ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση της Διεπαφής Αποσφαλμάτωσης OpenMP (OMPD) στον παραλληλοποιητικό μεταγλωττιστή OMPi

Φοιτητής: *Νικόλαος Γεωργίου*

Επιβλέπων Καθηγητής: *Βασίλειος Δημακόπουλος*

Ιωάννινα, Ιούλιος 2026

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

Κεφάλαιο 1: Εισαγωγή 2

1.1 Το πρόβλημα 2

1.2 Η λύση: OMPD 2

1.3 Αντικείμενο της διπλωματικής 3

1.4 Δομή του κειμένου 3

Κεφάλαιο 2: Θεωρητικό Υπόβαθρο 4

2.1 Το OpenMP 4

2.1.1 Μοντέλο εκτέλεσης 4

2.2 Η δυσκολία της αποσφαλμάτωσης παράλληλων προγραμμάτων 4

2.3 Το OMPD 5

2.3.1 Υλοποιήσεις του OMPD σε υπάρχοντες μεταγλωττιστές 5

Κεφάλαιο 3: Αρχιτεκτονική Υλοποίησης 7

3.1 Επισκόπηση αρχιτεκτονικής 7

3.2 Οργάνωση αρχείων 9

3.3 Βασικές δομές δεδομένων 9

3.3.1 Ο πίνακας συμβόλων 9

3.3.2 Handles 10

3.3.3 Η δομή ompd_parallel_region_t 10

3.3.4 Οι τρεις λίστες 10

3.4 Σημεία ενσωμάτωσης στο runtime 10

3.5 Πώς διαβάζει η libompd τη μνήμη του runtime 11

3.6 Ενσωμάτωση στο build system 12

Κεφάλαιο 4: Τύποι, Handles, Callbacks και Αρχικοποίηση 13

4.1 Βασικοί τύποι δεδομένων 13

4.2 Opaque handles 14

4.3 Δομή callbacks 15

4.4 Δημοσιευμένα σύμβολα του runtime 16

4.5 Αρχικοποίηση OMPD DLL 18

4.6 Συνολική εικόνα της αρχικοποίησης 19

Κεφάλαιο 5: Thread Handles και Thread State 21
5.1 Δομές δεδομένων νημάτων 21
5.1.1 ompd_thread_node_t 21
5.1.2 ort_eecb_t — Thread Control Block 21
5.1.3 _ompd_thread_handle 22
5.2 Διαχείριση thread list (runtime side) 23
5.2.1 ompd_thread_begin 23
5.2.2 ompd_thread_end 23
5.2.3 ompd_parallel_thread_join 23
5.3 Συναρτήσεις thread handles (DLL side) 24
5.4 Απαρίθμηση και ανάκτηση κατάστασης νήματος 25
5.4.1 ompd_enumerate_states 25
5.4.2 ompd_get_state 25
5.5 ICV queries με scope = thread 26
5.6 Περιορισμοί 26
Κεφάλαιο 6: Parallel Region Handles 27
6.1 Η δομή ompd_parallel_region_t 27
6.2 Runtime hooks (ompd_state.c) 28
6.2.1 ompd_parallel_begin 28
6.2.2 ompd_parallel_end 29
6.2.3 Notification breakpoints 29
6.3 DLL-side συναρτήσεις parallel handles 29
6.3.1 ompd_get_curr_parallel_handle 29
6.3.2 ompd_get_enclosing_parallel_handle 29
6.3.3 ompd_get_task_parallel_handle 30
6.3.4 ompd_rel_parallel_handle 30
6.3.5 ompd_parallel_handle_compare 30
6.3.6 ompd_get_parallel_info 30
6.4 ompd_get_thread_in_parallel 30
6.5 Ενσωμάτωση στο parallel.c 31

6.7 Περιορισμοί και μελλοντική εργασία 31
Κεφάλαιο 7: Task Handles και ICV Queries 33
7.1 Δομές δεδομένων tasks 33
7.1.1 ort_task_node_t και ort_task_icvs_t 33
7.1.2 _ompd_task_handle 34
7.2 Runtime hooks για tasks (ompd_state.c) 34
7.3 DLL-side συναρτήσεις task handles 34
7.3.1 ompd_get_curr_task_handle 34
7.3.2 ompd_get_generating_task_handle 35
7.3.3 ompd_get_scheduling_task_handle 35
7.3.4 ompd_get_task_in_parallel 35
7.3.5 ompd_get_task_function 35
7.3.6 ompd_get_task_frame 35
7.3.7 ompd_rel_task_handle και ompd_task_handle_compare 36
7.4 ICV Queries (§5.5.9) 36
7.4.1 Οι 14 ICVs της υλοποίησης 36
7.4.2 Dispatch ανά scope 37
7.5 ompd_get_icv_string_from_scope (§5.5.9.3) 37
7.6 ompd_enumerate_icvs (§5.5.9.1) 37
7.7 Περιορισμοί και μελλοντική εργασία 38
Κεφάλαιο 8: Display Control Variables 39
8.1 Ορισμός και ρόλος 39
8.2 Υλοποίηση ompd_get_display_control_vars() 40
8.3 Υλοποίηση ompd_rel_display_control_vars() 41
8.4 Μοντέλο ιδιοκτησίας μνήμης 42
8.5 Σύγκριση με ompd_get_icv_from_scope() 42
8.6 Περιορισμοί και μελλοντικές επεκτάσεις 42
Κεφάλαιο 9: Δοκιμές και Αξιολόγηση 44
9.1 Μεθοδολογία δοκιμών 44
9.2 Υποδομή δοκιμών 44

9.2.1 Μακροεντολές CHECK και CHECK_RC 44
9.2.2 Δομή κάθε test harness 45
9.2.3 In-process callbacks 45
9.3 Φάσεις δοκιμών 46
9.3.1 Φάσεις 1–3: Infrastructure (test_ompd.c) 46
9.3.2 Φάση 4: DLL API (test_ompd4.c) 47
9.3.3 Φάση 5: Thread & Task Handles (test_ompd5.c) 48
9.3.4 Φάση 6: ICV Scopes (test_ompd6.c) 48
9.3.5 Φάση 7: Full Integration (test_ompd7.c) 48
9.3.6 Φάση 8: Device Handles (test_ompd8.c) 49
9.4 Ενσωμάτωση με το Meson build system 49
9.5 Αποτελέσματα δοκιμών 50
9.6 Κάλυψη API 50
9.7 Περιορισμοί της υποδομής δοκιμών 52
9.7.1 Παράδειγμα: Επαλήθευση υπό πραγματική διακοπή (GDB) 52
Κεφάλαιο 10: Συμπεράσματα και Μελλοντική Εργασία 54
10.1 Τι υλοποιήθηκε 54
10.2 Σχεδιαστικές επιλογές 55
10.3 Περιορισμοί 55
10.4 Μελλοντική εργασία 56
10.5 Κλείνοντας 56
Βιβλιογραφία 1

Ευχαριστώ πολύ τον κ. Δημακόπουλο για την συμβολή του στο θέμα και την επίβλεψη της εργασίας, καθώς επίσης την οικογένεια μου και κυρίως τα αδέρφια μου Όλγας και Θάνου για την στήριξη τους στη διάρκεια αυτής

Κεφάλαιο 1: Εισαγωγή

Τα σύγχρονα υπολογιστικά συστήματα διαθέτουν πλέον εκ σχεδιασμού πολλαπλούς επεξεργαστικούς πυρήνες. Η αξιοποίησή τους απαιτεί παράλληλο προγραμματισμό: την ταυτόχρονη εκτέλεση τμημάτων ενός προγράμματος από διαφορετικά νήματα ή διεργασίες, με στόχο τη μείωση του χρόνου εκτέλεσης και την αποδοτικότερη χρήση των διαθέσιμων πόρων.

Το OpenMP[1] είναι το επικρατέστερο πρότυπο για παράλληλο προγραμματισμό κοινής μνήμης. Επιτρέπει στον προγραμματιστή να εισάγει παραλληλισμό σε υπάρχοντα σειριακά προγράμματα C, C++ και Fortran μέσω οδηγιών μεταγλωττιστή, χωρίς να απαιτείται ριζική αναδιάρθρωση του κώδικα. Η βιβλιοθήκη χρόνου εκτέλεσης που συνοδεύει τον μεταγλωττιστή αναλαμβάνει τη δημιουργία και τον συντονισμό των νημάτων, την κατανομή της εργασίας και τον συγχρονισμό.

1.1 Το πρόβλημα

Η αποσφαλμάτωση ενός παράλληλου προγράμματος OpenMP δεν μοιάζει με την αποσφαλμάτωση ενός σειριακού. Ο προγραμματιστής βλέπει νήματα που δημιουργούνται και εξαφανίζονται, φράγματα που μπλοκάρουν χωρίς προφανή αιτία, και εργασίες (tasks) που εκτελούνται σε άλλο νήμα από αυτό που τις δημιούργησε. Ο debugger (π.χ. GDB) βλέπει μόνο νήματα του λειτουργικού συστήματος και διευθύνσεις μνήμης. Η σύνδεση ανάμεσα στις δύο οπτικές λείπει.

Η αιτία είναι γνωστή. Η βιβλιοθήκη χρόνου εκτέλεσης του OpenMP κρατά εσωτερικές δομές (ομάδες νημάτων, ουρές εργασιών, μεταβλητές ελέγχου) στον χώρο διευθύνσεων του ίδιου του προγράμματος. Ο debugger, χωρίς γνώση της εσωτερικής διάταξης αυτών των δομών, δεν μπορεί να τις ερμηνεύσει. Το αποτέλεσμα είναι ότι ο χρήστης αναγκάζεται να συμπεράνει μόνος του ποιο νήμα λειτουργικού συστήματος αντιστοιχεί σε ποιο νήμα OpenMP, ή ποιο task ανήκει σε ποια ομάδα.

1.2 Η λύση: OMPD

Το OpenMP 5.0 εισήγαγε το OMPD (OpenMP Debugger Interface) [1], ένα πρότυπο API που επιτρέπει σε έναν debugger να ζητά πληροφορίες για την κατάσταση ενός προγράμματος OpenMP. Η ιδέα είναι απλή. Μαζί με τη βιβλιοθήκη χρόνου εκτέλεσης παρέχεται μία επιπλέον δυναμικά φορτώσιμη βιβλιοθήκη, η οποία γνωρίζει την εσωτερική διάταξη των δομών και απαντά στα ερωτήματα του debugger. Έτσι ο debugger αποκτά τη

δυνατότητα να παρουσιάσει την κατάσταση σε επίπεδο OpenMP, χωρίς να χρειάζεται να γνωρίζει τις λεπτομέρειες της συγκεκριμένης βιβλιοθήκης.

1.3 Αντικείμενο της διπλωματικής

Η εργασία υλοποιεί την υποστήριξη OMPD για τον μεταγλωττιστή OMPi [6], ο οποίος αναπτύσσεται στο Πανεπιστήμιο Ιωαννίνων. Η υλοποίηση αποτελείται από δύο μέρη. Το πρώτο είναι η ενσωμάτωση επιλεγμένων ενεργοποιητών (callbacks) και μιας δομής μεταδεδομένων μέσα στο `libort`, την υπάρχουσα βιβλιοθήκη χρόνου εκτέλεσης του OMPi. Το δεύτερο είναι η ανάπτυξη μιας ξεχωριστής βιβλιοθήκης, του `libompd`, η οποία υλοποιεί το API που ορίζει το πρότυπο και επικοινωνεί με το `libort` μέσω του debugger.

Η υλοποίηση είναι πλήρης, καλύπτοντας όλο το εύρος των προδιαγραφών του OMPD κι έχει δοκιμαστεί με επιτυχία μέσω του GNU Debugger (gdb).

1.4 Δομή του κειμένου

Το κείμενο οργανώνεται σε εννέα κεφάλαια. Το Κεφάλαιο 2 παρουσιάζει το θεωρητικό υπόβαθρο, δηλαδή το μοντέλο εκτέλεσης του OpenMP και τη σχέση του με τον debugger. Το Κεφάλαιο 3 περιγράφει την αρχιτεκτονική του OMPi, στο βαθμό που είναι απαραίτητο για την κατανόηση της υλοποίησης. Το Κεφάλαιο 4 δίνει τη συνολική εικόνα του OMPD. Τα Κεφάλαια 5 έως 7 παρουσιάζουν αναλυτικά τις τρεις οικογένειες χειρισμών (thread, parallel, task handles). Το Κεφάλαιο 8 αφορά τις συναρτήσεις πρόσβασης σε μεταβλητές ελέγχου (ICVs). Τέλος, το Κεφάλαιο 9 περιέχει τα συμπεράσματα και προτάσεις για μελλοντική εργασία.

Κεφάλαιο 2: Θεωρητικό Υπόβαθρο

2.1 Το OpenMP

Το OpenMP υπάρχει από το 1997 και αποτελεί το κυρίαρχο πρότυπο για προγραμματισμό κοινόχρηστης μνήμης σε C, C++ και Fortran. [3] Ο προγραμματιστής προσθέτει οδηγίες (directives) στον κώδικα και ο μεταγλωττιστής παράγει τον παράλληλο κώδικα αυτόματα. Με αυτόν τον τρόπο, ο ίδιος πηγαίος κώδικας μπορεί να μεταγλωττιστεί είτε σειριακά είτε παράλληλα, ανάλογα με τις επιλογές του μεταγλωττιστή.

2.1.1 Μοντέλο εκτέλεσης

Το OpenMP ακολουθεί το μοντέλο fork-join. Το αρχικό νήμα εκτελεί τον κώδικα σειριακά μέχρι να συναντήσει μια οδηγία `#pragma omp parallel`. Σε αυτό το σημείο δημιουργείται μία ομάδα νημάτων, όλα τα νήματα εκτελούν την περιοχή παραλληλίας και στο τέλος συγχρονίζονται σε ένα φράγμα (barrier) πριν συνεχίσει η εκτέλεση σειριακά. Οι περιοχές παραλληλίας μπορούν να είναι φωλιασμένες, δηλαδή ένα νήμα της ομάδας να ανοίξει με τη σειρά του νέα ομάδα.

Εκτός από τις κλασικές περιοχές παραλληλίας, το OpenMP υποστηρίζει και το μοντέλο εργασιών (tasks). Ένα task είναι ένα αυτοτελές κομμάτι εργασίας που δημιουργείται από ένα νήμα και εκτελείται είτε αμέσως είτε αργότερα από οποιοδήποτε νήμα της ομάδας. Το μοντέλο αυτό είναι απαραίτητο για ακανόνιστους υπολογισμούς, όπου ο φόρτος δεν είναι γνωστός εκ των προτέρων.

Η συμπεριφορά του χρόνου εκτέλεσης ρυθμίζεται από ένα σύνολο εσωτερικών μεταβλητών ελέγχου (Internal Control Variables, ICVs), οι οποίες περιγράφονται αναλυτικά στο Κεφάλαιο 8.

2.2 Η δυσκολία της αποσφαλμάτωσης παράλληλων προγραμμάτων

Ο debugger, όπως ο GDB [7], έχει σχεδιαστεί για σειριακά προγράμματα. Βλέπει νήματα του λειτουργικού συστήματος, διευθύνσεις μνήμης και τιμές μεταβλητών, αλλά δεν έχει καμία γνώση για την έννοια της «ομάδας νημάτων» ή του «task». Όταν το πρόγραμμα εισέρχεται σε μια περιοχή παραλληλίας, ο χρήστης βλέπει έναν αυξημένο αριθμό νημάτων λειτουργικού συστήματος και τίποτα άλλο.

Τα προβλήματα που προκύπτουν είναι τρία. Πρώτον, δεν υπάρχει τρόπος να συσχετιστεί ένα νήμα λειτουργικού συστήματος με το OpenMP νήμα που εκτελεί, ούτε να εντοπιστεί η ομάδα στην οποία ανήκει. Δεύτερον, τα tasks είναι εντελώς αόρατα. Ένα task μπορεί να έχει δημιουργηθεί από ένα νήμα και να εκτελείται σε άλλο, χωρίς ο debugger να

μπορεί να το δείξει. Τρίτον, οι μεταβλητές ελέγχου (ICVs) φαίνονται μόνο ως πεδία μέσα σε εσωτερικές δομές της βιβλιοθήκης, και η ερμηνεία τους απαιτεί γνώση της συγκεκριμένης υλοποίησης.

Η συνέπεια είναι ότι ο χρήστης χρειάζεται να γνωρίζει εσωτερικές λεπτομέρειες της βιβλιοθήκης χρόνου εκτέλεσης για να διαβάσει ακόμη και τις πιο βασικές πληροφορίες. Αυτό δεν είναι αποδεκτό για ένα τυποποιημένο API όπως το OpenMP.

2.3 Το OMPD

Το OMPD αποτελεί την απάντηση του προτύπου OpenMP στο παραπάνω πρόβλημα. Ορίζει ένα API μέσω του οποίου ο debugger ζητά πληροφορίες, χωρίς να χρειάζεται να γνωρίζει την εσωτερική διάταξη των δομών.[4] Η αρχιτεκτονική περιλαμβάνει τρεις οντότητες.

Η πρώτη είναι ο debugger, ο οποίος παίζει τον ρόλο του «τρίτου εργαλείου» (third-party tool). Ο debugger δεν γνωρίζει τίποτα για την εσωτερική δομή του χρόνου εκτέλεσης. Απλώς φορτώνει τη βιβλιοθήκη OMPD και καλεί τις συναρτήσεις της.

Η δεύτερη είναι η βιβλιοθήκη χρόνου εκτέλεσης OpenMP, η οποία εκτελείται μέσα στο πρόγραμμα που αποσφαλματώνεται. Αυτή κρατά τις εσωτερικές δομές (ομάδες, νήματα, tasks) και εκθέτει μια περιγραφή τους μέσα από μία κεντρική δομή μεταδεδομένων. Η δομή αυτή δηλώνει το μέγεθος και την τοποθεσία (offset) κάθε πεδίου, ώστε να μπορεί να διαβαστεί εξωτερικά.

Η τρίτη είναι η βιβλιοθήκη OMPD, η οποία φορτώνεται από τον debugger και λειτουργεί ως μεταφραστής ανάμεσα στις δύο πλευρές. Δεν έχει πρόσβαση στη μνήμη του προγράμματος απευθείας. Αντί αυτού, ζητά από τον debugger να διαβάσει συγκεκριμένες διευθύνσεις μέσω μιας συλλογής callbacks. Ο debugger παρέχει την πρόσβαση, η βιβλιοθήκη OMPD ερμηνεύει τα δεδομένα, και ο χρήστης βλέπει το αποτέλεσμα σε επίπεδο OpenMP.

Με αυτόν τον διαχωρισμό, ο ίδιος debugger μπορεί να αποσφαλματώνει προγράμματα που έχουν μεταγλωττιστεί με διαφορετικές υλοποιήσεις OpenMP, εφόσον κάθε υλοποίηση παρέχει τη δική της βιβλιοθήκη OMPD.

2.3.1 Υλοποιήσεις του OMPD σε υπάρχοντες μεταγλωττιστές

Το OMPD, ως πρότυπο, δεν επιβάλλει συγκεκριμένη υλοποίηση· κάθε μεταγλωττιστής OpenMP είναι ελεύθερος να αναπτύξει τη δική του βιβλιοθήκη χρόνου εκτέλεσης και τη δική του βιβλιοθήκη OMPD, αρκεί να τηρεί τη διεπαφή που ορίζει η προδιαγραφή. Δύο ευρέως διαθέσιμοι μεταγλωττιστές έχουν ήδη ενσωματώσει τέτοιες υλοποιήσεις.

