

Εθνικό Μετσοβίο Πολγτεχνείο Σχολή Ηλεκτρολογών Μηχανικών και Μηχανικών Υπολογιέτων Τομέας Τεχνολογίας Πληροφορικής και Υπολογιέτων

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

Διπλωματική Εργάσια

του

Στέφανου Κόφφα

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

Εργαστηριο Μικρομπολογιστών και Ψηφιακών Σύστηματών 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

Δ ιπλωματική Εργάσια

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 Αν. Καθηγητής Ε.Μ.Π. Καθηγητής Ε.Μ.Π.

Γεώργιος Οικονομάκος Επ. Καθηγητής Ε.Μ.Π.

(Υπογραφή)

N----**I**Z----

Στεφανός 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 difficulty 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

Υλοποίηση Συστήματος Δυναμικής Διαχείρισης μνήμης σε FPGA μέσω τεχνικών Υψηλού Επιπέδου Σύνθεσης

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). 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 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



Σχήμα 1: Αρχιτεκτονική ενός FPGA



Σχήμα 2: Αρχιτεκτονική ενός σύγχρονου FPGA



Σχήμα 3: Διαδιχασία υψηλού επιπέδου σύνθεσης με το Vivado HLS

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 directives). Σε αυτή τη διπλωματική για την υψηλού επιπέδου σύνθεση θα χρησιμοποιηθεί το Vivado HLS. Όπως φαίνεται και στο σχήμα 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 βελτιστοποπίησης (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 πρόχειται 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 πίναχες από δείχτες 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, ή σχεδίαση ενός συστήματος γίνεται σε κάποια γλώσσα υψηλού επιπέδου, όπως C/C++, με

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].

Τα FPGA αποτελούν μία ελχυστική 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.
- 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.

Όπως φαίνεται και στο σχήμα 4 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.



Σχήμα 4: Τροποποιημένη διαδικασία χρήσης του Vivado HLS σύμφωνα με το Σύστημα Δυναμικής Διαχείρισης μνήμης[2]

Το αρχιτεκτονικό πρότυπο ενός συστήματος πολλών επιταχυντών υλικού που χρησιμοποιεί τον δυναμικό διαχειριστή μνήμης φαίνεται στο σχήμα 5. 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 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 (πλαίσιο κώδικα 3.2) 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 6 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.



Many-accelerators System Interconnection Network



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.

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 7), αποθηκεύεται το μέγεθος του εκάστοτε ελεύθερου μπλοκ αλλά και η θέση του επόμενου ελεύθερου μπλοκ. Στο σχήμα 8 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 9.

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

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



Σχήμα 6: Υλοποίηση 1: bit map πίναχας για τον έλεγχο των ελεύθερων και δεσμευμένων byte του σωρού [2]

Number of bytes	Position of the Next Free block
-----------------	---------------------------------

Σχήμα 7: Δομή επικεφαλίδας των ελεύθερων μπλοκ μνήμης για την δεύτερη υλοποίηση



Σχήμα 8: Στιγμιότυπο μνήμης πριν την δημιουργία της λίστας των ελεύθερων μπλοκ (λευκά:ελεύθερα, μαύρα:δεσμευμένα)

ελεύθερα μπλοκ. Τα σημαντικά σημεία αυτών των τεχνικών είναι τα εξής:

 Δέσμευση μνήμης: Στο σχήμα 10 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



Σχήμα 9: Στιγμιότυπο μνήμης μετά την δημιουργία της λίστας των ελεύθερων μπλοκ (λευκά:ελεύθερα, μαύρα:δεσμευμένα)



Σχήμα 10: Διαδικασία δέσμευσης μνήμης

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 μπλοχ. 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 11) θα πρέπει να δεσμευτεί ολόκληρο το μπλοκ. Όμως αυτή η πληροφορία είναι γνωστή μόνο στον allocator και πρέπει να διατηρηθεί με τη βοήθεια



Σχήμα 11: 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





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



Σχήμα 13: 2η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συχγωνεύεται μόνο με το επόμενο ελεύθερο μπλοχ



Σχήμα 14: 3η περίπτωση αποδέσμευσης: Το μπλοχ που ελευθερώνεται (γχρι) συχγωνεύεται μόνο με το προηγούμενο ελεύθερο μπλοχ

τερο μέρος του σχήματος 11) 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



Σχήμα 15: 2η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) δεν συχγωνεύεται με κάποιο γειτονικό ελεύθερο μπλοκ

μπλοκ που ελευθερώνεται πρέπει να συγχωνευτεί με κάποιο γειτονικό του. Έτσι προκύπτουν 4 περιπτώσεις. Στην πρώτη (σχήμα 12) το μπλοκ που πρόκειται να ελευθερωθεί πρέπει να συγχωνευτεί με το προηγούμενο και το επόμενο ελεύθερο μπλοκ. Όπως φαίνεται και στο σχήμα 12 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 13), το μπλοκ που ελευθερώνεται πρέπει να συγχωνευτεί με το προηγούμενο της λίστας (σχήμα 14) και το μπλοκ που ελευθερώνεται απλά προστίθεται στη λίστα χωρίς να συγχωνευτεί με κάποιο άλλο (σχήμα 15).

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

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 δεσμέυσεις και 4 αποδεσμεύσεις.
- Πείραμα 2: Σενάριο με 10 δεσμεύσεις και 4 αποδεσμεύσεις.
- Πείραμα 3: Σενάριο με 256 επαναλήψεις από 2 δεσμεύσεις σταθερού μεγέθους και 2 αποδεσμεύσεις.
- Πείραμα 4: Σενάριο με 128 επαναλήψεις από 4 δεσμεύσεις σταθερού μεγέθους και 4 αποδεσμεύσεις.



Σχήμα 16: 1η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα



Σχήμα 17: 1η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ



Σχήμα 18: 1η ομάδα πειραμάτων: δέσμευση LUT

Από το σχήμα 16 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



Σχήμα 19: 1η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 20: 1η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

που γίνονται σε αυτό το πείραμα είναι λίγες στο πλήθος, αλλά σχετιχά μεγάλες, οπότε αυτή η μεγάλη διαφορά στην απόδοση των δύο υλοποιήσεων οφείλεται στις χρονοβόρες διαδιχασίες ενημέρωσης των bit του bit map πίναχα. Στο δεύτερο πείραμα η διαφορά της απόδοσης των δύο υλοποιήσεων είναι πιο μιχρά χαι η ενημέρωση των bit του bit map πίναχα πραγματοποιείται σε πολύ λιγότερους χύχλους. Τα άλλα δύο πειράματα χρησιμοποιούν μιχρές δεσμεύσεις μνήμης μιχρού χαι σταθερού μεγέθους και όπως είναι φυσιχό οι αλγόριθμοι First Fit χαι Next Fit υπερισχύουν έναντι του Best Fit. Ο απόλυτος αριθμός των φλιπ-φλοπ (σχήμα 17) χαι των LUT (σχήμα 18) που χρησιμοποιούνται εξαρτάται σε μεγάλο βαθμό από το πείραμα που εχτελείται. Έν γένει η υλοποίηση του Best Fit δεσμεύει χατά 18% λιγότερα φλιπ-φλοπ, 5% λιγότερα LUT από τον First Fit, χαι 13% λιγότερα LUT από τον Next Fit. Οι μπλοχ RAM (σχήμα 20) που χρησιμοποιούνται από τους First Fit χαι Next Γit χρειάζονται τον bit map πίναχα ο οποίος έχει μέγεθος ίσο με το $\frac{1}{8}$ του heap. Τα DSP48E που απαιτούνται είναι τα ίδια χαι για τις τρεις υλοποιήσεις (σχήμα 19).

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:



Σχήμα 21: 2η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα



Σχήμα 22: 2η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ

- Πείραμα 5: Σενάριο με 10 επαναλήψεις από 16 δεσμεύσεις σταθερού μεγέθους και 16 αποδεσμεύσεις με πιθανότητα 50%.
- Πείραμα 6: Σενάριο με 30 επαναλήψεις από 8 δεσμεύσεις σταθερού μεγέθους και 8 αποδεσμεύσεις με πιθανότητα 10%.
- Πείραμα 7: Σενάριο με 50 επαναλήψεις από 8 δεσμεύσεις σταθερού μεγέθους και 8 αποδεσμεύσεις με 30% πιθανότητα.
- Πείραμα 8: Σενάριο με 50 επαναλήψεις από 8 δεσμεύσεις σταθερού μεγέθους και 8 αποδεσμεύσεις με 10% πιθανότητα.

Σε αυτή την κατηγορία πειραμάτων ο Best Fit παρουσιάζεται πέντε φορές πιο γρήγορος από τους άλλους δύο αλγορίθμους (σχήμα 21). Αυτό συμβαίνει γιατί ο τρόπος που αναζητείται στον Best Fit η κατάλληλη ελεύθερη περιοχή μνήμης είναι πολύ πιο αποδοτικός και αυξάνεται η ταχύτητά του όσο τα δεδομένα στη μνήμη αυξάνονται. Ο χρόνος εχτέλεσης των

17



Σχήμα 23: 2η ομάδα πειραμάτων: δέσμευση LUT



Σχήμα 24: 2η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 25: 2η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

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 22) και των LUT (σχήμα 23) που χρησιμοποιούνται



Σχήμα 26: 3η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα

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. Έν 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 25) που χρησιμοποιούνται από τους First Fit και Next Fit είναι κατά μία περισσότερες γιατί χρειάζονται τον bit map πίναχα 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 24).

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 χάθε δέσμευσης προχύπτει από μία ομοιόμορφη χατανομή στο διάστημα [5,256].
- Πείραμα 10:Σενάριο με 84 επαναλήψεις από 5 δεσμεύσεις τυχαίου μεγέθους και 5 αποδεσμεύσεις με 10% πιθανότητα. Το τυχαίο μέγεθος της χάθε δέσμευσης προχύπτει από μία ομοιόμορφη χατανομή στο διάστημα [5,256].

Αυτή η ομάδα πειραμάτων περιέχει πειράματα που μοιάζουν πιο πολύ με μία πραγματιχή περίπτωση σε σχέση με τις προηγούμενες ομάδες. Η δέσμευση τυχαίου μεγέθους δημιουργεί ελεύθερα χομμάτια διαφορετιχού μεγέθους στη μνήμη χαι η μιχρή πιθανότητα ελευθέρωσης των δειχτών 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 26). Το διαφορετικό μέγεθος των μπλοκ αναγκάζουν τους First Fit και Next Fit να πραγματοποιούν όλο και μεγαλύτερες αναζητήσεις στον bit map πίναχα το οποίο χαταναλώνει αρχετούς χύχλους. Ο First Fit είναι πιο γρήγορος από τον Next Fit (x1.5) γιατί οι πιο πολλές δεσμεύσεις μνήμης συγχεντρώνονται



Σχήμα 27: 3η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ



Σχήμα 28: 3η ομάδα πειραμάτων: δέσμευση LUT



Σχήμα 29: 3η ομάδα πειραμάτων: δέσμευση DSP48E

στο ένα άχρο της με αποτέλεσμα να δημιουργούνται μεγαλύτερα χενά στο άλλο άχρο της. Έτσι υπάρχει μεγαλύτερη πιθανότητα για τον First Fit να πραγματοποιήσει τη δέσμευση μνήμης χωρίς να χρειαστεί να ψάξει σε ολόχληρο τον πίναχα bit map [11]. Ο απόλυτος αριθμός των φλιπ-φλοπ (σχήμα 27) και των LUT (σχήμα 28) που χρησιμοποιούνται εξαρτάται σε μεγάλο βαθμό από το πείραμα που εχτελείται. 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



Σχήμα 30: 3η ομάδα πειραμάτων: δέσμευση μπλοκ RAM

Fit και 5% λιγότερα LUT από τον Next Fit. Οι μπλοκ RAM (σχήμα 30) 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 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 29).

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]. Το πείραμα αυτό (πλαίσιο κώδικα 4.12) 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 4.13). Τα πειράματα της κατηγορίας αυτής είναι παραλλλαγές της συνθέσιμης εκδοχής του πειράματος του Larson.

- 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 31). Όπως και στις προηγούμενες



Σχήμα 31: 4η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα



Σχήμα 32: 4η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ



Σχήμα 33: 4η ομάδα πειραμάτων: δέσμευση LUT



Σχήμα 34: 4η ομάδα πειραμάτων: δέσμευση DSP48E



Σχήμα 35: 4
η ομάδα πειραμάτων: δέσμευση μπλο
κ ${\rm RAM}$

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 32) και των LUT (σχήμα 33) που χρησιμοποιούνται εξαρτάται σε μεγάλο βαθμό από το πείραμα που εκτελείται. Η υλοποίηση του Best Fit δεσμεύει κατά 10% λιγότερα φλιπ-φλοπ 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 35) 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 34).

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

\mathbf{A}	ckno	wledge	ments	1
Υ	λοπα τεχ	οίηση Σ νικών	Ουστήματος Δυναμικής Διαχείρισης μνήμης σε FPGA μέσω Υψηλού Επιπέδου Σύνθεσης	3
A	bstra	ict		25
п	εριεχ	χόμενα	x	28
к	ατάλ	ογος	Σχημάτων	31
\mathbf{Li}	sting	s		34
1	Intr	oducti	ion	35
	1.1	Introd	uction to FPGA	35
		1.1.1	Early History of Programmable Logic	35
		1.1.2	Latest Trends	37
		1.1.3	Areas of use	39
		1.1.4	Architecture	40
		1.1.5	Benefits of the FPGA	44
	1.2	Thesis	Overview	46
		1.2.1	Chapters Organization	46
2	FPO	GA De	sign Flow and High Level Synthesis	49
	2.1	Introd	uction	49
	2.2	Develo	opement Phases	49
		2.2.1	Design	49
		2.2.2	Simulation	51
		2.2.3	Synthesis	51
		2.2.4	Implementation	53
		2.2.5	Programming	53
	2.3	High I	Level Synthesis	53
		2.3.1	HLS Compiler	54

		2.3.2	HLS Design Methodology	57
3	Dyn	amic I	Memory Management Framework for HLS	69
	3.1	Introdu	uction	69
	3.2	The D	MM-HLS Framework	69
		3.2.1	Dynamic Memory Management in Vivado-HLS	70
		3.2.2	DMM-HLS Overview	72
		3.2.3	DMM-HLS Details	74
	3.3	The D	MM-HLS Enhanchement	98
		3.3.1	Next Fit	98
		3.3.2	Best Fit	103
4	\mathbf{Exp}	erimer	nts and Results	123
	4.1	Testing	g Environment	123
4.2 Experimental analysis		mental analysis	124	
		4.2.1	First group of experiments	124
		4.2.2	Second group of experiments	128
		4.2.3	Third group of experiments	135
		4.2.4	Larson Test	139
5	The	sis Coi	nclusion	149
	5.1	Genera	l Remarks	149
	5.2	Future	Work	150
Bi	bliog	raphy		152

Κατάλογος Σχημάτων

1	Αρχιτεκτονική ενός FPGA	4
2	Αρχιτεκτονική ενός σύγχρονου FPGA	4
3	Διαδικασία υψηλού επιπέδου σύνθεσης με το Vivado HLS	5
4	Τροποποιημένη διαδικασία χρήσης του Vivado HLS σύμφωνα με το Σύστημα	
	Δυναμικής Διαχείρισης μνήμης[2]	8
5	Αρχιτεκτονικό πρότυπο για συστήματα πολλών επιταχυντών υλικού μετά από	
	την εφαρμογή του συστήματος δυναμικής διαχείρισης μνήμης[2]	9
6	Υλοποίηση 1: bit map πίναχας για τον έλεγχο των ελεύθερων χαι δεσμευμένων	
	byte του σωρού [2]	10
7	Δ ομή επιχεφαλίδας των ελεύθερων μπλοχ μνήμης για την δεύτερη υλοποίηση .	10
8	Σ τιγμιότυπο μνήμης πριν την δημιουργία της λίστας των ελεύθερων μπλοχ (λευ-	
	κά:ελεύθερα, μαύρα:δεσμευμένα)	10
9	Σ τιγμιότυπο μνήμης μετά την δημιουργία της λίστας των ελεύθερων μπλοχ	
	(λευχά:ελεύθερα, μαύρα:δεσμευμένα)	11
10	Διαδικασία δέσμευσης μνήμης	11
11	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	
	επιχεφαλίδα δέσμευσης τότε δημιουργείται μία λίστα με αυτές τις λέξεις	12
12	1η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συγχωνε-	
	ύεται και με το προηγούμενο αλλά και με το επόμενο ελεύθερο μπλοκ	12
13	2η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συχγωνε-	
	ύεται μόνο με το επόμενο ελεύθερο μπλοκ	13
14	3η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) συχγωνε-	
	ύεται μόνο με το προηγούμενο ελεύθερο μπλοχ	13
15	2η περίπτωση αποδέσμευσης: Το μπλοκ που ελευθερώνεται (γκρι) δεν συχγω-	
	νεύεται με κάποιο γειτονικό ελεύθερο μπλοκ	14
16	1η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα	15
17	1η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ	15
18	1η ομάδα πειραμάτων: δέσμευση LUT	15
19	1η ομάδα πειραμάτων: δέσμευση DSP48E	16
20	1η ομάδα πειραμάτων: δέσμευση μπλοκ RAM	16
21	2η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα	17

22	2η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ	17
23	2η ομάδα πειραμάτων: δέσμευση LUT	18
24	2η ομάδα πειραμάτων: δέσμευση DSP48E	18
25	2η ομάδα πειραμάτων: δέσμευση μπλοκ RAM	18
26	3η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα	19
27	3η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ	20
28	3η ομάδα πειραμάτων: δέσμευση LUT	20
29	3η ομάδα πειραμάτων: δέσμευση DSP48E	20
30	3η ομάδα πειραμάτων: δέσμευση μπλοκ RAM	21
31	4η ομάδα πειραμάτων: Χρόνος προσομοίωσης σε λογαριθμική κλίμακα	22
32	4η ομάδα πειραμάτων: δέσμευση φλιπ-φλοπ	22
33	4η ομάδα πειραμάτων: δέσμευση LUT	22
34	4η ομάδα πειραμάτων: δέσμευση DSP48E	23
35	4η ομάδα πειραμάτων: δέσμευση μπλοκ RAM	23
1 1		26
1.1	PAI [7]	30 37
1.2	All programmable system on chip: Zyng series ^[17]	38
1.0	Simplified Instration of a logic coll	- <u>J</u> O
1.4	Basic FPCA Architecture	40
1.0	Modern FPGA Architecture	41
1.0	Example 4 input I UT [1]	42
1.7	D fip-flop structure [15]	42
1.0	Structure of DSP Block [15]	40
1.3 1 10	Structure of an addressable shift register [15]	44
1.10	Structure of an addressable sint register [15]	44
2.1	FPGA Development Phases [12]	50
2.2	Design Time vs Performance for different designs	51
2.3	Simulation Levels $[12]$	52
2.4	Synthesis stages $[12]$	52
2.5	Implementation steps $[12]$	53
2.6	Example code for three operations $[15]$	55
2.7	Execution of example code on a $Processor[15]$	55
2.8	Execution of HLS code on an FPGA[15] \ldots \ldots \ldots \ldots \ldots \ldots	56
2.9	HLS synthesis output for code in listing 2.1 [14] \ldots \ldots \ldots \ldots	59
2.10	HLS synthesis output for code in listing 2.2 [14]	60
2.11	HLS synthesis output for code in listing 2.2 $[14]$	60
2.12	Dataflow optimization [14]	61
2.13	Clock period and margin $[14]$	63
2.14	Loop Unrolling details $[14]$	65
2.15	Function and Loop pipelining behaviour [14]	65
2.16	Array partitioning[14]	66
2.17	Loop merging example $[14]$	
------	---	
3.1	Memory Bottleneck of Kmeans clustering algorithm[2]	
3.2	Extended Vivado HLS flow with DMM implementation[2]	
3.3	Many accelerator FPGA based systems architecture enchanced with DMM-	
	HLS framework $[2]$	
3.4	Example memory snapshot	
3.5	Free Block's header structure	
3.6	Best Fit's memory snapshot	
3.7	Unused words scenario	
3.8	Allocation process	
3.9	First Free Scenario	
3.10	Second Free Scenario	
3.11	Third Free Scenario	
3.12	Fourth Free Scenario	
4.1	First group of experiments: Simulation Time in logarithmic scale 127	
4.2	First group of experiments: Number of flip-flops	
4.3	First group of experiments: number of LUTs	
4.4	First group of experiments: Number of DSP48E	
4.5	First group of experiments: Number of block RAMs	
4.6	Second group of experiments: Simulation Time in logarithmic scale 134	
4.7	Second group of experiments: Number of flip-flops	
4.8	Second group of experiments: number of LUTs	
4.9	Second group of experiments: Number of DSP48E	
4.10	Second group of experiments: Number of block RAMs	
4.11	Third group of experiments: Simulation Time in logarithmic scale 138	
4.12	Third group of experiments: Number of flip-flops	
4.13	Third group of experiments: number of LUTs	
4.14	Third group of experiments: Number of DSP48E	
4.15	Third group of experiments: Number of block RAMs	
4.16	Fourth group of experiments: Simulation Time in logarithmic scale 146	
4.17	Fourth group of experiments: Number of flip-flops	
4.18	Fourth group of experiments: number of LUTs	
4.19	Fourth group of experiments: Number of DSP48E	
4.20	Fourth group of experiments: Number of block RAMs	

Listings

2.1	Demonstration code for parallel operations
2.2	Demonstration code for pipelined operations
3.1	Configuration File[2] $\ldots \ldots 75$
3.2	Struct MemLuvCore[2]
3.3	Struct MemLuvStats[2]
3.4	Struct MemLuvConf[2]
3.5	Function MemluvSetBitFreelist[2]
3.6	Function MemluvClearBitFreelist[2]
3.7	Function MemluvTestBitFreelist[2]
3.8	Function MemluvSetFreelist [2]
3.9	$Function MemluvClearFreelist()[2] \dots \dots$
3.10	Allocation Wrapper[2]
3.11	First Fit Implementation[2]
3.12	Function CurMemluvFree[2]
3.13	Function MemluvCalcPtrDistanceFromBase[2]
3.14	Function CurMemluvFreeBody[2]
3.15	Next Fit MemLuvCore
3.16	Next Fit implementation
3.17	Next Fit's implementation of function CurMemluvFreebody
3.18	Changes in struct MemluvCore for Best Fit
3.19	Best Fit's initialization
3.20	Best Fit's CurMemluvAlloc
3.21	Best Fit's implementation
3.22	Best Fit's allocation mechanism
3.23	Function find_previous_unused
3.24	Best Fit's CurMemluvFree()
3.25	Function MemluvCalcPtrDistanceFromBase _b $f()$
3.26	Function CurMemluvFreeBody_bf()
3.27	Function chk_for_unused()
3.28	Function free_memory()
4.1	Test Bench's main function $[2]$
4.2	Test 1

4.3	Test 2										•				•	•	•					•				•		•			•		•		1	25
4.4	Test 3							•						•	•	•	•	•		•		•		•		•							•		1	25
4.5	Test 4							•						•	•	•	•	•		•		•		•		•							•		1	26
4.6	Test 5										•			•	•	•	•	•				•				•		•			•		•		1	29
4.7	Test 6										•			•	•	•	•	•				•				•		•			•		•		1	30
4.8	Test 7					•	•	•			•	•	•	•	•	•	•	•		•		•	•	•				•			•	•	•		1	31
4.9	Test 9							•						•	•	•	•	•	•	•		•		•		•		•	•				•		1	32
4.10	Test 9 $$					•	•	•			•	•	•	•	•	•	•	•		•		•	•	•				•			•	•	•		1	36
4.11	Test 10						•	•	•		•			•	•	•	•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•		1	36
4.12	Larson	Τ	es	\mathbf{t}	[5]	•	•			•	•	•	•	•	•	•	•		•		•	•	•				•			•	•	•		1	39
4.13	Test 11																																		1	41

Chapter 1

Introduction

1.1 Introduction to FPGA

The Field Programmable Gate Array (FPGA) is a type of integrated circuit that can be reprogrammed many times after fabrication resulting in a different operation each time. Nowadays, FPGAs consist of so many logic cells, that a large variety of software algorithms can be implemented. The traditional process of programming an FPGA ressembles this of an Integrated Circuit (IC) but the device's nature offers great advantages. FPGAs' performance is similar to that of Application Specific Integrated Circuits (ASICs) but the cost of programming them is much lower. Their ability to be easily reconfigured gives the opportunity to quickly fix erroneous behaviours and leads to a more adaptive hardware that can be adjusted each and every time to the needs and restrictions of the market. [15]

1.1.1 Early History of Programmable Logic

FPGAs are closely related to the development of the integrated circuit which took place in the 1960s. The first precursors of FPGAs were the cellular arrays that were made as arrays of interconnected simple logic cells with fixed functionality. This technology evolved and Maitra cascade was created. This circuit was an one-dimensional variable cell function array. Each cell of this array could operate in one of 16 different functions of two or fewer variables and thus the ability to change the functionality of a circuit after its fabrication was been created. [8]. Following the Maitra cascade, the most important steps towards FPGAs were the cutpoint array and the cobweb array [8]. The cutpoint array consisted of a number of vertical Maitra cascades. Each cell supported 8 combinational functions but the connections between the cells were fixed. The cobweb array was a cell array in which both the functionality of the cells and the interconnection structure could be changed via its parameters. [8].

The next important step towards the FPGAs was the invention of read-only memories (ROM). ROM was an electronic device in which binary information could be permanently stored by properly configuring the interconnection structure of its internal elements. After this configuration, the information remains "saved" in the device even after the power



Figure 1.1: PLA [4]

supply is cut. The closest predeseccors of modern FPGAs are the programmable ROMs (PROMS). These devices were created by the manufacturers without any data within them, so a user could configure them according to his/her needs with an electric pulse applied to a specific port of the device. More specifically, two types of the PROMs were most important.

The first was the programmable logic array (PLA). As shown at figure 1.1, this device consisted of one AND plane and one OR plane, which were both programmable. The second was the programmable array logic (PAL). As it is shown at figure 1.2, this device consisted of an AND plane, which could be programmed and a fixed OR plane. If the function to be implemented can be expressed with only two levels of logic, these devices provide a perfect solution.

The first static memory-based FPGA, which allowed both logic and interconnection manipulation via a stream of configuration bits was proposed by Wahlstrom in 1967[13]. Technical difficulties, though, with regards to the area of a device using an SRAM, postponed the invention of the first FPGA untill 1984. This year, the co-founders of Xilinx, R. Freeman and B. Vonderschmitt, invented the first modern era FPGA which consisted of 64 logic blocks and 58 inputs and outputs. [4].

