemluvAlloc(blk_size, 0);
172           blksize[victim] = blk_size;
173           break;
174         case(13):
175           MemluvFree((void *)p13, free_size, 0);
176           p13=(char *)MemluvAlloc(blk_size, 0);
177           blksize[victim] = blk_size;
178           break;
179         case(14):
180           MemluvFree((void *)p14, free_size, 0);
181           p14=(char *)MemluvAlloc(blk_size, 0);
182           blksize[victim] = blk_size;
183           break;
184         case(15):
185           MemluvFree((void *)p15, free_size, 0);
186           p15=(char *)MemluvAlloc(blk_size, 0);
187           blksize[victim] = blk_size;
188           break;
189         case(16):
190           MemluvFree((void *)p16, free_size, 0);
191           p16=(char *)MemluvAlloc(blk_size, 0);
192           blksize[victim] = blk_size;
193           break;
194         case(17):
195           MemluvFree((void *)p17, free_size, 0);
196           p17=(char *)MemluvAlloc(blk_size, 0);
197           blksize[victim] = blk_size;
198           break;
199         case(18):
200           MemluvFree((void *)p18, free_size, 0);
201           p18=(char *)MemluvAlloc(blk_size, 0);
202           blksize[victim] = blk_size;
203           break;
204         case(19):
```

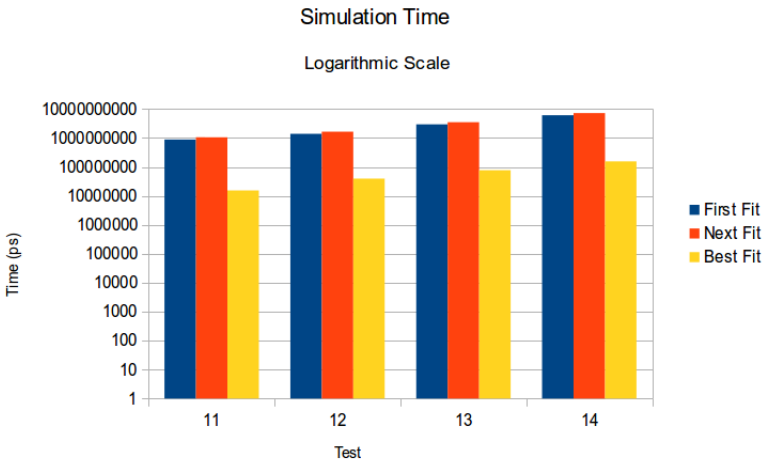
```
205           MemluvFree((void *)p19, free_size, 0);
206           p19=(char *)MemluvAlloc(blk_size, 0);
207           blksize[victim] = blk_size;
208         default:
209           continue;
210       }
211     }
212     result = (TB_UINT_T) i;
213 #endif
```

Listing 3.13: Test 11

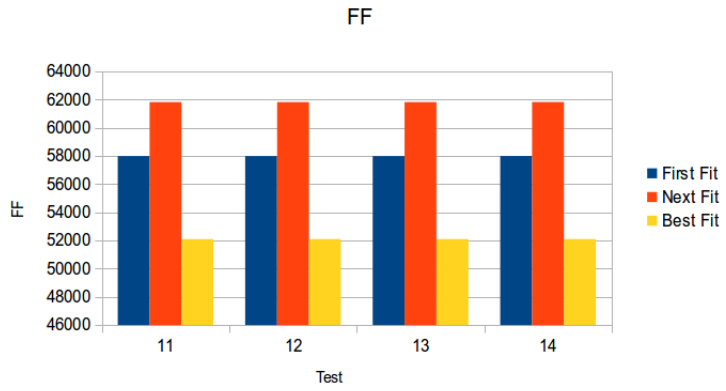
Για τη δημιουργία των πειραμάτων αυτής της κατηγορίας χρησιμοποιήθηκε σαν πρότυπο ένα πείραμα που έχει δημιουργηθεί από τον Per-Åke Larson και τον Murali Krishnan για την αξιολόγηση δυναμικών διαχειριστών μνήμης παράλληλων αρχιτεκτονικών[6] (listing 3.12). Το πείραμα αυτό (πλαίσιο κώδικα 3.12) διατηρεί έναν πίνακα από δείκτες και σε κάθε επανάληψη του κύριου βρόχου του επιλέγει έναν δείκτη τυχαία, ελευθερώνει τη μνήμη στην οποία δείχνει (`free()`)και στη συνέχεια εκτελεί καινούρια δέσμευση μνήμης τυχαίου μεγέθους (`malloc()`). Επειδή όμως το Vivado HLS δεν μπορεί να μεταγλωττίσει σε γλώσσα περιγραφής υλικού το πείραμα αυτό επακριβώς, δημιουργήθηκε ένα παρόμοιο πείραμα το όποιο είναι κατάλληλο για σύνθεση σε FPGA (πλαίσιο κώδικα 3.13). Τα πειράματα της κατηγορίας αυτής είναι παραλλλαγές της συνθέσιμης εκδοχής του πειράματος του Larson και ο κώδικάς τους δεν συμπεριλαμβάνεται σε αυτό το κεφάλαιο για συντομία.