Ο μεταγλωττιστής Clang, μέσω του χρόνου εκτέλεσης LLVM OpenMP, διαθέτει την πληρέστερη μέχρι σήμερα υλοποίηση. Η βιβλιοθήκη `libompd` βρίσκεται ενσωματωμένη

στον ίδιο τον κώδικα του LLVM project και χτίζεται προαιρετικά μέσω κατάλληλης επιλογής στο σύστημα παραγωγής.[8] Συνοδεύεται από πρόσθετο πρόσθετο (plugin) για τον GDB, γραμμένο σε Python, το οποίο προσθέτει εντολές ειδικά για το OpenMP, όπως `ompd icv`, `ompd parallel` και `ompd bt`, επιτρέποντας στον χρήστη να εξετάζει την κατάσταση του προγράμματος με όρους OpenMP και όχι με όρους νημάτων λειτουργικού συστήματος.

Ο GCC, μέσω της βιβλιοθήκης χρόνου εκτέλεσης libgomp, διαθέτει αντίστοιχη, αν και λιγότερο ώριμη -- υλοποίηση, αναπτυγμένη στο πλαίσιο του έργου GOMP με στόχο την κάλυψη του πέμπτου κεφαλαίου της προδιαγραφής OpenMP 5.0.[9] Η ενεργοποίησή της γίνεται μέσω της μεταβλητής περιβάλλοντος `GOMP_DEBUG`, ενώ βασικές συναρτήσεις του χρόνου εκτέλεσης, όπως η `GOMP_parallel`, καλούν συναρτήσεις ειδοποίησης (`ompd_bp_parallel_begin`, `ompd_bp_parallel_end`) κατά την είσοδο και έξοδο από παράλληλες περιοχές — μοτίβο αντίστοιχο με αυτό που ακολουθείται και στην υλοποίηση του OMPi.

Πέρα από τις υλοποιήσεις σε μεταγλωττιστές, το ίδιο το πρότυπο OMPD και οι αρχές σχεδίασής του τεκμηριώνονται σε σειρά εργασιών του Joachim Protze και συνεργατών του στο RWTH Aachen, οι οποίοι υπήρξαν από τους βασικούς συντελεστές της προδιαγραφής.[4] Ιδιαίτερο ενδιαφέρον για την παρούσα εργασία παρουσιάζει η δημοσίευσή τους σχετικά με την υποδομή δοκιμών για υλοποιήσεις OMPD, η οποία πραγματεύεται ακριβώς το ζήτημα της επαλήθευσης μιας υλοποίησης OMPD χωρίς πλήρη πρόσβαση σε πραγματικό out-of-process debugging περιβάλλον, ζήτημα που επανέρχεται και στο Κεφάλαιο 9 της παρούσας εργασίας.

Η ύπαρξη πολλαπλών, ανεξάρτητων υλοποιήσεων του OMPD σε ευρέως χρησιμοποιούμενους μεταγλωττιστές επιβεβαιώνει στην πράξη τον στόχο του προτύπου: ένας debugger που υλοποιεί την πλευρά του OMPD API μπορεί, καταρχήν, να αποσφαλματώνει προγράμματα ανεξαρτήτως του πού μεταγλωττίστηκαν, εφόσον η αντίστοιχη βιβλιοθήκη χρόνου εκτέλεσης παρέχει τη δική της βιβλιοθήκη OMPD.

Κεφάλαιο 3: Αρχιτεκτονική Υλοποίησης

Ο OMPi είναι ένας source-to-source μεταγλωττιστής OpenMP που αναπτύχθηκε στο Πανεπιστήμιο Ιωαννίνων. Αντί να παράγει απευθείας εκτελέσιμο κώδικα, μετασχηματίζει το πρόγραμμα εισόδου, δηλαδή C κώδικας με οδηγίες OpenMP, σε ισοδύναμο C κώδικα που χρησιμοποιεί κλήσεις βιβλιοθήκης χρόνου εκτέλεσης. Το αποτέλεσμα μεταγλωττίζεται στη συνέχεια από έναν συνηθισμένο μεταγλωττιστή (gcc, clang κ.λπ.).

Η αρχιτεκτονική του OMPi οργανώνεται σε τρία επίπεδα:

- Μεταγλωττιστής (`compiler/`) — αναλύει τον κώδικα εισόδου και εφαρμόζει τους μετασχηματισμούς OpenMP μέσω ενός custom AST
- Βιβλιοθήκη χρόνου εκτέλεσης (`libort/`) — παρέχει τις υπηρεσίες OpenMP κατά την εκτέλεση (νήματα, tasks, barriers, worksharing)
- Modules συσκευών (`devices/`) — υποστηρίζουν offloading σε επιταχυντές (CUDA, OpenCL, Vulkan)

Η υλοποίηση του OMPD στον OMPi χωρίζεται σε δύο ευδιάκριτα κομμάτια. Το ένα ζει μέσα στο ίδιο το πρόγραμμα που αποσφαλματώνεται, ως επέκταση της υπάρχουσας βιβλιοθήκης χρόνου εκτέλεσης. Το άλλο φορτώνεται από τον debugger και μιλάει με το πρώτο μέσω αιτημάτων ανάγνωσης μνήμης. Το κεφάλαιο αυτό περιγράφει πώς οργανώθηκαν τα δύο κομμάτια, ποιες νέες δομές δεδομένων εισήχθησαν, σε ποια σημεία του υπάρχοντος κώδικα έγιναν προσθήκες, και πώς η υποστήριξη ενεργοποιείται στο build system.

3.1 Επισκόπηση αρχιτεκτονικής

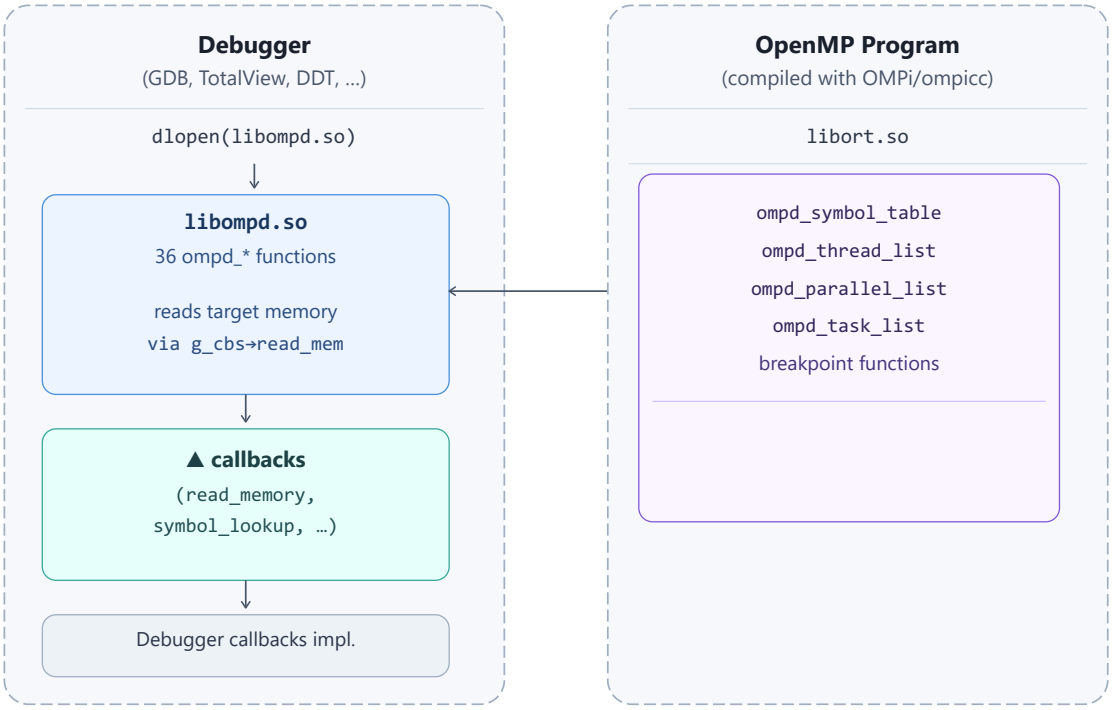
Ο διαχωρισμός σε δύο βιβλιοθήκες δεν είναι σχεδιαστική επιλογή του OMPi, αλλά απαίτηση του ίδιου του προτύπου. Η §5.1 του OpenMP 5.1 [2] ορίζει ότι η πλευρά του runtime πρέπει να παρέχει μόνο την κατάσταση, ενώ η μετάφρασή της σε απαντήσεις προς τον debugger γίνεται από μια ξεχωριστή βιβλιοθήκη που φορτώνεται δυναμικά. Έτσι ο ίδιος debugger μπορεί να δουλέψει με πολλές υλοποιήσεις OpenMP, αρκεί κάθε μία να συνοδεύεται από τη δική της βιβλιοθήκη OMPD.

Το πρώτο κομμάτι είναι η libort.so, η βιβλιοθήκη χρόνου εκτέλεσης του OMPi, στην οποία προστέθηκαν τρία πράγματα. Διατηρεί συνδεδεμένες λίστες με τα ενεργά νήματα, parallel regions και tasks, εκθέτει μεταδεδομένα για τις εσωτερικές δομές της, και ορίζει σημεία σηματοδότησης για τα σημαντικά συμβάντα του OpenMP. Δεν εκτελεί κανένα είδος

«OMPD λογικής»· απλώς κρατά την κατάσταση σε μορφή που μπορεί να διαβαστεί από τα έξω.

Το δεύτερο είναι η libompd.so, μια ανεξάρτητη βιβλιοθήκη που φορτώνει ο debugger μέσω dlopen(). Υλοποιεί ολόκληρο το σύνολο των 35 routines του OMPD API όπως ορίζεται στο OpenMP 5.1 §5.5, συν μία βοηθητική συνάρτηση (ompd_get_parallel_info) που εκθέτει απευθείας τα team_size και level μιας parallel region. Η libompd δεν συμπεριλαμβάνει τα εσωτερικά headers της libort και δεν έχει καμία απευθείας πρόσβαση στη μνήμη της διεργασίας-στόχου. Κάθε ανάγνωση γίνεται μέσω ενός πίνακα callbacks που της δίνει ο debugger κατά την αρχικοποίηση.

Ο διαχωρισμός αυτός φαίνεται στο Σχήμα 3.1. Ο debugger χρειάζεται να γνωρίζει μόνο το κοινό header ompd.h. Δεν βλέπει ούτε τη δομή των handles της libompd, ούτε φυσικά τη δομή των εσωτερικών αντικειμένων της libort.



Σχήμα 3.1: Αρχιτεκτονική OMPD — διαχωρισμός debugger και OpenMP program.

Η υποστήριξη ενεργοποιείται κατά τη μεταγλώττιση με την επιλογή Meson -Dompd=true. Η επιλογή αυτή ορίζει το preprocessor symbol OMPD_ENABLED, το οποίο περιβάλλει όλον τον νέο κώδικα στα αρχεία ort.c, parallel.c και tasks.c. Όταν η επιλογή δεν δίνεται, τίποτα από όσα περιγράφονται παρακάτω δεν μπαίνει στην παραγόμενη βιβλιοθήκη. Η επίδραση στον υπάρχοντα runtime είναι μηδενική όταν το OMPD δεν χρειάζεται.

3.2 Οργάνωση αρχείων

Ολόκληρη η υλοποίηση βρίσκεται συγκεντρωμένη στον κατάλογο libort/ompd/. Το ompd.h είναι το τυποποιημένο API header του OpenMP 5.1 §5 και μοιράζεται ανάμεσα σε libort, libompd και debugger. Το ompd_types.h περιέχει τους εσωτερικούς τύπους του OMPi, δηλαδή τα handles, τα struct metadata και τη δομή του symbol table. Τα ompd_state.h και ompd_state.c υλοποιούν την πλευρά της libort, με τα δημοσιευμένα globals, τις τρεις συνδεδεμένες λίστες, τα hooks και τα breakpoints. Το ompd_callbacks.c προσφέρει μια έτοιμη υλοποίηση της ompd_callbacks_t για in-process χρήση, η οποία χρειάζεται για τα unit tests. Τέλος, το ompd_dll.c περιέχει την υλοποίηση των συναρτήσεων του API και είναι το μόνο αρχείο που περνάει στη libompd.

Πέρα από αυτά, τροποποιήθηκαν τρία αρχεία του υπάρχοντος runtime. Το ort.c προσθέτει τις κλήσεις αρχικοποίησης και τερματισμού, το parallel.c τα hooks για parallel regions και νήματα, και το tasks.c τα hooks για τη δημιουργία και την εκτέλεση tasks. Κάθε προσθήκη είναι τυλιγμένη σε #ifdef OMPD_ENABLED.

3.3 Βασικές δομές δεδομένων

Τέσσερις κατηγορίες δομών προστέθηκαν. Η πρώτη είναι ένας κεντρικός δείκτης προς όλα τα υπόλοιπα, τρεις συνδεδεμένες λίστες κρατούν την τρέχουσα κατάσταση, τα handles αποτελούν την εξωτερική εικόνα των αντικειμένων του runtime, και μια νέα δομή περιγράφει κάθε ενεργή parallel region.

3.3.1 Ο πίνακας συμβόλων

Η δομή ompd_symbol_table_t αποθηκεύεται ως global μεταβλητή στη libort με σταθερό όνομα ompd_symbol_table, ώστε ο debugger να μπορεί να την εντοπίσει με symbol lookup. Μόλις τη βρει, η libompd τη διαβάζει ολόκληρη σε μία κλήση και αποκτά μια τοπική εικόνα της κατάστασης: τις διευθύνσεις των τριών λιστών και τους μετρητές τους, τον δείκτη στις global ICVs, και ένα σύνολο metadata που περιγράφουν τη διάταξη των εσωτερικών δομών του runtime.

Τα metadata είναι το κλειδί για την ανεξαρτησία της libompd. Κάθε πεδίο της ort_eecb_t, της ort_task_node_t και της ort_icvs_t δηλώνεται με όνομα, offset και μέγεθος, οπότε η libompd μπορεί να πλοηγείται σε αυτές τις δομές χωρίς να συμπεριλάβει ποτέ το ort_prive.h. Αν αλλάξει η διάταξη των πεδίων, αρκεί η libort να ξαναχτιστεί· η libompd συνεχίζει να δουλεύει χωρίς τροποποίηση.

3.3.2 Handles

Το πρότυπο ορίζει τέσσερις τύπους αδιαφανών handles: για address space, νήμα, parallel region και task. Ο OMPi τους υλοποιεί στο ompd_types.h. Κάθε handle είναι μια μικρή δομή που ζει στη μνήμη του debugger, εκχωρημένη μέσω του callback alloc_memory. Περιέχει έναν back-pointer προς το address space handle και τη διεύθυνση-στόχο της αντίστοιχης δομής runtime. Το address space handle κρατά επιπλέον και το τοπικό αντίγραφο του πίνακα συμβόλων. Επειδή η μνήμη ανήκει στον debugger, η αποδέσμευση γίνεται μέσω των αντίστοιχων ompd_rel_* συναρτήσεων, οι οποίες απλώς καλούν το free_memory callback.

3.3.3 Η δομή ompd_parallel_region_t

Ο βασικός runtime του OMPi δεν διατηρούσε δική του αναπαράσταση της τρέχουσας parallel region: οι πληροφορίες ήταν σκορπισμένες στα EECB των νημάτων. Για τις ανάγκες του OMPD εισήχθη μια νέα δομή που συγκεντρώνει όσα χρειάζεται να βλέπει ο debugger: το EECB του master, το μέγεθος και το επίπεδο της ομάδας, έναν δείκτη στη γονική region, και έναν πίνακα με τα EECB όλων των νημάτων της ομάδας. Οι λεπτομέρειες της δομής αναλύονται στο Κεφάλαιο 6, όπου και χρησιμοποιείται.

3.3.4 Οι τρεις λίστες

Η libort διατηρεί τρεις καθολικές συνδεδεμένες λίστες: μία για τα ενεργά νήματα (ompd_thread_list), μία για τις ενεργές parallel regions (ompd_parallel_list), και μία για τα ενεργά tasks (ompd_task_list). Κάθε λίστα ενημερώνεται από τα αντίστοιχα hooks και αποτελεί το σημείο εκκίνησης για τα ερωτήματα του debugger. Και οι τρεις προστατεύονται από το ίδιο lock, το ompd_state_lock, τύπου ee_lock_t — τον τύπο που ορίζει η στρώση αφαίρεσης Execution Environment του OMPi — ώστε η υλοποίηση του OMPD να παραμένει ανεξάρτητη από το συγκεκριμένο threading backend.

3.4 Σημεία ενσωμάτωσης στο runtime

Ο στόχος ήταν να αλλάξει ο υπάρχων κώδικας όσο το δυνατόν λιγότερο. Όλες οι προσθήκες περιορίζονται σε τρία αρχεία και όλες είναι κλήσεις προς συναρτήσεις του ompd_state.c, που κρύβουν εσωτερικά κάθε λεπτομέρεια της OMPD λογικής.

Στο ort.c προστέθηκαν δύο κλήσεις. Η ompd_state_init() καλείται μετά την αρχικοποίηση του threading layer και δέχεται δείκτη στις global ICVs. Συμπληρώνει τον πίνακα συμβόλων, εγγράφει το αρχικό νήμα στη λίστα νημάτων, και στο τέλος σηματοδοτεί στον debugger ότι το runtime είναι έτοιμο μέσω της ειδικής συνάρτησης ompd_dll_locations_valid(). Η ompd_state_finalize() αποδεσμεύει τις λίστες και μηδενίζει τον πίνακα συμβόλων.

Στο parallel.c προστέθηκαν πέντε κλήσεις, κατανεμημένες ανάμεσα στη δημιουργία και στον τερματισμό μιας παράλληλης περιοχής. Η ompd_parallel_begin() δεσμεύει τον κόμβο της νέας region και τον εισάγει στη λίστα. Η ompd_parallel_thread_join() καλείται από κάθε νήμα της ομάδας, συμπεριλαμβανομένου του master, κατά την είσοδό του στην περιοχή και καταγράφει το EECB του στη σωστή θέση του πίνακα team_members. Οι ompd_thread_begin() και ompd_thread_end() εισάγουν και αφαιρούν τα worker νήματα από τη λίστα νημάτων. Η ompd_parallel_end() αποδεσμεύει την κεφαλίδα της region στο τέλος.

Ο OMPi έχει τρεις διαφορετικές διαδρομές εξόδου από μια παράλληλη περιοχή, ανάλογα με το αν αυτή εκτελέστηκε ως task, ως βρόχος, ή απευθείας. Και οι τρεις χρειάστηκε να συμπεριλάβουν την κλήση προς την ompd_parallel_end(), ώστε καμία περιοχή να μην αφήνεται πίσω στη λίστα.

Στο tasks.c προστέθηκαν τρεις κλήσεις: μία όταν δημιουργείται ένα task, μία όταν αρχίζει η εκτέλεσή του, και μία όταν ολοκληρώνεται. Κάθε μία από αυτές ενημερώνει τη λίστα tasks και ενεργοποιεί το αντίστοιχο breakpoint, ώστε ο debugger να μπορεί να σταματήσει στις συγκεκριμένες στιγμές αν ο χρήστης το ζητήσει.