This was the start of the FPGA market, which was then populated by a significant number of vendors, including Xilinx, Altera, Actel, Lattice, Crosspoint, Algotronix, Prizm, Plessey, Toshiba, Motorola, and IBM. The number of vendors has diminished over the yars and now the market is dominated by Xilinx and Altera whose combined market share



Figure 1.2: PAL [7]

amounts to c. 67% of the total market. This market has seen an impressive growth and it is estimated that its size will be equal to \$8.2 billion by 2020 [9].

1.1.2 Latest Trends

During the last two decades, FPGAs developed dramatically achieving both a great improvement in their performance and an extension of their capabilities. Important architectural changes took place and, nowadays, FPGAs have grown from being a simple glue logic component to representing a complete System on Programmable Chip (SoPC) comprising on-board physical processors, dedicated DSP hardware, on-chip memory and highspeed I/O. For example, Xilinx's Zynq®-7000 familly combines the concurrent nature of an FPGA device with an ARM ® high end microcontroller and its features (memory and memory controlers, I/O and peripherals, hard-core implementations of 32-bit processor), as shown at figure 1.3[16]. This architecture helps developers to create complex designs, which combine serial with parallel execution leading to faster throughput and latency. In addition, by allowing embedded designers to design in a familiar ARM environment, they



Figure 1.3: All programmable system-on-chip: Zynq series[17]

can benefit from the time-to-market advantages of an FPGA platform compared to more traditional design cycles associated with ASICs.

Another important feature added to modern FPGAs is the ability of run-time reconfiguration. This feature is used when the functionality of an FPGA should be changed while it operates. Run-time reconfiguration (RTR) might affect either only parts of a design (partial RPR) or the whole chip (full RPR). Run-time reconfiguration can reduce power consumption and area impact of a design, especially if it can be modularized. For instance, if a device should be both a transmitter and a receiver but not done simultaneously, each module can be loaded when needed. The standard FPGA process would require both modules to be implemented on the device resulting in more space and power consumption, because logic components still consume energy even when they remain idle. [10]

1.1.3 Areas of use

In the beginning, FPGAs were used only for ASIC prototyping and in designs that the time-to-market was really important. In the second case, the designers would replace the FPGA with an equivalent ASIC at the first opportunity.

As their size, capabilities and speed increased, their areas of use rose respectively. The introduction of dedicated multipliers into FPGA architectures made them suitable choices for Digital Signal Processing (DSP) applications. The full system-on-chip approach creates a solution that is applicable to a wide range of engineering fields.

Nowadays, FPGAs are used in the following technological fields:

- Aerospace and defense: especially Avionics & UAV, Military Communications & Public Safety Radio, Missiles and Munitions, Secure Solutions, Situational Awareness, Space, Small Form Factor and Battery Powered Software Defined Radio Public Safety & Military Mobile Radios and 24 Channel Radar.
- ASIC Prototyping: ASIC prototyping and emulation have been boosted with the invention of all programmable System on Chip (SoC). Using this technology, the need for multi-chip partitioning is eliminated, risks for developing large ASIC designs are mitigated and power consumption is reduced.
- Audio: modern FPGAs are a perfect choice for DSP designs required in audio processing, compression, interfacing and conversion. Their parallel nature also facilitates the simultaneous and efficient process of multiple audio channels, which leads to excellent performance and quality.
- Automotive: a relative new field that FPGAs are used. Their many to implement driver assistance through real-time image analytics (object and pedestrian detection, lane departure etc.). Besides they are key element of intelligent transport means which can distribute video and data around the vehicle.
- Consumer Electronics: the good time-to-market feature that FPGAs support and the smart vision applications that can be implemented, gives the opportunity for FPGA based systems to meet the requirements of the rapidly evolving field of consumer electronics.
- *Datacenter*: FPGA based solutions are designed for high-bandwidth, low-latency servers, networking, and storage applications in order to bring higher value into cloud deployments.
- *High Performance Computing*: modern FPGAs are used in High Performance Computing as software accelerators (taking over a portion of the computationally intensive tasks of a software application which runs on a CPU) or as hardware accelerators (providing high throughput data processing at a fraction of GPUs' power).



Figure 1.4: Simplified Ilustration of a logic cell

- *Industrial*: FPGAs are also used to enhance factory automation frameworks and industrial imaging applications.
- *Medical*: FPGAs are used in ultrasound, Computerized Tomography Scanners, Magnetic Resonance Imaging (MRI), X-ray, Positive Emission Topography (PET) and Surgical Systems because they can support the increasing needs in processing power of large image files.
- Smarter Networks both wired and wireless: smarter networks need the ability to efficiently and quickly adapt to changes in network traffic demands, configuration, network utilization, and market requirements. For the implementation of all these abilities, reprogrammability is needed so that FPGAs are a perfect choice for this field.
- *Smarter Vision*: FPGAs are the right choice for the majority of computer vision algorithms, which are computationally intensive but can be easily parallelized.

```
[17]
```

1.1.4 Architecture

An FPGA consists of the following elements:

- Look-up table (LUT): This element performs logic operations.
- Flip-Flop (FF): This element stores the result of the LUT.
- Interconnection structure: This structure connects elements to one another.
- *I/O ports*: These ports move data from and to the FPGA.

The combination of these elements results in the basic FPGA component, which is the Configurable Logic Block (CLB). As shown at the figure 1.4, a typical block can



Figure 1.5: Basic FPGA Architecture

be created using 4-input LUT, a full adder (FA), and a D flip-flop (DFF). Each FPGA manufacturer defines its own basic structure, which may be different depending on the applications that each FPGA device targets at. The resulting structure is shown at figure 1.5, where the basic building blocks are organized in an array in which the interconnection structure between them can be changed in order to perform various functions required by the design.

As mentioned at 1.1.2, modern FPGAs combine their basic structure along with additional computational and data storage blocks in order to increase the efficiency of the device. These elements are:

- Embedded memories for additional data storage
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

The resulting architecture is shown at figure 1.6.



Figure 1.6: Modern FPGA Architecture



a'b'c'd' + abcd + abc'd' = 1000 0000 0000 1001 = 0x8009

Figure 1.7: Example 4-input LUT [1]

1.1.4.1 LUT

A LUT is the basic building block of an FPGA and it is responsible for implementing any logical function of N inputs. In the context of combinatorial logic, LUT is a truth



Figure 1.8: D flip-flop structure [15]

table. It consists of 2^N SRAM output bits, which can return all the possible outcomes of the input values according to the function that is implemented. As shown at the figure 1.7, a LUT can be made by connecting the memory cells to multiplexers, and depending on the input values, only one cell is enabled and returned as output. At the figure the 16:1 (i.e. 2^4 :1) multiplexer is implemented as a tree of 2:1 multiplexers. LUTs can also be used as data storage elements.

1.1.4.2 Flip-Flop

This is the FPGAs' basic storage element. It is always paired with a LUT to assist logic pipelining and data storage [15]. As shown at the figure 1.8, this component includes a data input, clock input, clock enable, reset and data output. If the signal clk_en is 1, the goal of the flip-flop is to propagate the input value to the output in every clock pulse.

1.1.4.3 DSP48 Block

This is the most complex computational element embedded in Xilinx FPGAs and is shown at figure 1.9. This element is an Arithmetic Logic Unit (ALU) that consists of three stages of operations. The first stage is an add/subtract, the second is a multiplier and the third is an add/subtract/accumulate engine.

1.1.4.4 Memory Elements

Instead of the LUTs which were discussed above, FPGAs include memory elements that can be used as random access memory (RAM), read only memory (ROM) or shift registers.

The BRAM is a dual port memory module embedded into the FPGAs. It is used to store data. Xilinx's FPGAs can support two sizes, 18K or 36K bits. Owing to the



Figure 1.9: Structure of DSP Block [15]



Figure 1.10: Structure of an addressable shift register [15]

fact that this is a dual port element it can support up to two operations simoultaneously allowing for paralel accesses to different memory locations. These BRAMs can be used to implement both RAMs (when data should be read and written) and ROMs (when data should be read and not modified).

The shift register as shown at figure 1.10 is a chain of D flip-flops sharing the same clock, in which the output of each flip-flop is connected to the data input of the next flip-flop in the chain. This circuit is usually used for simple delay of data, for grouping of serial data so that they can be processed in parallel and for stack structures.

1.1.5 Benefits of the FPGA

When compared to processor architectures, FPGAs offer great parallel processing capabilities. In a processor, the performance of the assembly produced by a high level language's compiler depends largely on the location of the data to be processed in the memory hierarchy. When the data to be processed is located in the cache memory, a load or a store operation needs significant fewer clock cycles compared to those needed when the data is in the main memory of the hard drive. As a result, software engineers spend a considerable amount of time in order to create algorithms, which raise the number of cache hits by exploiting the spatial locality of data. Furthermore, in processors, the code is run sequentially and even independent operations cannot be computed simoultaneously. The FPGA, however, is inherently a parallel processing device and every algorithm to be implemented must be reformed for parallel processing. When an FPGA is programmed, it can create more than one Arithmetic Logic Units (ALUs) that can operate in parallel, resulting in a more efficient architecture than that of single processors, which have only one ALU. Another important fact concerning FPGAs is that every time that they are programmed, a custom memory architecture is created that consists of storage banks as close as possible to the point of operation.

These actions are performed by the hardware designer manually or by the High Level Synthesis (HLS) tool automatically so that the capabilities of the FPGA are used (HLS will be discussed in depth in the next chapter). Either the HLS compiler or the hardware designer tries to adapt every design to its FPGA-effective through the processes of scheduling, pipelining and dataflow.

1.1.5.1 Scheduling

Scheduling is the process of identifying the data dependencies between different operations in order to decide which of them can be executed in parallel. A designer should analyze dependencies between operations and group together those that can be executed in the same clock cycle.

1.1.5.2 Pipelining

Pipelining is a design technique, which allows multiple operations to overlap. In this process, the computational circuit is divided to a chain of indepentent stages which can be executed in the same cycle. The output of each stage is the input of the next one. Using registers, the output of each stage is fed to the next one. As a consequence, in every clock cycle a different operation can start its execution and the throughput can be significantly increased.

1.1.5.3 Dataflow

Dataflow is another digital design technique, which enables the parallel execution of functions in the same program. Two scenarios are well suited for the dataflow technique. The first is when two functions need independent data sets for their execution and do not communicate with each other. In that simple case, the functions can be executed without any difficulties, in parallel. The more complex case is when one function feeds its results to another function (consumer-producer scenario). This scenario is more complicated but the paralellism is still feasible however not as efficient as the first one.

1.2 Thesis Overview

"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" [2].

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 [2].

In this diploma thesis:

- A dynamic memory management library of the recent literature was evaluated and extended. The functionality of this library was studied thoroughly in order to propose possible extensions.
- Architectural optimizations such as pipeline, dataflow and array partition were applied to computationally intensive algorithms via Vivado HLS. This step targeted at the familiarization with Vivado HLS tool and at the better understanding of its methodology.
- Next Fit algorithm was added in the DMM-HLS interface. Two different implementations of this algorithm were created and analyzed.
- Best Fit algorithm was added in the DMM-HLS interface. For the implementation of this algorithm an alternative approach of the manager's core logic was developed.
- The proposed architecture was evaluated by a variety of test cases in order to verify its functionality and to compare its performance with the existing implementation.

1.2.1 Chapters Organization

This diploma thesis is organized according to the following structure:

• *Chapter 1*: In this chapter an introduction to FPGAs is presented. This introduction starts from past inventions that led to modern FPGAs. Then, it demonstrates the technological fields in which modern FPGAs are used. Finally, their architecture is described and the benefits that they offer are presented.

- *Chapter 2*: This chapter describes the FPGA design flow, i.e. every step that is required for the programming of an FPGA device, and then gives an insightful view in High Level Synthesis process. First, Vivado (R) HLS'¹ logic is described. Then, the recommended design methodology that leads to more efficient designs is analyzed.
- *Chapter 3*: In this chapter the Dynamic Memory Management (DMM) interface is shown. First, the basic functionality is analyzed and the rest of the chapter explains the exact implementation of the DM manager for every different feature (First Fit, Next Fit, Best Fit).
- *Chapter 4*: This chapter consists of two sections. In the first one, the test cases that were created are analyzed. The second one, presents the simulation and sythesis results that were yielded by these experiments. An analysis of these results is also included in this chapter.
- *Chapter 5*: This is the last chapter and it concludes the findings of this study and highlights the future extensions arising from the use of the proposed methodologies.

¹the tool that is used in this diploma thesis

Chapter 2

FPGA Design Flow and High Level Synthesis

2.1 Introduction

Traditionally, FPGA development process is divided in the following stages: design, simulation, synthesis, implementation and programming. As shown at figure 2.1, these stages may overlap or interact whith each other. A brief description of each stage is given in section 2.2. Lately, however, the field of High Level Synthesis (HLS) has been developed. HLS can transform a specification of a high level language such as C, C++ or Java into a Register Transfer Level (RTL) implementation that can be synthesized in an FPGA. As shown at figure 2.2a, the time needed for an optimized version of RTL usually exceeds the time limit of the project. But figure 2.2b shows that the design time of a project through HLS is minimized and meets the time requirements [15].

2.2 Development Phases

2.2.1 Design

Design is the initial stage of the FPGA development process. This stage can be a conversion of a schematic to Hardware Description Language (HDL), various modifications to an existing HDL design or even the creation of a new design entirely from "scratch". During this process, many decisions should be made. First, a "design package" should be created. This describes the system's requirements and it is the outcome of predesign activities. The most important of them are:

- Creation of architecture
- The partitioning the design into sections and the i efficient assignment of them efficiently to the developers
- Decision on what the system should do through clear and unambiguous requirements



Figure 2.1: FPGA Development Phases [12]

- Creation of timing and other diagrams
- Decision on the design format (HDLs, Shcematic, Combination)
- Decision on the Manufacturer (Xilix, Altera, Quicklogic etc.) and the actual device part number (depending on the field of the application, on environmental conditions, temperature range and design size)
- Decision on the tools that should be used in this process. Design format, cost, and the fact that a design is shared among developers are the major considerations at this stage.

The next step in design process is the evaluation of the "design package" by the developers. This step is important because the developers need to understand thouroughly what firmware ¹ should be made. Any ambiguities should be clarified and the "design package" should be always updated with all the changes that occur through the development process.

¹HDL is a language that describes hardware and it is written in general by hardware engineers.



(a) Design Time vs Application Performance (b) Design Time vs Application Performance with RTL Design Entry

with Vivado HLS Compiler

Figure 2.2: Design Time vs Performance for different designs

2.2.2Simulation

Simulation is the process that verifies that the design operates as it should. Specifically, during the simulation, stimulus² is applied to the design and the output is observed. The stimulus can be either manually given to the simulator or produced automatically through graphical or HDL test benches. As shown at figure 2.3, the simulation process can take place after design, synthesis or implementation which creates three levels of simulation. The first level uses the RTL that was produced by the design stage in order to tackle logic and syntax errors. This level of simulation does not contain timing information. Functional simulation uses the either the netlist or the code along with some timing information generated automatically by the synthesis tool. It checks whether the design is altered by the synthesis process. The last and most accurate simulation level uses the code or the netlist that was created by the implementation tool. This simulation uses actual timing information of the signals and hence it is the only one, which can demonstrate timing problems of a design [12].

2.2.3Synthesis

Synthesis is the process, which takes a design and associates it with internal FPGA resources. It can be decomposed into three stages as shown at figure 2.4. The first is the design check and resource association, in which the design is checked for syntax and synthesis errors and every code's logical abstraction, such as mathematical operators, is replaced with the relevant logic elements. The next stage is the optimization. A synthesis tool first tries to adapt the design to the existing hardware and then it uses some algorithms in order to find and remove redundant logic, and to decide which will be the best timing policy and which will be the clock's speed. The last stage is the technology mapping, in which the optimized version of the design is mapped to the resources that the specific

²input data signals.



Figure 2.3: Simulation Levels[12]



Figure 2.4: Synthesis stages [12]

device offers. The synthesis stage can produce netlists (both functional simulation and design netlists), status reports (hardware resources usage, clock and timing information, critical paths, warnings, errors etc.) and schematic views that depict the design with generic symbols, such as adders, multipliers, counters, AND gates and OR gates [12].



Figure 2.5: Implementation steps[12]

2.2.4 Implementation

During the implementation, the physical layout of the design is determined and the netlist is mapped to FPGA resources, which are interconnected to its internal logic and I/O resources. This process is usually broken into three different steps as shown at figure 2.5. Translation is the first step, during which the netlist is combined with the design constraints and a new file is created. This file will be the input to the next step, which is known as mapping. During this step, the logic of the design is mapped to the FPGA's logic cells, the I/O cells and to other resources. This step produces Native Circuit Design (NCD) file that is used by the next step, known as place and route. Place and route process interconnects the design as it is stated in the NCD file and, if told so, it produces the gate level simulation and timing files. The next step takes the output file from place and route and generates a bitstream file, which is used to program the FPGA [12].

2.2.5 Programming

The last development phase is programming. Programming is the process of configuring the device with the bitstream file, which is generated by the implementation phase. Download cables, as they are called, connect the device with the workstation and pass the bitstream file from the workstation to the FPGA. This process takes as input a bitstream file and produces a programmed device. Some devices support In-System Programming (ISP); they can be programmed whithout being removed from the rest of the system. Designers should be very careful while programming a device because errors introduced in this phase are not easily spotted and tackled [12].

2.3 High Level Synthesis

High level synthesis transforms a specification given in a high level language (C, C++, Java) into an RTL implementation, which can be synthesized in an FPGA. HLS is a bridge beetween software and hardware domains and both can benefit from its advantages. First, hardware designers can take advantage of the productivity benefits that a higher level of abstraction offers. HLS also provides software developers with the capability to accelerate computationally intensive parts of their algorithms using the FPGA parallel architecture.

Vivado (R) HLS that is used in this diploma thesis, transforms only C specifications (C, C++, SystemC). The major benefits of Vivado (R) HLS design methodology are:

- The high level of abstraction that saves crucial time needed for implementation details
- The verification of the functional correctness can be done to the high level of abstraction and every bug can be identified and fixed more easily compared to hardware description languages.
- The optimization directives can create specific high performance hardware implementations.
- Many different designs can be evaluated by using different optimization directives. This enables an easy design space exploration and increases the likelihood of finding the optimal solution.
- Using the HLS, a designer can create portable C code, which can be easily retargeted at a different device just by changing an HLS' parameter.

The two processes which are the beating heart of HLS are scheduling and binding. Scheduling phase decides which operations are executed in each clock cycle. It takes into consideration the clock frequency, timing information from the target device's technology library and any user defined optimization directives. The clock period shows how many operations can be completed in every cycle so, if a design is mapped to different devices, the scheduling decisions can be different. Binding determines which hardware resource will be used for each scheduled operation. HLS, during the binding process uses specific information about the target device in order to create the optimal solution.

Another important phase of the HLS is the extraction and implementation of control logic and I/O ports. Function arguments create ports in the final RTL design and arrays are implemented by default as block RAMs ³. HLS creates the required data and addresses ports and any chip-enable or write-enable signals. [14]

2.3.1 HLS Compiler

Vivado[®] HLS is a compiler, which provides an environment similar to those for application development. It shares components with compilers for interpretation, analysis and optimization of C/C++ programs. The main difference between Vivado[®] HLS and a common C/C++ compiler is the execution target of the application. Vivado[®] HLS' nature enables software engineers to optimize their code for throughput, power, and latency. HLS compiler analyzes a program in terms of operations, conditional statements, loops and functions [15].

³other options are available through optimization directives.

A[i]	=	B[i]	*	C[i];
D[i]	=	B[i]	*	E[i];
F[i]	=	A[i]	+	D[i];

Figure 2.6: Example code for three operations[15]



Figure 2.7: Execution of example code on a Processor[15]

2.3.1.1 Operations

Operations refer to both the arithmetic and logic elements that contribute to the computation of a resulting value. Comparison statements are excluded because they are examined in section 2.3.1.2. HLS creates a targeted architecture for each algorithm in C/C++ unlike the general purpose compilers, which must stick to a specific architecture. This gives more leeway to the HLS designer who can now affect application performance in terms of throughput, latency and power. If the code of figure 2.6 was executed in a processor the steps would be completed sequentially because the architecture has only one EXE stage as shown at figure 2.7. On the contrary, even HLS' default behaviour analyzes the dependencies of the code, creates a custom memory hierarchy, which contains the data as close to the point of its consumption and finally produces better results as shown at figure 2.8 [15].

2.3.1.2 Conditional Statements

Conditional statements are the control flow statements implemented by if-else or case statements. In a processor, branch operations introduce idle cycles in the execution



Figure 2.8: Execution of HLS code on an FPGA[15]

pipeline because the condition must be calculated before the fetch of the next instruction. In an FPGA, however, all the possible circuits are created and a branch is just the selection between two different routes. Therefore their impact is significantly less[15].

2.3.1.3 Loops

Loops are a common programming construct and they are well supported by the HLS compiler. In a processor compiler, the iterations of the loop are sequentially executed, which means that one iteration starts only when the previous one has been completed. The standard behaviour of Vivado (R) HLS is the same but optimization directives allow the HLS designer to create pipelined architectures, which result in less computational latency and increased input data rate. When the HLS tries to achieve this results, it resolves data dependencies by altering the operations of the loop body and resource contentions by instantiating more copies of the resources. In case it cannot complete this process, it asks the user for algorithm changes.

2.3.1.4 Functions

HLS can fully parallelize the execution of functions using the dataflow optimization. This option instructs the HLS compiler to create independent hardware modules, that are capable of concurrent execution and self-synchronization during data transfer. Hence, by sacrificing a small amount of hardware resources, significant speedup can be achieved.

2.3.1.5 Dynamic Memory Allocation

The programming languages C and C++ allow their applications to dynamically allocate and deallocate memory when needed. It is not necessary for the compiler to know the amount of memory to be requested during the compile time. The HLS compiler, though, cannot support this functionality because it creates custom hardware circuits and specific memory architecture for each application. As a result, code provided to the HLS must use only compile time analyzable memory allocation. This feature may increase the total memory footprint of a software application but this fact should not bother HLS designers because, when HLS is used, the designer targets at transforming the algorithm in a way that will give the best possible hardware implementation [15].

2.3.1.6 Pointers

A pointer is an address to a memory location. Pointers are widely used in C/C++ as function parameters, for array handling, for dynamic memory allocation. They are usually combined with type casting. HLS compiler supports pointers that can be fully analyzed during compile time. An analyzable pointer's usage can be computed with pen and paper whithout the need of runtime parameters [15].

2.3.2 HLS Design Methodology

If hardware designers want to exploit HLS capabilities to their maximum, they should follow an appropriate design methodology. Such an approach could lead to less time to market and better performance. In the following sections a collection of good practices about design validation, hardware efficient C code, synthesis strategies, design analysis, optimization, and verification is described.

2.3.2.1 The C Test Bench

The test bench is a C/C++ program which calls the function that will be synthesized multiple times, provides appropriate input arguments and checks the results produced. This program is responsible for the C simulation and the verification of whether a design is correct. The simulation can be run before the synthesis. At this point, it checks only the validity of the software implementation of the algorithm. If the test bench is self checking (automatically checks the results produced to the expected values) it can be used by Vivado (R) HLS inorder to execute C/RTL co-simulation. Co-simulation verifies the post-synthesis results and decides if something went wrong during the synthesis process. The test bench is the main() function of the source code and its return value determines whether the design is correct in the following ways:

- It is set to zero if the results are correct
- It is set to a non-zero value if the results are wrong
- Attention should be paid if there is no return statement in function main() of the test bench. If this is the case then the C standard sets the return value to zero and the simulation will always succeed.

In HLS designs, the gratest amount of time is usually consumed when a designer tries to debug a wrong C specification through RTL simulation. RTL simulation is orders of magnitude slower than C simulation and more difficult to debug. Therefore, the time spent for the creation of a C test bench and for execution of C simulation offers great productivity boost to the whole process and should not be omitted.

2.3.2.2 Language Support

A designer who uses Vivado[®] HLS should keep in mind two important facts about every implementation in FPGA:

- An FPGA is a fixed size resource, and thus objects cannot be dynamically created and destroyed.
- There is no underlying operating system (OS) and all FPGA's communication should be performed via input and output ports.

For the above mentioned two reasons there are two categories of constructs in every HLS C design: unsupported and partially supported.

The unsupported are:

- System Calls: There is no OS to interact. Some common system calls are automatically excluded by the HLS without any error (abort(), atexit(), exit(), fprintf(), printf(), perror(), putchar() and puts())
- Dynamic Objects: Everything that is going to be synthesized must be of a known size at compile time. So malloc(), alloc(), free(), new, delete are not supported by the HLS. For the same reason, recursion is also not supported.

The partially supported constructs are:

- **Pointers**: Vivado (R) HLS supports pointer casting beetween only native C types. Pointer arrays are not supported. Pointer to pointer is supported for synthesis but not when used as an argument to the top level function.
- Memory Functions: The memcpy() and memset() are supported but only when constant values are used.

When a software implementation contains unsupported constructs, the synthesis process will fail and therefore everything that will be synthesized should be modified.