- *Πείραμα 11*: Σενάριο Larson με 20 επαναλήψεις του `while` βρόχου. Δηλαδή 40 δεσμεύσεις μνήμης τυχαίου μεγέθους και 20 αποδεσμεύσεις.
- *Πείραμα 12*: Σενάριο Larson με 50 επαναλήψεις του `while` βρόχου. Δηλαδή 70 δεσμεύσεις μνήμης τυχαίου μεγέθους και 50 αποδεσμεύσεις.
- *Πείραμα 13*: Σενάριο Larson με 100 επαναλήψεις του `while` βρόχου. Δηλαδή 120 δεσμεύσεις μνήμης τυχαίου μεγέθους και 100 αποδεσμεύσεις.
- *Πείραμα 14*: Σενάριο Larson με 200 επαναλήψεις του `while` βρόχου. Δηλαδή 220 δεσμεύσεις μνήμης τυχαίου μεγέθους και 200 αποδεσμεύσεις.

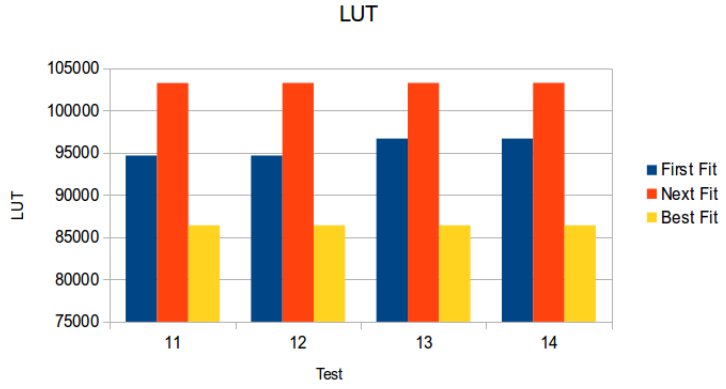
Σε αυτήν την περίπτωση ο Best Fit παρουσιάζεται 42 φορές πιο γρήγορος από τον First Fit και 50 φορές πιο γρήγορος από τον Next Fit (σχήμα 3.16). Όπως και στις προηγούμενες περιπτώσεις αυτό συμβαίνει γιατί οι πράξεις, που απαιτούνται από την αρχιτεκτονική που χρησιμοποιεί ο Best Fit, για την εύρεση του κατάλληλου μπλοκ, είναι πολύ λιγότερες από αυτές που απαιτούνται από την αρχιτεκτονική των First Fit και First Fit. Παρατηρούμε επίσης ότι ο χρόνος αυξάνεται γραμμικά ανάλογα με το πλήθος των επαναλήψεων του βρόχου `while`. Για τους λόγους που αναφέρθηκαν στην προηγούμενη ομάδα πειραμάτων ο First Fit παρουσιάζει και πάλι, καλύτερη απόδοση από τον Next Fit. Ο απόλυτος αριθμός των φλιπ-φλοπ (σχήμα 3.17) και των LUT (σχήμα 3.18) που χρησιμοποιούνται εξαρτάται σε μεγάλο βαθμό από το πείραμα που εκτελείται. Η υλοποίηση του Best Fit δεσμεύει κατά 10% λιγότερα φλιπ-φλοπ



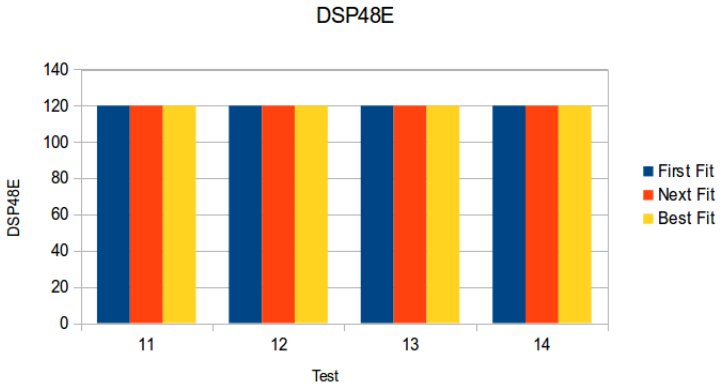
Σχήμα 3.16: 4η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα



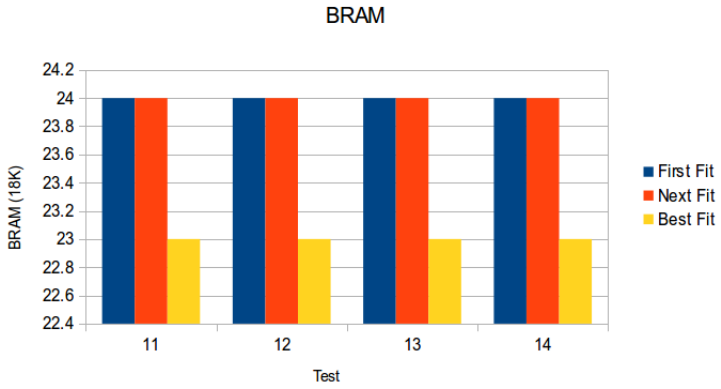
Σχήμα 3.17: 4η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ



Σχήμα 3.18: 4η ομάδα πειραμάτων: δέσμευση LUT



Σχήμα 3.19: 4η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 3.20: 4η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