3.5 Πώς διαβάζει η libompd τη μνήμη του runtime

Από σχεδιασμό, η libompd δεν γνωρίζει τίποτα για τις εσωτερικές δομές του OMPi. Ολόκληρη η επικοινωνία με τη διεργασία-στόχο γίνεται σε τέσσερα βήματα που αποτελούν τον πυρήνα του μοντέλου:

Όταν ο debugger φορτώνει τη libompd, καλεί την ompd_initialize() δίνοντάς της έναν πίνακα callbacks. Η libompd αποθηκεύει αυτόν τον πίνακα σε μια καθολική μεταβλητή και από εκείνη τη στιγμή και μετά κάθε αίτημα πρόσβασης στη μνήμη περνά από εκεί. Στη συνέχεια καλεί την ompd_process_initialize(), η οποία αναζητά μέσω του callback symbol_addr_lookup τη διεύθυνση του ompd_symbol_table στη διεργασία-στόχο, και στη συνέχεια καλεί το read_memory για να διαβάσει ολόκληρη τη δομή σε μία κίνηση. Μετά από αυτό, η libompd έχει ένα τοπικό αντίγραφο του πίνακα συμβόλων και ξέρει τα πάντα για τη διάταξη των εσωτερικών δομών.

Για κάθε επόμενο ερώτημα, η libompd χρησιμοποιεί μια βοηθητική συνάρτηση που δέχεται τη διεύθυνση μιας δομής, ένα metadata descriptor και το όνομα ενός πεδίου. Η συνάρτηση βρίσκει το πεδίο στα metadata, υπολογίζει τη διεύθυνσή του ως διεύθυνση βάσης συν offset, και καλεί το read_memory για ακριβώς όσα bytes χρειάζεται. Έτσι η libompd δεν πλοηγείται ποτέ σε εσωτερικές δομές με γνώση της C δήλωσης· πλοηγείται πάντα με βάση τα δημοσιευμένα metadata.

Υπάρχει ένα ακόμη στοιχείο στο μοντέλο, πιο πραγματιστικό. Η υλοποίηση του symbol_addr_lookup στο ompd_callbacks.c περιλαμβάνει μια σύντομη διαδρομή για τα ονόματα που χρησιμοποιούνται πιο συχνά, όπως το ompd_symbol_table και οι τρεις λίστες. Για αυτά επιστρέφει απευθείας τη διεύθυνση χωρίς να καλέσει το dlsym. Αυτό χρειάζεται για τα unit tests, όπου η libort μπορεί να είναι στατικά συνδεδεμένη και τα εσωτερικά σύμβολα να μην είναι ορατά με dlsym(RTLD_DEFAULT).

Το αποτέλεσμα είναι ότι η ίδια libompd λειτουργεί χωρίς αλλαγές σε τρία πολύ διαφορετικά σενάρια. Σε live debugging ο debugger υλοποιεί τα callbacks πάνω από ptrace. Σε ανάλυση core dump τα υλοποιεί πάνω από απλή ανάγνωση αρχείου. Και στα in-process unit tests που συνοδεύουν την υλοποίηση, τα callbacks είναι λίγο πιο πολύπλοκα memcpy, αφού debugger και «στόχος» είναι η ίδια διεργασία.

3.6 Ενσωμάτωση στο build system

Η υποστήριξη OMPD ορίστηκε ως προαιρετική επιλογή στο Meson. Όταν ενεργοποιείται, κάνει τρία πράγματα. Προσθέτει τα ompd_state.c και ompd_callbacks.c στις πηγές της libort και ορίζει το -DOMPD_ENABLED στα c_args, ώστε να ξεκλειδώσουν τα conditional blocks στα τρία τροποποιημένα αρχεία του υπάρχοντος runtime. Χτίζει τη libompd.so από το ompd_dll.c και την εγκαθιστά δίπλα στη libort, χωρίς να τη συνδέει μαζί της. Τέλος, δηλώνει πέντε εκτελέσιμα δοκιμών (test_ompd4 έως test_ompd8) με build_by_default: false, τα οποία τρέχουν μέσω meson test --suite ompd.

Η τυπική εντολή για build με OMPD είναι η meson setup build -Dompd=true ακολουθούμενη από meson compile -C build. Χωρίς την επιλογή, ο κώδικας OMPD δεν μεταγλωττίζεται καν, και ο OMPi συμπεριφέρεται όπως πριν.

Κεφάλαιο 4: Τύποι, Handles, Callbacks και Αρχικοποίηση

Πριν περάσουμε στις τρεις οικογένειες handles του OMPD, αναλύουμε το βασικό λεξιλόγιο: ποιους τύπους ορίζει το πρότυπο, πώς αναπαρίστανται οι διευθύνσεις, ποια είναι η callback table που μεταφέρει την επικοινωνία με τον debugger, και ποια σύμβολα δημοσιεύει ο runtime ώστε να φορτωθεί η libompd. Όλα αυτά εμφανίζονται από εδώ και πέρα σε κάθε μέρος του κώδικα. Σε αυτό το κεφάλαιο υλοποιούνται τα κεφάλαια §5.2-5.3 του προτύπου.

4.1 Βασικοί τύποι δεδομένων

Το §5.3 του προτύπου ορίζει μια σειρά από scalar τύπους που η libompd και ο debugger ανταλλάσσουν μεταξύ τους. Όλοι ζουν στο `ompd.h` και είναι 64-bit, ώστε να καλύπτουν χωρίς πρόβλημα την περίπτωση όπου debugger και target τρέχουν σε διαφορετικά bitness, για παράδειγμα 64-bit GDB που εξετάζει 32-bit binary.

Η `ompd_addr_t` είναι μια διεύθυνση στον χώρο μνήμης του target process και υλοποιείται ως `uint64_t`. Η `ompd_word_t` (`int64_t`) μεταφέρει τιμές δεδομένων όπως ICV values. Η `ompd_seg_t` δηλώνει τμήμα μνήμης και στον OMPi είναι πάντα μηδέν, αφού οι αρχιτεκτονικές που υποστηρίζει το runtime (x86-64, ARM64) χρησιμοποιούν επίπεδο address space· το πεδίο υπάρχει για συμμόρφωση και για μελλοντικές πλατφόρμες όπως ορισμένοι DSP. Η `ompd_size_t` δείχνει αριθμό bytes, η `ompd_wait_id_t` αναγνωριστικό αναμονής (barrier, mutex), η `ompd_device_t` αναγνωριστικό συσκευής, και η `ompd_thread_id_t` το native ID ενός νήματος, τυπικά `pthread_t` ή Linux TID.

Επειδή το OMPD υποστηρίζει και segmented memory, οι διευθύνσεις δεν περνούν ποτέ μόνες τους. Όλη η μνήμη του target αναπαρίσταται μέσω της σύνθετης δομής `ompd_address_t`, που φέρει το ζευγάρι `(segment, address)`:

```
typedef struct ompd_address_t {
    ompd_seg_t  segment;   /* 0 σε flat address spaces */
    ompd_addr_t address;
} ompd_address_t;
```

Στην πράξη, κάθε φορά που η libompd ζητά να διαβαστεί μνήμη του target, γεμίζει τη δομή με `{ 0, target_addr }` και την περνά στο memory callback του debugger. Δύο σταθερές συμπληρώνουν την εικόνα: η `ompd_thread_id_pthread = 1` δηλώνει ότι ο τύπος native ID είναι `pthread_t`, η περίπτωση του pthreads EE backend του OMPi, ενώ η `ompd_thread_id_lwp = 2` δηλώνει Linux TID και χρησιμοποιείται από τα process-based backends.

Το enum `ompd_rc_t` κωδικοποιεί όλες τις δυνατές τιμές επιστροφής. Δεν είναι πληροφορία που τη μαθαίνει κανείς απ' έξω, αλλά καθώς κάθε function της OMPD επιστρέφει `ompd_rc_t` και η spec απαιτεί συγκεκριμένη συμπεριφορά για κάθε περίπτωση σφάλματος. Ο πίνακας που ακολουθεί αποτελεί οδηγό. Το `ompd_rc_ok` δηλώνει επιτυχία· το `ompd_rc_unavailable` ότι η πληροφορία δεν είναι διαθέσιμη, για παράδειγμα null pointer στο runtime· το `ompd_rc_stale_handle` ότι το handle δείχνει σε παλιά κατάσταση όπως task που έχει ολοκληρωθεί· και το `ompd_rc_bad_input` ότι το όρισμα είναι μη έγκυρο, null pointer ή άγνωστο ICV id. Το `ompd_rc_error` καλύπτει εσωτερικά σφάλματα όπως αδύνατη ανάγνωση μνήμης, το `ompd_rc_unsupported` ότι η συγκεκριμένη υλοποίηση δεν υποστηρίζει τη λειτουργία, και το `ompd_rc_needs_state_tracking` ότι απαιτείται ενεργοποίηση του `OMPD_ENABLED`. Τέλος, το `ompd_rc_incompatible` σηματοδοτεί ασυμβατότητα έκδοσης, το `ompd_rc_nomem` αποτυχία εκχώρησης μνήμης, το `ompd_rc_incomplete` αποκοπή string όταν το `read_string` δεν βρει NUL terminator μέσα στο όριο που του δόθηκε, και το `ompd_rc_callback_error` αποτυχία στη διαδρομή ενός callback.

4.2 Opaque handles

Πάνω στους scalar τύπους χτίζονται τέσσερα opaque handles που αναπαριστούν τα entities του OpenMP runtime: ένα address space handle για τη διεργασία, ένα thread handle για κάθε νήμα, ένα parallel handle για κάθε parallel region, και ένα task handle για κάθε task. Το πρότυπο τα δηλώνει στο `ompd.h` ως απλά forward declarations· οι πραγματικές δομές βρίσκονται στο `ompd_types.h` του OMPi και είναι ορατές μόνο από τη libompd.

Το `ompd_address_space_handle_t` αναπαριστά μια διεργασία OpenMP, live ή core dump. Στο εσωτερικό κουβαλάει το context που έδωσε ο debugger, το τοπικό αντίγραφο του symbol table που έχει διαβαστεί μία φορά κατά την αρχικοποίηση, και τρία flags για device handles: `is_device`, `dev_idx`, `dev_shared`. Το `ompd_thread_handle_t` περιέχει address space handle, thread context, τη διεύθυνση του `ort_eecb_t` στο target (`eecb_addr`), και το native thread ID (`native_id`, `native_id_size`). Τα δύο τελευταία handles είναι πιο λιτά: το `ompd_parallel_handle_t` έχει address space handle και τη διεύθυνση της `ompd_parallel_region_t` στο target (`region_addr`), ενώ το `ompd_task_handle_t` έχει address space handle και τη διεύθυνση του `ort_task_node_t` (`task_addr`).

Κάθε handle εκχωρείται από τη libompd μέσω του `alloc_memory` callback του debugger. Αυτή η λεπτομέρεια ακούγεται τυπική, στην πραγματικότητα όμως μεταφέρει την κυριότητα: τη μνήμη την κατέχει ο debugger, τον κύκλο ζωής τη χειρίζεται ο debugger, και η libompd ποτέ δεν αποδεσμεύει από μόνη της. Υπάρχουν αντίστοιχες

`ompd_rel_*_handle()` συναρτήσεις που τις καλεί ο debugger όταν κρίνει. Από την οπτική του debugger, τα handles είναι αδιαφανή: δεν χρειάζεται και δεν θέλει να γνωρίζει τη δομή τους, απλώς τα κρατά και τα περνά πίσω στη libompd σε επόμενες κλήσεις.

Παράλληλα με τα handles, το πρότυπο ορίζει δύο τύπους context που ανήκουν στον debugger και όχι στη libompd: `ompd_address_space_context_t` και `ompd_thread_context_t`. Είναι κι αυτά opaque, και η libompd τα διακινεί χωρίς να γνωρίζει τι περιέχουν. Στον OMPi, για τις ανάγκες των unit tests, ορίζονται απλά:

```
struct _ompd_aspace_cont {
    int is_local;              /* πάντα 1 για in-process χρήση */
};
struct _ompd_thread_cont {
    ompd_thread_id_t kind;     /* pthread ή lwp */
    uint64_t         native_id;
};
```

Ένας πραγματικός debugger όπως ο GDB θα έχει πολύ πιο εμπλουτισμένη δομή, με ptrace file descriptors, RPC connections για remote debugging και τα συναφή, αλλά αυτό είναι δουλειά του debugger.

4.3 Δομή callbacks

Η `ompd_callbacks_t` είναι το σημείο όπου οι δύο πλευρές, debugger και libompd, χειραγωγούν αρμοδιότητες. Ορίζεται στο §5.4.6 του προτύπου και περιέχει έντεκα function pointers· η libompd αποθηκεύει έναν δείκτη σε αυτή τη δομή κατά την αρχικοποίηση και τη συμβουλεύεται για κάθε λειτουργία που εξαρτάται από τη διεργασία-στόχο.

Τα callbacks πέφτουν σε πέντε κατηγορίες. Η πρώτη αφορά τη διαχείριση μνήμης και περιλαμβάνει το `alloc_memory(nbytes, **ptr)` και το `free_memory(*ptr)`. Η libompd ζητά μνήμη από τον debugger για strings και structs που πρέπει να επιστρέψει, και ο debugger είναι αυτός που τελικά την αποδεσμεύει. Η δεύτερη κατηγορία είναι η πρόσβαση στη μνήμη του target: `read_memory`, `write_memory` και `read_string`. Στην πράξη η libompd χρησιμοποιεί μόνο το `read_memory`, αφού ποτέ δεν τροποποιεί την κατάσταση του target. Το `read_string` έχει την ιδιαιτερότητα να επιστρέφει `ompd_rc_incomplete` όταν δεν βρει NUL terminator μέσα στο πλαίσιο του `nbytes`. Η τρίτη κατηγορία περιορίζεται σε ένα callback, το `symbol_addr_lookup`, το οποίο επιλύει ένα όνομα συμβόλου σε διεύθυνση στο target· κρίσιμο για να βρεθεί ο `ompd_symbol_table`. Το όρισμα `file` μπορεί να είναι `NULL` ώστε η αναζήτηση να καλύψει όλες τις φορτωμένες βιβλιοθήκες.

Η τέταρτη κατηγορία αφορά μεγέθη τύπων και thread contexts. Το `sizeof_type` συμπληρώνει μια `ompd_device_type_sizes_t` με τα μεγέθη των βασικών C τύπων στο target, απαραίτητο όταν debugger και target έχουν διαφορετικά ABIs. Το `get_thread_context_for_thread_id` δημιουργεί thread context για το νήμα με δοσμένο native ID. Η πέμπτη κατηγορία είναι η μετατροπή δεδομένων: τα `device_to_host` και `host_to_device` χειρίζονται διαφορές byte order μεταξύ target και debugger, ενώ το `print_string` εμφανίζει diagnostic μηνύματα από τη libompd όπου επιθυμεί ο debugger.

Για τις ανάγκες των unit tests ο OMPi συνοδεύει τη libompd με μια έτοιμη in-process υλοποίηση της callback table στο `ompd_callbacks.c`, την `ompd_local_callbacks`. Επειδή debugger και OpenMP program μοιράζονται την ίδια διεργασία, τα read/write υλοποιούνται ως `memcpy`, και τα `device_to_host` / `host_to_device` είναι επίσης `memcpy` αφού δεν υπάρχει διαφορά endianness να γεφυρωθεί. Το `sizeof_type` επιστρέφει τα `sizeof()` του host, που συμπίπτουν με του target. Το `symbol_addr_lookup` έχει ένα fast path με στατικό πίνακα από έξι γνωστά ονόματα (όπως `ompd_symbol_table` και `ompd_thread_list`) που επιστρέφουν απευθείας τη C διεύθυνση μέσω `&symbol`. Αυτή η διαδρομή είναι αναγκαία επειδή το `dlsym(RTLD_DEFAULT)` αποτυγχάνει για statically-linked βιβλιοθήκες όταν το πρόγραμμα δεν έχει κτιστεί με `-rdynamic`. Από το αρχείο εξάγονται δύο σύμβολα προς τα tests: ο πίνακας `ompd_local_callbacks` και ο context `ompd_local_aspace_context`. Με αυτά τα δύο και μόνο, ένα test μπορεί να αρχικοποιήσει το OMPD DLL χωρίς να χρειάζεται εξωτερικός debugger.

4.4 Δημοσιευμένα σύμβολα του runtime

Πριν φτάσουμε στην αρχικοποίηση της libompd, εξετάζουμε ποια σύμβολα εκθέτει η libort προς τον debugger, ώστε ο τελευταίος να μπορεί να φορτώσει το DLL. Η δουλειά γίνεται σχεδόν εξ ολοκλήρου στο `ompd_state.c`.

Το πρώτο σύμβολο είναι το `ompd_dll_locations`: ένας πίνακας από C strings που τερματίζει σε `NULL` και περιέχει διαδρομές προς πιθανά αντίγραφα της `libompd.so`. Σύμφωνα με το §5.2 του προτύπου, μόλις ο debugger εντοπίσει αυτό το σύμβολο μέσα στη φορτωμένη libort, αρχίζει να επαναλαμβάνει τον πίνακα και να δοκιμάζει `dlopen()` σε κάθε διαδρομή ώσπου να φορτωθεί κάποια. Η τιμή του `OMPD_DLL_PATH` ορίζεται από το Meson κατά τη μεταγλώττιση, επομένως ο debugger βρίσκει πάντα τη `libompd.so` της τρέχουσας εγκατάστασης χωρίς ρυθμίσεις από τον χρήστη.

Το κεντρικότερο σύμβολο είναι ο `ompd_symbol_table`, τύπου `ompd_symbol_table_t`. Αρχικοποιείται μέσα στην `ompd_state_init()` και περιέχει όλα όσα χρειάζεται η libompd για να ξεκινήσει: την έκδοση API

(`version_major = 5`, `version_minor = 1`), τη διαδρομή προς τη `libompd.so` (εναλλακτικό μονοπάτι ως προς το `ompd_dll_locations`), τους δείκτες στις linked lists του runtime (`thread_list_head`, `parallel_list_head`, `task_list_head`, `device_list_head`) μαζί με τους counters τους, έναν δείκτη στο `ort_icvs_t` του runtime για ICV queries σε scope global, και — το πιο χρήσιμο — μεταδεδομένα δομών (`*_meta`) για τα `ort_eecb_t`, `ort_task_node_t`, `ort_icvs_t` και `ort_task_icvs_t`. Τα μεταδεδομένα κουβαλούν μέσα τους ονόματα πεδίων, offsets και μεγέθη.

Η συμπλήρωση των metadata γίνεται με μια απλή μακροεντολή (`FIELD`).

Το ενδιαφέρον της προσέγγισης βρίσκεται στο ότι τα `offsetof` και `sizeof` υπολογίζονται από τον compiler του runtime, οπότε τα metadata παραμένουν συνεπή με την πραγματική διάταξη της δομής στη μνήμη ανεξάρτητα από compiler alignment ή μελλοντικές αλλαγές στα πεδία. Η libompd δεν χρειάζεται ποτέ να συμπεριλάβει το `ort_prive.h`: γνωρίζει τη διάταξη μέσω των metadata.