2.3.2.3 Understanding Concurrent Hardware

The HLS compiler can create unique hardware circuits for every operation in a C specification. As a result, more than one operations can be simultaneously computed. In the code shown at listing 2.1, the design that is produced by HLS will not calculate y0, y1, and y2 sequentially but it will perform two of the calculations in parallel. This is the

```
+ Latency (clock cycles):

    * Summary:

    Latency | Interval | Pipeline|

    min | max | min | max | Type |

    +---+--+---+

    2 | 2 | 3 | 3 | none |
```

Figure 2.9: HLS synthesis output for code in listing 2.1 [14]

reason why the compiler creates more than one multiplier circuits. Assuming that the clock frequency is adjusted to fit only one multiplication, the design will be completed in 2 clock cycles and not in 3. Therefore it takes 2 cycles to output the results and the design can consume input every 3 cycles (Initiation Interval) as shown at the figure 2.9.

```
#include "foo.h"
  int foo(char a, char b, char d){
2
  #pragma HLS INTERFACE ap_ctrl_none register port=return
3
    int y0, y1, y2;
5
    y0 = a * b;
    y1 = c * y0;
7
    y^2 = d * y^1;
9
10
    return y2;
11
  ł
```

Listing 2.1: Demonstration code for parallel operations

With a more detailed analysis, it can be observed that, during the first two multiplications the third multiplier remains idle and, during the third multiplication the first two multipliers remain idle. Concurrent hardware is capable of dealing with this situation because it supports pipelined operation. This way of operation uses all the resources at the same time (figure 2.10). The initiation interval is reduced to one (figure 2.11) only with the addition of line 5 in the previous code (listing 2.2). This results in a high performance design (high throughput), which can consume input in every clock cycle.

```
#include "foo.h"
\mathbf{2}
  int foo(char a, char b, char d){
  #pragma HLS INTERFACE ap_ctrl_none register port=return
3
  #pragma HLS PIPELINE
4
5
    int y0, y1, y2;
6
    y0 = a * b;
7
    y1 = c * y0;
8
    y^2 = d * y^1;
9
10
    return y2;
11
12 }
```



Figure 2.10: HLS synthesis output for code in listing 2.2 [14]

Latency (clock * Summary:	cycles):	
++ Latency min max	Interval min max	Pipeline Type
	1 1	function

Figure 2.11: HLS synthesis output for code in listing 2.2 [14]

Besides the operations, through Vivado (\mathbb{R}) HLS, tasks can be also pipelined and run in parallel. Figure 2.12 demonstrates the dataflow optimization, which schedules each function to start its operation when the required data is available. In this example the initiation interval is decreased from 8 cycles to 5. This optimization directive helps Vivado (\mathbb{R}) HLS to understand which tasks ⁴ are independent and should be scheduled in parallel execution.

2.3.2.4 Default Synthesis Strategies

The creation of high performance designs with HLS requires a thorough understanding of its default behaviour and the optional optimizations that it supports.

First, Vivado (R) HLS uses the target device and decides how many operations can be performed in every clock cycle. After this decision it tries to optimize the design's performance with the following order.

• *Interval*: It tries to minimize the interval beetween new inputs and to increase the data throughput rate.

⁴in this example the tasks shown are functions but this optimization can be applied beetween functions and loops, and beetween loops.



Figure 2.12: Dataflow optimization [14]

- Latency: Its next step is the minimization of latency.
- Area: The last optimization which is performed is the minimization of the area occupied by the created design whithout changing its achieved interval and latency.

The default synthesis behaviour can be summarized in the following steps:

- Synthesis of Top-Level Function Arguments: The top-level function arguments are synthesized into data ports with an optional I/O protocol. This protocol is used to synchronize the data communication beetween the data port and other hardware resources. For example, a valid port can indicate when the data of its associated port is valid for reading of writing. The type of the protocol used is decided by the type of the C argument. Pass-by-value scalar arguments and pass-by-refernce inputs give data ports with no I/O protocol. Pass-by-reference outputs give data output ports with associated output valid ports. Pass-by-reference arguments which are both read and written, are partitioned to separate input and output ports following the above rules. Array arguments are implemented as block-RAM memory ports. An I/O protocol associated with the top-level function itself is also synthesized. This protocol permits the function operation and indicates when the output data is ready and when the function can consume new input data.
- Synthesis of sub-functions: Functions are synthesized into hierarchical blocks in the final RTL design and are scheduled to be executed as early as possible. As a result, every C function will by default be represented by a unique block. In some cases, though, Vivado (R) HLS may automatically inline small functions in order to

achieve better results. This functionality can be also applied manually with the inline optimization directive. Any user defined optimization stops at function boundaries unless a recursive option is set.

- Synthesis of loops:Loops are by default kept rolled. This means that the loop body is synthesized once and it is executed sequentially until it reaches its execution limit.
- Synthesis of arrays: Arrays are by default synthesized into block-RAM. Block-RAM is made of blocks of 18K-bit. Each block RAM will use as many 18K-bit blocks as required to implement each array. For example an array of 1024 integers will require

$$\frac{1024 * 32}{18000} = 1.8$$

blocks. Vivado (R) HLS automatically decides if a RAM should be implemented as either single or dual port for better latency results. The resource optimization directive can be used to manually declare the number of ports to be implemented. Vivado (R) HLS does not partition large arrays into smaller block RAMs and makes no attempt to merge small arrays into a single larger block RAM. Only small arrays are automatically partitioned to individual registers inorder to improve the quality of the results. This task is left to the designer through the relevant optimization directives.

- Synthesis of structs: Structs by default are decomposed into their member elements.
- Synthesis of operators: C operators such as +, *, and / are synthesized into hardware cores. HLS automatically determines which core will best suit each operation but the designer can use the resource optimization directive to manually specify one.
- Specifying the clock frequency: For C/C++ designs only a single clock is supported. HLS uses the user defined clock period and the device target information in order to create estimates of the timing of operations. It is not possible for the HLS to know the exact placement and routing, so it introduces the use of clock uncertainty. Clock uncertainty is a time margin used to ensure that HLS creates a design with not too much logic attached to each clock cycle. Thus, the timing requirements can be met even if not ideal placement and routing take place. Figure 2.13 depicts this concept.
- Specifying the Reset: An important decision for hardware designers is the selection of the reset behaviour. The reset behaviour should not be confused with the initialization behaviour. The latter refers to the initial values of variables or arrays and it is automatically implemented by the HLS. However, there is no way that a variable can return to its initial state during the execution of the design.



Figure 2.13: Clock period and margin [14]

This behaviour can be achieved with a reset. HLS allows four settings in the reset behaviour: none, control, state and all. None adds no reset to the design. Control is the default setting and ensures all control registers - those used in state machines and to generate I/O protocol signals - are reset. State adds a reset signal to the control registers plus any registers or memories derived from static and global variables in C code. Last, all is used to add a reset to all registers and memories in the design.

2.3.2.5 Design Optimization

The best way to exploit Vivado (R) HLS' capabilities is first to synthesize the design without any user defined optimization directives. The next step is to review the reports produced by the HLS tool and to apply different combinations of optimization directives in order to create an architecture with the best performance. This last step is called design space exploration. The following sections describe the optimizations that can be applied by Vivado (R) HLS.

The first category of optimizations is related to the design's throughput. ⁵ The most important optimization directive of this category is task pipelining. As demonstrated in section 2.3.2.3, pipeline allows operations to be executed in parallel and minimizes the initiation interval of a design. Pipeline optimization can be applied to loops and functions. There is a slight difference in the implemented hardware beetween these two cases as shown at figure 2.15. A pipelined function will continuously read new inputs and write new outputs but a pipelined loop causes a "bubble" in the data stream because it must complete all the iterations before it starts a new execution. Attention should be paid when a pipeline directive is applied to a loop with other loops inside its body. HLS will unroll (create hardware for each iteration of the loop) all the enclosed loops and may significantly increase the area impact of the design. In addition, if the inner loops have varible bounds that cannot be unrolled, the pipeline will fail. Loop unrolling though, can

⁵how many clock cycles needed for the design to accept new input data.

UNROLL optimization directive. As depicted at figure 2.14, a loop can remain rolled, can be partially unrolled or can be completely unrolled. At figure 2.14, the rolled loop will execute each iteration in seperate clock cycles. In this case, only one multiplier is needed and each block RAM can be a single port. In the partially unrolled loop, the initianion interval is halved (2 cycles) but 2 multipliers along with dual port block RAMs are needed. Last, if this loop is completely unrolled, the execution will be completed in one clock cycle, given that all the reads can be performed in parallel and that 4 multipliers are available. Another way of increasing the throughput of a design is the partitioning of arrays. By default, the maximum simultaneous operations for an array are 2 (dual port block RAM) and this can be a limiting factor to the performance of memory intensive algorithms. Therefore Vivado (R) HLS can be instructed to partition an array into multiple smaller arrays that support 2 operations individually. The different ways of partitioning an array are (figure 2.16):

- block: The original array is split into equally sized blocks of succesive elements.
- **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.
- **complete**: Every element of the original array is mapped to registers.

In some cases throughput could be increased without array partitioning but simply by declaring an array as FIFO. This optimization is very convenient when the elements of an array are sequentially accessed. Loop pipelining can be prevented by loop carry dependencies. Vivado (R) HLS' dependence analysis of complex designs can be too conservative and can create false dependencies. Using the DEPENENCE optimization directive, a hardware designer can help the HLS tool to overcome such false dependencies and create a more efficient implementation. As shown in section 2.3.2.3, dataflow optimization can lead to parallel execution of tasks and can improve significantly the overall throughput.

The second category of optimizations targets the minimization of latency ⁶. A designer can use latency constraints through the LATENCY directive and Vivado® HLS will try to create a hardware implementation, which meets the required latency specifications. If it cannot satisfy the constraints posed it relaxes the limits and tries to achieve the best possible result. Another way to increase latency is to merge sequential loops. Every loop creates a different state in the Finite State Machine (FSM) of the design. Hence, multiple sequential loops create unnecessary clock cycles in the design. As shown at figure 2.17 if the two loops are merged into one, the execution will drop from 11 to 6 clock cycles. In this example, just an **else** statement inside the add loop will solve the problem but most of the times it will not be that obvious. Consequently, Vivado® HLS can automatically merge loops via the LOOP_MERGE optimization directive.

The last group of optimizations tend to minimize the area impact of the design implementation. First, a hardware designer can use custom data types and bit-widths. If a

⁶the number of cycles needed for the operation to be completed.

```
void top(...) {
    ...
    for_mult:for (i=3;i>=0;i--) {
        a[i] = b[i] * c[i];
     }
    ...
}
```



Figure 2.14: Loop Unrolling details [14]



Figure 2.15: Function and Loop pipelining behaviour [14]

variable needs only 12 bits and is declared as 32 bit, it will result in larger and slower 32 bit operators, which potentially leads to large initiation intervals. Thus, a good practice would be to use the appropriate precision for every operation and to pay special attention to complex mathematical operations like multiplications, divisions, or modulus by using



Figure 2.16: Array partitioning[14]



Figure 2.17: Loop merging example [14]

the varibale width required by the application. Another optimization that Vivado (R) HLS supports is the function inlining. Through HLS INLINE directive the tool removes the function hierarchy and replaces a function call in the C specification with its code. This may decrease the area occupied by the design by allowing the components within the inlined function to be better shared and optimized with the logic of the calling function. Vivado (R) HLS automatically inlines small functions but for bigger ones this has to be manually set. Area reduction can be also achieved by mapping many small arrays into one large array. An array in the FPGA is mapped to block-RAM as stated in section 2.3.2.4. In some cases, the arrays are so small that do not use the full 18K bit block. This can lead to larger block-RAM allocations than those needed. The HLS user can solve this problem with the ARRAY_MAP directive which mapps multiple arrays into a larger one. Two ways are supported:

• Horizontal Mapping: a new array is created by concatenating the original arrays.
Physically, this creates a large array with more elements.

• Vertical Mapping: the words of the original arrays are concatenated. Physically, this is implemented by one array with larger word width.

A hardware designer, via Vivado[®] HLS can manually specify the number of operators used for specific operations that can be beneficial for the area impact of the design (AL-LOCATION directive). It is also possible for the designer to explicitly instruct the HLS tool regarding the cores that it should use for each operation (RESOURCE directive).

2.3.2.6 Writing Hardware Efficient Code

Apart from either the automatic or the custom optimizations that can be applied to the design with Vivado® HLS, there are some important steps in the development that can lead to a more efficient implementation.

First, the input data reads and the output data writes should be as few as possible. The input and output ports support only a limited number of parallel operations and are usually the performance bottlenecks of the designs. The best way to overcome this problem is to read data once and use caches if the data must be re-used.

Array accesses should be also as few as possible. Arrays are implemented by block-RAMsi, which can support, in best case, only two parallel operations⁷. This creates an important bottleneck in the overall performance. Besides the array optimization techniques that were mentioned in section 2.3.2.5 (array partitioning and array merging), algorithms should be also modified. Small localized caches can be used for storing results such as accumulations or group of data needed for every operation.

Another good practice, is to create conditional branching inside pipelined tasks rather than conditionally execute tasks. Conditionals are just separate paths in the pipeline. The data can flow rapidly from the pipelined task to the branch that is executed and this results in better performing implementation.

Chapter 3

Dynamic Memory Management Framework for HLS

3.1 Introduction

As mentioned in chapter 2, Vivado (R) HLS cannot synthesize designs that require dynamic allocation and deallocation of memory. It only permits static memory allocation. Thus, in the entire execution window of an application the maximum memory that is needed should be allocated. While static allocation works fine for a limited number of accelerators it does not scale to many-accelerator design paradigm. At figure 3.1, it is clearly presented that when the dataset workload is increased, the available memory resources are the first to be exhausted. In addition, as the number of the Kmeans accelerators is increased the memory resources are exhausted long before another resource is in shortage. So it is safe to declare that the BRAM memory is the main limiting factor of higher accelerator densities and large datasets.

In order to alleviate this "resource under-utilization" problem, a dynamic memory manager (DMM) for HLS is created. It targets at many-accelerator FPGA based systems and enables each accelerator to dynamically adapt its allocated memory according to the runtime requirements of memory [2]. This dynamic memory manager uses the First Fit algorithm for allocation and deallocation of memory but in this diploma thesis it will be extended with the implementation of Next Fit (section 3.3.1) and Best Fit (section 3.3.2) algorithms.

3.2 The DMM-HLS Framework

In this section the DMM-HLS framework is presented. First the DMM-HLS flow that is integrated in Vivado-HLS and the architectural implications on the organization of many-accelerator systems are presented. Then, the implemented DMM mechanisms are described [2].



Figure 3.1: Memory Bottleneck of Kmeans clustering algorithm^[2]

3.2.1 Dynamic Memory Management in Vivado-HLS

As shown in figure 3.2 the standard Vivado (R)-HLS design flow is slightly altered with the addition of the DMM-extension. The DMM-HLS extension works in the high level source code, and thus it keeps minimum implementation overhead to the designers. The first and only step for them is the source-to-source code modification. Through this step, the C source code of a design must be changed in order to include the DMM library. Also, every static allocation which is included in the initial design, must be transformed to a dynamic one using the function calls provided by the DMM-HLS API. It should be mentioned here that if a static allocation is not benefitted by its dynamic counterpart it should not be changed because the framework supports both the dynamic and the static allocation patterns. Then, the augmented code by the DMM-HLS function calls is synthesized into RTL implementation through the back end of Vivado (R) HLS tool. DMM-HLS framework extends the architectural template supported in Vivado (R) HLS by supporting many-accelerator systems and allowing the on-chip BRAM to be dynamically allocated among accelerators at runtime. This altered architecture is shown in figure 3.3 and consists of two subsystems: the processor subsystem and the accelerators subsystem. The processor subsystem consists of IPs¹ such as the processor (e.g. ARM A9 in Zynq) and it executes three code segments: the application control flow, the computationally nonintensive code kernels, and the code which should be executed on CPU due to the need of high accuracy or high FPGA implementation cost (e.g. floating point division). The accelerators subsystem holds the computationally intensive code kernels of the application. The accelerators are modeled in high level language (C) and synthesized using Vivado (R) HLS. As mentioned earlier the DMM-HLS framework supports the description of accelerators with both static and dynamically allocated data stored in the on-chip BRAMs[2].

¹Intelectual Property



Figure 3.2: Extended Vivado HLS flow with DMM implementation[2]



(b) Freelist organization (bitmap array)

Figure 3.3: Many accelerator FPGA based systems architecture enchanced with DMM-HLS framework[2]

DMM-HLS supports fully-parallel memory access paths, by grouping BRAM modules into unique memory banks, named heaps (figure 3.3). Every heap instantiation has its own DM allocator. The composition of heaps is described in a C header configuration file, enabling new heap instantiations to be easily described. Every heap-i is highly parameterizable on a number of design options, most important of which are the heap depth D_i^{H}

(the total number of unique addresses), the heap word length L_i^H (the number of bytes of every single word of heap), the allocation alignment A_i (the minimum number of bytes per allocation so that every new allocation starts from a unique address) and the metadata header size H_i (the number of bytes reserved on first address(es) of every allocation to store meta-data related to the allocation, e.g. allocation length). In a similar manner we define the free depth D_i^F and free word length L_i^F for the FreeBitMap memory.

The DM allocator for HLS supports arbitrary size allocation/deallocation, i.e. enables allocation on the same heap of any simple data type (integers, floats, doubles etc.), as well as more complex data types (structs of same or combined simple data types and 1-D arrays). A FreeBitMap structure, i.e. bit-map free-list holding information about the free space inside this heap, tracks the occupied memory space. FreeBitMap, is an array of registers, every bit of which maps to a singe byte of the Heap. The term "maps" refers to the allocation status (allocated or free) of this byte. Because one bit is related to one Heap's byte the freelist array will have $D_i^H \frac{words}{heap} * L_i^H \frac{bytes}{word} = D_i^H * L_i^H \frac{bytes}{heap} = D_i^H * L_i^H \frac{bits}{heap} * \frac{1}{8}$ i.e. 8 times smaller size.

Figure 3.3b depicts the organization of FreeBitMap, using an example of the allocation of 1 integer (4 bytes on x86). The first 2 words of the heap are used for metadata. More specifically the first word contains the size in bytes of the allocated data, and the second word contains the index of this allocation in the freelist bit map. Thus, six bytes are allocated and six bits of the freelist are marked (set to 1) [2].

3.2.2 DMM-HLS Overview

"The DM allocator can be employed through the DMM API, which is composed of two main function calls for memory allocation and deallocation. A similar to glibc malloc/free function call API is adopted:

- void* HlsMalloc(size_t size, uint heap_id)
- void HlsFree(void* ptr, uint heap_id)

where **size** is the requested allocation size in bytes, **heap_id** is the identification number of the heap on which allocation shall occur and ***ptr** is the pointer which shall be freed up. HLSMALLOC and HLSFREE are shown below"[2].

"The allocator implements a first fit algorithm to find a free continuous memory space according to requested bytes. This algorithm searches the memory and stops its execution at the first free block with the requested or larger size. A pointer to its start is returned. Since two extra bins to hold meta-data related to allocation are reserved, the first fit function returns the index of *FreeBitMap*, so that there are continuous (size + 2) free positions (equal to 0) at *FreeBitMap*[index] untill *FreeBitMap*[index + size + 2] (line 3). Next the allocator calculates the address of heap based on this index (line 4). However the heap word length L_i^H defines the smallest unit of memory access, i.e. each memory address specifies different number of bytes. Thus in order to allow any arbitrary allocation size we implemented a L_i^H -byte aligned memory address access. For that scope the mapped address *CandidateAddr* is padded with extra bytes in order to be aligned (line 5). Next the corresponding bits in *FreeBitMap* for the aligned address space reserved are marked (set to 1), (line 6) and finally the meta-data are written to the first two bins (lines 7,8). Finally the *AlignedAddr*+2 is returned in order to prevent the user to ovewrite the allocation metadata written in the first two bins" [2].

void^{*} HLSMALLOC(*size_t size*, *uint heap_id*)

- \triangleright Input: *size*: Requested allocation size in bytes
- \triangleright **Input**: *heap_id*: The identification number of heap
- ▷ **Output**: Generic pointer pointing to allocated memory address
- ▷ **Data**: * *HeapArray*,* *FreeBitMapArray*
- 1 $Heap_struct^* cur_heap \leftarrow HeapArray[heap_id]$
- 2 $FreeBitMap_struct^* cur_fbm \leftarrow FreeBitMapArray[heap_id]$
- 3 int $FrBMi \leftarrow FIRSTFIT(size + 2, cur_fbm)$
- 4 $int^* CandidateAddr \leftarrow MAPADDR(FrBMi, cur_heap)$
- 5 $[int^*AlignedAddr, int offset] \leftarrow PADDING(CandidateAddr)$
- 6 MARKBITMAP($FrBMi, size + offset + 2, cur_fbm$)

 \triangleright 1st Addr: alloc. size

- 7 $cur_heap[AlignedAddr] \leftarrow size$ \triangleright 2nd Addr: position in the freelist 8 $cur_heap[AlignedAddr + 1] \leftarrow FrBMi$

9 return (void*) AllignedAddr + 2

"The HLSFREE function has minimum execution overhead since the meta-data for every allocation store all necessary information for deallocation (lines 3, 4). It should be mentioned here that the pointer returned to the user by the allocator points after the metadata, therefore the ptr-2 and ptr-1 positions of the memory are read. The function exits with the deallocation of corresponding bits in *FreeBitMap* for the requested pointer (line 5). The UNMARKBITMAP function unmarks (sets to 0) the corresponding bits of the free list array" [2].

"Finally, the shared hardware interface of HLSMALLOC and HLSFREE limits execution parallelism when multiple accelerators request to allocate or free memory simultaneously, even if the accesses affect data across different heaps. To overcome this issue, function inlining of the HLSMALLOC and HLSFREE is applied ². This directive increases resource occupation, but also allows the unconstrained parallel access on heaps from many-accelerators" [2].

void HLSFREE(**void**^{*}*ptr*, *uint heap_id*)

- \triangleright **Input**: *ptr*: The pointer to free
- \triangleright **Input**: *heap_id*: The identification number of heap
- \triangleright **Data**: * *HeapArray*,* *FreeBitMapArray*
- 1 $Heap_struct^* cur_heap \leftarrow HeapArray[heap_id]$
- 2 $FreeBitMap_struct^* cur_fbm \leftarrow FreeBitMapArray[heap_id]i$ \triangleright 1st Addr: alloc. size
- 3 int size $\leftarrow cur_heap[ptr 2]$ \triangleright 2nd Addr: position in the freelist
- 4 int $FrBMi \leftarrow cur_heap[ptr-1]$
- 5 UNMARKBITMAP $(FrBMi, size + offset + 2, cur_fbm)$

3.2.3 DMM-HLS Details

In this section details about the C implementation of the DMM-HLS extension are presented.

3.2.3.1 Configuration File of DMM-HLS extension

Listing 3.1 shows the parameters that can be adjusted through the configuration file of the DMM-HLS library. More specifically the depth (line 3) of the heap, its type (line 31), and the size of each word (line 38) can be manually set. The last two constants are used in the if clause in line 162 which defines the type of the array which represents the memory heap. In a similar way the word length of the freelist bit-map array is controlled (lines 47, 189). The size of the freelist bit-map is calculated automatically as mentioned in section 3.2.1.

This configuration file provides a group of settings which control some aspects of the DM manager's behaviour. First, in line 65, the allocation/deallocation algorithm is chosen. Moreover, line 107, controls an optional check regarding the limits of a pointer that is going to be freed. If this flag is set then the DM manager checks the address of a given pointer and returns an error if it does not belong to the current heap. Furthermore, if the flag in line 113 is set, then every bit in the freelist is checked before freed (set to 0) and if it is 0 then the free operation fails and a FREE_FAIL_UNREGISTERED_BIT enum is returned by the DM-manager. Also, the CHCK_MALLOC_FAIL_DUE_FRAGMENTATION (line

²using the "pragma AP inline" preprocessor directive of Vivado-HLS

119) checks if the current allocation request has fallen due to fragmentation issues; i.e. free memory of the requested size exists but not in a continuous block. This flag should be used only when the HW_DEBUG_MEMLUV (line 16) is set to 1. Another important flag is the FREELIST_HANDLE_POLICY (line 60) which defines the behaviour of the freelist set and clear processes and affects significantly the execution time of the allocator. Three different values are supported: 0, 1, and 2. With 0 (slower process), one write occurs for every different bit that is set or cleared. With 1 (faster than 0), one write with a mask is occured for every freelist register that is affected but many cycles are lost in the step-by-step (one cycle per bit) calculation of the mask. Number 2 results in the most effective implementation. This implementation calculates the masks of the first and last freelist register and for every intermediate register the mask that is used is ~zero or zero directly because all the bits must marked/unmarked.

Another group of options is related with debugging mechanisms of software execution. More specifically DEBUG_MEMLUV (line 8), DEBUG_MEMLUV_LEVEL (line 13) and DEBUG_PRINT_STREAM (line 131) are used to control the debugging information that is printed and its output stream (stdout or logfile). Also, the percentage (0 or 100) of the memory contents that are going to be printed by the debugging interface (line 125) can be customized.