από τον First Fit και 15% λιγότερα φλιπ-φλοπ από τον Next Fit, 9% λιγότερα LUT από τον First Fit και 16% λιγότερα LUT από τον Next Fit. Οι μπλοκ RAM (σχήμα 3.20) που χρησιμοποιούνται από τους First Fit και Next Fit είναι κατά μία περισσότερες γιατί χρειάζονται τον bit map πίνακα ο οποίος έχει μέγεθος ίσο με το $\frac{1}{8}$ του heap. Η σημαντική αύξηση στον αριθμό των μπλοκ RAM που παρατηρείται και για τους τρεις αλγορίθμους στην ομάδα αυτή οφείλεται στη χρήση πινάκων για τις τυχαίες τιμές που χρειάζονται για την δέσμευση και την αποδέσμευση. Τα DSP48E που απαιτούνται είναι τα ίδια και για τις τρεις υλοποιήσεις (σχήμα 3.19).

Κεφάλαιο 4

Συμπεράσματα

4.1 Γενικές παρατηρήσεις

Αυτή η διπλωματική συμβάλλει στην βελτιστοποίηση του συστήματος μνήμης των FPGA για συστήματα πολλαπλών επιταχυντών. Για αυτόν τον σκοπό προτείνεται βιβλιοθήκη δυναμικής διαχείρισης μνήμης. Αυτή η βιβλιοθήκη έχει υλοποιηθεί σε συνθέσιμο C κώδικα, για να μπορεί να ενσωματωθεί στην μοντέρνα διαδικασία προγραμματισμού των FPGA (π.χ. Υψηλού Επιπέδου Σύνθεση). Διαφορετικές υλοποιήσεις αυτού του συστήματος δυναμικής διαχείρισης μνήμης διερευνήθηκαν. Η πρώτη υλοποίηση χεησιμοποιεί εναν bit map πίνακα, του οποίου κάθε bit αντιπροσωπεύει ένα byte της μνήμης. Τα bit που είναι 1 συμβολίζουν ένα δεσμευμένο byte ενώ αυτά που είναι 0 ένα ελεύθερο byte. Αυτή η υλοποίηση χρησιμοποιείται από τους αλγορίθμους *First Fit* και *Best Fit*. Σε αυτήν την υλοποίηση το σύστημα δυναμικής διαχείρισης μνήμης ψάχνει διαδοχικά όλα τα bit μέχρι να βρει τον απαιτούμενο αριθμό ελεύθερων bit. Η δεύτερη υλοποίηση χρησιμοποιεί μία εντελώς διαφορετική προσέγγιση. Μία λίστα από ελεύθερα block μνήμης δημιουργείται με τη χρήση επικεφαλίδων στα ελεύθερα block μνήμης οπότε η freelist bitmap δεν χρειάζεται. Η επικεφαλίδα χρειάζεται δύο λέξεις μνήμης. Η πρώτη λέξη έχει το μέγεθος (σε bytes) του ελεύθερου μπλοκ και η δεύτερη περιέχει την θέση του επόμενου ελεύθερου block. Αυτήν την υλοποίηση χρησιμοποιεί ο *Best Fit*