Πέρα από τα δεδομένα, το `ompd_state.c` εξάγει εννέα notification breakpoint functions. Είναι κενές συναρτήσεις, με μοναδικό σκοπό να αποτελούν σταθερά σημεία όπου ο debugger μπορεί να τοποθετήσει software breakpoints. Ορίζονται με `__attribute__((noinline))` και ένα compiler memory barrier στο εσωτερικό:

```
#define OMPD_NOINLINE   __attribute__((noinline))
#define OMPD_BP_BODY    do { asm volatile("" ::: "memory"); } while (0)

void OMPD_NOINLINE ompd_dll_locations_valid(void) { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_parallel_begin(void)   { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_parallel_end(void)     { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_thread_begin(void)     { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_thread_end(void)       { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_task_begin(void)       { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_task_end(void)         { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_device_begin(void)     { OMPD_BP_BODY; }
void OMPD_NOINLINE ompd_bp_device_end(void)       { OMPD_BP_BODY; }
```

Το memory barrier `asm volatile("" ::: "memory")` δεν παράγει assembly, αλλά εμποδίζει τον compiler να αναδιατάξει εγγραφές μνήμης μετά την κλήση. Έτσι, όταν ο debugger σταματήσει σε ένα από αυτά τα σημεία (π.χ. `ompd_bp_parallel_begin`), η κατάσταση των OMPD lists στο runtime είναι ήδη πλήρως ενημερωμένη και ό,τι διαβάσει ο debugger είναι συνεπές.

Η ompd_dll_locations_valid έχει ξεχωριστό ρόλο. Καλείται στο τέλος της ompd_state_init() και σηματοδοτεί στον debugger ότι το runtime έχει ολοκληρώσει την αρχικοποίησή του, οπότε η libompd μπορεί να φορτωθεί. Στην πράξη, ο GDB μέσω του OMPD plugin τοποθετεί breakpoint εκεί και περιμένει.

4.5 Αρχικοποίηση OMPD DLL

Η αρχικοποίηση της libompd γίνεται σε δύο διακριτά βήματα: πρώτα αρχικοποιείται η ίδια η DLL με την ompd_initialize, και κατόπιν δημιουργείται handle για τη συγκεκριμένη διεργασία-στόχο μέσω της ompd_process_initialize. Γύρω τους υπάρχουν τέσσερις μικρότερες βοηθητικές συναρτήσεις.

Η ompd_initialize παίρνει δύο ορίσματα, μια έκδοση API και έναν δείκτη σε ompd_callbacks_t, και είναι η πρώτη κλήση που κάνει ο debugger μετά το dlopen(libompd.so). Αν τα callbacks είναι NULL επιστρέφει ompd_rc_bad_input. Αν η ζητούμενη έκδοση είναι μικρότερη της OMPD_CURRENT_API_VERSION (202011UL, που αντιστοιχεί στο OpenMP 5.1), επιστρέφει ompd_rc_unsupported. Αν περάσει τους ελέγχους, αποθηκεύει τους callbacks σε μια DLL-global μεταβλητή g_cbs και θέτει g_inited = 1. Από εκεί και πέρα, κάθε εξαγόμενη συνάρτηση της libompd ξεκινά με δύο guard macros:

```
#define CHECK_INIT()  do { if (!g_inited || !g_cbs) return
ompd_rc_error;     } while (0)
#define CHECK_NULL(p) do { if (!(p))                return
ompd_rc_bad_input; } while (0)
```

Με αυτά αποκλείεται ολόκληρη η κατηγορία σφαλμάτων που θα προέκυπταν αν ο debugger καλούσε μια OMPD function πριν την αρχικοποίηση ή με null pointers.

Οι τρεις βοηθητικές συναρτήσεις είναι λιτές. Η ompd_get_api_version(handle, *version) γράφει στη *version την τιμή 202011UL ώστε ο debugger να επιβεβαιώνει την έκδοση μετά τη φόρτωση. Η ompd_get_version_string(handle, **string) επιστρέφει δείκτη στο static string "OMPi OMPD 5.1". Σημείωση που αξίζει εδώ: αυτή είναι η μοναδική εξαίρεση στον κανόνα ότι «η μνήμη είναι του debugger». Ο δείκτης δείχνει σε ROM string και δεν πρέπει να αποδεσμευθεί. Τέλος, η ompd_finalize καθαρίζει την κατάσταση (g_cbs = NULL, g_inited = 0) και επιστρέφει ompd_rc_unsupported αν κληθεί χωρίς προηγούμενο ompd_initialize.

Η ompd_process_initialize είναι η πιο ενδιαφέρουσα από όλες. Δουλειά της είναι να δημιουργήσει ένα address space handle και να φορτώσει τοπικά τον ompd_symbol_table του target. Δουλεύει σε τέσσερα βήματα. Πρώτα καλεί το

`symbol_addr_lookup` με όνομα `"ompd_symbol_table"` για να βρει τη διεύθυνση της δομής στο target. Μετά εκχωρεί ένα νέο handle μέσω του `alloc_memory` και επιστρέφει `ompd_rc_nomem` αν αποτύχει η εκχώρηση. Στο τρίτο βήμα καλεί `read_memory` ώστε να αντιγράψει ολόκληρη τη δομή στο πεδίο `ah->symtab` του handle, μία και μόνη ανάγνωση ανεξάρτητα από το πόσα μεταδεδομένα κουβαλάει το symbol table. Στο τέταρτο βήμα ελέγχει την έκδοση: αν `ah->symtab.version_major != 5`, αποδεσμεύει το handle και επιστρέφει `ompd_rc_incompatible`. Αυτή η συμπεριφορά είναι κρίσιμη για να αποτραπεί η σύνδεση μιας libompd 5.1 με runtime παλαιότερης έκδοσης. Όταν η συνάρτηση επιστρέψει με επιτυχία, η libompd γνωρίζει πλέον τα offsets όλων των ενδιαφερόντων πεδίων του `ort_eecb_t`, του `ort_task_node_t` και του `ort_icvs_t`, τις διευθύνσεις των linked lists, και τον δείκτη στις global ICVs, όλα αυτά μέσα από μία και μόνο κλήση `read_memory`.

Όταν το target πρόγραμμα κάνει χρήση target offloading, ο debugger μπορεί επίσης να δημιουργήσει handle για κάποια συσκευή μέσω της `ompd_device_initialize`. Η υλοποίηση διατρέχει την `ompd_device_list` του target με `dll_read` ψάχνοντας τον κόμβο με το ζητούμενο `dev_id`. Όταν τον βρει, δημιουργεί ένα device-tagged address space handle (με `is_device = 1`) και αντιγράφει το symtab από το process handle. Το όρισμα `kind` (τύπος αναγνωριστικού συσκευής) αναγνωρίζεται από τη συνάρτηση αλλά αγνοείται στην παρούσα υλοποίηση· η αντιστοίχιση γίνεται μόνο βάσει `dev_id`.

4.6 Συνολική εικόνα της αρχικοποίησης

Όλα τα παραπάνω συγκολλιούνται σε μια συγκεκριμένη χρονική σειρά που ξεκινά από την εκκίνηση του OpenMP program και τελειώνει όταν ο debugger είναι έτοιμος να κάνει OMPD queries. Όταν τρέξει η `main()`, η `_ort_init()` του runtime καλεί την `ompd_state_init()`, η οποία αρχικοποιεί τον `ompd_symbol_table`, καταχωρεί το initial thread, και γεμίζει τα metadata. Στο τέλος της εκτελείται η `ompd_dll_locations_valid` ως notification breakpoint, σταματώντας τυπικά τον debugger ο οποίος έχει βάλει εκεί breakpoint από το ξεκίνημα. Σε αυτό το σημείο ο debugger φορτώνει τη `libompd.so`, καλεί `ompd_initialize` και κατόπιν `ompd_process_initialize`. Από εκεί και κάτω, για κάθε `#pragma omp parallel`, το runtime καλεί διαδοχικά την `ompd_parallel_begin` (που ενημερώνει το ompd state και το list των parallel regions) και την `ompd_bp_parallel_begin` (notification breakpoint). Αντίστοιχα ζευγάρια υπάρχουν για το τέλος του parallel region και για τα tasks. Όταν τελικά καλείται η `exit()`, η `ompd_state_finalize` του runtime απελευθερώνει τους πόρους και ο debugger καλεί `ompd_finalize` για να καθαρίσει την DLL.

Αυτή η σειρά εξασφαλίζει ένα πράγμα που είναι λιγότερο προφανές απ' όσο μοιάζει: η libompd δεν φορτώνεται ποτέ πριν το runtime έχει ολοκληρώσει την αρχικοποίηση του `ompd_symbol_table`. Αν ο debugger προσπαθούσε να φορτώσει το DLL νωρίτερα, η `ompd_process_initialize` θα αποτύγχανε είτε με `ompd_rc_incompatible` (επειδή το `version_major` θα ήταν ακόμα μηδέν), είτε με σφάλμα στο symbol lookup. Το breakpoint στην `ompd_dll_locations_valid` είναι ακριβώς το σημείο όπου ο debugger μαθαίνει ότι μπορεί να προχωρήσει.

Κεφάλαιο 5: Thread Handles και Thread State

Κάθε νήμα ενός OpenMP program έχει μια ταυτότητα από δύο οπτικές: από τη μεριά του runtime, είναι ένα `ort_eecb_t` με επίπεδο, θέση στην ομάδα και τρέχον task· από τη μεριά του debugger, είναι ένα opaque handle που τον αντιπροσωπεύει χωρίς να του επιτρέπει να δει τα εσωτερικά. Εδώ εξηγείται πώς συνδέονται οι δύο πλευρές: ποιες δομές διατηρεί το runtime ώστε ο debugger να μπορεί να εντοπίσει και να εξετάσει οποιοδήποτε νήμα, και πώς υλοποιούνται οι πέντε συναρτήσεις thread handles που ορίζει το §5.5.5 του προτύπου.

5.1 Δομές δεδομένων νημάτων

Η υποστήριξη thread handles στηρίζεται σε δύο συμπληρωματικές δομές. Η πρώτη ζει στη μεριά του runtime και αποτελεί κόμβο της linked list που κρατά ο OMPi για κάθε ενεργό νήμα. Η δεύτερη ζει στη μεριά του debugger, εκχωρείται από τη libompd κατά request, και δεν έχει δικαίωμα να δει τα εσωτερικά headers του runtime.

5.1.1 ompd_thread_node_t

Κάθε ενεργό νήμα OpenMP αντιπροσωπεύεται από έναν κόμβο στη globally linked list `ompd_thread_list` που διατηρείται στη libort:

```
typedef struct ompd_thread_node_s {
    uint64_t                    native_tid; /* pthread_t as uint64 */
    void                       *eecb;       /* ort_eecb_t* in target */
    struct ompd_thread_node_s *next;
} ompd_thread_node_t;
```

Το πεδίο `native_tid` αποθηκεύει το `pthread_t` του νήματος μετατραπμένο σε `uint64_t` — ο λόγος είναι ότι το πρότυπο OMPD ορίζει τα thread IDs ως `uint64_t` ανεξάρτητα πλατφόρμας. Χρησιμεύει ως κλειδί αναζήτησης: όταν ο debugger ζητά handle για ένα native thread ID, η libompd διατρέχει τη λίστα συγκρίνοντας `node->native_tid` με το ID που έδωσε.

Το πεδίο `eecb` είναι δείκτης στο `ort_eecb_t` (Thread Control Block) του νήματος. Μέσω αυτού η libompd διαβάζει ICVs, επίπεδο nesting, τρέχον task και άλλες πληροφορίες για το νήμα. Η λίστα προστατεύεται από το `ompd_state_lock` και η κεφαλή της αντικατοπτρίζεται στο πεδίο `thread_list_head` του `ompd_symbol_table`, ώστε να είναι ορατή στη libompd μέσω ανάγνωσης μνήμης.

5.1.2 ort_eecb_t — Thread Control Block

Η `ort_eecb_t` (Execution Environment Control Block) είναι η κεντρική δομή του OMPi για κάθε νήμα. Ορίζεται στο `libort/ort_prive.h` και τα πεδία που

χρησιμοποιεί το OMPD είναι τα `thread_num` (θέση στην ομάδα, 0 = master), `num_siblings` (μέγεθος ομάδας), `level` (συνολικό επίπεδο nesting), `activelevel` (μόνο τα μη-εκφυλισμένα επίπεδα), `parent` (δείκτης στο EECB του γονικού νήματος), και `tasking.current_executing_task` (τρέχον task).

Το `current_executing_task` αξίζει ιδιαίτερη αναφορά: βρίσκεται σε nested struct (`ort_eecb_t.tasking.current_executing_task`), οπότε το offset του δεν μπορεί να υπολογιστεί αυτόματα από τη μακροεντολή `FIELD` που είδαμε στο Κεφ. 4. Υπολογίζεται ρητά στην `ompd_state_init()`:

```
uint32_t cet_off = (uint32_t)(
    offsetof(ort_eecb_t, tasking) +
    offsetof(ort_tasking_t, current_executing_task));
strncpy(eecb_meta.fields[idx].name, "current_executing_task", 63);
eecb_meta.fields[idx].offset = cet_off;
eecb_meta.fields[idx].size   = sizeof(void *);
```

5.1.3 _ompd_thread_handle

Το handle που επιστρέφεται στον debugger ορίζεται στο `ompd_types.h` και η μνήμη του εκχωρείται μέσω `g_cbs->alloc_memory`, οπότε ανήκει στον debugger:

```
struct _ompd_thread_handle {
    ompd_address_space_handle_t *ah;             /* address space
handle */
    ompd_thread_context_t       *thread_context; /* tool-provided
context */
    ompd_address_t               eecb_addr;      /* target ort_eecb_t
addr */
    uint64_t                     native_id;
    ompd_size_t                  native_id_size;
};
```

Το `eecb_addr` είναι ο κεντρικός σύνδεσμος μεταξύ handle και runtime: κάθε ICV query, task query και state query χρησιμοποιεί αυτή τη διεύθυνση ως αφετηρία για ανάγνωση πεδίων του `ort_eecb_t` μέσω της `dll_read_field()`. Αν το `eecb_addr` δεν έχει επιλυθεί (παραμένει 0), οι queries επιστρέφουν `ompd_rc_unavailable` χωρίς crash.

5.2 Διαχείριση thread list (runtime side)

Η `ompd_thread_list` ενημερώνεται από τρεις hook συναρτήσεις στο `ompd_state.c`, που καλούνται από το `parallel.c` υπό `OMPD_ENABLED`. Καλύπτουν τον πλήρη κύκλο ζωής ενός νήματος μέσα σε parallel region.

5.2.1 ompd_thread_begin

Καλείται από το `parallel.c` για κάθε μη-master νήμα αφού το EECB του έχει πλήρως αρχικοποιηθεί. Η διαδικασία είναι η εξής: εκχωρείται κόμβος `ompd_thread_node_t` με `malloc`, συμπληρώνονται τα πεδία `native_tid` και `eecb`, και υπό `ompd_state_lock` ο κόμβος προστίθεται στην αρχή της `ompd_thread_list`, ενημερώνεται το `thread_list_head` στον `ompd_symbol_table`, και αυξάνεται ο `thread_count`. Αν η εκχώρηση αποτύχει, η `thread_count` ενημερώνεται κι εκεί ώστε να παραμένει συνεπής — η συνάρτηση δεν κάνει abort. Τελικά καλείται το `ompd_bp_thread_begin()`, όπου ο debugger μπορεί να τοποθετήσει breakpoint.

5.2.2 ompd_thread_end

Καλείται πριν το EECB του νήματος ανακτηθεί. Υπό `ompd_state_lock` γίνεται γραμμική αναζήτηση στη λίστα με κριτήριο `cur->eecb == eecb`, ο κόμβος αποσυνδέεται, ενημερώνεται το `thread_list_head`, και η `thread_count` μειώνεται με guard `> 0` για αποφυγή underflow. Αν ο κόμβος δεν βρεθεί (δεν θα έπρεπε να συμβεί), η `thread_count` μειώνεται κι εκεί και το breakpoint καλείται κανονικά. Στο τέλος εκτελείται το `ompd_bp_thread_end()`.

5.2.3 ompd_parallel_thread_join

Καλείται για κάθε νήμα (master και workers) κατά την είσοδό του στην parallel region. Ρόλος της είναι να γεμίσει τον πίνακα `team_members[]` της parallel region, που χρησιμοποιεί η `ompd_get_thread_in_parallel` για να βρει το i-οστό νήμα μιας ομάδας. Υπό lock, εκτελείται `team_members[thread_num] = eecb`.

Το master νήμα χρειάζεται ειδική μεταχείριση: δεν δημιουργεί νέο κόμβο, αλλά ενημερώνει τον υπάρχοντα. Η λογική είναι ότι το master είναι ήδη στη `ompd_thread_list` με το εξωτερικό του EECB· όταν μπει σε parallel region, αποκτά νέο EECB. Για να παραμείνει ο κόμβος ακριβής, αποθηκεύεται το παλιό EECB στο `master_prev_eecb` της parallel region και ο `node->eecb` ενημερώνεται με το νέο:

```
/* ompd_state.c — ompd_parallel_thread_join(), master special case */
if (thread_num == 0) {
    uint64_t self = (uint64_t)(uintptr_t)pthread_self();
```

```
    for (node = ompd_thread_list; node; node = node->next) {
        if (node->native_tid == self) {
            ompd_parallel_list->master_prev_eecb = node->eecb;
            node->eecb = eecb;  /* update to inner EECB */
            break;
        }
    }
}
```

Όταν τελειώσει η parallel region, η `ompd_parallel_end()` αναστρέφει αυτή την ενέργεια: επαναφέρει `node->eecb = master_prev_eecb` ώστε ο master να εμφανίζεται ξανά με το εξωτερικό EECB του.

5.3 Συναρτήσεις thread handles (DLL side)

Η libompd εκθέτει πέντε συναρτήσεις για τη διαχείριση thread handles, οι οποίες αντιστοιχούν στο §5.5.5 του προτύπου.

Η `ompd_get_thread_handle` δημιουργεί handle από native thread ID. Αφού εκχωρήσει μνήμη για το handle, διατρέχει την `ompd_thread_list` στη μνήμη του target διαβάζοντας έναν-έναν τους κόμβους:

```
while (node_ptr != 0) {
    dll_read(ah, node_ptr,                          8, &node_tid);
    dll_read(ah, node_ptr + 8,              ptr_size, &node_eecb);
    dll_read(ah, node_ptr + 8 + ptr_size, ptr_size, &node_next);
    if (node_tid == th->native_id) {
        th->eecb_addr.address = node_eecb;
        break;
    }
    node_ptr = node_next;
}
```

Επειδή η libompd δεν συμπεριλαμβάνει εσωτερικά headers του runtime, η πλοήγηση γίνεται με ρητά offsets (8 bytes για `native_tid`, `ptr_size` bytes για `eecb` και `next`). Αξίζει να σημειωθεί ότι η συνάρτηση επιστρέφει `ompd_rc_ok` ακόμα και αν δεν βρεθεί το νήμα στη λίστα και το `eecb_addr` παραμείνει 0 — το handle είναι έγκυρο, απλώς μετέπειτα queries θα επιστρέψουν `ompd_rc_unavailable`. Πρόκειται για σκόπιμη fault-tolerant σχεδίαση.

Η `ompd_get_thread_in_parallel` επιστρέφει handle για το i-οστό νήμα μιας ομάδας, διαβάζοντας `team_members[thread_num]` από τη δομή της parallel region.

Επειδή δεν έχει πρόσβαση σε native OS ID, επιστρέφει handle με `thread_context = NULL`. Αυτό σημαίνει ότι η `ompd_get_thread_id` δεν λειτουργεί για τέτοια handles, αλλά τα ICV queries, τα task queries και η κατάσταση νήματος λειτουργούν κανονικά μέσω του `eecb_addr`.