Last but not least, the return values for every possible outcome of the DM manager are defined (line 138)

```
/** The depth of internal buffer, i.e. the number of addresses
2
  */
  #define MEMLUV_DEPTH 4096
3
  /** Print debug messages of mem allocation functions, on SW only
5
     execution
   *
6
   */
7
  #define DEBUG_MEMLUV 0
8
ç
  /** The level of debugging. DEBUG_MEMLUV should be 1
10
  * 0->None, 1-> Light Debug, 2-> Medium Debug, 3-> Insane Debug
11
  */
12
13 #define DEBUG_MEMLUV_LEVEL 3
14
15 /** Instatiate a HW debug monitor. Introduces extra HW resources overhead */
  #define HW_DEBUG_MEMLUV 0
16
17
  /** Replace memluv allocator with system's allocator, i.e. glibc calloc */
18
 #define SIM_WITH_GLIBC_MALLOC 0
19
20
  /** The statistics/debug messages are supressed for timing measurements.
21
22
     NOTE:
            On hls this define has no impact since it is only applied to
     non-synthesizable code.
23
  *
24
   */
25 #define SIM_MINIMUM_OVERHEAD 0
```

```
26
27 /** The type of the internal memory core. Curently uint and float are
28 * supported, no way to support them simultaneously
  * 1-> uint 2-> float (deprecated in Vivado 2013.4)
29
30 */
31 #define CORE_TYPE 1
32
33 /** The width of internal memory core. i.e. the legnth of every line of
34 * internal memory. Supported 8,16,32,64. Whatever the internal size,
    the requested memory type may be of any length/type.
35
36 * LLVM take cares of data (un)packing, whatever the endianess.
37 */
38 #define CORE_WIDTH 32
39
_{40} /** The type of most internal pointers, i.e. base, addr, next etc. */
41 #define uint_t uint32_t
42 #define int_t int32_t
43
44 /** The width of free list, i.e. the legnth of every line of bitmap array.
45 * Supported 8,16,32,64
46 */
47 #define FREELIST_WIDTH 32
48
49 /** The policy of handling the freelist set/clear processes.
  * 0 -> Set/Clear per bit - One write to freelist for every bit
50
           (many cycles)
51
     1 -> Set/Clear per register (and per bit for non-alinged requests),
  *
52
           the mask is bit-computed (many cycles) but write happens as
53
           many times as the number of registers are affected,
54
           plus bit-process writes for non-aligned requests.
55
     2 -> Set/Clear per register. The mask is computed directly
56
           (in 6 cycles with seq. exec.) and we succeed the minimum number of
57
           writes to freelist - one per affected register.
58
59 */
60 #define FREELIST_HANDLE_POLICY 2
61
62 /** The fitting algorithm of MemLuv.
63 * 0 -> First fit algorithm.
64
  */
65 #define MEMLUV_FIT_ALGORITHM 0
66
67 /** The policy of first fit algorithm of MemLuv (applicable only when
      MEMLUV_FIT_ALGORITHM==0).
68
      0 -> Search bit-by-bit starting from position 0 to bitmap length.
69
           On first succesfull empty bin, the next search starts from the next
70
           bit. If it fails the next search starts from the next bit.
71
72
  * 1 -> Search bit-by-bit starting from position 0 to bitmap length.
           On first succesfull empty bin, the freelist registers are checked
73 *
           with masks. If it fails the next search starts from the next bit.
74
75 */
```

```
76 #define MEMLUV_FIRST_FIT_POLICY 1
77
  /** 1,2,4,8 bytes address alingment. Optimally should be log2(CORE_WIDTH) */
78
  #define ALIGNMENT 4
79
80
   /** The number of sizeof(CORE_UINT_T) bytes reserved at start of every
81
   * allocation to store metadata. 0,2 are supported so far for x86.
82
      0 -> No header is used
83
      2 -> Header of two fields is used: On any address, the 1st previous
84
      address holds the index in freelist and the 2nd previous address
85
      holds the size of allocation in bytes. i.e. :
86
            0x0 -> [size_in_bytes]
87
            0x1 -> [index_in_freelist]
88
    *
            0x2 -> [1st_data_stored]...
89
            Note: The number of bits of CORE_UINT_T dictates the maximum
90
            allocation/free of 2<sup>(sizeof(CORE_UINT_T)x8)</sup> bytes regardless
91
            the core size (MEMLUV_DEPTHxsizeof(CORE_UINT_T) bytes).
92
   *
   */
93
94 #define MEMLUV_ALLOC_HEADER 0
95
  /** 1 -> Inline all logic inside alloc/free. Results in a fast, yet
96
            resource hungry circuit. Can the target FPGA accept the overhead?
97
   * 0 -> Results to inlining of the main alloc/free wrappers, with limited
98
            logic inside them (computation intensive logic has been moved
99
            to shared functions/enities)
   *
100
   */
101
102 #define MEMLUV_HLS_INLINE 1
103
104 /** Force checking if requested free pointer is a valid address for
   * current hw heap
105
106
107 #define CHCK_FREE_OUT_OF_BOUNDS 0
108
109 /** Force checking of freelist value prior to marking zero bits. In case
   * a request free position is not '1', a FREE_FAIL_UNREGISTERED_BIT enum
110
   * variable is asserted on memluv_action_status_t
111
112
113 #define CHCK_FREELIST_BEFORE_FREE 0
114
115 /** Force checking if malloc fails due to fragmentation issues, although
   * there are available bytes for the requested size, but not continusoly.
116
117
   * HW_DEBUG_MEMLUV should be '1'
   */
118
119 #define CHCK_MALLOC_FAIL_DUE_FRAGMENTATION 0
120
121 /** The percentage of lines of memory core dump which will be printed on
122
   * MemluvDumpCore() and MemluvDumpFreeList() calls.
   * Valid integer values (no check is performed): 0-100
123
   */
124
125 #define MEMELUV.DUMP.SHOW.PERCENTAGE 100
```

```
126
   /** The output stream in which all messages are reported.
127
   * 0 -> stdout (standard output)
128
   * 1 -> file (./memluv_run_<pid>.log)
129
   */
130
  #define DEBUG_PRINT_STREAM 0
131
132
   /** Enabling the report of fragmentation.
133
   * HW (working with freelist sequential parsing.)
134
   */
135
136 #define DEBUG_REPORT_FRAGMENTATION 0
137
138
139 typedef enum {IDLE,
                MALLOC_SUCCESS,
140
                MALLOC_FAIL,
141
                MALLOC_FAIL_DUE_FRAGMENTATION,
142
                FREE_SUCCESS,
143
                FREE_FAIL,
144
                FREE_FAIL_UNREGISTERED_BIT,
145
                FREE_FAIL_POINTER_OUT_OF_BOUNDS} memluv_action_status_t;
146
  typedef enum {ALL, THIS} memluv_struct_handle_t;
147
148
   /*-----
                                            _____
149
   | Software IEC/IEEE floating-point types.
150
   *-----
                                                  ----*/
151
  typedef unsigned int float32;
152
  typedef unsigned long long float64;
153
154
  /** rounds up to the nearest multiple of ALIGNMENT
155
   * limitation: works only with power of 2 size
156
   */
157
158 #define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))
159
  #define SIZE_T_SIZE (ALIGN(sizeof(uint_t)))
160
161
162
  \#if CORE_TYPE = 1
    \#if CORE_WIDTH == 8
163
164
      #define CORE_UINT_T uint8_t
    #elif CORE_WIDTH == 16
165
166
      #define CORE_UINT_T uint16_t
167
    #elif CORE_WIDTH == 32
168
      #define CORE_UINT_T uint32_t
    #elif CORE_WIDTH == 64
169
      #define CORE_UINT_T uint64_t
170
    #else
171
172
      #define CORE_UINT_T UNSUPPORTED_CORE_WIDTH
    #endif
173
    #define var_type_t uint_t
174
175 #elif CORE_TYPE == 2
```

78

```
\#if CORE_WIDTH == 32
176
       #define CORE_UINT_T float32
177
       #define var_type_t float32
178
     #elif CORE_WIDTH = 64
179
       #define CORE_UINT_T float64
180
181
       #define var_type_t float64
    #else
182
       #define CORE_UINT_T UNSUPPORTED_CORE_WIDTH
183
     #endif
184
   #else
185
    #define CORE_UINT_T UNSUPPORTED_CORE_TYPE
186
  #endif
187
188
189 #if FREELIST_WIDTH == 8
    #define FREELIST_UINT_T uint8_t
190
     #define FREELIST_UINT_TD uint16_t
191
192 \# elif FREELIST_WIDTH == 16
    #define FREELIST_UINT_T uint16_t
193
     #define FREELIST_UINT_TD uint32_t
194
195 #elif FREELIST_WIDTH == 32
     #define FREELIST_UINT_T uint32_t
196
     #define FREELIST_UINT_TD uint64_t
197
198 #elif FREELIST_WIDTH == 64
    #define FREELIST_UINT_T uint64_t
199
     #define FREELIST_UINT_TD uint64_t
200
201 #else
    #define FREELIST_UINT_T UNSUPPORTED_FREELIST_WIDTH
202
203 #endif
204
205 /** log2(ALIGNMENT), dummy but avoid either gcc warning or extra registers
   * for look-up table, or log cordic module :)
206
   */
207
208 \# i f ALIGNMENT = 8
   #define LOG_ALIGN 3
209
210 \#elif ALIGNMENT == 4
   #define LOG_ALIGN 2
211
212 \# elif ALIGNMENT = 2
   #define LOG_ALIGN 1
213
214 \# elif ALIGNMENT == 1
   #define LOG_ALIGN 0
215
216 #else
217
    #define LOG_ALIGN UNSUPPORTED_ALIGNMENT
218 #endif
219
220 typedef struct MemLuvConf MemLuvConf;
221 typedef struct MemLuvStats MemLuvStats;
222 typedef struct MemLuvCore MemLuvCore;
223
224 /** A pointer to MemLuvConf0, declared in global.c. It is initialized
225 * through MemluvInit().
```

226 */ 227 MemLuvConf *CurMemLuvConf;

Listing 3.1: Configuration File^[2]

3.2.3.2 Global Structs Used by the allocator

The DM manager uses three C structs which keep important information organized. These are: struct MemLuvCore (listing 3.2), struct MemLuvStats (listing 3.3) and struct MemLuvConf (listing 3.4). The first two are tied with every heap that is implemented but the last one is a global struct that contains global configuration options for the DMM-HLS extension.

Struct MemLuvCore (listing 3.2) is the main struct that the allocator uses. It contains the memory heap (line 4) and the corresponding freelist array (line 10). This struct also contains the number of freelist addresses and bits (lines 7, 8) along with the core's size in bytes and depth (lines 12, 13). This four variables, inform the allocator in the runtime, about the limits of the freelist and the heap and come in handy in debug messages and iterations. alloc_rqst, free_rqst, tot_rqst (line 18) are some counters useful for statistics. More specifically, alloc_rqst counts the allocation requests that have been served so far, free_rqst counts the free requests and tot_rqst represents the total number requests that the DM manager has completed. stats (line 27) keeps track of the DM manager's statistics, its fields are shown in listing 3.3 and analyzed in the following paragraph. ***base** (line 14) is a pointer to the first address of the heap (base = &(MemLuvCore->core[0])). aligned and aligned_pad (line 16) stores the position (starting from 0) of the first byte and the first word respectively, of the free block that is found, after the alignment process has taken place. ibitpos and ibitpos_pad (line 19) are used for the same reason but as temporary values. log_align instructs how many times the byte position should be shifted to the right in order to give the word position. For example if a word length is $4 = 2^2$ bytes and the byte position is 2048 then the word position is 2048 >> 2 = 512. This struct also saves the status code of every request that is served (line 20). The possible return values are shown in listing 3.1 and line 138. Finally, some simulation specific and 24) fields are defined. More specifically, the name (line 23) and the file descriptor (line 22) of a file that is used by the allocator for debug information. Also a control variable (line 24) which is set to 0 after the creation of this file is defined.

```
//1 bit represents 1 distinct memory location
9
    FREELIST_UINT_T freelist [MEMLUV_DEPTH*sizeof(CORE_UINT_T)/(sizeof(
10
        FREELIST_UINT_T) *8) ];
11
    uint_t size;
12
13
     uint_t depth;
    var_type_t *base;
14
15
    uint_t aligned, aligned_pad;
16
    uint8_t log_align;
17
    uint16_t alloc_rqst , free_rqst , tot_rqst;
18
    uint_t ibitpos, ibitpos_pad;
19
    memluv_action_status_t action_status;
20
21 #ifndef __SYNTHESIS__
    FILE *fd;
22
    char filename [64];
23
    unsigned int file_init;
24
25 \# endif
26 #if HW_DEBUG_MEMLUV==1
    MemLuvStats stats;
27
28 #endif
29 };
```

Listing 3.2: Struct MemLuvCore[2]

In listing 3.3 the struct MemLuvStats is demonstrated. This is used to keep track of various statistics during the execution window of the DM manager. The fields in lines 4, 5, 6, 7, 9 are self descriptive. The manager also keeps track of the additional bytes that have been allocated due to alignment issues (line 8). Furthermore, there is a per request flag (line 3) which is set to 1 when the whole memory is searched and no free block of the size requested is found. The DM manager also knows during the runtime how many free bytes exist beetween the first byte of the memory and the last allocated byte (line 10).

```
struct MemLuvStats
  {
2
    uint_t max_address_reached;
3
    uint_t used_percentage;
4
    uint_t total_bytes_requested;
5
    uint_t total_bytes_allocated;
6
    uint_t total_addresses_allocated;
7
    uint_t total_fragmented_bytes;
8
    uint_t total_bytes_for_headers;
9
    uint_t total_request_fragmented_bytes;
10
11 \};
```

Listing 3.3: Struct MemLuvStats[2]

The last struct that is by the DMM-HLS extension is the struct MemLuvConf (listing 3.4). This contains global information that is not tied with one heap only. It contains the total number of heaps that are implemented by the DMM-HLS extension (line 3) and the

file stream that the debug messages are printed (line 4). For the simulation only it also keeps the pid^3 which is used to create a custom name ⁴ to the debug file.

```
struct MemLuvConf
1
 {
2
    uint8_t num_hw_heaps;
3
   FILE *dbg_fd;
4
   #ifndef __SYNTHESIS__
5
    pid_t pid;
6
   #endif
7
 };
8
```

Listing 3.4: Struct MemLuvConf[2]

3.2.3.3 Functions which Manage Freelist

In this section the interface which is responsible for the management of the freelist is demonstrated. It consists of 5 functions that will be analyzed in the following paragraphs.

The function MemluvSetBitFreelist() (listing 3.5) takes two arguments, the freelist array and an index (var_type_t k), and sets the k-th bit to 1. First the freelist register that the k-th bit belongs to is calculated (line 9) and then the bit's position in the register (line 10). In the next steps a new variable is created which sets to 1 the appropriate bit (lines 11, 12). Finally, this variable is OR-ed with the respective freelist register (line 13). In a similar way, the function MemluvClearBitFreelist() (listing 3.6) sets the k-th bit to zero, and the function MemluvTestBitFreelist() (listing 3.7) returns 1 if the k-th bit is 1 or 0 otherwise.

```
void MemluvSetBitFreelist( FREELIST_UINT_T A[ ], var_type_t k ) {
2 #ifdef __SYNTHESIS_
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
  #else
5
6 #pragma AP inline
  #endif
7
  #endif
8
     FREELIST_UINT_T i = (FREELIST_UINT_T)(k/(sizeof(FREELIST_UINT_T)*8));
9
     FREELIST_UINT_T pos = k\%(sizeof(FREELIST_UINT_T) *8);
10
     FREELIST_UINT_T flag = 1;
                                      // flag = 0000.....00001
11
     flag = (FREELIST_UINT_T)(flag << pos); // flag = 0000...010...000</pre>
                                                                              (
12
         shifted k positions)
     A[i] = A[i] | flag;
                                     // Set the bit at the k-th position in A[i]
13
14
  }
```

Listing 3.5: Function MemluvSetBitFreelist[2]

```
1 void MemluvClearBitFreelist( FREELIST_UINT_T A[ ], var_type_t k ) {
```

 $^{^{4}}memluv_run_ < pid > .log$

```
2 #ifdef __SYNTHESIS__
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 #else
6 #pragma AP inline
7 #endif
 #endif
8
     FREELIST_UINT_T i = (FREELIST_UINT_T)(k/(sizeof(FREELIST_UINT_T)*8));
9
     FREELIST_UINT_T pos = k\%(sizeof(FREELIST_UINT_T) * 8);
10
     FREELIST_UINT_T flag = 1;
                                       // flag = 0000.....00001
^{11}
     flag = (FREELIST_UINT_T)(flag << pos); // flag = 0000...010...000
                                                                            (
12
         shifted k positions)
     flag = (FREELIST_UINT_T)(~flag); // flag = 1111...101...111
13
     A[i] = A[i] \& flag; // RESET the bit at the k-th position in A[i]
14
15 }
```

Listing 3.6: Function MemluvClearBitFreelist[2]

```
1 uint8_t MemluvTestBitFreelist( FREELIST_UINT_T A[ ], var_type_t k ) {
2 #ifdef __SYNTHESIS__
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 #else
6 #pragma AP inline
 #endif
7
8 #endif
     FREELIST_UINT_T \ i = (FREELIST_UINT_T)(k/(sizeof(FREELIST_UINT_T)*8));
9
     FREELIST_UINT_T pos = k\%(sizeof(FREELIST_UINT_T)*8);
10
     FREELIST_UINT_T flag = 1;
                                     // flag = 0000.....00001
11
     flag = (FREELIST_UINT_T)(flag << pos); // flag = 0000...010...000
                                                                             (
12
         shifted k positions)
                               // Test the bit at the k-th position in A[i]
     if ( A[i] & flag )
13
        return 1;
14
15
     else
        return 0;
16
17 }
```

Listing 3.7: Function MemluvTestBitFreelist[2]

Function MemluvSetFreelist() (listing 3.8) is used to set to 1 all the freelist bits that represent the bytes of an allocation request. It has three arguments: the freelist array (FREELIST_UINT_T A[]), the position of the starting bit (var_type_t start_pos), and the position of the ending bit (var_type_t end_pos). As shown in the code (listing 3.8), this can be done with three different ways depending on FREELIST_HANDLE_POLICY. If it is 0 (line 9), then every bit is set to 1 separately. This leads to one write for every bit and is the slowest implementation. If it is 1 (line 14), then mask registers and logic operations on every freelist register are used. More specifically, every mask's bit is computed separately and saved in an mask variable (line 25). When this mask is ready it sets to 1 all the bits of the respective freelist register (line 28). If FREELIST_HANDLE_POLICY is 2 (line 33), mask

registers and logical operations are used again, but in a more efficient way. First, the starting and ending freelist registers are calculated (lines 39, 40). Then, the position of the starting register's first bit and the position of the last register's last bit are calculated (lines 41, 42). If the starting and ending register refer to the same register (line 45), then only one mask is calculated which is OR-ed with the respective freelist register is calculated (line 52) and OR-ed with the register (line 54), next, all the intermediate registers are directly OR-ed with a mask wich consists of ones only (line 57), and finally, the last's register mask is calculated (line 59) and OR-ed with the respective register (line 61). The last one, is the more efficient way of marking the bits because it uses write per register and because it calculates only two masks (starting, ending) and all the intermediate are directly applied. In a similar way is implemented the function MemluvClearFreelist() which is used in free requests and sets the freelist bits to 0 (listing 3.9).

```
void MemluvSetFreelist(FREELIST_UINT_T A[ ], var_type_t start_pos,
      var_type_t end_pos) {
2 #ifdef __SYNTHESIS__
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 # else
6 #pragma AP inline
7 #endif
8 #endif
  #if FREELIST_HANDLE_POLICY == 0
9
    var_type_t i;
10
    for (i = start_pos; i < = end_pos; i++) {
11
      MemluvSetBitFreelist(A, i);
12
    }
13
14 #elif FREELIST_HANDLE_POLICY == 1
15
    var_type_t i;
    FREELIST_UINT_T j=0;
16
    FREELIST_UINT_T pos;
17
    FREELIST_UINT_T flag;
18
    FREELIST_UINT_T reg_val=0;
19
    for (i = start_pos; i < = end_pos; i++) {
20
      j = (FREELIST_UINT_T)(i/(sizeof(FREELIST_UINT_T)*8));
21
      pos = i\%(sizeof(FREELIST_UINT_T) * 8);
22
                           // flag = 0000.....00001
      flag = 1;
23
      flag = (FREELIST_UINT_T)(flag \ll pos);
                                                // flag = 0000...010...000
                                                                                   (
24
          shifted i positions)
      reg_val = reg_val | flag;
                                      // Set the bit at the i-th position in A[j
25
          1
      Dprintf(3, "DEBUG: Inside MemluvSetFreelist: bin=%u, reg=%u, pos=%u,
26
          flag=%u, reg_val=%u\n", i, j, pos, flag, reg_val);
27
      if (pos=FREELIST_WIDTH-1) {
        A[j] = A[j] | reg_val; // Only (i%FREELIST_WIDTH) writes to freelist
28
29
         reg_val=0;
30
```

```
31
    A[j] = A[j] | reg_val; // plus one last write to freelist
32
_{33} #elif FREELIST_HANDLE_POLICY == 2
    var_type_t start_reg , end_reg;
34
    var_type_t start_pos_in_reg , end_pos_in_reg;
35
    var_type_t i;
36
    FREELIST_UINT_T mask, zero=0;
37
    FREELIST_UINT_TD flag=1; /* Double the size of FREELIST_UINT_T, since we
38
        need to calculate the number 2^FREELIST_WIDTH */
    start_reg = (FREELIST_UINT_T)(start_pos/(sizeof(FREELIST_UINT_T)*8));
39
    end_reg = (FREELIST_UINT_T)(end_pos/(sizeof(FREELIST_UINT_T)*8));
40
    start_pos_in_reg = start_pos%(sizeof(FREELIST_UINT_T)*8);
^{41}
    end_pos_in_reg = end_pos%(sizeof(FREELIST_UINT_T)*8);
42
    Dprintf(3, "DEBUG: Inside MemluvSetFreelist: start_pos=%u, end_pos=%u,
43
        start_reg=%u, end_reg=%u, start_pos_in_reg=%u, end_pos_in_reg=%u\n",
        start_pos, end_pos, start_reg, end_reg, start_pos_in_reg,
        end_pos_in_reg);
    switch ( end_reg - start_reg ) {
44
    case 0:
45
      // power of 2 with shift: i.e. 2^{30} == 1 << 30 == 1073741824
46
      mask = (FREELIST_UINT_T)(((flag <<(end_pos_in_reg+1))-1) - ((flag <<
47
          \operatorname{start_pos_in_reg}(-1);
      Dprintf(3, "DEBUG: mask=%u\n", mask);
48
      A[start_reg] = A[start_reg] | mask;
49
      break;
50
    default:
51
      mask = (FREELIST_UINT_T)((~zero) - ((flag << start_pos_in_reg) - 1));
52
      Dprintf(3, "DEBUG: start mask=%u\n", mask);
53
54
      A[start_reg] = A[start_reg] | mask;
      for (i=start_reg+1; i<end_reg; i++) {</pre>
55
        Dprintf(3, "DEBUG: Marking entire reg %u\n", i);
56
        A[i] = (FREELIST_UINT_T)(~zero);
57
      }
58
      mask = (FREELIST_UINT_T) (((flag <<(end_pos_in_reg+1))-1));
59
      Dprintf(3, "DEBUG: end mask=%u\n", mask);
60
      A[end_reg] = A[end_reg] | mask;
61
62
      break;
    }
63
64 \# else
    Dprintf(0, "Unsupported FREELIST_HANDLE_POLICY on stc/memluv.h. Aborting
65
        \ldots \langle n \rangle n;
66
    exit(-1);
67 #endif
68 }
```

Listing 3.8: Function MemluvSetFreelist [2]

```
void MemluvClearFreelist(MemLuvCore* CurMemLuvCore, var_type_t start_pos,
var_type_t end_pos) {
#ifdef __SYNTHESIS__
#if MEMLUV_HLS_INLINE == 0
```

```
4 #pragma AP inline off
  #else
\mathbf{5}
6 #pragma AP inline
7 #endif
8 #endif
9 #if FREELIST_HANDLE_POLICY == 0
    var_type_t i;
10
    for (i=start_pos; i <= end_pos; i++) {
11
12 #if CHCK_FREELIST_BEFORE_FREE==1
      if (MemluvTestBitFreelist(CurMemLuvCore->freelist, i) == 0) {
13
        CurMemLuvCore->action_status = FREE_FAIL_UNREGISTERED_BIT;
14
         Dfprintf(2, CurMemLuvConf->dbg_fd, "\nDEBUG: Useless free request on
15
            freelist position %u", i);
      }
16
17 #endif
      MemluvClearBitFreelist (CurMemLuvCore->freelist , i);
18
19
    ł
20 \# elif FREELIST_HANDLE_POLICY == 1
21
    var_type_t i;
    \label{eq:FREELIST_UINT_T} FREELIST_UINT_T \quad j = 0;
22
    FREELIST_UINT_T pos;
23
    FREELIST_UINT_T flag;
24
    FREELIST_UINT_T reg_val=(FREELIST_UINT_T)(^{\circ}0);
25
    for (i = start_pos; i < = end_pos; i++) {
26
      j = (FREELIST_UINT_T)(i/(sizeof(FREELIST_UINT_T)*8));
27
      pos = i\%(sizeof(FREELIST_UINT_T)*8);
28
      flag = 1;
                            // flag = 0000.....00001
29
      flag = (FREELIST_UINT_T)(flag << pos); // flag = 0000...010...000
                                                                                   (
30
          shifted i positions)
      flag = (FREELIST_UINT_T)(~flag);
                                            // flag = 1111...101...111
31
      reg_val = reg_val \& flag;
                                     // Set the bit at the i-th position in A[j
32
          Т
      Dprintf(3, "DEBUG: Inside MemluvSetFreelist: bin=%u, reg=%u, pos=%u,
33
          flag=%u, reg_val=%u\n", i, j, pos, flag, reg_val);
      if (pos==FREELIST_WIDTH-1) {
34
        CurMemLuvCore->freelist [j] = CurMemLuvCore->freelist [j] & reg_val; //
35
            Only (i%FREELIST_WIDTH) writes to freelist
        reg_val = (FREELIST_UINT_T)(~0);
36
37
      }
38
    }
39
    CurMemLuvCore->freelist [j] = CurMemLuvCore->freelist [j] & reg_val; // plus
         one last write to freelist
40 #elif FREELIST_HANDLE_POLICY == 2
    var_type_t start_reg , end_reg;
41
42
    var_type_t start_pos_in_reg , end_pos_in_reg;
    var_type_t i;
43
44
    FREELIST_UINT_T mask, zero=0;
    FREELIST_UINT_TD flag=1; /* Double the size of FREELIST_UINT_T, since we
45
        need to calculate the number 2^FREELIST_WIDTH */
    start_reg = (FREELIST_UINT_T)(start_pos/(sizeof(FREELIST_UINT_T)*8));
46
```

```
end_reg = (FREELIST_UINT_T)(end_pos/(sizeof(FREELIST_UINT_T)*8));
47
     start_pos_in_reg = start_pos%(sizeof(FREELIST_UINT_T)*8);
48
     end_pos_in_reg = end_pos%(sizeof(FREELIST_UINT_T)*8);
49
     Dprintf(3, "DEBUG: Inside MemluvSetFreelist: start_pos=%u, end_pos=%u,
50
        start_reg=%u, end_reg=%u, start_pos_in_reg=%u, end_pos_in_reg=%u\n",
        start_pos, end_pos, start_reg, end_reg, start_pos_in_reg,
        end_pos_in_reg);
    switch ( end_reg - start_reg ) {
51
    case 0:
52
      // power of 2 with shift: i.e. 2^30 == 1 << 30 == 1073741824
53
      mask = (FREELIST_UINT_T)(~(((flag <<(end_pos_in_reg+1))-1) - ((flag <<
54
          \operatorname{start_pos_in_reg}(-1)));
      Dprintf(3, "DEBUG: mask=%u\n", mask);
55
      CurMemLuvCore->freelist [start_reg] = CurMemLuvCore->freelist [start_reg]
56
          & mask;
      break;
57
    default:
58
      mask = (FREELIST_UINT_T)(~(((~zero)) - ((flag << start_pos_in_reg)-1)));
59
      Dprintf(3, "DEBUG: start mask=%u\n", mask);
60
      CurMemLuvCore->freelist[start_reg] = CurMemLuvCore->freelist[start_reg]
61
          & mask;
      for (\,i{=}start\_reg{+}1;\ i{<}end\_reg\,;\ i{+}+) {
62
         Dprintf(3, "DEBUG: Erasing entire reg %u\n", i);
63
         CurMemLuvCore->freelist[i] = zero;
64
65
      }
      mask = (FREELIST_UINT_T) (~(((flag <<(end_pos_in_reg+1))-1)));
66
      Dprintf(3, "DEBUG: end mask=%u\n", mask);
67
      CurMemLuvCore->freelist[end_reg] = CurMemLuvCore->freelist[end_reg] &
68
          mask;
      break;
69
    }
70
71 #else
    Dprintf(0, "Unsupported FREELIST_HANDLE_POLICY on stc/memluv.h. Aborting
72
        \ldots \n \ );
    \operatorname{exit}(-1);
73
74 #endif
75
```

Listing 3.9: Function MemluvClearFreelist()[2]

3.2.3.4 First Fit Allocation and Deallocation Mechanisms

In this section the allocation/deallocation interface of the DMM-HLS extension is analyzed. This interface is divided in two levels. In the first level are the functions CurMemluvAlloc() for allocation, and CurMemluvFree() for deallocation (listings 3.10 and 3.12 respectively). This level is responsible for every side task that has to be made for these requests such as finding the suitable padding, updating global variables or storing allocation/deallocation metadata in the heap. In the second level, are the functions MemluvFit() with MemluvFirstFit(), and CurMemluvFreeBody() (listing 3.9) for allocation and deallocation respectively. This level interacts with the freelist array, and it handles the search of free memory and the deallocation of allocated memory.