Η πειραματική αξιολόγηση έδειξε ότι η δεύτερη υλοποίηση είναι πιο αποδοτική. Παρόλο που ο *Best Fit* πρέπει να ψάξει όλη την μνήμη για να τερματίσει την εκτέλεσή του, εξαιτίας της αποδοτικότερης υλοποίησης που χρησιμοποιεί, ο χρόνος που απαιτείται είναι αισθητά μικρότερος από την πρώτη υλοποίηση. Οι διαδικασίες που απαιτούνται για την ενημέρωση της freelist είναι πολύ χρονοβόρες και απαιτούν πολλούς κύκλους μηχανής. Ένα ακόμα ενδιαφέρον συμπέρασμα είναι ότι ο *Next Fit* είναι πιο αργός από τον *First Fit*. Ο *First Fit* τήνει να δεσμεύει μνήμη στην αρχή της μνήμης οπότε προς το τέλος της συγκεντρώνονται μεγάλα ελεύθερα block. Αντίθετα ο *Next Fit* ακυρώνει αυτήν την χρήσιμη συμεριφορά επειδή κατανέμει τις δεσμεύσεις σε όλο το μήκος της μνήμης. Οπότε αν προκύπτουν συχνά αιτήματα μεγάλου μεγέθους τότε ο *First Fit* ξεπερνά σε απόδοση τον *Next Fit*.

4.2 Μελλοντικές Επεκτάσεις

Μία υποσχόμενη προσέγγιση για την εξέλιξη του συστήματος δυναμικής διαχείρισης μνήμης θα ήταν η προσαρμογή των αλγορίθμων *First Fit* και *Next Fit* στην αρχιτεκτονική του *BestFit*. Με αυτήν την προσαρμογή η χρονική απόδοση τους θα βελτιωθεί αισθητά και οι χρόνοι εκτέλεσης θα είναι πολύ μικρότεροι από του *Best Fit*. Μία ακόμα πιθανή βελτίωση θα ήταν η δημιουργία ενός συστήματος δέσμευσης μνήμης που θα υποστηρίζει όλους τους αλγορίθμους και θα επιλέγει δυναμικά ποιον απο τους τρεις θα χρησιμοποιήσει. Αυτός ο συνδυασμός θα μπορούσε να χρησιμοποιεί αρχικά τον πιο γρήγορο αλγόριθμο, ο οποίος πιθανότατα είναι ο *First Fit*, και όταν ο κατακερματισμός της μνήμης περάσει κάποιο κατώφλι θα πρέπει να χρησιμοποιηθεί ο *Best Fit*. Αυτό το δυναμικό σύστημα δέσμευσης μνήμης μπορεί να δώσει τα καλύτερα αποτελέσματα σε σχέση με τον χρόνο προσομοίωσης και τον αριθμό των επιταχυντών που υποστηρίζονται. Επίσης ο *Best Fit* τήνει να αυξάνει τον αριθμό των μικρών block μνήμης το οποίο δεν είναι επιθυμητό. Μία καλή πρακτική είναι να τροποποιηθεί έτσι ώστε να πραγματοποιεί μία δέσμευση όταν το μέγεθος του ελεύθερου μπλοκ ικανοποιεί την ακόλουθη συνθήκη.

$$size\_requested \leqslant block's\_size \leqslant size\_requested + K$$

, όπου K ένας θετικός ακέραιος. Η μόνη τροποποίηση που απαιτείται στο σύστημα είναι η καταγραφή του ακριβούς μεγέθους της κάθε δέσμευσης η οποία μπορεί να υλοποιηθεί πολύ εύκολα με την χρήση επικεφαλίδων δέσμευσης [3].