Η `ompd_rel_thread_handle` αποδεσμεύει το handle καλώντας `free_memory` στον thread context και στο ίδιο το handle. Η `ompd_thread_handle_compare` συγκρίνει δύο handles βάσει `eecb_addr.address`, ώστε handles από διαφορετικές πηγές (π.χ. ένα από `ompd_get_thread_handle` και ένα από `ompd_get_thread_in_parallel`) να συγκρίνονται ορθά αν αναφέρονται στο ίδιο νήμα. Τέλος, η `ompd_get_thread_id` επιστρέφει αντίγραφο του `native_id` που αποθηκεύτηκε στο handle κατά τη δημιουργία του — η τρέχουσα υλοποίηση ελέγχει το `kind` και επιστρέφει `ompd_rc_unsupported` αν δεν είναι `ompd_thread_id_pthread` , που είναι ο μόνος τύπος που υποστηρίζεται από το pthreads backend του OMPi.

5.4 Απαρίθμηση και ανάκτηση κατάστασης νήματος

5.4.1 ompd_enumerate_states

Η `ompd_enumerate_states` επιτρέπει στον debugger να ανακαλύψει ποιες καταστάσεις νήματος υποστηρίζει αυτή η υλοποίηση, χωρίς να τις κωδικοποιεί εκ των προτέρων. Ακολουθεί το cursor pattern του προτύπου: `current_state = 0` σημαίνει «ξεκίνα από την αρχή», κάθε κλήση επιστρέφει το επόμενο στοιχείο μαζί με το `more_enums` flag που δηλώνει αν υπάρχουν κι άλλα.

Υποστηρίζονται επτά καταστάσεις: `ompt_state_work_serial` (0x001) για εκτέλεση σειριακού κώδικα, `ompt_state_work_parallel` (0x002) για εκτέλεση μέσα σε parallel region, `ompt_state_idle` (0x004) για αδρανές νήμα σε αναμονή εργασίας, `ompt_state_wait_barrier` (0x020), `ompt_state_wait_taskwait` (0x040), `ompt_state_wait_taskgroup` (0x080) και `ompt_state_wait_mutex` (0x100). Τα ονόματα ακολουθούν την ονοματολογία OMPT για συμβατότητα με debuggers που γνωρίζουν και τις δύο διεπαφές. Αν το `current_state` δεν βρεθεί στον πίνακα, η συνάρτηση επιστρέφει `ompd_rc_bad_input`.

5.4.2 ompd_get_state

Η `ompd_get_state` επιστρέφει την τρέχουσα κατάσταση ενός νήματος. Η παρούσα υλοποίηση είναι stub που επιστρέφει πάντα `ompt_state_work_parallel` (0x002) και `wait_id = 0`. Η πλήρης υλοποίηση απαιτεί per-thread state tracking ενσωματωμένο στο runtime: κάθε νήμα θα πρέπει να διατηρεί μια μεταβλητή στο

`ort_eecb_t` που ενημερώνεται κατά την είσοδο σε barrier, taskwait, mutex και άλλες καταστάσεις αναμονής. Αυτό προϋποθέτει ουσιαστικά ενσωμάτωση στοιχείων της OMPT[5] διεπαφής στο runtime και παραμένει για μελλοντική εργασία.

5.5 ICV queries με scope = thread

Η πλήρης ανάλυση των ICV queries γίνεται στο Κεφάλαιο 7· εδώ σημειώνουμε μόνο πώς χρησιμοποιείται το thread handle ως αφετηρία. Για `scope = thread`, δύο ICVs διαβάζονται απευθείας από το EECB: το `ompd_icv_level_var` μέσω του πεδίου `level` (συνολικό επίπεδο nesting) και το `ompd_icv_active_levels_var` μέσω του `activelevel` (μόνο τα μη-εκφυλισμένα επίπεδα). Και για τα δύο χρησιμοποιείται η `dll_read_field(ah, eecb_addr, eecb_meta, field_name, &val, 4)`.

Οι υπόλοιπες ICVs, όπως `nthreads`, `dynamic`, `schedule` και `bind`, διαβάζονται από το `ort_task_icvs_t` του τρέχοντος task: πρώτα διαβάζεται το `current_executing_task` pointer από το EECB, και στη συνέχεια τα πεδία του `task_icvs_t` στη θέση `task_ptr + task_icvs_offset`.

5.6 Περιορισμοί

Τρεις περιορισμοί στα thread handles. Πρώτος, η `ompd_get_state` επιστρέφει πάντα `ompt_state_work_parallel`, επειδή απαιτεί per-thread state tracking στο `ort_eecb_t` που δεν έχει υλοποιηθεί ακόμα. Δεύτερος, η `ompd_get_thread_id` δεν υποστηρίζει άλλα IDs πέρα των pthreads (πχ LWP), οπότε σε αυτά δίνει αποτέλεσμα `ompd_rc_unsupported`. Τρίτος, τα handles που παράγει η `ompd_get_thread_in_parallel` έχουν `thread_context = NULL`, οπότε η `ompd_get_thread_id` επιστρέφει `ompd_rc_bad_input` για αυτά — οι υπόλοιπες λειτουργίες λειτουργούν κανονικά.

Κεφάλαιο 6: Parallel Region Handles

Για να μπορεί ο debugger να εξετάζει μια ομάδα νημάτων, χρειάζεται ένα αντικείμενο που συγκεντρώνει τα στοιχεία της ομάδας ως σύνολο: το μέγεθός της, το επίπεδο nesting, τα EECB των μελών της. Αυτό το αντικείμενο είναι η `ompd_parallel_region_t`, μια νέα δομή που εισήχθη αποκλειστικά για τις ανάγκες του OMPD. Γύρω από αυτήν οργανώνεται το κεφάλαιο: η δομή της, τα runtime hooks που τη συντηρούν ενημερωμένη, οι συναρτήσεις της libompd που εξυπηρετούν τα ερωτήματα του debugger, και τα σημεία κλήσης στο `parallel.c`.

6.1 Η δομή ompd_parallel_region_t

Το βασικό runtime του OMPi δεν κρατούσε ενιαία αναπαράσταση μιας parallel region: οι σχετικές πληροφορίες ήταν σκορπισμένες στα `ort_eecb_t` των νημάτων. Για το OMPD αυτό δεν αρκεί, γιατί ο debugger χρειάζεται να απαντά σε ερωτήματα τύπου «πόσα νήματα έχει αυτή η ομάδα» χωρίς να σαρώνει ολόκληρη τη λίστα νημάτων. Εισήχθη λοιπόν η `ompd_parallel_region_t` στο `ompd_types.h`:

```
typedef struct ompd_parallel_region_s {
    ompd_addr_t  master_eecb_addr;          /* search key    */
    struct ompd_parallel_region_s *parent;  /* parent region       */
    struct ompd_parallel_region_s *next;    /* next in list */
    int           team_size;
    int           level;
    void         *shared_data;
    void        **team_members;             /* ort_eecb_t array */
    void         *master_prev_eecb;
} ompd_parallel_region_t;
```

Το `master_eecb_addr` είναι η διεύθυνση του `ort_eecb_t` του master νήματος, αποθηκευμένη ως ακέραιος 64-bit. Χρησιμεύει ως μοναδικό κλειδί αναζήτησης στην `ompd_parallel_end()`: ο master γνωρίζει το δικό του EECB και το χρησιμοποιεί για να βρει και να αφαιρέσει τον κόμβο από τη λίστα.

Τα πεδία `parent` και `next` σχηματίζουν τη δομή λίστας. Το `next` συνδέει τους κόμβους σε μονής κατεύθυνσης λίστα με κεφαλή την `ompd_parallel_list`. Το `parent` δείχνει στη γονική parallel region, επιτρέποντας στη libompd να πλοηγηθεί στην ιεραρχία nesting μέσω `ompd_get_enclosing_parallel_handle()`.

Τα `team_size` και `level` ανακτώνται απευθείας από τα ερωτήματα `ompd_get_parallel_info()` και από ICV queries με scope=parallel. Ο πίνακας

`team_members` εκχωρείται με `calloc(team_size, sizeof(void*))` και συμπληρώνεται σταδιακά: κάθε νήμα εγγράφει το EECB του στη θέση `team_members[thread_num]` κατά την εκκίνησή του, μέσω της hook `ompd_parallel_thread_join()`. Μόλις ο πίνακας είναι πλήρης, η `ompd_get_thread_in_parallel()` μπορεί να εντοπίσει οποιοδήποτε νήμα της ομάδας. Το `master_prev_eecb` αποθηκεύει το αρχικό EECB του master πριν εισέλθει στην parallel region, ώστε να αποκατασταθεί στη `thread_list` μετά το πέρας της.

Η λίστα `ompd_parallel_list` ταξινομείται innermost-first: ο πιο πρόσφατος κόμβος βρίσκεται πάντα στην κεφαλή. Αυτό σημαίνει ότι η `ompd_get_curr_parallel_handle()` επιστρέφει πάντα την εσώτερη ενεργή region απλώς διαβάζοντας την κεφαλή, χωρίς επαναληπτική αναζήτηση.

Το DLL-side handle `_ompd_parallel_handle` είναι σκόπιμα απλό:

```
struct _ompd_parallel_handle {
    ompd_address_space_handle_t *ah;
    ompd_address_t                region_addr;   /* target ompd_parallel_region_t */
};
```

Αποθηκεύει μόνο τη διεύθυνση της `ompd_parallel_region_t` στο target. Κάθε πεδίο διαβάζεται on-demand μέσω `dll_read()` και `offsetof()`, οπότε το handle αντικατοπτρίζει πάντα την τρέχουσα κατάσταση στη μνήμη.

6.2 Runtime hooks (ompd_state.c)

6.2.1 ompd_parallel_begin

Η `ompd_parallel_begin()` καλείται από το `parallel.c` μόνο για πραγματικά παράλληλες εκτελέσεις, δηλαδή όταν `team_size > 1`. Οι serial regions παρακάμπτονται, γιατί δεν δημιουργούν ομάδα νημάτων και δεν χρειάζεται να εμφανίζονται στον debugger.

Η συνάρτηση εκχωρεί έναν νέο κόμβο `ompd_parallel_region_t` με `malloc`, συμπληρώνει τα πεδία `master_eecb_addr`, `team_size` και `shared_data`, και εκχωρεί τον πίνακα `team_members` με `calloc(team_size, sizeof(void*))`. Στη συνέχεια εισάγει τον κόμβο στη λίστα `ompd_parallel_list` υπό `ompd_state_lock`: ορίζει `parent` ως την τρέχουσα κεφαλή, ενημερώνει το `level` ως `parent->level + 1` (ή 1 αν δεν υπάρχει γονική region), και τοποθετεί τον νέο κόμβο στην κεφαλή. Μετά την αποδέσμευση του lock καλείται η `ompd_bp_parallel_begin()`, ώστε ο debugger να βλέπει συνεπή κατάσταση όταν σταματά εκεί.

6.2.2 ompd_parallel_end

Η `ompd_parallel_end()` καλείται σε τρία διαφορετικά σημεία του `parallel.c` [γραμμές 388, 414, 455] — ένα για κάθε διαδρομή εξόδου: task-based, loop-based και άμεση εκτέλεση. Η τριπλή τοποθέτηση δεν είναι επανάληψη, αλλά αρχιτεκτονική απόφαση: ο OMPi έχει τρεις ξεχωριστές διαδρομές εξόδου από μια parallel region και η OMPD υλοποίηση πρέπει να καλύπτει και τις τρεις, ώστε κανένας region descriptor να μην παραμένει στη λίστα ανεξάρτητα από τον τρόπο εξόδου.

Εσωτερικά, η συνάρτηση σαρώνει τη λίστα αναζητώντας τον κόμβο με `master_eecb_addr` ίσο με τη διεύθυνση του master EECB. Αν δεν βρεθεί — αυτό συμβαίνει για serial regions που παρέλειψαν την `ompd_parallel_begin()` — η συνάρτηση επιστρέφει αθόρυβα. Όταν βρεθεί, αποσυνδέει τον κόμβο, επαναφέρει το master EECB στη `thread_list` αν `master_prev_eecb != NULL`, αποδεσμεύει τον πίνακα `team_members` και τον κόμβο, και καλεί `ompd_bp_parallel_end()`.

6.2.3 Notification breakpoints

Τα `ompd_bp_parallel_begin()` και `ompd_bp_parallel_end()` ορίζονται ως κενές `noinline` συναρτήσεις με compiler memory barrier. Καλούνται μετά την αποδέσμευση του `ompd_state_lock` για έναν σαφή λόγο: ο debugger σταματά σε αυτά τα breakpoints και, για να διαβάσει τη `ompd_parallel_list`, στέλνει ερωτήματα OMPD στο target — αν το lock ήταν ακόμα κρατημένο, κάθε τέτοιο ερώτημα θα κόλλαγε.

6.3 DLL-side συναρτήσεις parallel handles

6.3.1 ompd_get_curr_parallel_handle

Επιστρέφει handle για την εσώτερη ενεργή parallel region. Αναζητά τη διεύθυνση της καθολικής λίστας `ompd_parallel_list` στο target μέσω `symbol_addr_lookup`, διαβάζει την κεφαλή με `dll_read()`, και αν είναι μη-μηδενική δημιουργεί ένα νέο handle με `region_addr.address` ίσο με αυτή την κεφαλή. Αν η λίστα είναι κενή, επιστρέφεται `ompd_rc_unavailable`.

6.3.2 ompd_get_enclosing_parallel_handle

Πλοηγείται στην ιεραρχία nesting επιστρέφοντας handle για τη γονική parallel region. Διαβάζει το πεδίο `parent` από την `ompd_parallel_region_t` που δείχνει το δοθέν handle: αν είναι μηδέν, η region είναι στο εξώτατο επίπεδο και επιστρέφεται `ompd_rc_unavailable`. Αλλιώς δημιουργεί handle για τη γονική region. Η αλυσίδα `parent → parent → …` αντιστοιχεί ακριβώς στη δυναμική ιεραρχία nesting, ακόμα και αν έχουν δημιουργηθεί και αποδεσμευτεί ενδιάμεσες regions.

6.3.3 ompd_get_task_parallel_handle

Επιστρέφει handle για την parallel region που περιβάλλει ένα task. Στην παρούσα υλοποίηση ακολουθεί την ίδια λογική με την `ompd_get_curr_parallel_handle()`: επιστρέφει την κεφαλή της `ompd_parallel_list`. Σε μελλοντική φάση θα χρησιμοποιεί δείκτη από τον κόμβο `ort_task_node_t` προς τη συγκεκριμένη parallel region στην οποία δημιουργήθηκε το task.

6.3.4 ompd_rel_parallel_handle

Αποδεσμεύει ένα parallel handle καλώντας `free_memory`. Το handle δεν κατέχει άλλους πόρους: ο back-pointer `ah` ανήκει στον debugger και το `region_addr` είναι απλώς ένας ακέραιος.

6.3.5 ompd_parallel_handle_compare

Συγκρίνει δύο parallel handles βάσει του `region_addr.address`, δηλαδή της διεύθυνσης της `ompd_parallel_region_t` στη μνήμη του target. Κάθε region έχει μοναδική `malloc`'d διεύθυνση, οπότε η σύγκριση διευθύνσεων ισοδυναμεί με σύγκριση ταυτότητας region. Επιστρέφει -1, 0 ή 1 στο `*cmp_value`.

6.3.6 ompd_get_parallel_info

Ανακτά το μέγεθος ομάδας και το επίπεδο nesting μιας parallel region. Τα δύο ορίσματα εξόδου είναι ανεξάρτητα προαιρετικά — αν κάποιο είναι `NULL`, παραλείπεται. Η ανάγνωση γίνεται μέσω `dll_read()` στις θέσεις `region_addr + offsetof(ompd_parallel_region_t, team_size)` και `…level` αντίστοιχα. Η χρήση `offsetof()` εξασφαλίζει φορητή πρόσβαση ανεξάρτητα από compiler alignment: ακόμα και αν προστεθούν πεδία στη δομή μελλοντικά, οι υφιστάμενοι υπολογισμοί παραμένουν σωστοί.

6.4 ompd_get_thread_in_parallel

Η `ompd_get_thread_in_parallel()` ανήκει τυπικά στις thread functions, αλλά χρησιμοποιεί αποκλειστικά τη δομή `ompd_parallel_region_t` και περιγράφεται εδώ. Επιτρέπει στον debugger να αποκτήσει thread handle για οποιοδήποτε νήμα μιας ομάδας χωρίς να γνωρίζει το native OS ID του.

Αρχικά ελέγχεται αν το `thread_num` είναι εντός ορίων: αν `thread_num < 0` ή `thread_num >= team_size`, επιστρέφεται `ompd_rc_bad_input`. Το `team_size` διαβάζεται από την `ompd_parallel_region_t` μέσω `dll_read()`. Στη συνέχεια διαβάζεται ο δείκτης `team_members` από τη δομή: αν είναι μηδέν, η `ompd_parallel_thread_join()` δεν έχει εκτελεστεί ακόμα και επιστρέφεται

`ompd_rc_unavailable`. Αλλιώς διαβάζεται το EECB pointer από τη θέση `team_members[thread_num]` και δημιουργείται το thread handle.

Τα handles που προέρχονται από αυτή τη συνάρτηση δεν έχουν native OS thread ID (`native_id_size == 0`), οπότε μια επακόλουθη κλήση `ompd_get_thread_id()` επιστρέφει `ompd_rc_unavailable`. Αυτό είναι αναμενόμενο: ο OMPi δεν αποθηκεύει per-thread το `pthread_t` στη δομή `ompd_parallel_region_t` και αυτή η πληροφορία δεν είναι διαθέσιμη μέσω του πίνακα `team_members`.

Σημείωση σχετικά με το χρόνο: ο πίνακας `team_members[]` συμπληρώνεται σταδιακά καθώς τα νήματα ξεκινούν. Αν ο debugger σταματήσει πολύ νωρίς — πριν εκτελεστεί η `ompd_parallel_thread_join()` για κάποια νήματα — μερικές θέσεις μπορεί να είναι ακόμα NULL, και η συνάρτηση επιστρέφει `ompd_rc_unavailable` γι' αυτές.

6.5 Ενσωμάτωση στο parallel.c

Ο Πίνακας 6.1 συνοψίζει όλα τα σημεία κλήσης των OMPD hooks στο `parallel.c`.

Γραμμή	Κλήση hook	Συνθήκη	Σκοπός
33–35	`#include "ompd/ompd_state.h"`	#ifdef OMPD_ENABLED	Δήλωση prototypes
104	`ompd_thread_begin(t)`	eeid != 0	Εγγραφή worker thread στη thread_list
104	`ompd_parallel_thread_join(t, eeid)`	πάντα	Συμπλήρωση team_members[eeid]
122	`ompd_thread_end(t)`	eeid != 0	Αφαίρεση worker thread από thread_list
188	`ompd_parallel_begin(me, teamsize)`	teamsize > 1	Εισαγωγή νέας parallel region
388, 414, 455	`ompd_parallel_end(me)`	πάντα	Αφαίρεση region (3 exit paths)

Πίνακας 6.1: Σημεία κλήσης OMPD hooks στο libort/parallel.c.

6.7 Περιορισμοί και μελλοντική εργασία

Η `ompd_get_curr_parallel_handle()` επιστρέφει πάντα την innermost ενεργή parallel region παγκοσμίως — την κεφαλή της `ompd_parallel_list` — και όχι αυτήν στην οποία εκτελείται το συγκεκριμένο νήμα. Σε προγράμματα με βαθιά nested parallelism και

ταυτόχρονες εξωτερικές regions αυτό δίνει λανθασμένο αποτέλεσμα· χρειάζεται per-thread παρακολούθηση της ενεργής region.