In listing 3.10 the function CurMemluvAlloc() is shown. Its arguments are: the structure (CurMemLuvCore) that represents the heap for this specific allocation and the number of bytes (nbytes) that are requested. First, the allocator updates the statistics counters (lines 8, 9). Then, the first byte of the available memory space is found (line 15). If the status code is MALLOC_SUCCESS, then appropriate padding creates the aligned address (lines 26, 27, 28). On the contrary, when malloc fails the first address of the memory heap is returned (line 35). In both cases, if HW_DEBUG_MEMLUV is set to 1, then statistics are updated (lines 30, 37). If header is used, then the metadata are writtern to the core (lines 53, 57) and the address to be returned is incremented by two words (line 43). Finally the function returns a pointer to the relevant free memory space (line 66).

```
void* CurMemluvAlloc(MemLuvCore *CurMemLuvCore, uint_t nbytes) {
  #ifdef __SYNTHESIS__
2
  #pragma AP inline
3
  #endif
4
       var_type_t *addr=NULL, *returned;
\mathbf{5}
       uint_t cand_base;
6
       uint_t allocated = 0;
       CurMemLuvCore \rightarrow alloc_rqst++;
8
       CurMemLuvCore->tot_rqst++;
9
10
       /* 0x1: Find a valid position on freelist of continuous nbytes,
11
                given a fit algorithm. Bytes for headers are computed
12
                internally in these algorithms.
        *
13
14
        */
       cand_base = MemluvFit(CurMemLuvCore, nbytes);
15
16
  #if HW_DEBUG_MEMLUV==1
17
       allocated=CurMemLuvCore->ibitpos_pad+1-cand_base;
18
  #endif
19
20
       if (CurMemLuvCore->action_status == MALLOC_SUCCESS) {
21
22
           /* 0x2: If malloc succeeded, make padding of address
23
                    according to alingment and do some statistics
24
            */
25
           CurMemLuvCore \rightarrow aligned = ReqPadding(cand_base);
26
           CurMemLuvCore->aligned_pad = CurMemLuvCore->aligned>>(CurMemLuvCore
27
               \rightarrow \log_{-} \operatorname{align});
           addr = (((var_type_t*)CurMemLuvCore->base) + CurMemLuvCore->
28
               aligned_pad);
  #if HW_DEBUG_MEMLUV==1
29
30
           MemluvAllocUpdateStas(CurMemLuvCore, nbytes, allocated);
31 #endif
32
       }
       else {
33
```

```
// 0x3: If malloc failed, return the address of base heap
34
           addr = CurMemLuvCore->base;
35
  #if HW_DEBUG_MEMLUV==1
36
           MemluvAllocUpdateStas(CurMemLuvCore, nbytes, allocated);
37
  #endif
38
      }
39
40
      // 0x4: The address is increased by MEMLUV_ALLOC_HEADER
41
               positions if we use header
      11
42
      returned = addr + MEMLUV_ALLOC_HEADER;
43
  #if MEMLUV_ALLOC_HEADER > 0
44
45
      /* 0x5: If we use header, write to the first two addresses
46
               the length of the requested allocation and the
47
               index of the requested allocation on freelist bitmap,
48
               respectively
49
        */
50
51
      //The first word stores the length in bytes of the respective allocation
52
      CurMemLuvCore->core [(cand_base>>CurMemLuvCore->log_align)] = (
53
          CORE_UINT_T) (nbytes);
54
      //The second word stores the index of the requested allocation
55
      //(first byte) on freelist bitmap
56
      CurMemLuvCore->core [(cand_base>>CurMemLuvCore->log_align)+1] = (
57
          CORE_UINT_T) cand_base;
  #endif
58
      DumpStatisticsToFile (CurMemLuvCore
59
  #if USE_MEMLUV==1 && HW_DEBUG_MEMLUV==1
60
61
                             .1
                             , allocated
62
63 #endif
                            );
64
      // Ox6: Return the address in the form of a generic (void*) pointer.
65
      return ((void*)returned);
66
67
  }
```

Listing 3.10: Allocation Wrapper[2]

The algorithm used by the DMM-HLS extension, for allocation and deallocation, is First Fit. First Fit, searches the heap's bytes sequentially, and when a free block of bytes, that is larger or equal than the requested, is found, the algorithm stops and returns a pointer to its first byte. The implementation of the First Fit algorithm is shown in listing 3.11. Depending on the MEMLUV_FIRST_FIT_POLICY option the DM manager supports two ways of finding a free block.

In the first way, the freelist bits are checked one by one. More specifically, the respective loop (line 17) starts from the first bit and ends when the last bit is checked. In the loop body, if a free bit is found (line 20) and it is the first after one or more allocated (line 24), then its position is saved and a counter counts the free bits after (line 30). In contrast,

if the current bit is reserved the counter and the logic values are initialized again (lines 34, 35, 36, 37). In every step of the loop, the counter is checked and if it has reached the requested value (line 41), the algorithm adds the appropriate padding to the starting byte (line 45), marks the freelist bits (line 49), updates the status code (line 50) and returns the position of the first free byte ⁵. If the loop ends without finding any free block of the requested size then the status code is set to MALLOC_FAIL (line 62), and 0 is returned.

The second way of finding a free memory block uses registers which examine the bits, through logical operations. This way is more complex, but more efficient. The coresponding loop (line 82), has the same limits. In every step, though, the possible ending bit is calculated (line 91), along with the starting and ending freelist registers (lines 93, 94) and the starting and ending positions in the starting and ending registers respectively (lines 95, 96). Also, the counter of the free bytes is initialized to zero (line 97). Next, two cases are taken into consideration. In the first case the starting register of the freelist is the same with the ending register (line 113). Thus, only one mask has to be calculated (line 114). The following example demonstrates the calculation of this mask. If the starting position is 5 (starting from 0) and the ending position is 28 (24 bytes allocation) the mask will be calculated as follows,

$$mask = ((flag << (end_pos + 1)) - 1) - ((flag << (start_pos)) - 1)$$
(3.1)

$$= ((1 << 29) - 1) - ((1 << 5) - 1)$$
(3.2)

$$= 0001111111111111111111111100000 \tag{3.5}$$

and the candidate bits will be set to 1 and the rest to zero. Next, this mask is AND-ed with the reverted register (line 115) and if the result is equal with the mask then the counter's value is updated, in order for the check in line 177 to succeed. The example's AND operation will succeed only if the inverted register has ones to the bits in positions 28 until 5, i.e. only if the register has zeros in bits in positions 28 until 5. Every check in this function is done through this process. The next case is when the starting bit and the ending bit belong to different registers (line 132). In the beginning of this case, the first mask is calculated (line 134), and used for the register check (line 140). If this check is successful a possible free block is found. Thus, its free bytes are added to the counter (line 141). Then, every intermediate register is checked (line 151) and added to the counter (line 153). Finally, if all the intermediate registers were free, the algorithm checks the last register (lines 166, 167) and updates the counter (line 168). If any of these register checks fails, then the algorithm continues is search from the next bit. The remaining part of this process is identical with that of the previous paragraph.

uint_t MemluvFirstFit(MemLuvCore *CurMemLuvCore, uint_t nbytes) {
#ifdef __SYNTHESIS__

 $^{^{5}}$ For the word position this value must be shifted LOG_ALIGN times to the right

```
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 #else
6 #pragma AP inline
7 #endif
8 #endif
9 #if MEMLUV_FIT_ALGORITHM == 0
10 #if MEMLUV_FIRST_FIT_POLICY==0
    uint_t free_bins=0, cand_base=0;
11
    uint8_t first_visit=0;
12
    CurMemLuvCore \rightarrow ibitpos = 1;
13
14
    // 0x1: Go to a while loop to check every sinlge bit if it is
15
    11
             candidate for allocation
16
    while (CurMemLuvCore->ibitpos<=CurMemLuvCore->freelist_total_bits) {
17
18
      // 0x2: Check this bit
19
      if (MemluvTestBitFreelist(CurMemLuvCore->freelist, CurMemLuvCore->
20
          ibitpos -1) == 0) \{
21
         // 0x3: If this bit is the first free found on freelist, start count
22
         11
                free positions
23
        if (first_visit == 0) {
24
           cand_base = CurMemLuvCore \rightarrow ibitpos -1;
25
           first_visit = 1 is almost identical to First Fit the check that
26
27 has been added in the end of the \verb!while! loop
28 (;
        }
29
30
        free_bins++;
      }
31
      // 0x4: If current bit is reserved, cancel the count of free positions
32
      11
               up to now
33
      else {
34
        free_bins = 0;
35
         cand_base = 0;
36
         first_visit = 0;
37
38
      }
      // 0x5: Check if free bins found up to now are equivalent to the bins
39
40
      11
               requested
      if (free_bins == nbytes + MEMLUV_ALLOCHEADER*sizeof(CORE_UINT_T)) {
^{41}
42
43
         // 0x6: If we found requested bins, make padding according to
44
         11
                 alingment
         CurMemLuvCore \rightarrow ibitpos_pad = ReqPadding(CurMemLuvCore \rightarrow ibitpos) -1;
45
46
         // Ox7: Mark requested free bins as reserved in freelist, malloc
47
48
         11
                 succceeded
         MemluvSetFreelist (CurMemLuvCore->freelist, cand_base, CurMemLuvCore->
49
             ibitpos_pad);
         CurMemLuvCore->action_status = MALLOC_SUCCESS;
50
```

```
51
         // 0x8: Return the position of first free bin, to allocator
52
         return cand_base;
53
      }
54
      // 0x9: Check next bin
55
      CurMemLuvCore \rightarrow ibitpos++;
56
57
    }
    /* Oxa: If never returned on step Ox8, and freelist was scanned
58
             entirely, there are no continuous free bins for current
59
     *
             request, thus allocation failed.
     *
60
     */
61
    CurMemLuvCore->action_status = MALLOC_FAIL;
62
    CurMemLuvCore \rightarrow ibitpos_pad = 0;
63
    return (0);
64
65
66 #elif MEMLUV_FIRST_FIT_POLICY==1
    var_type_t free_bins=0, cand_bin_start=0, cand_bin_end, i;
67
    var_type_t start_reg , end_reg;
68
    var_type_t start_pos_in_reg , end_pos_in_reg;
69
    \label{eq:FREELIST_UINT_T_mask, zero=0;} FREELIST_UINT_T mask, zero=0;
70
71
    /* NOTE: Double the size of FREELIST_UINT_T,
72
        since we need to calculate the number 2°FREELIST_WIDTH
     *
73
     */
74
    FREELIST_UINT_TD flag = 1;
75
76
    uint8_t search_footer=0;
77
    CurMemLuvCore \rightarrow ibit pos = 1;
78
79
    // Ox1: Go to a while loop to check every sinlge bit if it is candidate
80
             for allocation
81
    while (CurMemLuvCore->ibitpos<=CurMemLuvCore->freelist_total_bits) {
82
83
      /* 0x2: Given this bit, check: i) the last bit based on current
84
                                            request (bytes + header)
85
                                         ii) the registers that start and
86
87
                                               last bit are indexed
                                         iii) the position in these registers
88
89
        */
       cand_bin_start=CurMemLuvCore->ibitpos -1;
90
       cand_bin_end = (var_type_t) (cand_bin_start+nbytes+MEMLUV_ALLOCHEADER*
91
           sizeof(CORE_UINT_T) - 1);
92
       start_reg = (FREELIST_UINT_T)(cand_bin_start/(sizeof(FREELIST_UINT_T)*8)
93
           );
       end_reg = (FREELIST_UINT_T)(cand_bin_end/(sizeof(FREELIST_UINT_T)*8));
94
95
       start_pos_in_reg = cand_bin_start%(sizeof(FREELIST_UINT_T)*8);
       end_pos_in_reg = cand_bin_end%(sizeof(FREELIST_UINT_T) *8);
96
       free_bins=0;
97
98
```

```
99
       // 0x3: Check if requested continuous bins from cand_bin_start to
       11
                cand_bin_end are free
100
       switch ( end_reg - start_reg ) {
101
102
         /* 0x4: If the request affects bits of the same register,
103
          *
                  e.g. check bits 5-13
104
105
                  Register 0: MSB=15 [xxxx xxxx xxxx xxxx] LSB=0
106
107
                  we create a mask with zeros on the space
108
                  MSB[cand_bin_end,cand_bin_start]LSB and we check
109
                  for free positions by ANDIng this mask with the inverted
110
                  freelist register (since we check for zeros)
111
          */
112
         case 0:
113
           mask = (FREELIST_UINT_T)((((flag <<(end_pos_in_reg+1))-1) - ((flag <<
114
                \operatorname{start_pos_in_reg}(-1)));
           if ((~CurMemLuvCore->freelist[start_reg] & mask) == mask) {
115
              CurMemLuvCore->ibitpos=cand_bin_end+1;
116
              free_bins=(var_type_t)(nbytes + MEMLUV_ALLOC_HEADER*sizeof(
117
                  CORE_UINT_T));
            }
118
            break;
119
120
         /* 0x5: If the request affects bits of different registers,
121
                  e.g. check bits 5-42
          *
122
123
             Register 0: MSB=15 [xxxx xxxx xxxx] LSB=0 (bits 5-15)
124
125
             Register 1: MSB=31 [xxxx xxxx xxxx] LSB=16 (ii: bits 16-31)
          *
126
127
             Register 2: MSB=47 [xxxx xxxx xxxx] LSB=32 (iii:bits 32-42)
128
129
          *
              We do the job in three steps: see i, ii, iii below
130
          */
131
         default:
132
133
         search_footer = 0;
         mask = (FREELIST_UINT_T)(((\ \ zero\)) - ((flag << start_pos_in_reg)-1));
134
135
         /* 0x5 i) Create a mask with zeros on the space [MSB,cand_bin_start]
136
                     and check for free positions by ANDIng this mask with the
137
138
          *
                     inverted freelist register (since we check for zeros)
139
          */
         if ((~CurMemLuvCore->freelist[start_reg] & mask) == mask) {
140
            free_bins = FREELIST_WIDTH - start_pos_in_reg;
141
            \operatorname{search}_{-}\operatorname{footer}=1;
142
143
            /* continue to footer in case the request is not big enough to
            * search for intermediate registers
144
             */
145
146
```

```
147
            /* 0x5 ii) Check if intermediate registers (start_reg+1,end_reg) are
                         zero (free). This step may be skipped if the request
148
                         affects bits of continuous registers (i.e. 23,24)
149
              *
             */
150
            for (i=start_reg+1; i<end_reg; i++) {</pre>
151
                   (CurMemLuvCore->freelist[i] == zero) {
              if
152
                 free_bins + = FREELIST_WIDTH;
153
                 \operatorname{search}_{-}\operatorname{footer}=1;
154
155
              }
               else {
156
                 search_footer = 0;
157
                 break;
158
159
              }
            }
160
            /* 0x5 iii) Create a mask with zeros on the space [cand_bin_end,LSB]
161
                          and check for free positions by ANDIng this mask with
162
              *
                          the inverted freelist register
             *
163
             */
164
            if (\text{search_footer} = 1) {
165
              mask = (FREELIST\_UINT\_T) (((flag << (end\_pos\_in\_reg+1))-1));
166
              if ((~CurMemLuvCore->freelist[end_reg] & mask) == mask) {
167
                 free_bins += end_pos_in_reg+1;
168
                 CurMemLuvCore->ibitpos=cand_bin_end+1;
169
              }
170
            ł
171
          }
172
          break;
173
        }
174
175
        // Ox6: Check if free bins found up to now are equivalent to the bins
        11
                 requested
176
        if (free_bins == nbytes + MEMLUV_ALLOC_HEADER*sizeof(CORE_UINT_T)) {
177
178
          // 0x7: If we found requested bins, make padding according to
179
          11
                   alingment
180
          CurMemLuvCore \rightarrow ibitpos_pad = ReqPadding(CurMemLuvCore \rightarrow ibitpos) - 1;
181
182
          // 0x8: Mark requested free bins as reserved in freelist, malloc
183
                   succeeded
          11
184
          MemluvSetFreelist (CurMemLuvCore->freelist , cand_bin_start ,
185
              CurMemLuvCore->ibitpos_pad);
          CurMemLuvCore->action_status = MALLOC_SUCCESS;
186
187
          // 0x9: Return the position of first free bin, to allocator
188
          return cand_bin_start;
189
190
        }
191
192
        // Oxa: Check next bin
        CurMemLuvCore->ibitpos++;
193
194
195
```

```
/* Oxb: If never returned on step 0x9, and freelist was scanned entirely,
196
              there are no continuous free bins for current request, thus
197
              allocation failed.
198
      */
199
     CurMemLuvCore->action_status = MALLOC_FAIL;
200
     CurMemLuvCore \rightarrow ibitpos_pad = 0;
201
     return (0);
202
   #else
203
     Dprintf(0, "ERROR: Unsupported MEMLUV_FIRST_FIT_POLICY=%u on src/memluv.h.
204
          Aborting...\n\n", MEMLUV_FIRST_FIT_POLICY);
     exit(-1);
205
206 #endif
207 #else
     return 0;
208
209 #endif
   }
210
```

Listing 3.11: First Fit Implementation[2]

In listing 3.12 CurMemluvFree() is shown. This function is pretty simple and it implements the first level of the DM manager's freeing interface. Its arguments are the current HW heap struct (*CurMemLuvCore), a pointer to the block that will be freed (*ptr) and the size in bytes of this block (nbytes). If allocation headers are used, the number of bytes for every alocation are saved as metadata so the last argument is valid only when allocation headers are not used (MEMLUV_ALLOC_HEADER = 2). First, this function increments the request counters (lines 12, 13), and sets the status code of this request (line 16). Then, it stores in the variables data_alloc_length and index the size in bytes of the block that has to be freed, and its index in the freelist. This is can be done with two different ways. The first, uses the allocation header (lines 18, 19), and the second, uses the nbytes function argument and calculates manually the index in the freelist of this pointer. (lines 21, 22) Finally, CurMemluvFreeBody() is called.

```
void CurMemluvFree(MemLuvCore *CurMemLuvCore, CORE_UINT_T *ptr, uint_t
      nbytes ) {
2 #ifdef __SYNTHESIS__
3 #if MEMLUV_HLS_INLINE == 1 || MEMLUV_ALLOC_HEADER > 0
4 #pragma AP inline
  #else
5
6 #pragma AP inline off
  #endif
7
 #endif
8
9
             data_alloc_length , index;
10
    uint_t
11
    CurMemLuvCore \rightarrow free_rqst ++;
12
13
    CurMemLuvCore \rightarrow tot_rqst++;
14
15
    /* Always mark a succesfull free unless check is forced */
    CurMemLuvCore->action_status = FREE_SUCCESS;
16
```

```
#if MEMLUV_ALLOC_HEADER > 0
17
    data_alloc_length=CurMemLuvCore->core [(uint_t)(ptr-CurMemLuvCore->base)
18
        -2];
    index=CurMemLuvCore->core [(uint_t)(ptr-CurMemLuvCore->base)-1];
19
20 \# else
^{21}
    data_alloc_length=nbytes;
    index = MemluvCalcPtrDistanceFromBase(CurMemLuvCore, ptr);
22
  #endif
23
    CurMemluvFreeBody(CurMemLuvCore, ptr, data_alloc_length, index);
^{24}
  }
25
```

Listing 3.12: Function CurMemluvFree[2]

The function which calculates the index in freelist of the first byte which is pointed by a pointer is shown in listing 3.13. This process is necessary because in Vivado® HLS the direct subtraction of addresses ⁶ cannot be synthesized. Thus, MemluvCalcPtrDistanceFrom Base() increments a counter by sizeof(CORE_UINT_T) (line 18) for every address that is before the address that ptr points to (line 17).

```
uint_t MemluvCalcPtrDistanceFromBase(MemLuvCore *CurMemLuvCore, CORE_UINT_T
2
      *ptr) {
3 #ifdef __SYNTHESIS__
4 \# i f MEMLUV_HLS_INLINE == 0
5 #pragma AP inline off
6 #else
7 #pragma AP inline
  #endif
8
  #endif
9
10
11
    uint_t i, distance=0;
    if (CurMemLuvCore->action_status == FREE_SUCCESS) {
12
      for (i=0; i<CurMemLuvCore->depth; i++) {
13
14
        //While this statement is synthesized, the difference cannot be
15
16
        //directly computed to a variable
        if (ptr > &CurMemLuvCore->core[i]) {
17
           distance+=(uint_t) sizeof (CORE_UINT_T);
18
         }
19
20
         else {
           //break statement causes vivado internal error
21
           i+=CurMemLuvCore->size /(uint_t)sizeof(CORE_UINT_T);
22
         }
23
      }
24
25
    ł
26
    return distance;
27 }
```

Listing 3.13: Function MemluvCalcPtrDistanceFromBase[2]

CurMemluvFreeBody() is shown in listing 3.14. First of all, it checks the validity of the address given (line 12). Then, the allocation length is updated according to the header length (line 19) and the index is written to CurMemLuvCore->stats.result (line 20). Next, in order to create the correct ending position in the freelist appropriate padding is added (line 22). Finally, the respective freelist bits are set to 0 (line 23), and the statistics are updated (line 26) and written to the debug file (line 28).

```
void CurMemluvFreeBody (MemLuvCore *CurMemLuvCore, CORE_UINT_T *ptr, uint_t
      data_alloc_length , uint_t index) {
2 #ifdef __SYNTHESIS__
3 \# if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 #else
6 #pragma AP inline
7 #endif
  #endif
8
    uint_t ptr_index_in_freelist=0, end_index_in_freelist=0, true_alloc_length
9
        ;
10
11 #if CHCK_FREE_OUT_OF_BOUNDS==1
    if ((ptr < &CurMemLuvCore->core[0]) || (ptr > &CurMemLuvCore->core[
12
        CurMemLuvCore->depth])) {
      CurMemLuvCore->action_status = FREE_FAIL_POINTER_OUT_OF_BOUNDS;
13
    }
14
    else
15
16 #endif
17
    {
      ptr_index_in_freelist = index;
18
      true_alloc_length = data_alloc_length + (uint_t)(MEMLUV_ALLOC_HEADER*
19
          sizeof(CORE_UINT_T));
      MemLuvDebugUpdate(CurMemLuvCore, 2, (uint_t)index);
20
21
      end_index_in_freelist = ReqPadding(ptr_index_in_freelist +
22
          true_alloc_length) - 1;
      MemluvClearFreelist (CurMemLuvCore, ptr_index_in_freelist,
23
          end_index_in_freelist);
      CurMemLuvCore->ibitpos_pad = end_index_in_freelist+1-
24
          ptr_index_in_freelist;
25 #if HW_DEBUG_MEMLUV==1
      MemluvFreeUpdateStas(CurMemLuvCore, data_alloc_length);
26
  #endif
27
      DumpStatisticsToFile (CurMemLuvCore
28
            #if USE_MEMLUV==1 && HW_DEBUG_MEMLUV==1
29
               , 2
30
               ,CurMemLuvCore->ibitpos_pad
31
32
            #endif
             );
33
34
    }
35 }
```

Listing 3.14: Function CurMemluvFreeBody[2]

3.3 The DMM-HLS Enhanchement

The initial implementation of the DMM-HLS extension is described in section 3.2. It is a system created from scratch, and problems from every aspect of the library should be tackled. Also, Vivado HLS limitations made it more difficult and time consuming to debug. Therefore, only one algorithm was supported, First Fit. But, when the system was finally stable, the need to optimize its behaviour was emerged. The purpose of this diploma thesis is to add to the system both Next Fit and Best Fit implementations, and examine their behaviour.