Η μνήμη υλοποιείται από το Vivado® HLS σαν μία block-RAM. Αυτή η block-RAM μπορεί να είναι είτε single port (μόνο ένα read και ένα write μπορούν να πραγματοποιηθούν ταυτόχρονα) είτε dual port (μόνο δύο read και ένα write μπορούν να πραγματοποιηθούν ταυτόχρονα). Αυτός ο μηχανισμός περιορίζει την απόδοση ενός παράλληλου συστήματος. Μία πιθανή τροποποίηση που μπορεί να βελτιώσει την απόδοση ενός συστήματος πολλαπλών επιταχυντών υλικού είναι ο χωρισμός της μνήμης σε μικρότερα κομματια που θα συμπεριφέρονται σαν ανεξάρτητες block-RAM. Αυτό μπορεί να πραγματοποιηθεί πολύ εύκολα μέσα απο Vivado® HLS μέσω της εφαρμογής του array partition directive [1]. στην μνήμη. Ο πιο αποδοτικός χωρισμός της μνήμης βέβαια εξαρτάται κάθε φορά απο την εφαρμογή και μπορεί να προκύψει μέσα απο την προσεκτική αναζήτηση του χώρου λύσεων (design space exploration).

Το Vivado® HLS υποστηρίζει πολλές οδηγίες βελτιστοποίησης οι οποίες επηρεάζουν σημαντικά την αρχιτεκτονική του RTL κυκλώματος. Διαφορετικοί συνδυασμοί αυτών των οδηγιών θα μπορούσαν να συγκριθούν και να επιλεχθεί ο αποδοτικότερος από αυτούς.

[1] `#pragma HLS ARRAY_PARTITION variable=heap_core ...`

Bibliography

[1] Altera Corporation. Fpga architecture. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01003.pdf`, 2016. Last accessed on 20/03/2016.

[2] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris. Dynamic memory management in vivado-hls for scalable many accelerator architectures. *Proceedings of 11th International Applied Reconfigurable Computing Symposium*, 9040:117–128, April 2015.

[3] D. E. Knuth. *Fundamental Algorithms*. Addison-Wesley, third edition, 1968.

[4] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008. doi:10.1561/1000000005.

[5] Larson. Larson Test source code. `https://github.com/emeryberger/Malloc-Implementations/blob/master/allocators/streamflow/streamflow/larson.cpp`, 2012. Last accessed on 8/05/2016.

[6] P.-A. Larson and M. Krishnan. Memory allocation for long-running server applications. In *ISMM '98 Proceedings of the 1st international symposium on Memory management*, pages 176–185. ACM Press, 1998. doi:10.1145/301589.286880.

[7] M. M. Mano and M. D. Ciletti. *Digital Design*. Pearson, 2013.

[8] R. C. Minninck. A survey of microcellular research. *Journal of the Association of Computing Machinery*, 14(2):203–241, 1967. doi:10.1145/321386.321387.

[9] Research and Markets. FPGA Market by Type (High-End, Mid-End, Low-End), Verticals (Telecommunication, Industrial, AD, Automotive, Others), Architecture (Sram, Flash, Antifuse), Technology Node (28nm-10nm, 45/40nm, Others), and Geography - Forecast to 2022. `http://www.researchandmarkets.com/research/xc57px/fpga_market_by`, 2016. Last accessed on 20/03/2016.

[10] B. Schulz, C. Paiz, J. Hagemeyer, S. Mathapati, M. Porrmann, and J. Bocker. Run-time reconfiguration of fpga-based drive controllers. *2007 European Conference on Power Electronics and Applications*, pages 1–10, 2007. doi:10.1109/EPE.2007.4417686.

[11] J. E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the Association of Computing Machinery*, 18(8):433–440, 1975. doi:10.1145/360933.360949.

[12] G. R. Smith. *FPGAs 101: Everything you need to know to get started.* Elsevier Inc, 2010.

[13] S. E. Wahlstrom. Programmable logic arrays - cheaper by the millions. *Electronics*, 40(25):90–95, 1967. doi:10.1145/321386.321387.

[14] Xilinx, Inc. Vivado Design Suite User Guide, High Level Synthesis. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf`, 2014. Last accessed on 28/03/2016.

[15] Xilinx, Inc. Introduction to FPGA Design with Vivado High Level Synthesis (HLS). `http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`, 2015. Last accessed on 20/03/2016.

[16] Xilinx, Inc. Zynq-7000, All Programmable Soc-Technical Reference Manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2015. Last accessed on 20/03/2016.

[17] Xilinx, Inc. Fpga applications. `http://www.xilinx.com`, 2016. Last accessed on 20/03/2016.