Η `ompd_get_task_parallel_handle()` δεν ανακτά την πραγματική parallel region στην οποία δημιουργήθηκε το task, αλλά επιστρέφει απλώς την innermost ενεργή region μέσω της ίδιας λογικής με την `ompd_get_curr_parallel_handle()`. Για σωστή υλοποίηση χρειάζεται δείκτης από τον κόμβο `ort_task_node_t` προς τη parallel region στην οποία δημιουργήθηκε το task.

Κεφάλαιο 7: Task Handles και ICV Queries

Το OMPD API χωρίζει τα queries σε δύο κατηγορίες που συνδέονται στενά: τη διαχείριση task handles (§5.5.10 του προτύπου) και την ανάκτηση Internal Control Variable (ICV) τιμών ανά scope (§5.5.9). Οι δύο κατηγορίες αλληλοσυνδέονται γιατί ορισμένα ICV queries χρησιμοποιούν task handles ως σημείο αναφοράς, και η λογική dispatch ανά scope ενσωματώνει πρόσβαση τόσο σε EECBs όσο και σε task nodes. Το κεφάλαιο αυτό καλύπτει και τα δύο: πρώτα τις δομές δεδομένων και τα hooks, μετά τις DLL συναρτήσεις, και στο τέλος τον μηχανισμό ICV queries.

7.1 Δομές δεδομένων tasks

7.1.1 ort_task_node_t και ort_task_icvs_t

Ο OMPi αναπαριστά κάθε task ως κόμβο τύπου `ort_task_node_t` (ορισμός στο `libort/ort_prive.h`). Η libompd δεν συμπεριλαμβάνει απευθείας αυτό το header, αλλά πλοηγείται στη δομή μέσω ενός metadata descriptor που αρχικοποιεί η `ompd_state_init()`. Ο Πίνακας 7.1 παρουσιάζει τα πεδία που εκθέτει η υλοποίηση.

Πεδίο	Τύπος	Σημασία για OMPD
`func`	`void *(*)(void *)`	Η outlined task function (NULL για implicit task).
`parent`	`ort_task_node_t *`	Generating task — ο κόμβος που δημιούργησε αυτό το task.
`isfinal`	`int`	Αν == 1, το task εκτελείται final (χωρίς νέα tasks).
`num_children`	`volatile int`	Αριθμός ενεργών child tasks.
`rtid`	`int`	Runtime task ID.

Πίνακας 7.1: Πεδία ort_task_node_t που εκθέτει η OMPD υλοποίηση.

Τα per-task ICVs αποθηκεύονται σε nested struct τύπου `ort_task_icvs_t` μέσα στο `ort_task_node_t`. Η libompd ανακτά τις τιμές τους μέσω ενός δεύτερου metadata descriptor, `task_icvs_meta`, που αρχικοποιεί η `ompd_state_init()` δηλώνοντας τα πεδία `nthreads`, `dynamic`, `rtschedule`, `threadlimit` και `proc_bind`. Η διεύθυνση της `ort_task_icvs_t` υπολογίζεται ως `task_addr + task_icvs_offset`, όπου `task_icvs_offset` είναι το `offsetof(ort_task_node_t, icvs)` που αποθηκεύεται στον πίνακα συμβόλων κατά την αρχικοποίηση.

7.1.2 _ompd_task_handle

Το DLL-side handle `_ompd_task_handle` αποθηκεύει μόνο τη διεύθυνση του `ort_task_node_t` στο target:

```
struct _ompd_task_handle {
    ompd_address_space_handle_t *ah;
    ompd_address_t               task_addr;  /* target ort_task_node_t
*/
};
```

Κάθε πεδίο διαβάζεται on-demand μέσω `dll_read_field()`. Η ταυτότητα ενός task handle καθορίζεται αποκλειστικά από το `task_addr.address`.

7.2 Runtime hooks για tasks (ompd_state.c)

Τρεις hook συναρτήσεις καλούνται από το `tasks.c` υπό `OMPD_ENABLED`. Η `ompd_task_create()` [tasks.c:~176] καλείται μετά την εισαγωγή του task στην ουρά εκτέλεσης: αυξάνει τον μετρητή `task_count` υπό lock. Δεν ενεργοποιεί breakpoint, γιατί η δημιουργία task δεν είναι notification event κατά §5.6 του προτύπου.

Η `ompd_task_begin()` καλείται αμέσως πριν την εκτέλεση της task function. Εκεί ο debugger μπορεί να διαβάσει `ompd_get_curr_task_handle()` και να δει το τρέχον task. Ενεργοποιεί `ompd_bp_task_begin()`. Η `ompd_task_end()` καλείται αμέσως μετά την ολοκλήρωση, μειώνει τον `task_count` και ενεργοποιεί `ompd_bp_task_end()`.

Ο μετρητής `task_count` στον πίνακα συμβόλων παρακολουθεί τον αριθμό tasks που έχουν δημιουργηθεί αλλά δεν έχουν ακόμα ολοκληρωθεί. Η πλήρης λίστα `ompd_task_list` παραμένει NULL προς το παρόν — η συντήρησή της αποτελεί μελλοντική εργασία.

7.3 DLL-side συναρτήσεις task handles

7.3.1 ompd_get_curr_task_handle

Επιστρέφει handle για το task που εκτελεί αυτή τη στιγμή το νήμα. Διαβάζει το πεδίο `current_executing_task` από το EECB του νήματος. Αν το EECB δεν έχει επιλυθεί (`eecb_addr.address == 0`) ή αν το πεδίο είναι μηδέν, επιστρέφεται `ompd_rc_unavailable`. Αλλιώς δημιουργείται handle με `task_addr.address` ίσο με τον δείκτη που διαβάστηκε. Η συμπεριφορά αυτή είναι σύμφωνη με το §5.5.10.1 του προτύπου, που επιτρέπει `ompd_rc_unavailable` όταν η πληροφορία δεν είναι διαθέσιμη.

7.3.2 ompd_get_generating_task_handle

Επιστρέφει handle για το task που δημιούργησε το δοθέν task μέσω `#pragma omp task`. Διαβάζει το πεδίο `parent` από το `ort_task_node_t`: αν είναι `NULL`, το task είναι initial implicit task και δεν έχει generating task, οπότε επιστρέφεται `ompd_rc_unavailable`. Η σχέση generating task αποτυπώνεται από το πεδίο `parent`, το οποίο ο OMPi ορίζει κατά τη δημιουργία κάθε task ως δείκτη στο task του νήματος που εκτέλεσε την `#pragma omp task`.

7.3.3 ompd_get_scheduling_task_handle

Το §5.5.10.5 του προτύπου ορίζει τον «scheduling task» ως το task που ήταν ενεργό στο task scheduling point όπου προγραμματίστηκε το δοθέν task. Αυτός μπορεί να διαφέρει από τον generating task σε σενάρια work stealing. Στον OMPi, η συνάρτηση ανακατευθύνεται στη `ompd_get_generating_task_handle()`, γιατί δεν τηρείται ξεχωριστή παρακολούθηση του scheduling task. Η διάκριση αυτή παραμένει για μελλοντική εργασία.

7.3.4 ompd_get_task_in_parallel

Επιστρέφει handle για το task που εκτελεί ένα νήμα μέσα σε parallel region. Αρχικά ελέγχεται ότι `0 ≤ thread_num < team_size`, όπου `team_size` διαβάζεται από την parallel region — αν το `thread_num` είναι εκτός ορίων, επιστρέφεται `ompd_rc_bad_input`. Στη συνέχεια διαβάζεται το EECB pointer από `team_members[thread_num]`, και από το EECB διαβάζεται το `current_executing_task` για να δημιουργηθεί το task handle.

7.3.5 ompd_get_task_function

Επιστρέφει την entry point (function pointer) του task διαβάζοντας το πεδίο `func` από τον task metadata descriptor. Αν το `func` είναι `NULL` — αυτό συμβαίνει για implicit tasks που δεν έχουν outlined function — επιστρέφεται `ompd_rc_unavailable`. Ο debugger μπορεί να χρησιμοποιήσει αυτή τη διεύθυνση για να αναζητήσει στο symbol table την outlined task function και να εμφανίσει stack backtraces για tasks.

7.3.6 ompd_get_task_frame

Επιστρέφει `ompd_rc_unsupported`. Η ανάκτηση stack frame information απαιτεί ενσωμάτωση με DWARF/CFI και πρόσβαση στο call stack του task μέσω OMPT. Το §5.5.10.6 του προτύπου επιτρέπει ρητά `ompd_rc_unsupported` ως έγκυρο return code για αυτή τη συνάρτηση.

7.3.7 ompd_rel_task_handle και ompd_task_handle_compare

Η `ompd_rel_task_handle()` καλεί `free_memory` — το handle δεν κατέχει άλλους πόρους. Η `ompd_task_handle_compare()` συγκρίνει τα `task_addr.address` δύο handles: δύο ίσες διευθύνσεις σημαίνουν το ίδιο `ort_task_node_t`, δηλαδή ταυτόσημα tasks.

7.4 ICV Queries (§5.5.9)

Η `ompd_get_icv_from_scope()` είναι η κεντρικότερη συνάρτηση ICV queries. Δέχεται ένα opaque handle (address space, thread, parallel ή task), ένα scope, και ένα ICV ID, και επιστρέφει αριθμητική τιμή (`ompd_word_t`). Εσωτερικά χρησιμοποιεί dispatch ανά scope.

7.4.1 Οι 14 ICVs της υλοποίησης

Η libompd απαριθμεί 14 ICVs μέσω του static πίνακα `g_icvs[]`.

ICV ID	Όνομα	Scope	Αναγνώσιμη
`ompd_icv_dynamic_var`	dynamic-var	address_space	✓
`ompd_icv_nthreads_var`	nthreads-var	address_space	✓
`ompd_icv_run_sched_var`	run-sched-var	address_space	✓
`ompd_icv_stacksize_var`	stacksize-var	address_space	✓
`ompd_icv_wait_policy_var`	wait-policy-var	address_space	✓
`ompd_icv_thread_limit_var`	thread-limit-var	address_space	✓
`ompd_icv_max_active_levels_var`	max-active-levels-var	address_space	✓
`ompd_icv_cancellation_var`	cancel-var	address_space	✓
`ompd_icv_default_device_var`	default-device-var	address_space	✓
`ompd_icv_num_procs_var`	num-procs-var	address_space	✓
`ompd_icv_schedule_var`	def-sched-var	thread	✓
`ompd_icv_active_levels_var`	active-levels-var	thread	✓
`ompd_icv_level_var`	levels-var	thread	✓
`ompd_icv_bind_var`	bind-var	thread	✓

Πίνακας 7.2: ICVs που απαριθμεί η ompd_enumerate_icvs().

7.4.2 Dispatch ανά scope

Για `scope=global` ή `scope=address_space`, το handle ερμηνεύεται ως `_ompd_aspace_handle*` και ο δείκτης `global_icvs` αναγιγνώσκεται εκ νέου από το live target, όχι από το αποθηκευμένο αντίγραφο, ώστε να αντικατοπτρίζονται αλλαγές που έγιναν κατά την εκτέλεση.

Για `scope=thread`, οι `levels-var` και `active-levels-var` διαβάζονται απευθείας από το EECB (πεδία `level` και `activelevel` αντίστοιχα). Οι υπόλοιπες (schedule, bind, nthreads, dynamic) ανακτώνται από το `ort_task_icvs_t` του `current_executing_task` μέσω μιας εσωτερικής βοηθητικής `icv_from_task_icvs()`.

Για `scope=task` ή `scope=implicit_task`, το handle ερμηνεύεται ως `_ompd_task_handle*` και τα ICVs διαβάζονται απευθείας από το `ort_task_icvs_t` στη διεύθυνση `task_addr + task_icvs_offset`. Η αντιστοίχιση ICV ID → πεδίο καλύπτει `nthreads`, `dynamic`, `rtschedule`, `threadlimit` και `proc_bind`.

Για `scope=parallel`, το handle ερμηνεύεται ως `_ompd_parallel_handle*`. Η `nthreads-var` αντιστοιχεί στο `team_size` της region, και οι `levels-var` και `active-levels-var` αντιστοιχούν και οι δύο στο `level` — ο OMPi θεωρεί κάθε ενεργή parallel region ως ενεργό επίπεδο.

7.5 ompd_get_icv_string_from_scope (§5.5.9.3)

Η συνάρτηση επιστρέφει ονομαστική (symbolic) αναπαράσταση μιας ICV. Εσωτερικά καλεί πρώτα `ompd_get_icv_from_scope()` για να πάρει αριθμητική τιμή, και στη συνέχεια την μετατρέπει σε string. Για `run-sched-var` επιστρέφεται το όνομα του schedule (`static`, `dynamic`, `guided`, `auto`). Για `bind-var` επιστρέφεται το όνομα της τιμής proc_bind (`false`, `true`, `primary`, `close`, `spread`). Για όλες τις άλλες ICVs επιστρέφεται ο αριθμός ως δεκαδικό string. Η μνήμη εκχωρείται μέσω `alloc_memory` και η αποδέσμευση είναι ευθύνη του debugger, σύμφωνα με §5.5.9.3.

7.6 ompd_enumerate_icvs (§5.5.9.1)

Επιτρέπει στον debugger να ανακαλύψει όλες τις ICVs που υποστηρίζει η υλοποίηση. Ακολουθεί το ίδιο cursor pattern με `ompd_enumerate_states()`: η πρώτη κλήση γίνεται με `current_icv = ompd_icv_undefined`, κάθε επόμενη επιστρέφει `next_id`, `next_name`, `next_scope` και `more`. Κάθε `next_name` εκχωρείται νέα μέσω `alloc_memory` και ανήκει στον debugger. Επιστρέφεται `ompd_rc_bad_input` αν το `current_icv` δεν αναγνωριστεί.

7.7 Περιορισμοί και μελλοντική εργασία

Η `ompd_get_task_frame()` επιστρέφει `ompd_rc_unsupported`. Η σωστή υλοποίηση απαιτεί ενσωμάτωση με DWARF stack unwinding και OMPT για tracking των task frame pointers.

Η `ompd_get_scheduling_task_handle()` ανακατευθύνεται στη `ompd_get_generating_task_handle()`. Σε σενάρια work stealing, το scheduling task μπορεί να διαφέρει από το generating task, αλλά ο OMPi δεν τηρεί ξεχωριστή εγγραφή γι' αυτόν.

Η `ompd_get_tool_data()` επιστρέφει `ompd_rc_unsupported`. Απαιτεί ενσωμάτωση OMPT tool data (`ompt_data_t`) στο runtime.

Κεφάλαιο 8: Display Control Variables

Το OpenMP ορίζει ένα σύνολο μεταβλητών περιβάλλοντος που καθορίζουν τη συμπεριφορά ενός προγράμματος από τη στιγμή της εκκίνησής του: `OMP_NUM_THREADS`, `OMP_SCHEDULE`, `OMP_PROC_BIND` και άλλες. Μέσω των Display Control Variables (DCVs) η OMPD παρέχει ένα τυποποιημένο τρόπο ανάγνωσης αυτών των ρυθμίσεων από ένα εργαλείο debugging, χωρίς να χρειάζεται πρόσβαση στο περιβάλλον της επιθεωρούμενης διεργασίας.

Η διαχείριση των DCVs ορίζεται στην §5.5.8 του προτύπου OpenMP 5.1 και υλοποιείται μέσω δύο συναρτήσεων: `ompd_get_display_control_vars()`, που δεσμεύει και επιστρέφει τις τιμές, και `ompd_rel_display_control_vars()`, που αποδεσμεύει τη μνήμη αφού το εργαλείο τελειώσει με αυτές.

8.1 Ορισμός και ρόλος

Τα DCVs αποτελούν ένα υποσύνολο των Internal Control Variables (ICVs) του OpenMP, συγκεκριμένα εκείνα που ελέγχονται μέσω μεταβλητών περιβάλλοντος. Σε αντίθεση με τα ICVs που ανακτώνται από την `ompd_get_icv_from_scope()` και απαιτούν συγκεκριμένα handles νημάτων ή tasks, τα DCVs αφορούν τη διαμόρφωση ολόκληρης της εκτέλεσης και επιστρέφονται ως απλές συμβολοσειρές «ΟΝΟΜΑ=τιμή». Ο χρήστης του debugger αποκτά έτσι με μία κλήση πλήρη εικόνα της αρχικής ρύθμισης.

Σύμφωνα με το πρότυπο (§5.5.8.1), η `ompd_get_display_control_vars()` επιστρέφει ένα NULL-terminated πίνακα συμβολοσειρών μορφής NAME=value. Εάν μια μεταβλητή δεν έχει οριστεί στο περιβάλλον, επιστρέφεται ως NAME= με κενή τιμή, ώστε ο πίνακας να έχει πάντα σταθερό πλήθος στοιχείων ανεξαρτήτως διαμόρφωσης.

Μεταβλητή Περιβάλλοντος	ICV που ελέγχει	Τυπική τιμή
`OMP_NUM_THREADS`	nthreads-var	`4`
`OMP_DYNAMIC`	dyn-var	`false`
`OMP_NESTED`	nest-var (deprecated)	`false`
`OMP_SCHEDULE`	def-sched-var	`static`
`OMP_THREAD_LIMIT`	thread-limit-var	`1024`
`OMP_MAX_ACTIVE_LEVELS`	max-active-levels-var	`2`
`OMP_PROC_BIND`	bind-var	`false`
`OMP_PLACES`	place-partition-var	
`OMP_STACKSIZE`	stacksize-var	

OMP_WAIT_POLICY	wait-policy-var	
OMP_CANCELLATION	cancel-var	false
OMP_DEFAULT_DEVICE	default-device-var	0
OMP_MAX_TASK_PRIORITY	max-task-priority-var	0
OMP_NUM_TEAMS	nteams-var	0
OMP_TEAMS_THREAD_LIMIT	teams-thread-limit-var	0
OMP_TARGET_OFFLOAD	target-offload-var	DEFAULT
OMP_AFFINITY_FORMAT	affinity-format-var	

Οι δεκαεπτά αυτές μεταβλητές καλύπτουν το σύνολο των OMP_* μεταβλητών που διαβάζει η libort: αριθμό νημάτων, δυναμική ρύθμιση, προγραμματισμό εργασιών, ανώτατα επίπεδα παραλληλισμού, δέσμευση νημάτων, τοποθέτηση νημάτων (OMP_PLACES), μέγεθος στοίβας, πολιτική αναμονής, ακύρωση, συσκευές και ομάδες.

8.2 Υλοποίηση ompd_get_display_control_vars()

Η `ompd_get_display_control_vars()` υλοποιείται στο `ompd_dll.c` και δέχεται τη λαβή χώρου διευθύνσεων (`as_handle`) και ένα double-pointer (`control_vars`) μέσα στον οποίο επιστρέφει τον NULL-terminated πίνακα:

```
ompd_rc_t
ompd_get_display_control_vars(
    ompd_address_space_handle_t  *as_handle,
    const char * const           **control_vars);
```

Το κρίσιμο σχεδιαστικό χαρακτηριστικό είναι η χρήση `getenv()` για ανάγνωση των τιμών. Αυτό σημαίνει ότι η συνάρτηση αντικατοπτρίζει τη διαμόρφωση όπως ήταν κατά την εκκίνηση, δηλαδή ό,τι είδε η libort όταν αρχικοποιήθηκε. Αλλαγές που έγιναν μέσω του OpenMP API κατά τη διάρκεια εκτέλεσης, όπως μια κλήση `omp_set_num_threads()`, δεν αντικατοπτρίζονται στα DCVs, γι' αυτό και ο ρόλος τους είναι διαγνωστικός και όχι δυναμικός.