3.3.1 Next Fit

First, Next Fit is implemented. This algorithm is identical to First Fit, with the only difference that it starts to search free memory space from the position that the previous allocation request stopped or the last free happened. If the whole freelist is searched without finding any suitable free blocks then the allocation process fails. This algorithm has the same behaviour with First Fit in the long term but it distributes the allocated bytes to the entire memory and not only in the beginning. Next Fit algorithm, is easily produced through First Fit. First, a new field must be added to the heap struct (listing 3.15). This variable will contain the position that the algorithm starts its search. If the last request was an allocation, this position will be the end of the allocation but if it was a free, it will be the firs byte of this freed block.

```
1 struct MemLuvCore
2 {
3 ...
4 #if MEMLUV_FIT_ALGORITHM==1
5 uint_t nf_cache_bin_in;
6 #endif
7 };
```

Listing 3.15: Next Fit MemLuvCore

Its implementation is shown at listing 3.16 and is almost identical to First's Fit. The first difference that should be highlighted is the starting position (line 23) of the algorithm. Another important difference is the loop's control variable (line 28) which is a counter. CurMemLuvCore->ibitipos cannot be used as control variable because when it reaches the maximum value (CurMemLuvCore->freelist_total_bits) it should be reset to 1 and let the loop continue its execution. This creates the last difference which is the modulo operation in line 158 and the increment of the corner case (ibitpos==0) in line 159.

```
uint_t MemluvNextFit(MemLuvCore *CurMemLuvCore, uint_t nbytes) {
2 #ifdef __SYNTHESIS__
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 #else
6 #pragma AP inline
  #endif
7
8 #endif
  #if MEMLUV_FIT_ALGORITHM==1
9
      var_type_t free_bins=0, cand_bin_start=0, cand_bin_end, i;
10
      var_type_t start_reg , end_reg;
11
      var_type_t start_pos_in_reg , end_pos_in_reg;
12
13
      var_type_t start_pos_of_next_fit;
14
15
      FREELIST_UINT_T mask, zero=0;
16
17
      FREELIST_UINT_TD flag = 1;
      uint8_t search_footer=0;
18
      uint8_t wrap_search;
19
      uint16_t number_of_bits_checked=0;
20
      uint_t bits_examined = 0;
21
22
      CurMemLuvCore \rightarrow ibitpos = CurMemLuvCore \rightarrow nf_cache_bin_in + 1;
^{23}
      start_pos_of_next_fit=CurMemLuvCore->ibitpos;
24
25
      // Ox1: Go to a while loop to check every sinlge bit if it is candidate
26
      // for allocation
27
      while (bits_examined <CurMemLuvCore->freelist_total_bits) {
28
29
30
           /* 0x2: Given this bit, check: i)
                                                  the last bit based on current
                                                  request (bytes + header)
31
32
            *
                                            ii) the registers that start and
                                                  last bit are indexed
33
            *
                                            iii) the position in these registers
34
            *
            */
35
36
           cand_bin_start=CurMemLuvCore->ibitpos -1;
37
           cand_bin_end=(var_type_t)(cand_bin_start+nbytes+MEMLUV_ALLOC_HEADER*
38
               sizeof(CORE_UINT_T) - 1);
39
           start_reg = (FREELIST_UINT_T)(cand_bin_start/(sizeof(FREELIST_UINT_T
40
               ) * 8));
           end_reg = (FREELIST_UINT_T)(cand_bin_end/(sizeof(FREELIST_UINT_T)*8)
41
               );
           start_pos_in_reg = cand_bin_start%(sizeof(FREELIST_UINT_T)*8);
42
           end_pos_in_reg = cand_bin_end%(sizeof(FREELIST_UINT_T) *8);
43
           free_bins=0;
44
45
           // Ox3: Check if requested continuous bins from cand_bin_start to
46
           // cand_bin_end are free
47
```

```
switch ( end_reg - start_reg ) {
48
49
           /* 0x4: If the request affects bits of the same register, (bits
50
              5-13)
                   Register 0: MSB=15 [xxxx xxxx xxxx] LSB=0 (bits 5-13)
51
52
                   we create a mask with zeros on the space
53
                   MSB[cand_bin_end,cand_bin_start]LSB and we check for free
54
                   positions by ANDIng this mask with the inverted freelist
55
                   register (since we check for zeros)
56
57
            */
58
           case 0:
59
               mask = (FREELIST_UINT_T) ((((flag <<(end_pos_in_reg+1))-1) - ((
60
                   flag \ll start_pos_in_reg(-1)));
               if ((~CurMemLuvCore->freelist[start_reg] & mask) == mask) {
61
                   CurMemLuvCore->ibitpos=cand_bin_end+1;
62
                   free_bins=(var_type_t)(nbytes + MEMLUV_ALLOC_HEADER*sizeof(
63
                       CORE_UINT_T));
               }else{
64
                 free_bins=1;
65
               }
66
               break;
67
68
           /* 0x5: Check bits of different registers, e.g. bits 5-42
69
70
                 Register 0: MSB=15 [xxxx xxxx xxxx xxxx] LSB=0
                                                                     (bits 5-15)
            *
71
72
73
                 Register 1: MSB=31 [xxxx xxxx xxxx] LSB=16
                                                                     (entire Reg)
74
                 Register 2: MSB=47 [xxxx xxxx xxxx] LSB=32 (bits 32-42)
75
76
                 We do the job in three steps: see i, ii, iii below
            *
77
            */
78
           default:
79
               search_footer = 0;
80
               mask = (FREELIST_UINT_T)(((~zero)) - ((flag << start_pos_in_reg))
81
                   -1));
82
               /* 0x5 i)
                          Create a mask with zeros on the space
83
                           [MSB, cand_bin_start] and check for free positions by
84
85
                *
                           ANDIng this mask with the inverted freelist register
                           (since we check for zeros)
86
                */
87
                   ((~CurMemLuvCore->freelist[start_reg] & mask) == mask) {
88
               if
                   free_bins = FREELIST_WIDTH - start_pos_in_reg;
89
90
                   search_footer = 1;
                   /* continue to footer in case the request is not big
91
                    * enough to search for intermediate registers
92
93
                    */
```

```
94
                      /* 0x5 ii) Check if intermediate registers (start_reg+1,
95
                                   end_reg) are zero (free). This step may be
96
                       *
                                   skipped if the request affects bits of continuous
97
                       *
                                   registers (i.e. 23,24)
98
                       *
                       */
99
100
                      for (i=start_reg+1; i<end_reg; i++) {</pre>
101
                          if (CurMemLuvCore->freelist[i] == zero) {
102
                               free_bins + = FREELIST_WIDTH;
103
                               \operatorname{search}_{-}\operatorname{footer}=1;
104
                          }
105
                          else {
106
                               free_bins=1;
107
                               \operatorname{search}_{-}\operatorname{footer}=0;
108
                               break;
109
                          }
110
                      }
111
112
                      /* 0x5 iii) Create a mask with zeros on the space
113
                                    [cand_bin_end,LSB] and check for free positions
114
                       *
                                    by ANDIng this mask with the inverted freelist
                       *
115
                                    register.
                       *
116
                       */
117
118
                      if (search_footer == 1) {
119
                          mask = (FREELIST\_UINT\_T) (((flag <<(end\_pos\_in\_reg+1))-1))
120
                          if ((~CurMemLuvCore->freelist[end_reg] & mask) == mask)
121
                               {
                               free_bins += end_pos_in_reg+1;
122
123
                               CurMemLuvCore->ibitpos=cand_bin_end+1;
                          }else{
124
                             free_bins=1;
125
                           }
126
                      }
127
                 }else{
128
                   free_bins=1;
129
130
                 }
                 break;
131
132
            }
133
134
            // Ox6: Check if free bins found up to now are equivalent to the
            // bins requested
135
            if (free_bins == nbytes + MEMLUV_ALLOC_HEADER*sizeof(CORE_UINT_T)) {
136
137
138
                 // 0x7: If we found requested bins, make padding according to
                 // alingment
139
                 CurMemLuvCore->ibitpos_pad = ReqPadding(CurMemLuvCore->ibitpos)
140
                      -1;
```

```
141
                 // 0x8: Update the value of the next starting point
142
                 if (CurMemLuvCore->ibitpos_pad>CurMemLuvCore->nf_cache_bin_in)
143
                     CurMemLuvCore->nf_cache_bin_in = CurMemLuvCore->ibitpos_pad;
144
145
146
                 // 0x9: Mark requested free bins as reserved in freelist,
147
                 // malloc succeeded
148
                 MemluvSetFreelist (CurMemLuvCore->freelist , cand_bin_start ,
149
                     CurMemLuvCore->ibitpos_pad);
                 CurMemLuvCore \rightarrow action\_status = MALLOC\_SUCCESS;
150
151
                 // Oxa: Return the position of first free bin, to allocator
152
                 return cand_bin_start;
153
            }
154
155
            // Oxb: Check next bin. Wrap the search to the start if the
156
            // end has been reached
157
            CurMemLuvCore->ibitpos=(CurMemLuvCore->ibitpos+free_bins)%(
158
                CurMemLuvCore->freelist_total_bits);
            CurMemLuvCore \rightarrow ibit pos +=((CurMemLuvCore \rightarrow ibit pos) ==0);
159
160
161
162
            bits_examined += free_bins;
163
       }
164
        /* Oxc: If never returned on step 0x9, and freelist was scanned
165
         * entirely, there are no continuous free bins for current request,
166
         * thus allocation failed.
167
         */
168
169
       CurMemLuvCore \rightarrow action\_status = MALLOC\_FAIL;
170
       CurMemLuvCore \rightarrow ibitpos_pad = 0;
171
   #endif
172
173
       return (0);
   }
174
```

Listing 3.16: Next Fit implementation

The only change in CurMemluvFreeBody() is shown in listing 3.17. Every free request that is served, updates the value of nf_cache_bin_in. Thus, the next allocation will start searching for free blocks from that address.

```
void CurMemluvFreeBody(MemLuvCore *CurMemLuvCore, CORE_UINT_T *ptr, uint_t
data_alloc_length, uint_t index) {
....
#if MEMLUV_FIT_ALGORITHM==1
CurMemLuvCore->nf_cache_bin_in = ptr_index_in_freelist;
#endif
....
}
```


Figure 3.4: Example memory snapshot

Number of bytes	Position of the Next Free block
-----------------	---------------------------------

Figure 3.5: Free Block's header structure

Listing 3.17: Next Fit's implementation of function CurMemluvFreebody

3.3.2 Best Fit

Best Fit algorithm searches every free block and finds the smaller free block with size equal or larger than the requested. Unlike Next Fit, this algorithm is implemented with a completely different approach from First Fit. The key parts are analyzed in section 3.3.2.1, and the implementation details are demonstrated in section 3.3.2.2.

3.3.2.1 Best Fit Key Parts

Figure 3.4 shows a possible state of the memory after dynamic storage allocation operations. The black areas are allocated blocks and the white ones are free. This example shows 5 blocks that are in use (reserved) together with 6 free (available) blocks. In First Fit and Next Fit, the state of the memory is represented by a bitmap array (1 bit = 1 memory byte) but Best Fit will use a different approach. It uses the available space to create a list of the free blocks. This is done by storing metadata in the first two words of every free block as shown in figure 3.5. The first word contains the block's size in bytes and the next word contains the position of the next free block. Therefore, the memory snapshot shown in figure 3.4, is tranformed by the DM manager as shown in figure 3.6. The arrows shown in figure 3.6 should not be confused with the actual C pointers; they just show the result of using the information that is stored in every free block.

The use of two words in every free block assumes that the minimum size of every free block is two words. For this reason, special attention should be given when the allocation is done from a free block that is exactly one word bigger than the requested



Figure 3.6: Best Fit's memory snapshot

size. The allocator cannot let this word represent a free block because the header metadata cannot be written. Thus, it should mark this word as allocated. This word should remain allocated as long as the rest of the block but it should never be used by the application because that will result in out of bounds access. When the allocation header is 0^7 , the DM manager, in order to fulfil every free request, besides the actual pointer, it also needs the size of the block that the pointer points to. However, the user cannot know if the allocator allocated an extra "unused" word so the DM manager must keep this informatuon internally. At figure 3.7 this scenario is demonstrated. The first picture shows a snapshot of the memory. The black areas are the allocated and the white areas are the free. The gray area demonstrates the scenario described above (the allocation that is going to happen is only one word smaller than the free block). The second array demonstrates how the DM allocator handles such cases. The allocation starts from the first word of this block as shown by the gray area. The last word of this block will be added to the unused words' list. In order to create this list the allocator stores the position of the next unused word in every unused word as shown symbolically by the picture's arrows. When the allocation header is 2^8 , the allocator does not create a list with the unused words. It just allocates the whole block and writes the number of bytes that were allocated in the allocation header.

Two scenarios are possible in every allocation. The first part of figure 3.8 shows the simple case. The gray area represents the allocation that is going to happen. This allocation starts from the last word of this free block because the metadata words are in the beginning. Therefore, in this case the allocator calculates the starting position of the returned pointer $((free_block's_end+1) - size_requested)$ and updates the size of the free block in the header metadata with the right value. The second part of figure 3.8 shows the special case when a block that is the same size as the request is found. In this case the allocator should find the previous free block and update its header, in order to point to the free block that is after the one that is allocated.

The deallocation process is more complex. In general the block that is going to be freed is beetween other free blocks. In this case 4 scenarios can occur. Their common attribute is that the DM manager should find the previous free block in the freelist. The

⁷#define MEMLUV_ALLOC_HEADER 0

⁸#define MEMLUV_ALLOC_HEADER 2



Figure 3.7: Unused words scenario



Figure 3.8: Allocation process

first case is when the block that is freed should merge with both the previous and the next free blocks (figure 3.9). The next cases are when the block that is freed should merge with either the next (figure 3.10) or the previous block (figure 3.11). Finally, in some cases the



Figure 3.9: First Free Scenario



Figure 3.10: Second Free Scenario

block that is freed is not adjacent to any other free blocks so it should be attached to the virtual free list with the appropriate values in its header (figure 3.12). Rarely a block that is freed may affect one of the list's ends. These "corner" cases are presented and explained with the actual code (listing 3.28)



Figure 3.12: Fourth Free Scenario

3.3.2.2 Best Fit Implementation

In the previous section the important parts of Best Fit were analyzed. In this section the actual C implementation is presented.

At listing 3.18 the changes that should be made to the struct MemLuvCore are shown. For Best Fit the freelist array is ommited from this struct, because the freelist is created through metadata headers in the actual memory. The rest of this struct remains the same except for 3 fields that were added. The first field is the position of the first free block (line 28). If the memory has no free blocks its value will be the MEMLUV_DEPTH. This will inform the DM manager when the memory is full. In general, core[freelist_start] will access the first word of the first free block. In a similar way the first unused word's position is kept (line 30). Finally, allocated (line 29) saves the number of bytes that were eventually allocated by the manager. This value is useful for statistics.

```
struct MemLuvCore
  {
2
    // Static buffer for new allocations
3
    CORE_UINT_T core [MEMLUV_DEPTH];
4
    uint8_t id;
\mathbf{5}
6
    uint_t size;
7
    uint_t depth;
8
    var_type_t *base;
9
    var_type_t *top;
10
11
    uint_t aligned, aligned_pad;
12
    uint8_t log_align;
13
    uint16_t alloc_rqst , free_rqst , tot_rqst;
14
    uint_t ibitpos, ibitpos_pad;
15
    memluv_action_status_t action_status;
16
 #ifndef __SYNTHESIS__
17
    FILE *fd;
18
    char filename [64];
19
    unsigned int file_init;
20
  #endif
21
22 #if HW_DEBUG_MEMLUV==1
    MemLuvStats stats:
23
24 #endif
    //This should always point to the first free block
25
    //if there is no free block then a special value
26
    //(MEMLUV_DEPTH) should be given to this variable
27
    uint_t freelist_start;
28
    uint_t allocated;
29
    uint_t unused_words_start;
30
31
  };
```

Listing 3.18: Changes in struct MemluvCore for Best Fit

When the first allocation occurs, the DM manager should do some initializations. As shown at listing 3.19, the first two words of the heap are filled with the appropriate values. The total heap's size in bytes is saved in the first word(line 21) and the value MEMLUV_DEPTH is saved in the second word (line 25). This value indicates that there is no other free blocks after the first. This function also saves the position of the first free block (line 28). Finally, MEMLUV_DEPTH is assigned to unused_words_start because the allocator have not created yet any unused words.

```
1 /** The initialize_memory() function is needed when the MemluvAlloc
2 * is called for the first time so as to create the appropriate
```

```
*
      header in the available memory (only for best fit).
3
      The header structure is shown below:
   *
\overline{4}
5
   *
               _____
6
   *
       | number of block's bytes | next free block
7
   *
                                                        - 1
            _____
   *
8
9
      @param CurMemLuvCore The employed HW heap structure MemLuvCore
10
   *
   */
11
  void initialize_memory(MemLuvCore *CurMemLuvCore){
12
13 #ifdef __SYNTHESIS__
    #if MEMLUV_HLS_INLINE == 0
14
      #pragma AP inline off
15
    #else
16
      #pragma AP inline
17
    #endif
18
19 #endif
    //available bytes of the HW heap
20
    CurMemLuvCore->core [0] = sizeof (CORE_UINT_T) *MEMLUV_DEPTH;
21
22
    //Next free block of the heap (the end of the heap in this case)
23
    //because the whole heap is free
24
    CurMemLuvCore = core [1] = MEMLUV_DEPTH;
25
26
    //update the position of the start of the heap
27
    CurMemLuvCore \rightarrow freelist\_start = 0;
28
29
    //update the starting position of unused words list
30
    CurMemLuvCore->unused_words_start = MEMLUV_DEPTH;
31
32
  }
```

Listing 3.19: Best Fit's initialization

At listing 3.20 Best Fit's CurMemluvAlloc() is shown. This function is responsible for the initialization of the memory heap (line 13) and every calculation that should be done outside Best Fit algorithm. Every time that is called, the statistics counters are incremented (lines 15, 16). The allocation should always be an integer amount of heap's words so the bytes requested by the user should be rounded up to meet this requirement (line 25). This rounded value is incremented according to header configuration (line 28) and then the allocation algorithm is called (line 30). The algorithm returns the position of the first word of the allocated block. If the allocation succeeded the block's address is calculated (lines 37, 52). On the contrary, if the allocation failed, the heap's base address is returned (line 44). Finally, if header is used, the number of bytes of each allocation is saved in the first word of the allocated block.

¹ void* CurMemluvAlloc(MemLuvCore *CurMemLuvCore, uint_t nbytes) {

^{2 #}ifdef __SYNTHESIS__

^{3 #}pragma AP inline

^{4 #}endif

```
var_type_t *addr=NULL, *returned;
\mathbf{5}
       uint_t cand_base;
6
       uint_t allocated =0;
\overline{7}
  #if HW_DEBUG_MEMLUV==1
8
       uint_t allocated =0;
9
10 #endif
11
    if(!(CurMemLuvCore->alloc_rqst))
12
       initialize_memory (CurMemLuvCore);
13
14
    CurMemLuvCore \rightarrow alloc_rqst ++;
15
    CurMemLuvCore \rightarrow tot_rqst++;
16
17
    uint_t aligned_bytes;
18
    uint_t modulo;
19
    uint_t word_size_in_bytes;
20
    word_size_in_bytes = sizeof(CORE_UINT_T);
21
    aligned_bytes = nbytes;
22
    modulo = nbytes % word_size_in_bytes
23
    if(modulo){
24
       aligned_bytes = nbytes + (word_size_in_bytes - modulo);
25
    }
26
  #if MEMLUV_ALLOC_HEADER > 0
27
    aligned_bytes += MEMLUV_ALLOC HEADER*sizeof(CORE_UINT_T);
28
29 \# endif
    cand_base = MemluvFit(CurMemLuvCore, aligned_bytes);
30
    allocated = CurMemLuvCore->allocated;
31
32
    if (CurMemLuvCore->action_status == MALLOC_SUCCESS) {
33
      /* 0x2: If malloc succeeded, calculate the address of the
34
               free block and do some statistics.
35
       */
36
      addr = (((var_type_t *)CurMemLuvCore->base) + cand_base);
37
38
  #if HW_DEBUG_MEMLUV==1
39
           MemluvAllocUpdateStas(CurMemLuvCore, nbytes, allocated);
40
41 #endif
    }else {
42
      // 0x3: If malloc failed, return the address of base heap
43
      addr = CurMemLuvCore->base;
44
  #if HW_DEBUG_MEMLUV==1
45
46
           MemluvAllocUpdateStas(CurMemLuvCore, nbytes, allocated);
47 #endif
  }
48
49
    //0x4: The address is increased by MEMLUV_ALLOC_HEADER positions
50
51
    //if we use header.
    returned = addr + MEMLUV_ALLOC_HEADER;
52
53
54 #if MEMLUV_ALLOC_HEADER > 0
```

```
/* 0x5: If we use header, write to the first two addresses the length
55
               of the requested allocation and the index of the requested
56
               allocation on freelist bitmap, respectively
57
        */
58
    CurMemLuvCore->core[cand_base] = allocated;
59
60
    DumpStatisticsToFile(CurMemLuvCore
61
  #if USE_MEMLUV==1 && HW_DEBUG_MEMLUV==1
62
                                              , 1
63
                                              , allocated
64
65 #endif
                            );
66
      // Ox6: Return the address in the form of a generic (void*) pointer.
67
      return ((void*)returned);
68
69
  ł
```

Listing 3.20: Best Fit's CurMemluvAlloc

The main logic of Best Fit implementation is shown at listing 3.21. The function MemluvBestFit() (listing 3.21) iterates over the heap's free blocks and finds the most suitable (according to Best Fit policy) free block for the allocation requested. This iteration is implemented through a while loop (line 40) which stops when the last free block is found. temp_difference (line 41) stores for every free block the difference between the number of bytes that exist in this free block and the number of bytes that were requested. Best Fit searches for the minimum of this value. When a possible minimum is found (line 49), the previous free block and its position are saved. Then it continues to the next free block (lines 55, 56, 57 58). This process can stop if a free block with the same size as the requested is found (line 42). If no free block is found this function returns MEMLUV_DEPTH (lines 29, 63). Otherwise the position of the first word that is allocated is returned (line 69). Every side effect of an allocation is managed by the function allocate_memory() which is demonstrated at listing 3.22.

```
/** This function searches for the suitable (best fit algorithm) block
      in the freelist if any exists.
   *
3
   *
      Oparam CurMemLuvCore The implemented HW heap struct MemLuvCore
5
      Oparam nbytes The number of bytes that must be allocated. (including
6
   *
                     allignement and header bytes)
   *
7
   */
8
  uint_t MemluvBestFit(MemLuvCore *CurMemLuvCore, uint_t nbytes){
9
10 #ifdef __SYNTHESIS__
11 #if MEMLUV_HLS_INLINE == 0
12 #pragma AP inline off
13 \# else
14 #pragma AP inline
15 #endif
16
  #endif
17
```

```
uint64_t min_difference , temp_difference;
18
    uint_t previous_free_block , current_free_block;
19
    uint_t min_position, start_of_allocated_block;
20
    uint_t previous_free_block_of_min;
21
    uint_t freelist_start;
22
    uint_t bytes_available, next_free_block;
23
    uint8_t found=0, found_zero=0;
^{24}
25
    if((CurMemLuvCore->freelist_start) == MEMLUV_DEPTH){
26
       CurMemLuvCore->action_status = MALLOC_FAIL;
27
       CurMemLuvCore \rightarrow allocated = 0;
28
       return MEMLUV.DEPTH;
^{29}
30
    }
    \min_{\text{difference}} = \text{UINT64}_{\text{MAX}};
^{31}
32
    freelist_start = CurMemLuvCore->freelist_start;
33
34
    previous_free_block = freelist_start;
35
    current_free_block = freelist_start;
36
    bytes_available = CurMemLuvCore->core[freelist_start];
37
    next_free_block = CurMemLuvCore->core[freelist_start+1];
38
39
    while(current_free_block != MEMLUV_DEPTH) {
40
       temp_difference = bytes_available - nbytes;
41
       if(!temp_difference){
42
         found_zero = 1;
43
         min_position = current_free_block;
44
         previous_free_block_of_min = previous_free_block;
45
         \min_{difference} = 0;
46
         break;
47
       }
48
       if( (temp_difference < min_difference) && (bytes_available > nbytes)){
49
         \min_{difference} = temp_{difference};
50
         \min_{\text{position}} = \text{current}_{\text{free}} + \text{block};
51
         previous_free_block_of_min = previous_free_block;
52
         found = 1;
53
       }
54
       previous_free_block = current_free_block;
55
       current_free_block = next_free_block;
56
       bytes_available = CurMemLuvCore->core[current_free_block];
57
58
       next_free_block = CurMemLuvCore > core[current_free_block + 1];
59
    }
    if(!found && !found_zero){
60
       CurMemLuvCore \rightarrow action\_status = MALLOC\_FAIL;
61
       CurMemLuvCore \rightarrow allocated = 0;
62
       return MEMLUV_DEPTH;
63
64
    }else{
       start_of_allocated_block = allocate_memory(min_position,
65
       previous_free_block_of_min, nbytes, min_difference, CurMemLuvCore);
66
67
```

```
68 CurMemLuvCore->action_status = MALLOC_SUCCESS;
69 return start_of_allocated_block;
70 }
71 }
```

Listing 3.21: Best Fit's implementation

Figure 3.8 shows the two possible scenarios in every allocation. Every task that should be performed for the completion of the allocation is handled by the function allocate_memory (listing 3.22). In the first scenario the freelist is not altered, and only the size of the block is decremented according to the allocation bytes. This case is handled by the else clause in line 69 which calculates the starting word of the allocation (line 71), and updates the header metadata (lines 70, 72). The second scenario is more complex because the block that is used for the allocation should be removed from the freelist. First, it should be checked whether the block that is removed from the list is in the middle of the list (line 50) or in the beginning (line 53). In the first case the header of the previous block should be updated with the correct values but in the second case only the global pointer which saves the start of the free list should be updated. After those changes the DM manager must check if the allocation that is going to happen creates an unused word (as described in section 3.3.2.2). The block of code that starts in line 57 handles this case. First the position of the unused word is calculated and then this word is added to the list of usused words. This word can be added either in the middle of the list (line 59) or in the beginning (line 63).

```
/** allocate_memory is used by the allocator when a suitable free block
      is found. This function makes the appropriate changes
   *
2
      to the freelist headers and to the global variable freelist start.
3
   *
      More specifically when two cases are taken into consideration:
   *
4
5
         When the available block is the same size as requested or 1 word
6
   *
         bigger (these cases are treated the same because header == 2 words).
7
   *
         If this block is the same with freelist_start, freelist_start
8
         should be updated with the next position of freelist. If the next
ç
   *
         position is MEMLUV_DEPTH our memory will be fully allocated.
10
   *
         If this block is not the freelist_start then the previous should
11
         point to the next free block (next(previous) = next(current))
12
13
         When the available block is larger than the size requested then its
14
         header should be updated with the correct values (size -= requested)
15
         This means that the free blocks are allocated from their end towards
16
17
         their start.
18
              min_position The start of the free block found
19
      @param
      @param previous_free_block_of_min The previous free block
20
^{21}
      @param
              bytes_requested The amount of bytes that the user requested
      @param min_difference How many bytes more exist in this block
22
23
      Oparam
              CurMemLuvCore The employed HW Heap structure
      Oretval assigned_position The index of the free blcok in the Heap
24
```

```
*/
25
  uint_t allocate_memory(uint_t min_position, uint_t
26
      previous_free_block_of_min ,
                           uint_t bytes_requested, uint_t min_difference,
27
                           MemLuvCore *CurMemLuvCore) {
28
  #ifdef __SYNTHESIS__
29
    #if MEMLUV_HLS_INLINE == 0
30
      #pragma AP inline off
31
    #else
32
      #pragma AP inline
33
    #endif
34
_{35} #endif
36
    uint_t bytes_available , next_free_block;
37
    uint_t start_of_allocated_block;
38
    uint_t bytes_in_two_words;
39
    uint_t previous_unused;
40
    uint_t this_unused_word;
41
    uint_t tmp;
42
43
    bytes_available = CurMemLuvCore->core[min_position];
44
    next_free_block = CurMemLuvCore->core[min_position + 1];
45
46
    bytes_in_two_words = 2*sizeof(CORE_UINT_T);
47
48
    if(min_difference < bytes_in_two_words){</pre>
49
      if(previous_free_block_of_min != min_position){
50
        CurMemLuvCore->core[previous_free_block_of_min + 1] = next_free_block;
51
         start_of_allocated_block = min_position;
52
      }else{
53
         start_of_allocated_block = min_position;
54
         CurMemLuvCore->freelist_start = next_free_block;
55
      }
56
57
      if(min_difference){
         this\_unused\_word = min\_position + (bytes\_requested >>
58
            TIMES_TO_GET_WORD_POSITION);
         if (CurMemLuvCore->unused_words_start < this_unused_word) {
59
           previous_unused = find_previous_unused (CurMemLuvCore,
60
               this_unused_word);
           CurMemLuvCore->core[this_unused_word] = CurMemLuvCore->core[
61
               previous_unused];
62
           CurMemLuvCore->core [previous_unused] = this_unused_word;
         }else{
63
           CurMemLuvCore->core[this_unused_word] = CurMemLuvCore->
64
               unused_words_start;
           CurMemLuvCore->unused_words_start = this_unused_word;
65
66
         }
      }
67
      CurMemLuvCore->allocated = bytes_requested + min_difference;
68
    }else{
69
```

```
70 bytes_available -= bytes_requested;
71 start_of_allocated_block = min_position + (bytes_available >>
72 TIMES_TO_GET_WORD_POSITION);
73 CurMemLuvCore->core[min_position] = bytes_available;
74 CurMemLuvCore->allocated = bytes_requested;
75 return start_of_allocated_block;
76 }
```

Listing 3.22: Best Fit's allocation mechanism

The function find_previous_unused() (listing 3.23) is used by allocate_memory() to find the unused word that is located just before the given position. It consists of a simple while clause (line 28) which iterates through the unused words' list and stops when an index bigger than given is reached.

```
/** This function is used to find the previous unused word
      in the unused words list.
2
   *
3
      NOTE: unused words are created when the allocated block
4
   *
      is one word bigger than the requested size. Because
5
   *
      the free blocks must have two word header the whole
6
      block is considered allocated by the allocator but
7
      the user cannot know about its existence. So a
8
      bookeeping should be implemented.
9
   *
   */
10
11
12 uint_t find_previous_unused (MemLuvCore *CurMemLuvCore, uint_t
      this_unused_word) {
13 #ifdef __SYNTHESIS__
    #if MEMLUV_HLS_INLINE == 0
14
      #pragma AP inline off
15
    #else
16
      #pragma AP inline
17
    #endif
18
19 #endif
20
    uint_t current, next, start;
21
22
    uint_t i;
23
    start = CurMemLuvCore->unused_words_start;
24
25
    current = CurMemLuvCore->unused_words_start;
    next = CurMemLuvCore->core[current];
26
27
    while(next < this_unused_word){</pre>
28
      current = next;
29
      next = CurMemLuvCore->core[current];
30
31
    }
32
    return current;
33 }
```

Listing 3.23: Function find_previous_unused

Best Fit's CurMemluvFree() is almost the same with the previous implementations and is shown at listing 3.24. The key differences are:

- The calculation of the byte's index that this pointer points to (line 18).
- The new implementation of Free's logic (line 27).

These 2 functions are shown at listings 3.25 and 3.26 respectively.

```
void CurMemluvFree(MemLuvCore *CurMemLuvCore, CORE_UINT_T *ptr, uint_t
      nbytes ) {
2 #ifdef ___SYNTHESIS__
3 #if MEMLUV_HLS_INLINE == 1 || MEMLUV_ALLOC_HEADER > 0
  #pragma AP inline
4
  #else
5
6 #pragma AP inline off
  #endif
7
  #endif
8
9
             data_alloc_length , index;
    uint_t
10
11
    CurMemLuvCore->free_rqst++;
12
    CurMemLuvCore \rightarrow tot_rqst++;
13
14
    /* Always mark a succesfull free unless check is forced */
15
    CurMemLuvCore->action_status = FREE_SUCCESS;
16
17
    index = MemluvCalcPtrDistanceFromBase_bf(CurMemLuvCore, ptr);
18
19
  #if MEMLUV_ALLOC_HEADER > 0
20
    data_alloc_length = CurMemLuvCore->core [(index>>TIMES_TO_GET_WORD_POSITION
^{21}
        ) -2];
22 \# else
    data_alloc_length = nbytes;
23
24 #endif
25
    CurMemluvFreeBody_bf(CurMemLuvCore, ptr, data_alloc_length,
26
                          index >> TIMES_TO_GET_WORD_POSITION);
27
28 }
```

Listing 3.24: Best Fit's CurMemluvFree()

MemluvCalcPtrDistanceFromBase_bf() (listing 3.25) finds through binary search the index of the first byte that *ptr points to. This implementation is way more efficient than First Fit's and Next Fit's (listing 3.13) which searches this index linearly.

```
1 uint_t calc_binary_search(MemLuvCore *CurMemLuvCore, CORE_UINT_T *ptr){
2 #ifdef __SYNTHESIS__
```

```
_{3} #if MEMLUV_HLS_INLINE == 0
4 #pragma AP inline off
5 # else
6 #pragma AP inline
  #endif
\overline{7}
  #endif
8
9
     uint_t distance=0;
10
     uint_t i, min, mid, max;
^{11}
12
    \min = 0; i = 0;
13
    max=CurMemLuvCore->depth;
14
15
    while (min<=max) {
16
       mid = min + (int)(max - min)/2;
17
       if(ptr > &(CurMemLuvCore->core[mid])){
18
         \min = \min + 1;
19
       }else if(ptr < &(CurMemLuvCore->core[mid])){
20
         \max = \min - 1;
21
       }else if(ptr == &(CurMemLuvCore->core[mid])){
22
         distance = (uint_t ) mid*sizeof(CORE_UINT_T);
23
         \min = \max + 1;
24
       }
25
    }
26
    return distance;
27
28 }
29
30 uint_t MemluvCalcPtrDistanceFromBase_bf(MemLuvCore *CurMemLuvCore,
      CORE_UINT_T * ptr) {
31 #ifdef __SYNTHESIS__
_{32} #if MEMLUV_HLS_INLINE == 0
33 #pragma AP inline off
34 #else
35 #pragma AP inline
36 #endif
37 #endif
38
     uint_t distance=0;
     if (CurMemLuvCore->action_status == FREE_SUCCESS) {
39
40
       distance = calc_binary_search (CurMemLuvCore, ptr);
^{41}
    }
42
    return distance;
43 }
```

Listing 3.25: Function MemluvCalcPtrDistanceFromBase_bf()

At listing 3.26 the function CurMemluvFreeBody_bf() is shown. This function calculates the correct value of the allocated block either with allocation header enabled (line 32) or whithout allocation header (lines 21, 37). It writes some statistics (lines 34, 39) and finally calls the function free_memory() (line 42). This function is shown at listing 3.28.

```
void CurMemluvFreeBody_bf (MemLuvCore *CurMemLuvCore, CORE_UINT_T * ptr,
1
2
                               uint_t data_alloc_length , uint_t index){
3 #ifdef __SYNTHESIS__
4 \# if MEMLUV_HLS_INLINE = 1
5 #pragma AP inline
6 #else
7 #pragma AP inline off
8 #endif
  #endif
9
10
    uint_t true_alloc_length;
11
12
13
  #if MEMLUV_ALLOC_HEADER == 0
14
    uint_t modulo;
15
    uint_t word_size_in_bytes;
16
    word_size_in_bytes = sizeof(CORE_UINT_T);
17
18
    modulo = data_alloc_length % word size in bytes;
19
    if(modulo){
20
      data_alloc_length += (word_size_in_bytes - modulo);
21
    }
22
23 #endif
24 #if CHCK_FREE_OUT_OF_BOUNDS == 1
    if (index >= MEMLUV_DEPTH) {
25
      CurMemLuvCore->action_status = FREE_FAIL_POINTER_OUT_OF_BOUNDS;
26
27
      return;
28
    }
29 #endif
30 #if MEMLUV_ALLOC_HEADER > 0
    index -= 2;
31
32
    true_alloc_length = CurMemLuvCore->core[index];
33 #if HW_DEBUG_MEMLUV == 1
    MemLuvDebugUpdate(CurMemLuvCore, 2, (uint_t) index);
34
35 \# endif
36 \# else
    true_alloc_length = data_alloc_length;
37
38 #if HW_DEBUG_MEMLUV == 1
    MemLuvDebugUpdate(CurMemLuvCore, 2, CurMemLuvCore->stats.lock);
39
40 \# endif
41 #endif
    free_memory(CurMemLuvCore, true_alloc_length, index);
42
43
44 #if HW_DEBUG_MEMLUV==1
      MemluvFreeUpdateStas(CurMemLuvCore, data_alloc_length);
45
46 \# endif
47
48
    DumpStatisticsToFile (CurMemLuvCore
49
        #if USE_MEMLUV==1 && HW_DEBUG_MEMLUV==1
50
```

```
51 ,2

52 ,CurMemLuvCore->ibitpos_pad

53 #endif

54 );

55 }
```

Listing 3.26: Function CurMemluvFreeBody_bf()

Before the analysis of the main logic of every free operation in Best Fit the function chk_for_unused() is demonstrated (listing 3.27). This function is needed only when the allocation header is not used⁹. It checks if an unused word belongs to the block that is going to be freed. If so, it increases the size of the free operation by one word (line 56). As explained in section 3.3.2.1, every unused word will be in the end of the allocated block. Thus, the possible position of an unused word is calculated as shown in line 53. The function has_unused() returns 1 if this position is a member of the unused words' list and remove this word from the list.

```
/** This function decides with the help of the unused words list
      whether this block during the allocation created an unused word.
   *
3
   *
      If so, this word is removed from the list.
      @param CurMemLuvCore The implemented HW heap
5
      Oparam possible_unused The index of a possible unused memory word
6
   *
7
      Oretval ans 1 if there is unused word, 0 otherwise
8
   *
   */
9
  uint8_t has_unused(MemLuvCore *CurMemLuvCore, uint_t possible_unused){
10
11
    uint8_t ans = 0;
12
    uint_t current, next;
13
    int i=0;
14
15
    current = CurMemLuvCore->unused_words_start;
16
    next = CurMemLuvCore->core[current];
17
    if(current == MEMLUV.DEPTH || possible_unused == MEMLUV.DEPTH){
18
      return 0;
19
    }else if(current == possible_unused){
20
      ans = 1;
21
      CurMemLuvCore->unused_words_start = next;
22
    }else{
23
      while(next < possible_unused){</pre>
24
        current = next;
25
        next = CurMemLuvCore->core[current];
26
      }
27
      if ((next == possible_unused) && (next != MEMLUV_DEPTH)) {
28
        CurMemLuvCore->core [current] = CurMemLuvCore->core [next];
29
         ans = 1;
30
31
```

```
32
33
    return ans;
34
  }
35
36
  /**
      Adds sizeof(CORE_UINT_T) bytes to the number of bytes that must
37
      be freed depending on the output of the function has_unused.
38
39
      @param CurMemLuvCore The HW heap struct
40
      Oparam nbytes The initial size in bytes
41
      @param position This blocks starts from core[position]
42
43
      Oretval nbytes the updated number of bytes
44
45
46
   */
  uint_t chk_for_unused (MemLuvCore *CurMemLuvCore, uint_t nbytes,
47
                          uint_t position){
48
49
    uint_t unused_start;
50
    uint_t possible_position;
51
52
    possible_position = position + (nbytes >> TIMES_TO_GET_WORD_POSITION);
53
54
    if(has_unused(CurMemLuvCore, possible_position)){
55
      nbytes += (uint_t) sizeof(CORE_UINT_T);
56
    }
57
    return nbytes;
58
59 }
```

Listing 3.27: Function chk_for_unused()

free_memory() (listing 3.28) frees the allocated block which starts at position position and has size **nbytes**. This function is responsible for every modification in the list of free blocks which may arise by every possible case (as explained in section 3.3.2.1). If allocation header is not used it should be checked if the allocation of this block has created an unused word (line 30). Then three corner cases are examined. The first (line 36) is when there are no other free blocks in the heap and only the header metadata (lines 37, 38) and the global pointer to free list's start should be configured (line 39). In the second case the block that is going to be freed is before the start of the list (line 40). Therefore, the global pointer which points to the first free block should be updated (line 49). The block's header is also filled with the correct metadata (lines 47, 48) even if it is merged with next one (line 43). The third corner case is when the block that is freed should be appended to the end of the list (line 50). In that case the block can either be merged with its previous affecting only the previous' size (line 53) or it is appended to the list with the appropriate values in its header (lines 56, 57). As discussed in section 3.3.2.1, the block that is going to be freed is located in the middle of the list. The else clause in line 59 is responsible for this scenarios. More specifically the scenarios in figures 3.9, 3.10, 3.11 and

3.12 are handled in lines 76, 72, 69 and 64 respectively.

```
1 /** free_memory frees the block that starts from core[position]
  * until core[nbytes/(sizeof(UINT_T))]
2
      This function is responsible for merging the block to be
3
      freed with adjacent ones (either with previous or next
\overline{4}
   *
      block)
\mathbf{5}
6
   * Cparam CurMemLuvCore The HW heap struct
\overline{7}
   * Oparam nbytes The number of bytes that should be freed
8
   * @param position The block starts from core[position]
9
  */
10
11 void free_memory (MemLuvCore *CurMemLuvCore, uint_t nbytes,
                    uint_t position){
12
13
    uint_t freelist_start;
14
    uint_t temp_next, temp_size;
15
    CORE_UINT_T *core;
16
    uint_t size_of_previous , next_of_previous;
17
    uint_t previous;
18
19
    Dfprintf(1, CurMemLuvConf->dbg_fd, "FREE_MEMORY:\n");
20
    if(position >= MEMLUV.DEPTH){
21
      Dfprintf(1, CurMemLuvConf->dbg_fd, "Error: Address too high\n");
22
      return;
23
    }
24
25
    freelist_start = CurMemLuvCore->freelist_start;
26
    core = \&(CurMemLuvCore -> core [0]);
27
28
29 #if MEMLUV_ALLOC_HEADER == 0
    nbytes = chk_for_unused(CurMemLuvCore, nbytes, position);
30
31 #endif
32
    //this line is needed for FreeUpdateStats
33
    CurMemLuvCore->ibitpos_pad = nbytes;
34
35
    if (freelist_start == MEMLUV_DEPTH) {
36
      CurMemLuvCore->core [position] = nbytes;
37
      CurMemLuvCore = core [position + 1] = freelist_start;
38
      CurMemLuvCore->freelist_start = position;
39
    }else if(freelist_start > position){
40
41
      temp_next = freelist_start;
      temp_size = nbytes;
42
      if ((position + (nbytes >> TIMES_TO_GET_WORD_POSITION)) == temp_next) {
43
         temp_size = core[temp_next] + nbytes;
44
        temp_next = core[temp_next + 1];
45
46
      }
      core[position] = temp_size;
47
      core[position + 1] = temp_next;
48
      CurMemLuvCore->freelist_start = position;
49
```

```
}else if(core[(previous = find_previous_free_block(CurMemLuvCore, position
50
        )) + 1] == MEMLUV_DEPTH) \{
      size_of_previous = core[previous];
51
      if((previous + (size_of_previous >> TIMES_TO_GET_WORD_POSITION)) ==
52
          position){
        core [ previous ] += nbytes;
53
      }else{
54
        core[previous + 1] = position;
55
        core[position] = nbytes;
56
        core[position + 1] = MEMLUV_DEPTH;
57
      }
58
    }else{
59
      previous = find_previous_free_block(CurMemLuvCore, position);
60
      size_of_previous = core[previous];
61
      next_of_previous = core[previous + 1];
62
      if( ((previous + (size_of_previous >> TIMES_TO_GET_WORD_POSITION)) !=
63
          position) &&
          ((position + (nbytes >> TIMES_TO_GET_WORD_POSITION)) !=
64
              next_of_previous)){
        core[previous + 1] = position;
65
        core[position + 1] = next_of_previous;
66
        core [position] = nbytes;
67
      }else if(((previous + (size_of_previous >> TIMES_TO_GET_WORD_POSITION)))
68
          == position) &&
               ((position + (nbytes >> TIMES_TO_GET_WORD_POSITION)) !=
69
                   next_of_previous)){
        core [previous] += nbytes;
70
      }else if(((previous + (size_of_previous >> TIMES_TO_GET_WORD_POSITION)))
71
          != position) &&
               ((position + (nbytes >> TIMES_TO_GET_WORD_POSITION)) ==
72
                   next_of_previous)){
        core[previous + 1] = position;
73
        core [position] = core [next_of_previous] + nbytes;
74
        core[position + 1] = core[next_of_previous + 1];
75
      }else{
76
        core[previous] += nbytes + core[next_of_previous];
77
78
        core[previous + 1] = core[next_of_previous + 1];
79
      }
80
    }
81 }
```

Listing 3.28: Function free_memory()

Chapter 4

Experiments and Results

This chapter demonstrates the process that is used in order to create a profile of the extended DMM-HLS. First, the testbench which is used by the Vivado R HLS is analyzed along with every framework's function call that it is required. Then, every allocation/deallocation test is analyzed. It should be noted here that the more realistic and accurate scenario is the larson test (section 4.2.4), which is the one that must be trusted more.

4.1 Testing Environment

As refered in section 2.3.2.1 Vivado (R) HLS uses a test bench for the evaluation of a design. Thus, a test bench is created in order to compare the three different implementations of the DM manager. Its main() function is shown in listing 4.1. The only part of this function that is synthesized is the function yadmm() (line 17) and it is analyzed in the following paragraph. The remaining operations are: the initialization of every data structure that is used by the DM manager (line 5), and the printing of usefull messages such as general information (total size, starting and ending address, freelist's size, freelist's depth, address and data width) about the heaps that are used (line 7), and all the contents of the heap (line 24) and the freelist array if used (line 25). These operations are not inluded in the time measurement of the simulation process.

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

```
val=RandMinMax(1,i);
13
  #else
14
       val=i;
15
  #endif
16
       returned = yadmm(val
17
                  #if HW_DEBUG_MEMLUV==1
18
                     , &mlvstats
19
                  #endif
20
                  );
21
     }
22
  #if SIM_WITH_GLIBC_MALLOC==0
23
    MemluvDumpFreeList(mlvcore, ALL);
^{24}
    MemluvDumpCore(mlvcore, ALL);
25
  #endif
26
    MemluvEnd();
27
     return 0;
^{28}
29 }
```

Listing 4.1: Test Bench's main function[2]

4.2 Experimental analysis

yadmm() consists of several different tests that use the DM manager interface. Every test is used for the performance evaluation of the different implementations of the DM manager. A preprocessor flag¹ enables a different test each time the simulation takes place. In this section every test will be analyzed.

4.2.1 First group of experiments

This group consists of 4 simple experiments. Despite their simplicity, these experiments highlighted some of the key differences between the two implementations of the DMM-HLS framework.

4.2.1.1 Test 1

The first test is shown at listing 4.2. It is just a simple scenario with four allocations and four frees. It does not evaluate thoroughly the DM manager but it provides a usefull first comparison between the different algorithms of the DM manager.

```
1 #if TEST==1
2 char *data1,*data2,*data3,*data4;
3 uint_t i;
4 data1 = (char *)MemluvAlloc(2048,0);
5 data2 = (char *)MemluvAlloc(2048,0);
6 MemluvFree(data1,2048,0);
7 data3 = (char *)MemluvAlloc(4096,0);
```

```
^{1}\mathrm{e.g.} #define TEST 1
```

```
8 data4 = (char *)MemluvAlloc(2040,0);
9 MemluvFree(data2,2048,0);
10 MemluvFree(data3,4096,0);
11 MemluvFree(data4,2040,0);
12 #endif
```



4.2.1.2 Test 2

The second test (listing 4.3), is used for the same purpose as the first one. It consists of ten allocations and three deallocations. This test is a bit more stressful than the previous one.

```
1 \# if TEST == 2
    char *data1,*data2,*data3,*data4;
2
    char *data5,*data6,*data7,*data8;
3
    char *data9,*data10,*data11,*data12;
4
\mathbf{5}
    data1 = (char *) MemluvAlloc(300, 0);
6
7
    data2 = (char *) MemluvAlloc(150, 0);
    data3 = (char *) MemluvAlloc(29, 0);
8
9
    data4 = (char *) MemluvAlloc(55, 0);
    data5 = (char *) MemluvAlloc(653, 0);
10
    data6 = (char *) MemluvAlloc(323, 0);
11
    data7 = (char *) MemluvAlloc(63, 0);
12
    data8 = (char *) MemluvAlloc(89, 0);
13
    data9 = (char *) MemluvAlloc(76, 0);
14
15
    MemluvFree((void *)data5, 653, 0);
16
17
    MemluvFree((void *)data3, 29, 0);
    MemluvFree((void *)data1, 300, 0);
18
    data5 = (char *) MemluvAlloc(50, 0);
19
  #endif
20
```

Listing 4.3: Test 2

4.2.1.3 Test 3

The fourth test case is shown at listing 4.4. It is a for loop that performs two allocations and two deallocations in every iteration.

```
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);
</pre>
```

```
MemluvFree((void *)p1, 32, 0);
MemluvFree((void *)p2, 32, 0);
}
#endif
```



4.2.1.4 Test 4

The fifth test case is shown at listing 4.5. It is a for loop that performs 4 allocations and deallocations in every iteration.

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

Listing 4.5: Test 4

4.2.1.5 Results

As we can see from figure 4.1, Best Fit is x1000 and x1300 faster than First Fit and Next Fit respectively at the first experiment. The memory allocations of this experiment are only 4 but they are relatively large. The significant difference in the performance of the different implementations highlights the time consuming task of updating the free list bit map array. In the second experiment the difference in the performance of the two implementations is insignificant because the size of the memory allocations is relatively small. Thus, fewer clock cycles are required to update the values of the bit map array. The rest of the experiments of this group use small memory allocations of constant size (32 bytes). In this case the bit map array is updated through an AND operation for every memory allocation/deallocation, because 32 bytes in the heap are represented with 32 bits in the bit map array, i.e. the width of one integer. In addition, every allocation starts from



Figure 4.1: First group of experiments: Simulation Time in logarithmic scale



Figure 4.2: First group of experiments: Number of flip-flops



Figure 4.3: First group of experiments: number of LUTs

the beginning of the heap. As a consequence, First Fit and Next Fit outperform Best Fit. The number of the flip-flops (figure 4.2) and the LUTs (figure 4.3) that are used depends strongly on the experiment's structure. In general, though, Best Fit requires 18% fewer



Figure 4.4: First group of experiments: Number of DSP48E



Figure 4.5: First group of experiments: Number of block RAMs

flip-flops, 5% fewer LUT from First Fit and 13% fewer LUT from Next Fit. First Fit and Next Fit need one more block-RAM than Best Fit (figure 4.5). This extra block-RAM is used to implement the bit map array that is required by this implementation. The size of this array is $\frac{(heap's \ size)}{8}$. The DSP48E that are required are the same for every implementation (figure 4.4).

4.2.2 Second group of experiments

This group of experiments consists of 4 test cases. In order to create larger memory footprint, in these test cases every free operation has specific possibility to be completed successfully.

4.2.2.1 Test 5

This test case (listing 4.6) introduces a new feature. In order to strech the heap to its limits every pointer that is allocated is freed with a 50% chance only. This tests the performance of the algorithms when the memory is full or almost full. RandMinMaxSyn(min, max, ...) is a pseudo-random synthesizable generator that returns random integers from the set [min, max]. Therefore, if the set is [0, 1] the chance of freeing a pointer is 50%.

```
//50% chance of freeing the pointer
1
_2 \# if TEST == 5
    int i;
3
4
    char *p0, *p1, *p2, *p3, *p4, *p5, *p6, *p7;
5
6
    char *p8, *p9, *p10, *p11, *p12, *p13, *p14, *p15;
7
    unsigned short lfsr_ptr = 0xACE1u;
8
9
    for (i=0; i<10; i++)
10
      p1 = (char *) MemluvAlloc(32, 0);
11
12
      p2 = (char *) MemluvAlloc(32, 0);
      p3 = (char *) MemluvAlloc(32, 0);
13
      p4 = (char *) MemluvAlloc(32, 0);
14
      p5 = (char *) MemluvAlloc(32, 0);
15
      p6 = (char *) MemluvAlloc(32, 0);
16
      p7 = (char *) MemluvAlloc(32, 0);
17
      p8 = (char *) MemluvAlloc(32, 0);
18
      p9 = (char *) MemluvAlloc(32, 0);
19
      p10 = (char *) MemluvAlloc(32, 0);
20
      p11 = (char *) MemluvAlloc(32, 0);
21
       p12 = (char *) MemluvAlloc(32, 0);
22
      p13 = (char *) MemluvAlloc(32, 0);
23
      p14 = (char *) MemluvAlloc(32, 0);
^{24}
       p15 = (char *) MemluvAlloc(32, 0);
25
26
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
27
         MemluvFree((void *)p1, 32, 0);
28
       }
29
30
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
         MemluvFree((void *)p2, 32, 0);
31
32
       }
       if(RandMinMaxSyn(0, 1, &lfsr_ptr, 0)){
33
         MemluvFree((void *)p3, 32, 0);
34
       }
35
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
36
         MemluvFree((void *)p4, 32, 0);
37
38
       }
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
39
         MemluvFree((void *)p5, 32, 0);
40
       }
41
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
42
         MemluvFree((void *)p6, 32, 0);
43
       }
44
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
45
         MemluvFree((void *)p7, 32, 0);
46
       }
47
       if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
48
         MemluvFree((void *)p8, 32, 0);
49
50
```

```
if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
51
         MemluvFree((void *)p9, 32, 0);
52
53
      }
      if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
54
         MemluvFree((void *)p10, 32, 0);
55
56
      }
      if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
57
         MemluvFree((void *)p11, 32, 0);
58
      }
59
      if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
60
         MemluvFree((void *)p12, 32, 0);
61
      }
62
      if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
63
         MemluvFree((void *)p13, 32, 0);
64
      }
65
      if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
66
         MemluvFree((void *)p14, 32, 0);
67
      }
68
      if(RandMinMaxSyn(0, 1, \&lfsr_ptr, 0))
69
         MemluvFree((void *)p15, 32, 0);
70
71
       }
72
73
    }
74 #endif
```



4.2.2.2 Test 6

In the same way the next test is created (listing 4.7). This test case frees the pointers only with 10% chance.

```
//10% chance of freeing each pointer
1
  #if TEST==6
2
    int i;
3
4
    char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;
5
    char *p8, *p9, *p10, *p11, *p12, *p13, *p14, *p15;
6
7
    unsigned short lfsr_ptr = 0xACE1u;
8
9
10
    for (i=0; i<30; i++)
      p1 = (char *) MemluvAlloc(32, 0);
11
      p2 = (char *) MemluvAlloc(32, 0);
12
      p3 = (char *) MemluvAlloc(32, 0);
13
      p4 = (char *) MemluvAlloc(32, 0);
14
      p5 = (char *) MemluvAlloc(32, 0);
15
      p6 = (char *) MemluvAlloc(32, 0);
16
      p7 = (char *) MemluvAlloc(32, 0);
17
      p8 = (char *) MemluvAlloc(32, 0);
18
```

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

Listing 4.7: Test 6

4.2.2.3 Test 7

In a similar way operates the next test (listing 4.8). Now the each pointer is freed with 30% chance.