Κάθε συμβολοσειρά «NAME=value» δεσμεύεται μέσω της `dll_alloc()`, που καλεί το callback `alloc_memory` του εργαλείου. Εάν κάποια δέσμευση αποτύχει, η συνάρτηση αποδεσμεύει ό,τι έχει ήδη δεσμευτεί πριν επιστρέψει σφάλμα, εξασφαλίζοντας σωστή συμπεριφορά ακόμα και σε συνθήκες εξάντλησης μνήμης:

```
for (i = 0; i < NDCVARS; i++) {
    const char *env_val = getenv(g_dcvar_names[i]);
    size_t nlen = strlen(g_dcvar_names[i]);
```

```
    size_t vlen = env_val ? strlen(env_val) : 0;
    rc = dll_alloc(nlen + 1 + vlen + 1, (void **)&vec[i]);
    if (rc != ompd_rc_ok) {
        for (j = 0; j < i; j++) dll_free(vec[j]);
        dll_free(vec);
        return rc;
    }
    memcpy(vec[i], g_dcvar_names[i], nlen);
    vec[i][nlen] = '=';
    if (env_val) memcpy(vec[i] + nlen + 1, env_val, vlen + 1);
    else         vec[i][nlen + 1] = '\0';
}
vec[NDCVARS] = NULL;
```

8.3 Υλοποίηση ompd_rel_display_control_vars()

Η `ompd_rel_display_control_vars()` αποδεσμεύει τη μνήμη που δέσμευσε η `ompd_get_display_control_vars()` και θέτει τον pointer σε NULL, αποτρέποντας τυχαία χρήση του:

```
ompd_rc_t
ompd_rel_display_control_vars(const char * const **control_vars)
{
    const char * const *vec;
    int i;
    CHECK_INIT();
    if (!control_vars || !*control_vars)
        return ompd_rc_ok;   /* already NULL — idempotent */
    vec = *control_vars;
    for (i = 0; vec[i] != NULL; i++)
        dll_free((void *)vec[i]);
    dll_free((void *)vec);
    *control_vars = NULL;
    return ompd_rc_ok;
}
```

Η υλοποίηση έχει δύο αξιοσημείωτα χαρακτηριστικά. Πρώτον, είναι idempotent: εάν `control_vars` ή `*control_vars` είναι ήδη NULL, επιστρέφει αμέσως `ompd_rc_ok` χωρίς να δράσει, οπότε είναι ασφαλές να κληθεί δύο φορές. Δεύτερον, η αποδέσμευση ακολουθεί αντίστροφη σειρά από τη δέσμευση: πρώτα ελευθερώνονται οι συμβολοσειρές και τελευταίος ο πίνακας pointers.

8.4 Μοντέλο ιδιοκτησίας μνήμης

Η `libompd.so` δεσμεύει όλη τη μνήμη μέσω του callback `alloc_memory` του εργαλείου, χρησιμοποιώντας την εσωτερική wrapper `dll_alloc()`. Έτσι, αν και η δέσμευση γίνεται από τη `libompd.so`, ο χώρος ανήκει στον allocator του εργαλείου. Το εργαλείο δεν πρέπει να καλεί `free()` απευθείας στον επιστρεφόμενο πίνακα, καθώς αυτό θα παράκαμπτε την αφαίρεση του OMPD. Η μοναδική ορθή αποδέσμευση γίνεται μέσω της `ompd_rel_display_control_vars()`.

Για `NDCVARS=17`, η συμμετρία είναι πλήρης: γίνονται 18 δεσμεύσεις — ένας πίνακας pointers και δεκαεπτά συμβολοσειρές — και αντίστοιχα 18 αποδεσμεύσεις. Αυτό το μοντέλο είναι συνεπές με τη φιλοσοφία ολόκληρης της OMPD διεπαφής: κάθε δέσμευση που εκτελεί η `libompd.so` για λογαριασμό του εργαλείου γίνεται μέσω callback, ώστε να λειτουργεί σωστά και σε out-of-process debugging όπου `libompd.so` και επιθεωρούμενη διεργασία δεν μοιράζονται χώρο μνήμης.

8.5 Σύγκριση με ompd_get_icv_from_scope()

DCVs και ICVs περιγράφουν παρεμφερείς πληροφορίες αλλά εξυπηρετούν διαφορετικές ανάγκες:

Χαρακτηριστικό	**ompd_get_display_control_vars**	**ompd_get_icv_from_scope**
Μορφή τιμής	Κείμενο (π.χ. "static,100")	Αριθμός ή enum
Scope	Χώρος διευθύνσεων (ολικό)	Thread / task / parallel
Handle	as_handle μόνο	thread / task / parallel handle
Πηγή δεδομένων	getenv() — τιμή εκκίνησης	Άμεση ανάγνωση μνήμης
Χρήση	Διαγνωστικό snapshot εκκίνησης	Τρέχουσα τιμή ICVs

Στην πράξη, τα DCVs χρησιμεύουν ως snapshot της αρχικής διαμόρφωσης, ενώ η `ompd_get_icv_from_scope()` επιτρέπει την ανάγνωση της τρέχουσας κατάστασης ενός συγκεκριμένου νήματος ή task κατά τη διάρκεια εκτέλεσης.

8.6 Περιορισμοί και μελλοντικές επεκτάσεις

Η τρέχουσα υλοποίηση έχει δύο περιορισμούς. Πρώτον, η `getenv()` επιστρέφει την τρέχουσα τιμή κατά τη στιγμή της κλήσης: εάν το πρόγραμμα έχει τροποποιήσει μεταβλητές περιβάλλοντος μετά την εκκίνηση, οι αλλαγές αυτές θα εμφανίζονται στα DCVs παρότι δεν αντιστοιχούν στην αρχική διαμόρφωση. Δεύτερον, αλλαγές ICVs που έγιναν μέσω του

OpenMP API δεν αντικατοπτρίζονται στα DCVs: γι' αυτή την πληροφορία το εργαλείο πρέπει να χρησιμοποιεί `ompd_get_icv_from_scope()`.

Κεφάλαιο 9: Δοκιμές και Αξιολόγηση

Η ορθότητα μιας υλοποίησης διεπαφής debugging δεν μπορεί να εγγυηθεί μόνο μέσω code review. Χρειάζεται μια συστηματική υποδομή που να ελέγχει κάθε επίπεδο της στοίβας OMPD ξεχωριστά: τα εσωτερικά δεδομένα του runtime, τα callbacks, τα handles, και τελικά τη συνολική συμπεριφορά κατά τη διάρκεια πραγματικής parallel εκτέλεσης. Για τη δοκιμή της υλοποίησης OMPD στον OMPi αναπτύξαμε αυτόνομα test harness σε οκτώ φάσεις.

9.1 Μεθοδολογία δοκιμών

Η υλοποίηση OMPD στο OMPi χρησιμοποιεί μια in-process δοκιμαστική προσέγγιση. Αντί να απαιτείται ένας πλήρης debugger (GDB, LLDB) για κάθε δοκιμή, γράφτηκαν αυτόνομα εκτελέσιμα test harness που φορτώνουν και χρησιμοποιούν τη libompd.so απευθείας εντός της ίδιας διαδικασίας.

Η προσέγγιση αυτή έχει αρκετά πλεονεκτήματα:

1. **Αυτοματοποίηση**: Οι δοκιμές εκτελούνται αυτόματα με το meson test, χωρίς να απαιτείται αλληλεπιδραστική session debugger.
2. **Ταχύτητα**: Κάθε δοκιμαστικό εκτελέσιμο τελειώνει σε δευτερόλεπτα, επιτρέποντας εκτέλεση κατά κάθε build.
3. **Πλήρης κάλυψη**: Δοκιμάζεται τόσο το runtime API (ompd_state.c) όσο και το DLL API (ompd_dll.c) σε κάθε επίπεδο.
4. **Ανεξαρτησία εργαλείου**: Δεν εξαρτάται από συγκεκριμένη έκδοση GDB ή άλλου debugger, οπότε τρέχει σε οποιοδήποτε σύστημα build.

Κάθε test harness παρέχει in-process υλοποίηση των OMPD callbacks (alloc_memory, free_memory, read_memory, symbol_addr_lookup κ.λπ.) που λειτουργούν απευθείας στη μνήμη της τρέχουσας διαδικασίας. Αυτό επιτρέπει να ελεγχθεί η ορθή συμπεριφορά της libompd.so ανεξάρτητα από τυχόν ιδιαιτερότητες συγκεκριμένου debugger.

9.2 Υποδομή δοκιμών

Η υποδομή αποτελείται από δύο στοιχεία: τα αρχεία δοκιμών (test_ompd*.c) και τη μακροεντολή CHECK().

9.2.1 Μακροεντολές CHECK και CHECK_RC

Κάθε αρχείο δοκιμών χρησιμοποιεί δύο κεντρικές μακροεντολές για τον έλεγχο αποτελεσμάτων:

```
#define CHECK(expr, msg) \
do { \
if (expr) { \
printf(" PASS %s\n", msg); \
passed++; \
} else { \
printf(" FAIL %s\n", msg); \
failed++; \
} \
} while (0)

#define CHECK_RC(rc, fn) \
CHECK((rc) == ompd_rc_ok, fn " returns ompd_rc_ok")
```

Κάθε δοκιμή διατηρεί μετρητές g_passes και g_failures. Στο τέλος κάθε test harness εκτυπώνεται η συνολική εικόνα και επιστρέφεται exit code 1 εάν υπάρχουν αποτυχίες — κάτι που το meson test αναγνωρίζει ως αποτυχία.

9.2.2 Δομή κάθε test harness

Κάθε αρχείο δοκιμών ακολουθεί τυποποιημένη δομή ενοτήτων που αντιστοιχούν στα γράμματα A, B, C, ... Κάθε ενότητα δοκιμάζει ένα συγκεκριμένο μέρος του OMPD API:

```
static void test_a_init(void)      { /* Αρχικοποίηση libompd */  }
static void test_b_handles(void)   { /* Handles χώρου διευθ. */ }
static void test_c_threads(void)   { /* Thread handles */        }
/* ... */

int main(void) {
    test_a_init();
    test_b_handles();
    test_c_threads();
    /* ... */
    printf("Passed: %d  Failed: %d\n", g_passes, g_failures);
    return g_failures ? 1 : 0;
}
```

9.2.3 In-process callbacks

Τα in-process callbacks υλοποιούνται στο αρχείο libort/ompd/ompd_callbacks.c. Το read_memory callback αντιγράφει μνήμη με memcpy() αντί για ptrace() που αρκεί για in-process δοκιμές:

```
static ompd_rc_t
cb_read_memory(ompd_address_space_context_t *ctx,
               ompd_thread_context_t *thr,
               const ompd_address_t *addr,
```

```
                ompd_size_t nbytes,
                void *buffer)
{
    memcpy(buffer, (const void *)(uintptr_t)addr->address, nbytes);
    return ompd_rc_ok;
}
```

Αντίστοιχη λογική ακολουθεί και το write_memory callback. Το symbol_addr_lookup χρησιμοποιεί dlsym() για επίλυση συμβόλων στον χώρο διευθύνσεων της τρέχουσας διαδικασίας.

9.3 Φάσεις δοκιμών

Οι δοκιμές οργανώνονται σε οκτώ φάσεις, κάθε μία εστιάζει σε διαφορετικό επίπεδο της στοίβας OMPD. Ο πίνακας που ακολουθεί δίνει συνοπτική εικόνα:

Αρχείο	Φάση	Θέμα	Assertions
test_ompd.c	1–3	Infrastructure, callbacks, metadata δομών	65+
test_ompd4.c	4	DLL API: αρχικοποίηση, handles, ICVs, DCVs	88
test_ompd5.c	5	Thread & task handles, team_members	45
test_ompd6.c	6	ICV queries σε όλα τα scopes	55
test_ompd7.c	7	Full integration με πραγματικά pthreads	48
test_ompd8.c	8	Device handles και device lifecycle	41

9.3.1 Φάσεις 1–3: Infrastructure (test_ompd.c)

Το test_ompd.c δοκιμάζει την υποδομή χαμηλού επιπέδου πριν εισαχθεί οποιαδήποτε λογική DLL, και απαιτεί μεταγλώττιση με ompicc επειδή χρησιμοποιεί OpenMP pragmas για την εκτέλεση parallel regions.

Η φάση 1 επαληθεύει ότι ο ompd_symbol_table περιέχει σωστά πεδία έκδοσης (major/minor) και ότι τα offset metadata που παράγει η μακροεντολή FIELD δίνουν ορθές τιμές για τα πεδία των ort_eecb_t, ort_task_node_t και ompd_parallel_region_t.

Η φάση 2 δοκιμάζει κάθε callback ξεχωριστά, ελέγχοντας ότι όλα επιστρέφουν ompd_rc_ok και ότι οι τιμές που διαβάζονται είναι σωστές.

Η φάση 3 ελέγχει ότι τα offsets κρίσιμων πεδίων που δημοσιεύει ο ompd_symbol_table συμφωνούν με τα πραγματικά sizeof/offsetof που επιστρέφει ο compiler.

9.3.2 Φάση 4: DLL API (test_ompd4.c)

Δοκιμάζει το σύνολο του δημόσιου OMPD DLL API όπως το βλέπει ένα εργαλείο debugging. Οι 8 ενότητες (A–H) καλύπτουν:

Ενότητα	Θέμα	Βασικές assertions
A	Αρχικοποίηση και version queries	ompd_initialize() → ompd_rc_ok, σωστό abi_version, σωστό vendor string
B	Process / address-space handle	ompd_process_initialize() → μη-NULL handle, ompd_get_omp_version() = 51
Γ	Thread handle	ompd_get_thread_handle() → μη-NULL, ompd_thread_handle_compare() = 0
Δ	State enumeration	ompd_enumerate_states() επιστρέφει τουλάχιστον 1 κατάσταση
E	Parallel region handle	ompd_get_curr_parallel_handle() μετά από ompd_parallel_begin(), ompd_get_parallel_info() → σωστό team_size
ΣΤ	ICV queries (global scope)	ompd_get_icv_from_scope() για nthreads, dynamic, run_sched επιστρέφει ompd_rc_ok
Z	Display control variables	ompd_get_display_control_vars() → μη-NULL, τουλάχιστον 1 τιμή, ompd_rel_display_control_vars() → NULL
H	Finalization	ompd_finalize() → ompd_rc_ok, CHECK_INIT() αποτυγχάνει μετά

9.3.3 Φάση 5: Thread & Task Handles (test_ompd5.c)

Εστιάζει στα thread handles, τα task handles και την πλήρωση του πίνακα team_members[] μέσω ompd_parallel_thread_join(), σε 9 ομάδες (A–I). Χρησιμοποιεί απευθείας κλήσεις στο internal ORT API χωρίς OpenMP pragmas. Ελέγχεται ότι το αρχικό νήμα έχει valid thread handle με ορθή eecb_addr, ότι το ompd_get_curr_task_handle() επιστρέφει μη-NULL handle για το implicit task, ότι το ompd_get_task_function() επιστρέφει διεύθυνση ≠ 0, και ότι μετά από ompd_parallel_thread_join() ο πίνακας team_members[] πληρώνεται σωστά για κάθε νήμα της ομάδας. Δοκιμάζεται επίσης η σχέση generating task, το ompd_get_task_in_parallel() και το ompd_task_handle_compare().

9.3.4 Φάση 6: ICV Scopes (test_ompd6.c)

Δοκιμάζει το ompd_get_icv_from_scope() σε όλα τα υποστηριζόμενα scopes. Στο global scope ελέγχονται 10 ICVs: nthreads, dynamic, run_sched, stacksize, wait_policy, thread_limit, max_active_levels, cancellation, default_device και num_procs, όλα επιστρέφουν ompd_rc_ok με τιμές συνεπείς με την αρχικοποίηση του runtime. Στο thread scope ελέγχονται level, active_levels, nthreads, dynamic, run_sched και thread_limit μέσω του implicit task του νήματος. Στα scopes task και implicit_task επαληθεύονται τα nthreads, dynamic και run_sched από τις per-task ICVs. Ελέγχεται επίσης η σωστή απόκριση σε αντικανονικές κλήσεις: το stacksize_var με thread handle (που δεν ανήκει στο task-ICVs path) επιστρέφει ompd_rc_bad_input, όπως ορίζει η §5.5.9.2 του προτύπου. Τέλος, η ομάδα Z δοκιμάζει το ompd_get_icv_string_from_scope() για schedule ICV, επαληθεύοντας ότι η επιστρεφόμενη συμβολοσειρά είναι γνωστό όνομα χρονοπρογραμματισμού ("static", "dynamic", "guided" ή "auto").

9.3.5 Φάση 7: Full Integration (test_ompd7.c)

Η πιο ολοκληρωμένη δοκιμή της στοίβας OMPD. Χρησιμοποιεί πραγματικά POSIX threads μέσω του pthreads EE backend, εκτελώντας parallel region μέσω _ort_execute_parallel() — το ίδιο runtime entry point που χρησιμοποιεί και ο ompicc-μεταφρασμένος κώδικας. Έτσι ασκείται ολόκληρη η υποδομή OMPD (ompd_thread_begin/end, ompd_parallel_begin/end, ompd_parallel_thread_join) σε πραγματικά νήματα. Οι 10 ομάδες δοκιμών (A–J) ελέγχουν: τη λίστα νημάτων πριν και κατά τη διάρκεια του parallel, την ορθότητα parallel region metadata (team_size, level), τα thread handles για όλα τα νήματα με μοναδικά eecb_addr, τα thread-scope και parallel-scope ICVs μέσα στο parallel, τα task handles εντός parallel, την ορθή εκκαθάριση μετά το τέλος, και τη συμπεριφορά nested parallelism σε επίπεδο 2. Δοκιμάζεται επίσης η αντίστροφη αναζήτηση νήματος από native thread ID και η σύγκριση ανίσων handles.

9.3.6 Φάση 8: Device Handles (test_ompd8.c)

Δοκιμάζει τη διαχείριση device handles σε 7 ομάδες (A–G), χρησιμοποιώντας προσομοιωμένη συσκευή χωρίς πραγματικό hardware. Ελέγχεται ότι η λίστα συσκευών είναι κενή πριν από ompd_device_begin() και περιέχει ακριβώς ένα στοιχείο μετά. Το ompd_device_initialize() χωρίς προηγούμενη κλήση ompd_process_initialize() επιστρέφει ompd_rc_needs_init, ενώ με valid process handle επιστρέφει ompd_rc_ok και μη-NULL device handle. Ελέγχεται η ορθή επιστροφή έκδοσης (51) από ompd_get_omp_version() για device handle, η ομαλή αποδέσμευση μέσω ompd_rel_address_space_handle(), και η αφαίρεση της συσκευής από τη λίστα μετά το ompd_device_end(). Τέλος, επαληθεύεται ότι τα symbols ompd_bp_device_begin και ompd_bp_device_end εντοπίζονται μέσω symbol_addr_lookup() με μη-μηδενική διεύθυνση.