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

```
p6 = (char *) MemluvAlloc(32, 0);
16
      p7 = (char *) MemluvAlloc(32, 0);
17
      p8 = (char *) MemluvAlloc(32, 0);
18
19
      if (RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3)
20
         MemluvFree((void *)p1, 32, 0);
21
      }
22
      if (RandMinMaxSyn(0, 9, &lfsr_ptr, 0) < 3) {
23
         MemluvFree((void *)p2, 32, 0);
^{24}
      }
25
      if (RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3)
26
         MemluvFree((void *)p3, 32, 0);
27
28
      }
      if (RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3)
29
         MemluvFree((void *)p4, 32, 0);
30
      }
31
       if (RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3) \{
32
         MemluvFree((void *)p5, 32, 0);
33
      }
34
      if(RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3) \{
35
         MemluvFree((void *)p6, 32, 0);
36
      }
37
      if (RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3)
38
         MemluvFree((void *)p7, 32, 0);
39
      }
40
      if (RandMinMaxSyn(0, 9, \&lfsr_ptr, 0) < 3)
41
         MemluvFree((void *)p8, 32, 0);
42
      }
43
44
    }
45 #endif
```

Listing 4.8: Test 7

4.2.2.4 Test 8

This is similar with the test in described in section 4.2.2.2. The only difference is that the blocks that were allocated are also written.

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

```
p2 = (char *) MemluvAlloc(32, 0);
13
      p2[31] = p1[31]+1;
14
      p3 = (char *) MemluvAlloc(32, 0);
15
      p3[31] = p2[31]+2;
16
       p4 = (char *) MemluvAlloc(32, 0);
17
      p4[31] = p3[31] + 3;
18
      p5 = (char *) MemluvAlloc(32, 0);
19
      p5[31] = p4[31]+4;
20
      p6 = (char *) MemluvAlloc(32, 0);
21
      p6[31] = p5[31]+5;
22
      p7 = (char *) MemluvAlloc(32, 0);
23
      p7[31] = p6[31]+6;
24
      p8 = (char *) MemluvAlloc(32, 0);
25
      p8[31] = p7[31] + 7;
26
27
       if (!RandMinMaxSyn(0, 9, \&lfsr_ptr, 0))
28
         MemluvFree((void *)p1, 32, 0);
29
      }
30
       if (!RandMinMaxSyn(0, 9, \&lfsr_ptr, 0))
31
         MemluvFree((void *)p2, 32, 0);
32
33
       }
       if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
34
         MemluvFree((void *)p3, 32, 0);
35
      }
36
       if (!RandMinMaxSyn(0, 9, \&lfsr_ptr, 0))
37
         MemluvFree((void *)p4, 32, 0);
38
39
       }
       if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
40
         MemluvFree((void *)p5, 32, 0);
41
42
       }
       if (!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)) {
43
         MemluvFree((void *)p6, 32, 0);
44
       }
45
       if(!RandMinMaxSyn(0, 9, &lfsr_ptr, 0)){
46
         MemluvFree((void *)p7, 32, 0);
47
      }
48
49
       if (!RandMinMaxSyn(0, 9, \&lfsr_ptr, 0))
         MemluvFree((void *)p8, 32, 0);
50
51
       }
52
    }
  #endif
53
```

Listing 4.9: Test 9

4.2.2.5 Results

In this group of experiments Best Fit's implementation is x5 times faster than First Fit's and Next Fit's (figure 4.6). Best Fit searches for free blocks in a very efficient way and its execution time decreases as the data in the memory increases. In the other imple-



Figure 4.6: Second group of experiments: Simulation Time in logarithmic scale



Figure 4.7: Second group of experiments: Number of flip-flops



Figure 4.8: Second group of experiments: number of LUTs

mentation though, the execution time is increased as the data in the memory is increased. Thus Best Fit is faster in these test cases. The simulation time in the experiments 6, 7, and 8 is the same for First Fit and Next Fit because the size of every free block that is



Figure 4.9: Second group of experiments: Number of DSP48E



Figure 4.10: Second group of experiments: Number of block RAMs

created is the same and both the algorithms complete their execution very fast. The time consuming part of updating the freelist bit map is included in both of them. The number of the flip-flops (figure 4.7) and the LUTs (figure 4.8) that are used depends strongly on the experiment's structure. Thus, experiment's 5 large loop body leads to a significant increase in resources that are required. In general, though, Best Fit requires 21% fewer flip-flops than First Fit, 18% fewer flip-flops than First Fit, and 10% fewer LUTs from First Fit and Next Fit. First Fit and Next Fit need one more block-RAM than Best Fit (figure 4.10). This extra block-RAM is used to implement the bit map array that is required by this implementation. The size of this array is $\frac{(heap's size)}{8}$. The DSP48E that are required are the same for every implementation (figure 4.9).

4.2.3 Third group of experiments

This group consists of 2 test cases. In addition to uncertain free operations randomsized memory allocations are introduced.

4.2.3.1 Test 9

The next test (listing 4.10) introduces a new feature. RandMinMaxSyn() is used for the size of every allocation. This creates a random-sized holes in the memory heap and it is a more realistic scenario than the previous ones.

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

Listing 4.10: Test 9

4.2.3.2 Test 10

This test is similar with the previous one. The random number generator, however, has been replaced by lookup tables. These tables (lookup_allocate[] and lookup_free[]) consist of random numbers that were produced by the generator (RandMinMaxSyn()). lookup_allocate has integers from 1 to 256 and lookup_free has integers from 0 to 10.

```
#if TEST==10
uint_t rand1, rand2, rand3, rand4, rand5;
int i;
char *p1, *p2, *p3, *p4, *p5;
unsigned short lfsr_ptr = 0xACE1u;
```

```
//i<84
7
    for (i=0; i<84; i++)
8
      rand1 = lookup_allocate[i * 5];
9
       p1 = (char *) MemluvAlloc(rand1, 0);
10
11
       rand2 = lookup_allocate[i*5+1];
12
      p2 = (char *) MemluvAlloc(rand2, 0);
13
14
      rand3 = lookup_allocate[i*5+2];
15
       p3 = (char *) MemluvAlloc(rand3, 0);
16
17
       rand4 = lookup_allocate[i*5+3];
18
      p4 = (char *) MemluvAlloc(rand4, 0);
19
20
       rand5 = lookup_allocate[i*5+4];
21
      p5 = (char *) MemluvAlloc(rand5, 0);
22
23
       if(!lookup_free[i]){
^{24}
         MemluvFree((void *)p1, rand1, 0);
25
      }
26
       if(!lookup_free[i+1]){
27
         MemluvFree((void *)p2, rand2, 0);
28
       }
29
       if(!lookup_free[i+2]){
30
         MemluvFree((void *)p3, rand3, 0);
31
       }
32
       if(!lookup_free[i+3]){
33
         MemluvFree((void *)p4, rand4, 0);
34
35
       }
       if(!lookup_free[i+4]){
36
         MemluvFree((void *)p5, rand5, 0);
37
38
      }
    }
39
40 #endif
```

Listing 4.11: Test 10

4.2.3.3 Results

These two test cases are closer to a real time scenario than the previous ones. Randomsized memory allocations create free blocks with different size in the heap and the small possibility of freeing each pointer increases the memory usage. Best Fit is x446 and x2952 faster than First Fit and x258 and x2052 faster than Next Fit in the tests 9 and 10 respectively (figure 4.11). The random-sized allocations force First Fit and Next Fit to search increasing areas of the bit map array which requires more clock cycles. First Fit, as Shore proved [11], tends to allocate blocks in the beginning of the memory heap and keeps large free blocks in the end. Therefore, it is more possible for First Fit to execute successfully an allocation without searching the whole bit map array. On the contrary



Figure 4.11: Third group of experiments: Simulation Time in logarithmic scale



Figure 4.12: Third group of experiments: Number of flip-flops



Figure 4.13: Third group of experiments: number of LUTs

Next Fit cancels this advantageous behaviour and it scatters the memory allocations in the whole memory. As a result, First Fit is faster than Next Fit (figure 4.11). The number of the flip-flops (figure 4.12) and the LUTs (figure 4.13) that are used depends strongly on the experiment's structure. In general, though, Best Fit requires 14% fewer flip-flops than First Fit and Next Fit, 3% fewer LUTs than First Fit, and 5% fewer LUTs from First Fit


Figure 4.14: Third group of experiments: Number of DSP48E



Figure 4.15: Third group of experiments: Number of block RAMs

and Next Fit. First Fit and Next Fit need one more block-RAM than Best Fit (figure 4.15). This extra block-RAM is used to implement the bit map array that is required by this implementation. The steep increase in the number of block-RAMs that is required in experiment 10 is owed to the lookup tables that are used when a random value is needed. The size of this array is $\frac{(heap's \ size)}{8}$. The DSP48E that are required are the same for every implementation (figure 4.14).

4.2.4 Larson Test

Per-Åke Larson and Murali Krishnan conducted a survey between different allocators in pursuit of a more scalable DM manager for parallel architectures [6]. They created a test that generates enough workload and yields reliable results. The source code of the main logic of this test is shown at listing 4.12. This test case consists of a loop (line 34) in which num_chunks deallocations and allocations are performed (lines 37 and 43 respectively) to random pointers (line 36) in every iteration. The lran2() function that is used (lines 36, 41), yields values from a uniform distribution with range from 10 to 1000. The program exits the loop when the time of the execution exceeds a specific threshold. The calculation of the time is done in line 49 and is based on the system call gettimeofday().

```
. . .
2 #define MAX_BLOCKS 1000000
  char * blkp [MAX_BLOCKS]
3
4
  void runloops(long sleep_cnt, int num_chunks ){
5
6
    int cblks ;
7
    int victim ;
8
    int blk_size ;
9
10
    long ticks_per_sec ;
11
12
    long start_cnt , end_cnt ;
    app_int64 ticks ;
13
    double duration ;
14
    double reqd_space ;
15
    //ULONG used_space ;
16
    int sum_allocs=0 ;
17
18
    QueryPerformanceFrequency( &ticks_per_sec ) ;
19
    QueryPerformanceCounter( &start_cnt) ;
20
21
    for( cblks=0; cblks<num_chunks; cblks++){</pre>
22
      if (max_size == min_size) {
^{23}
         blk_size = min_size;
24
      } else {
25
         blk_size = min_size+lran2(&rgen)%(max_size - min_size) ;
26
27
      }
28
      blkp[cblks] = (char *) malloc(blk_size) ;
29
       blksize[cblks] = blk_size;
30
       assert(blkp[cblks] != NULL) ;
31
32
    }
33
    while (TRUE) {
34
      for( cblks=0; cblks<num_chunks; cblks++){</pre>
35
         victim = lran2(&rgen)%num_chunks ;
36
         free(blkp[victim]);
37
         if (max_size == min_size) {
38
           blk\_size = min\_size;
39
40
         } else {
           blk_size = min_size+lran2(&rgen)%(max_size - min_size) ;
41
         }
42
         blkp[victim] = (char *) malloc(blk_size) ;
43
         blksize[victim] = blk_size;
44
         assert(blkp[victim] != NULL) ;
45
      }
46
47
      sum_allocs += num_chunks ;
48
       QueryPerformanceCounter(&end_cnt);
49
       ticks = end_cnt - start_cnt ;
50
```

1

```
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 4.12: Larson Test [5]

The Larson test was compiled by standard C/C++ compilers and run in general purpose processors. As a consequence, for the adaptation of this test case to vivado HLS' test bench some modifications are required. First, the code that is synthesized cannot have an array of pointers. Thus, blkp[] will be replaced by independent pointers. The number of these pointers, however, cannot be large, due to synthesis implications. In addition, system calls are not supported for FPGA synthesis and the termination condition of the loop should be changed. Thus, the while loop is replaced with a for loop with specified limits. Furthermore, the values that are drawn from the random generator are saved in lookup tables in order to facilitate the synthesis of the test case. The basic metric that is going to be used for this approach is not the number of allocations and deallocations that were completed². This number remains constant because it is defined as the for loop limit. Thus, the metric that is going to evaluate the performance of the different algorithms will be the time of simulation. A modified Larson test is presented at listing 4.13. This test consists of 20 pointers (p0-p19) and 20 iterations of the loop.

```
//larson test
  #if TEST==11
\mathbf{2}
3
    //char *
                         blkp[MAX_BLOCKS] ;
    char *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7,*p8,*p9;
4
    char *p10,*p11,*p12,*p13,*p14,*p15,*p16,*p17,*p18,*p19;
5
6
     uint32_t
                blksize[MAX_BLOCKS] ;
7
     uint32_t
                \min_{size} = 10, \max_{size} = 500;
8
9
    //number of seconds that the test lasts
10
11
    long sleep_cnt = 10;
12
    //number of memory chunks
13
    int num_chunks = 1000;
14
15
                cblks ;
16
    int
                victim ;
17
     int
     uint32_t
                blk_size ;
18
     uint32_t
                free_size;
19
     int
                sum_allocs=0 ;
20
^{21}
     int
                i = 0;
22
     blk_size = min_size+lookup_allocate[0];
^{23}
     blksize[0] = blk_size;
^{24}
```

²Larson used this one

```
p0=(char *) MemluvAlloc(blk_size, 0);
25
26
    blk_size = min_size + lookup_allocate[1];
27
    blksize[1] = blk_size;
28
    p1=(char *)MemluvAlloc(blk_size, 0);
29
30
    blk_size = min_size+lookup_allocate[2];
^{31}
    blksize[2] = blk_size;
32
    p2=(char *)MemluvAlloc(blk_size, 0);
33
34
    blk_size = min_size+lookup_allocate[3];
35
    blksize[3] = blk_size;
36
    p3=(char *)MemluvAlloc(blk_size, 0);
37
38
    blk_size = min_size+lookup_allocate [4];
39
    blksize[4] = blk_size;
40
    p4=(char *) MemluvAlloc(blk_size, 0);
41
42
    blk_size = min_size+lookup_allocate [5];
43
    blksize[5] = blk_size;
44
    p5=(char *) MemluvAlloc(blk_size, 0);
45
46
    blk_size = min_size+lookup_allocate[6];
47
    blksize[6] = blk_size;
48
    p6=(char *) MemluvAlloc(blk_size, 0);
49
50
    blk_size = min_size+lookup_allocate [7];
51
    blksize[7] = blk_size;
52
    p7=(char *) MemluvAlloc(blk_size, 0);
53
54
    blk_size = min_size + lookup_allocate[8];
55
    blksize[8] = blk_size;
56
    p8=(char *)MemluvAlloc(blk_size, 0);
57
58
    blk_size = min_size+lookup_allocate[9];
59
    blksize[9] = blk_size;
60
61
    p9=(char *) MemluvAlloc(blk_size, 0);
62
63
    blk_size = min_size + lookup_allocate[10];
    blksize[10] = blk_size;
64
    p10=(char *) MemluvAlloc(blk_size, 0);
65
66
    blk_size = min_size + lookup_allocate[11];
67
    blksize[11] = blk_size;
68
    p11=(char *) MemluvAlloc(blk_size, 0);
69
70
71
    blk_size = min_size + lookup_allocate[12];
    blksize[12] = blk_size;
72
    p12=(char *) MemluvAlloc(blk_size, 0);
73
74
```

```
75
     blk_size = min_size + lookup_allocate[13];
     blksize[13] = blk_size;
76
     p13=(char *) MemluvAlloc(blk_size, 0);
77
78
     blk_size = min_size + lookup_allocate[14];
79
     blksize[14] = blk_size;
80
     p14=(char *)MemluvAlloc(blk_size, 0);
^{81}
82
     blk_size = min_size + lookup_allocate[15];
83
     blksize[15] = blk_size;
84
     p15=(char *)MemluvAlloc(blk_size, 0);
85
86
     blk_size = min_size + lookup_allocate[16];
87
     blksize[16] = blk_size;
88
     p16=(char *)MemluvAlloc(blk_size, 0);
89
90
     blk_size = min_size + lookup_allocate[17];
91
     blksize[17] = blk_size;
92
     p17=(char *) MemluvAlloc(blk_size, 0);
93
94
     blk_size = min_size + lookup_allocate[18];
95
     blksize[18] = blk_size;
96
     p18=(char *)MemluvAlloc(blk_size, 0);
97
98
     blk_size = min_size+lookup_allocate[19];
99
     blksize[19] = blk_size;
100
     p19=(char *)MemluvAlloc(blk_size, 0);
101
102
     for (i=0; i<20; i++){
103
       victim = lookup_20_ints[i];
104
       free_size = blksize[victim];
105
       blk_size = min_size+lookup_allocate[i];
106
107
       switch(victim){
108
         case(0):
109
            MemluvFree((void *)p0, free_size, 0);
110
111
            p0=(char *) MemluvAlloc(blk_size, 0);
            blksize[victim] = blk_size;
112
113
            break;
          case(1):
114
            MemluvFree((void *)p1, free_size, 0);
115
116
            p1=(char *) MemluvAlloc(blk_size, 0);
117
            blksize[victim] = blk_size;
            break;
118
          case(2):
119
            MemluvFree((void *)p2, free_size, 0);
120
121
            p2=(char *) MemluvAlloc(blk_size, 0);
            blksize[victim] = blk_size;
122
            break;
123
          case(3):
124
```

```
MemluvFree((void *)p3, free_size, 0);
125
            p3=(char *)MemluvAlloc(blk_size, 0);
126
            blksize[victim] = blk_size;
127
            break;
128
          case(4):
129
            MemluvFree((void *)p4, free_size, 0);
130
            p4=(char *)MemluvAlloc(blk_size, 0);
131
            blksize[victim] = blk_size;
132
            break;
133
          case(5):
134
            MemluvFree((void *)p5, free_size, 0);
135
            p5=(char *)MemluvAlloc(blk_size, 0);
136
            blksize[victim] = blk_size;
137
            break;
138
          case(6):
139
            MemluvFree((void *)p6, free_size, 0);
140
            p6=(char *)MemluvAlloc(blk_size, 0);
141
            blksize[victim] = blk_size;
142
            break;
143
         case(7):
144
            MemluvFree((void *)p7, free_size, 0);
145
            p7=(char *) MemluvAlloc(blk_size, 0);
146
            blksize[victim] = blk_size;
147
            break;
148
          case(8):
149
            MemluvFree((void *)p8, free_size, 0);
150
            p8=(char *)MemluvAlloc(blk_size, 0);
151
            blksize[victim] = blk_size;
152
153
            break;
          case(9):
154
            MemluvFree((void *)p9, free_size, 0);
155
            p9=(char *)MemluvAlloc(blk_size, 0);
156
            blksize[victim] = blk_size;
157
            break;
158
          case(10):
159
            MemluvFree((void *)p10, free_size, 0);
160
            p10=(char *)MemluvAlloc(blk_size, 0);
161
            blksize[victim] = blk_size;
162
163
            break;
          case(11):
164
            MemluvFree((void *)p11, free_size, 0);
165
166
            p11=(char *)MemluvAlloc(blk_size, 0);
167
            blksize[victim] = blk_size;
            break:
168
          case(12):
169
            MemluvFree((void *)p12, free_size, 0);
170
171
            p12=(char *)MemluvAlloc(blk_size, 0);
            blksize[victim] = blk_size;
172
            break;
173
          case(13):
174
```

```
MemluvFree((void *)p13, free_size, 0);
175
            p13=(char *)MemluvAlloc(blk_size, 0);
176
            blksize[victim] = blk_size;
177
            break;
178
         case(14):
179
            MemluvFree((void *)p14, free_size, 0);
180
            p14=(char *)MemluvAlloc(blk_size, 0);
181
            blksize[victim] = blk_size;
182
            break;
183
         case(15):
184
            MemluvFree((void *)p15, free_size, 0);
185
            p15=(char *)MemluvAlloc(blk_size, 0);
186
            blksize[victim] = blk_size;
187
            break;
188
         case(16):
189
            MemluvFree((void *)p16, free_size, 0);
190
            p16=(char *)MemluvAlloc(blk_size, 0);
191
            blksize[victim] = blk_size;
192
            break;
193
         case(17):
194
            MemluvFree((void *)p17, free_size, 0);
195
            p17=(char *) MemluvAlloc(blk_size, 0);
196
            blksize[victim] = blk_size;
197
            break;
198
         case(18):
199
            MemluvFree((void *)p18, free_size, 0);
200
            p18=(char *)MemluvAlloc(blk_size, 0);
201
            blksize[victim] = blk_size;
202
203
            break;
         case(19):
204
            MemluvFree((void *)p19, free_size, 0);
205
            p19=(char *)MemluvAlloc(blk_size, 0);
206
            blksize[victim] = blk_size;
207
208
         default:
209
            continue;
       }
210
211
     }
     result = (TB_UINT_T) i;
212
213
   #endif
```

Listing 4.13: Test 11

The rest of the test cases use this approach but alter the number of iterations of the loop. More specifically, test 12 consists of a for loop with 50 iteratations, test 13 consists of a for loop with 100 iterations and test 14 consists of a for loop with 200 iterations.

4.2.4.1 Results

In this group Best Fit is x42 faster than First Fit and x50 faster than Next Fit (figure 4.16). As explained in the previous test cases, Best Fit's implementation is more efficient



Figure 4.16: Fourth group of experiments: Simulation Time in logarithmic scale



Figure 4.17: Fourth group of experiments: Number of flip-flops



Figure 4.18: Fourth group of experiments: number of LUTs

and requires fewer operations to complete its operation than First Fit's and Next Fit's. In this group of test cases the simulation time is icreased as the number memory allocation requests increases. As shown in before, First Fit completes its operation earlier than Next



Figure 4.19: Fourth group of experiments: Number of DSP48E



Figure 4.20: Fourth group of experiments: Number of block RAMs

Fit. The number of the flip-flops (figure 4.17) and the LUTs (figure 4.18) that are used depends strongly on the experiment's structure. Best Fit requires 10% fewer flip-flops than First Fit, 15% fewer flip-flop than Next Fit, 9% fewer LUTs than First Fit, and 16% fewer LUTs than Next Fit. First Fit and Next Fit need one more block-RAM than Best Fit (figure 4.20). This extra block-RAM is used to implement the bit map array that is required by this implementation. The large number of block-RAMs that is required is owed to the lookup tables that are used when a random value is needed. The size of this array is $\frac{(heap's \ size)}{8}$. The DSP48E that are required are the same for every implementation (figure 4.19).

Chapter 5

Thesis Conclusion

5.1 General Remarks

This diploma thesis contributes to the optimization of the FPGA memory subsystem in many-accelerator systems. To this direction, the adoption of Dynamic Memory Management (DMM) of on-chip memory, is proposed. This DMM library is developed in synthesizable C code in order to be highly utilized in modern FPGA programming environments, i.e. High Level Synthesis (HLS). Different implementations of this DMM-HLS interface are explored and compared. The first implementation uses a bit map array in which every bit represents a byte in the memory heap. The bits that are equal to 1 represent allocated bytes and those equal to 0 represent free bytes. This implementation is used by *First Fit* and *Next Fit* algorithms. When this approach is used, the DM manager searches bit by bit this array until it finds the requested size of free bytes. The second implementation uses a completely different approach. A list of the free blocks is created via metadata headers in the heap's free blocks and the bit map array is not needed. The header is created with the use of two words. The first is the size in bytes of this free block and the second is the position of the next free block. This approach is used by *Best Fit* algorithm.

The evaluation analysis showed that the second approach is more efficient. Despite the fact that *Best Fit* should search the whole list of the free blocks untill it terminates, the simulation time of this approach is significantly smaller than the first one. The operations that are required to handle the bit map array in the first approach are time consuming and need many clock cycles. Another intresting conclusion is that *Next Fit* is slower than *First Fit*. *First Fit* tends to allocate blocks towards one end of the memory heap and encourages large free blocks to grow at the other end. On the contrary, *Next Fit* cancel this behaviour because it distributes the allocations to the whole memory heap. Therefore, if allocations that are large compared to the average request occur then *First Fit* outperforms *Next Fit*.

5.2 Future Work

A promising approach regarding the evolution of the DMM-HLS framework is the implementation of both *First Fit* and *Next Fit* algorithms with *Best Fit's* core architecture. This will improve their time performance significantly and it will give better simulation times than the *Best Fit* implementation. The next improvement could be to create an allocator which supports all the allocation/deallocation algorithms and chooses dynamically in the runtime which one should be used. Such a combination, can use in the start the fastest algorithm, which probably will be *First Fit*, and when the memory fragmentation or usage is above a specified level the allocation/deallocation algorithm should be changed to *Best Fit*. This dynamic allocation/deallocation scheme can yield the best possible combination of results regarding the simulation time of the allocator and the number of accelerators supported. *Best Fit* tends to increase very small blocks of free memory which is undesirable. A good practice for *Best Fit*, may be, to perform a memory allocation if the size of a free block satisfies the following relation.

$size_requested \leq block's_size \leq size_requested + K$

, where K is a small positive integer. The overhead that is needed for this approach¹ can be easily accomplished with the use of headers [3].

The memory heap array is implemented by the Vivado (R) HLS as a block-RAM as mentioned in chapter 2. This block-RAM can be single port, which supports only one simultaneous read and write, or dual port, which supports two simultaneous reads and writes. Obviously, this mechanism forms an obstacle in parallel architectures. Thus, an improvement that can facilitate the parallel nature of many accelerator architectures is the partition of every heap array to multiple smaller ones. This can be done easily through the Vivado (R) HLS tool with the application of the array partition directive ² to the memory heap array. The best partition policy, though, can be application-specific and can be found after careful design space exploration.

Vivado[®] HLS supports many optimization directives which alter the architecture of the RTL design. An important step towards the optimization of the DMM-HLS framework's performance could be an extensive exploration of the different combinations of optimization directives (dataflow, pipeline, inline, array partition etc.) that can be applied to the framework. The best combination of optimization directives may lead to smaller simulation times and optimal circuits.

¹the exact size of the allocation should be kept by the DM manager

²#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.
- 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.