9.4 Ενσωμάτωση με το Meson build system

Οι δοκιμές ενσωματώνονται πλήρως στο Meson build system του OMPi μέσω του αρχείου libort/ompd/meson.build. Η δήλωση build_by_default: false εξασφαλίζει ότι τα εκτελέσιμα δοκιμών δεν χτίζονται κατά το συνηθισμένο meson compile, αλλά μόνο όταν εκτελείται meson test:

```
foreach tname : [
  'test_ompd4', 'test_ompd5', 'test_ompd6',
  'test_ompd7', 'test_ompd8',
]
  exe = executable(
    tname,
    tname + '.c',
    link_with:        [ libort, libompd ],
    dependencies:     [ dep('threads'), cc.find_library('dl') ],
    build_by_default: false,
  )
  test('ompd/' + tname, exe,
    suite:   'ompd',
    timeout: 60,
  )
endforeach
```

Για εκτέλεση των δοκιμών:

```
# Μεταγλώττιση και εκτέλεση όλων των OMPD δοκιμών
```

```
meson test -C build --suite ompd

# Εκτέλεση συγκεκριμένης δοκιμής με verbose output
meson test -C build ompd/test_ompd7 --verbose

# Χτίσιμο χωρίς εκτέλεση (για debugging)
meson compile -C build test_ompd7
```

Κάθε test εκτελείται με timeout 60 δευτερόλεπτα, που είναι αρκετό για το test_ompd7.c που εκκινεί πραγματικά pthreads, αλλά όχι αρκετά μεγάλο ώστε ένα ατέρμονο deadlock να μπλοκάρει το CI pipeline.

9.5 Αποτελέσματα δοκιμών

Εκτελώντας το meson test --suite ompd (test_ompd4–test_ompd8) σε σύστημα Linux x86-64 με GCC 12 και OMP_NUM_THREADS=4:

Δοκιμαστικό	Assertions	Passed	Failed
test_ompd4	88	88	0
test_ompd5	45	45	0
test_ompd6	55	55	0
test_ompd7	48	48	0
test_ompd8	41	41	0
Σύνολο	277	277	0

Όλες οι δοκιμές πέρασαν επιτυχώς.

9.6 Κάλυψη API

Ο παρακάτω πίνακας παρουσιάζει την κάλυψη των δημόσιων συναρτήσεων OMPD API από τις δοκιμές:

Συνάρτηση OMPD	Δοκιμάζεται σε	Κάλυψη
ompd_initialize	test_ompd4 §A	✓ Πλήρης
ompd_finalize	test_ompd4 §H	✓ Πλήρης
ompd_process_initialize	test_ompd4 §B	✓ Πλήρης
ompd_rel_address_space_handle	test_ompd8 §E	✓ Πλήρης

`ompd_get_omp_version`	test_ompd4 §B	✓ Πλήρης
`ompd_get_thread_handle`	test_ompd4/5 §Γ	✓ Πλήρης
`ompd_rel_thread_handle`	test_ompd5	✓ Πλήρης
`ompd_get_next_thread_handle`	test_ompd5	✓ Πλήρης
`ompd_thread_handle_compare`	test_ompd5	✓ Πλήρης
`ompd_get_state`	test_ompd4/5 §Δ	✓ Μερική
`ompd_get_thread_in_parallel`	test_ompd5	✓ Πλήρης
`ompd_get_curr_parallel_handle`	test_ompd4/7 §E	✓ Πλήρης
`ompd_get_enclosing_parallel_handle`	test_ompd4	✓ Πλήρης
`ompd_get_parallel_info`	test_ompd4/7	✓ Πλήρης
`ompd_rel_parallel_handle`	test_ompd4/7	✓ Πλήρης
`ompd_get_task_parallel_handle`	test_ompd4/5	✓ Μερική
`ompd_parallel_handle_compare`	test_ompd4/5	✓ Πλήρης
`ompd_get_curr_task_handle`	test_ompd5/7 §Γ	✓ Πλήρης
`ompd_get_generating_task_handle`	test_ompd5 §E	✓ Πλήρης
`ompd_get_scheduling_task_handle`	test_ompd5	✓ Μερική
`ompd_get_task_function`	test_ompd5 §Δ	✓ Πλήρης
`ompd_get_task_frame`		✗ Δεν δοκιμάζεται
`ompd_rel_task_handle`	test_ompd5/7	✓ Πλήρης
`ompd_get_task_in_parallel`	test_ompd5	✓ Πλήρης
`ompd_task_handle_compare`	test_ompd5	✓ Πλήρης
`ompd_get_icv_from_scope`	test_ompd4/6/7	✓ Πλήρης
`ompd_get_icv_string_from_scope`	test_ompd6 §Z	✓ Πλήρης
`ompd_enumerate_icvs`	test_ompd6	✓ Πλήρης
`ompd_get_display_control_vars`	test_ompd4 §Z	✓ Πλήρης
`ompd_rel_display_control_vars`	test_ompd4 §Z	✓ Πλήρης
`ompd_enumerate_states`	test_ompd4 §Δ	✓ Μερική
`ompd_device_initialize`	test_ompd8 §Γ	✓ Πλήρης
`ompd_get_tool_data`	—	✗ Δεν δοκιμάζεται

Από τις 33 συναρτήσεις OMPD API του OMPi, 27 δοκιμάζονται πλήρως, 4 μερικώς και 2 επιστρέφουν ompd_rc_unsupported χωρίς άλλη κάλυψη (ompd_get_task_frame, ompd_get_tool_data).

9.7 Περιορισμοί της υποδομής δοκιμών

Η in-process προσέγγιση, παρότι πρακτική, έχει ορισμένα όρια. Το σημαντικότερο είναι ότι τα callbacks χρησιμοποιούν memcpy() αντί για ptrace(), οπότε δεν δοκιμάζεται η ορθή λειτουργία σε πραγματικό out-of-process debugging· η ολοκληρωμένη επαλήθευση σε αυτό το επίπεδο θα απαιτούσε GDB script tests.

Ένα δεύτερο όριο αφορά την απομόνωση. Το μοντέλο OMPD προϋποθέτει σταματημένο target — ο debugger διακόπτει όλα τα νήματα πριν από κάθε ερώτημα. Στο in-process test harness, ο tester και το runtime μοιράζονται τον ίδιο χώρο διευθύνσεων και εκτελούνται ταυτόχρονα, οπότε οι δομές thread/parallel list δεν εξετάζονται ποτέ υπό τις συνθήκες απομόνωσης που θα είχε ένας πραγματικός debugger.

Τέλος, το test_ompd7.c τρέχει με OMP_NUM_THREADS=4 κατά προεπιλογή και το nested parallelism δοκιμάζεται μόνο σε επίπεδο 2. Δεν υπάρχουν stress tests για memory leaks υπό επαναλαμβανόμενες κλήσεις ompd_parallel_begin/end ή ompd_task_create/end.

9.7.1 Παράδειγμα: Επαλήθευση υπό πραγματική διακοπή (GDB)

Παρότι η κύρια δοκιμαστική υποδομή λειτουργεί in-process, μια συμπληρωματική δοκιμή πραγματοποιήθηκε με τη χρήση του GDB σε σύστημα Linux, όπου ο μεταγλωττισμένος δοκιμαστικός στόχος εκτελέστηκε ως ξεχωριστή διεργασία και ο debugger τοποθέτησε breakpoints απευθείας στις συναρτήσεις ειδοποίησης (notification functions) του libort.

Μετά τη φόρτωση του προγράμματος και τον ορισμό breakpoint στη συνάρτηση `ompd_bp_parallel_begin`, η εκτέλεση διακόπηκε τη στιγμή της δημιουργίας μιας παράλληλης περιοχής:

```
(gdb) break ompd_bp_parallel_begin
(gdb) run
...
Breakpoint 2, ompd_bp_parallel_begin ()
    at libort/ompd/ompd_state.c:546
546  OMPD_NOINLINE void ompd_bp_parallel_begin(void) { OMPD_BP_BODY; }
(gdb) print *ompd_parallel_list
$1 = {master_eecb_addr = 0, parent = 0x0, next = 0x0, team_size = 4,
      level = 1, shared_data = 0x0, team_members = 0x49bd80,
      master_prev_eecb = 0x40}
(gdb) print ompd_parallel_list->team_members[0]
$2 = (void *) 0x0
(gdb) print ompd_parallel_list->team_members[1]
$3 = (void *) 0x0
```

Στο σημείο αυτό η δομή `ompd_parallel_region_t` έχει ήδη δεσμεύσει τον πίνακα `team_members[]`, αλλά τα στοιχεία του παραμένουν `NULL` — τα εργατικά νήματα δεν έχουν ακόμη προσχωρήσει στην ομάδα. Θέτοντας breakpoint στη συνάρτηση `ompd_parallel_thread_join` και συνεχίζοντας την εκτέλεση, παρατηρήθηκε η σταδιακή συμπλήρωση του πίνακα, ένα στοιχείο ανά κλήση, μέχρι να γεμίσουν και οι τέσσερις θέσεις, επιβεβαιώνοντας το μοντέλο lazy filling που περιγράφηκε στην Ενότητα 6.4.

Η δοκιμή αυτή, αν και μη αυτοματοποιημένη, έχει ιδιαίτερη αξία: αντίθετα με το in-process test harness, εδώ ο GDB λειτουργεί ως πραγματικός εξωτερικός debugger: ο στόχος είναι πλήρως διακεκομμένος (stopped target) και η ανάγνωση της κατάστασης γίνεται μέσω του μηχανισμού ptrace του λειτουργικού συστήματος, όχι μέσω απευθείας πρόσβασης στη μνήμη της ίδιας διεργασίας. Επιβεβαιώνει, δηλαδή, ότι η OMPD υλοποίηση λειτουργεί σωστά και υπό τις πραγματικές συνθήκες απομόνωσης που περιγράφει το πρότυπο, έστω και χωρίς να καλύπτει συστηματικά όλο το εύρος των σεναρίων που καλύπτει το αυτοματοποιημένο test_ompd7.

Κεφάλαιο 10: Συμπεράσματα και Μελλοντική Εργασία

Η αποσφαλμάτωση ενός OpenMP προγράμματος θέτει ένα πρακτικό πρόβλημα: ο debugger βλέπει νήματα και μνήμη, αλλά όχι τη δομή της παραλληλίας που τα δημιουργεί. Για να παρακολουθεί ποιο νήμα ανήκει σε ποια ομάδα, σε ποιο task βρίσκεται ή πόσα επίπεδα παραλληλίας είναι ενεργά, χρειάζεται συνεργασία από τη βιβλιοθήκη χρόνου εκτέλεσης. Αυτή τη συνεργασία τυποποιεί το OMPD, μέσα από ένα σαφώς ορισμένο API μεταξύ runtime και εργαλείου.

Υλοποιήθηκε το OMPD στον OMPi, καλύπτοντας τη διαχείριση καταστάσεων στο runtime, το πλήρες DLL API και την αυτοματοποιημένη δοκιμαστική υποδομή. Παρακάτω συνοψίζουμε τι υλοποιήθηκε, ποιές σχεδιαστικές επιλογές έγιναν και τι μένει για μελλοντική εργασία.

10.1 Τι υλοποιήθηκε

Η `libompd.so` υλοποιεί 33 δημόσιες συναρτήσεις OMPD API που καλύπτουν τη διαχείριση address space, thread, parallel region και task handles, την ανάκτηση ICV τιμών ανά scope, τα display control variables και την καταμέτρηση καταστάσεων νημάτων. Η βιβλιοθήκη αποφεύγει εξάρτηση από τα ιδιωτικά headers του OMPi, καθώς η πληροφορία για τη διάταξη δομών (`ort_eecb_t`, `ort_task_node_t`, `ort_icvs_t`) έρχεται αποκλειστικά μέσω του `ompd_symbol_table[]`, ενός global descriptor που δημοσιεύεται από τη `libort.so` κατά την αρχικοποίηση.

Η κεντρική τεχνική προσθήκη είναι η `ompd_parallel_region_t`, μια νέα δομή που δεν υπήρχε στο αρχικό runtime. Ο OMPi δεν διατηρούσε ενιαία αναπαράσταση μιας parallel region, καθώς οι σχετικές πληροφορίες ήταν σκορπισμένες στα `ort_eecb_t` των νημάτων. Η νέα δομή συγκεντρώνει `team_size`, `level`, `shared_data` και τον πίνακα `team_members[]` σε ένα σημείο, σε linked list με innermost-first σειρά, και συντηρείται από το runtime μέσω των hooks `ompd_parallel_begin`, `ompd_parallel_end` και `ompd_parallel_thread_join`.

Εννέα notification breakpoints, υλοποιημένα ως άδειες `noinline` συναρτήσεις με compiler barriers, καλύπτουν τα κρίσιμα γεγονότα εκτέλεσης: έναρξη και λήξη parallel regions, δημιουργία, έναρξη και λήξη tasks, εκκίνηση και τερματισμό νημάτων, αρχικοποίηση συσκευών. Ο debugger τοποθετεί breakpoints σε αυτές τις συναρτήσεις ώστε να ειδοποιείται για κάθε γεγονός χωρίς να χρειάζεται polling.

Γράφτηκαν πέντε test harness με συνολικά ~277 assertions, ενσωματωμένα στο Meson build system. Επαληθεύουν κάθε επίπεδο της στοίβας OMPD, από τα callbacks και τα

metadata μέχρι τα ICV queries, τη διαχείριση nested parallel και τους device handles, χωρίς να απαιτείται εξωτερικός debugger για την εκτέλεσή τους.

10.2 Σχεδιαστικές επιλογές

Η `libompd.so` επικοινωνεί με τον χώρο μνήμης του target αποκλειστικά μέσω callbacks που παρέχει ο debugger, χωρίς να καλεί απευθείας `ptrace()` ή άλλες κλήσεις συστήματος. Αυτό δεν αποτελεί σχεδιαστική επιλογή του OMPi αλλά απαίτηση του προτύπου: η §5.1 ορίζει ρητά ότι η `libompd.so` πρέπει να παραμένει αγνωστική ως προς τον μηχανισμό μεταφοράς δεδομένων, με αποτέλεσμα η ίδια βιβλιοθήκη να λειτουργεί τόσο in-process (memcpy) όσο και out-of-process (ptrace) χωρίς τροποποίηση.

Ο πίνακας `team_members[]` πληρώνεται βαθμιαία μέσω `ompd_parallel_thread_join()`, καθώς κάθε νήμα γράφει τη δική του θέση κατά την εκκίνησή του. Η εναλλακτική, δηλαδή η δέσμευση ολόκληρου του πίνακα εκ των προτέρων, θα εισήγαγε race condition ανάμεσα στη δημιουργία της δομής και την εκκίνηση των νημάτων. Αντίστοιχα, τα Display Control Variables ανακτώνται μέσω `getenv()` αντί για ανάγνωση ICVs από τη μνήμη, καθώς το πρότυπο ορίζει ότι τα DCVs αντανακλούν την αρχική διαμόρφωση μέσω περιβάλλοντος.

Το `symbol_addr_lookup` callback αναζητά πρώτα στον `ompd_symbol_table[]` πριν καλέσει `dlsym()`. Αυτό έχει πρακτική σημασία για στατικά συνδεδεμένα binaries: σε `libort.a` τα εσωτερικά σύμβολα δεν είναι ορατά μέσω `dlsym()`, οπότε χωρίς αυτό το fast-path η αρχικοποίηση θα απέτυχε σε static builds.

10.3 Περιορισμοί

Η υλοποίηση καλύπτει το σύνολο του OMPD API για host processes, με τα εξής γνωστά όρια:

Περιορισμός	Επίπτωση	Προτεινόμενη λύση
Χωρίς out-of-process δοκιμές	Η ορθή λειτουργία με GDB/ptrace δεν επαληθεύεται αυτόματα	GDB Python script tests με DejaGnu ή pytest-gdb
team_members[] lazy filling	Ατελής πίνακας εάν debugger διακόψει νωρίς	Atomic bitfield για παρακολούθηση ποια νήματα έχουν join
Χωρίς task frame info	GDB δεν εμφανίζει call stack για tasks — ompd_get_task_frame() επιστρέφει ompd_rc _unsupported	Υλοποίηση με saved frame pointer από ort_task_node_t

10.4 Μελλοντική εργασία

Το πιο άμεσο επόμενο βήμα είναι ένα GDB Python plugin που φορτώνει τη `libompd.so` και εκθέτει εντολές υψηλού επιπέδου στο command-line του debugger, όπως `ompd-threads`, `ompd-parallel` και `ompd-tasks`. Δεδομένου ότι η `libompd.so` του OMPi είναι ήδη spec-compliant, ένα τέτοιο plugin θα λειτουργούσε χωρίς αλλαγές στη βιβλιοθήκη.

Πιο μακροπρόθεσμα, η ευθυγράμμιση με μεταγενέστερα πρότυπα θα απαιτούσε χαρτογράφηση νέων ICVs και breakpoints. Η ενοποίηση με OMPT (OpenMP Tools Interface), που υπάρχει ήδη σε ξεχωριστό branch, θα δημιουργούσε μια ολοκληρωμένη υποδομή debuggability και profiling για τον OMPi, καθώς οι δύο διεπαφές αλληλοσυμπληρώνονται: το OMPT παρέχει real-time ειδοποιήσεις εκτέλεσης, ενώ το OMPD επιτρέπει εκ των υστέρων επιθεώρηση κατάστασης.

10.5 Κλείνοντας

Η αρχιτεκτονική του OMPD κάνει την υλοποίηση παρεμβατικά μικρή. Το runtime αποκτά state-tracking μέσω linked lists και notification hooks, αλλά η κεντρική λογική εκτέλεσης παραμένει αναλλοίωτη. Η `libompd.so` από την πλευρά της είναι απομονωμένη από τις εσωτερικές δομές του OMPi μέσω του `ompd_symbol_table`, πράγμα που σημαίνει πως μια μελλοντική ανακατασκευή του runtime δεν θα χρειαστεί να ξαναγράψει τη DLL, μόνο να ενημερώσει τον descriptor.

Το OMPD παραμένει σχετικά νέο ως πρότυπο, με λίγες υλοποιήσεις εκτός του LLVM libomp. Η υλοποίηση στον OMPi αποτελεί δεύτερη spec-compliant αναφορά, βασισμένη σε διαφορετική αρχιτεκτονική runtime και διαφορετικές σχεδιαστικές επιλογές.

Βιβλιογραφία

[1] OpenMP Architecture Review Board, *OpenMP Application Programming Interface, Version 5.0*. OpenMP ARB, Nov. 2018. [Online]. Available: https://www.openmp.org/specifications/

[2] OpenMP Architecture Review Board, *OpenMP Application Programming Interface, Version 5.1*. OpenMP ARB, Nov. 2020. [Online]. Available: https://www.openmp.org/specifications/

[3] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, ser. Scientific and Engineering Computation. Cambridge, MA: MIT Press, 2008.

[4] J. Protze, I. Laguna, D. H. Ahn, J. DelSignore, A. Burton, M. Schulz, and M. S. Müller, "Lessons Learned from Implementing OMPD: A Debugging Interface for OpenMP," in *Proc. 11th Int. Workshop on OpenMP (IWOMP 2015)*, Lecture Notes in Computer Science, vol. 9342. Cham: Springer, 2015. doi: 10.1007/978-3-319-24595-9_7

[5] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis," in *Proc. 9th Int. Workshop on OpenMP (IWOMP 2013)*, Lecture Notes in Computer Science, vol. 8122. Berlin: Springer, 2013, pp. 171–185. doi: 10.1007/978-3-642-40698-0_13

[6] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A Portable C Compiler for OpenMP V.2.0," in *Proc. 5th European Workshop on OpenMP (EWOMP 2003)*, Aachen, Germany, 2003.

[7] Free Software Foundation, *Debugging with GDB: The GNU Source-Level Debugger*, 15th ed. [Online]. Available: https://www.gnu.org/software/gdb/documentation/

[8] LLVM Project, *libompd* implementation (Phabricator D100181–D100185). Available: https://reviews.llvm.org/D100181

[9] GNU Project, GNU Offloading and Multi-Processing Project (GOMP). Available: https://gcc.gnu.org/projects/gomp/