Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Τεχνικές Πρόβλεψης για ζωντανή μεταφορά Εικονικών Μηχανών (VM Live Migration) σε Περιβάλλοντα Υπολογιστικού Νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΗΜΗΤΡΙΟΣ ΚΑΛΟΓΕΡΟΠΟΥΛΟΣ

Επιβλέπων : Γεώργιος Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Τεχνικές Πρόβλεψης για ζωντανή μεταφορά Εικονικών Μηχανών (VM Live Migration) σε Περιβάλλοντα Υπολογιστικού Νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΗΜΗΤΡΙΟΣ ΚΑΛΟΓΕΡΟΠΟΥΛΟΣ

Επιβλέπων : Γεώργιος Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Ιουλίου 2019.

Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2019

Δημήτριος Καλογερόπουλος

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

Περίληψη

Μια από τις βασικές τεχνολογίες εικονικοποίησης της Υποδομής-ως-Υπηρεσία (IaaS) στην εποχή του Cloud Computing είναι η ζωντανή μεταφορά (live migration) των εικονικών μηχανών (VMs). Μέσω του live migration, προβλήματα όπως η ενοποίηση των servers και η εξισορρόπηση φόρτου μεταξύ των φυσικών μηχανημάτων μπορούν να συντονιστούν. Ωστόσο, η μη διαθεσιμότητα της υπηρεσίας κατά τη διάρκεια του VM live migration μπορεί να είναι σημαντική σε σχέση με τις προσδοκίες των πελατών για την απόκριση των υπηρεσιών καθώς και τα επίπεδα ποιότητας της υπηρεσίας (QoS). Αυτές οι μετρικές δηλώνονται σε συμφωνίες σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου (SLAs). Συγκεκριμένα, για το live migration με pre-copy τεχνική αντιγραφής της μνήμης, υπάρχει το ρίσκο της μη σύγκλισης του αλγορίθμου και επομένως τη μη μετάβασή του στο stop-and-copy στάδιο. Αυτή η κατάσταση συμβαίνει όταν το VM γράφει στις σελίδες μνήμης ταχύτερα από το ρυθμό μεταφοράς των σελίδων αυτών από τον αρχικό host στον host προορισμού. Καθώς οι Cloud πάροχοι υπηρεσιών δεν μπορούν να ρυθμίσουν το ρυθμό "βρώμικων" σελίδων (dirty page rate) της εφαρμογής που εκτελείται σε μια εικονική μηχανή, πρέπει να διαμορφώσουν τις συνθήκες τερματισμού του migration. Στην περίπτωση του QEMU/KVM λογισμικού ελέγχου, οι τροποποιήσιμες παράμετροι είναι η μέγιστη ταχύτητα μεταφοράς (*max-bandwidth*) και ο μέγιστος ανεκτός χρόνος μη λειτουργίας (*downtime-limit*). Λόγω του φυσικού δικτύου, το εύρος ζώνης έχει περιορισμένο άνω όριο και οι διαχειριστές δεν θέλουν να το εκμεταλλευτούν πλήρως. Επομένως, η παράμετρος *downtime-limit* θα πρέπει να διαμορφωθεί ώστε το pre-copy live migration να συγκλίνει και να ολοκληρωθεί με επιτυχία.

Οι εφαρμογές με εκτεταμένες εγγραφές στη μνήμη είναι δύσκολο να μεταφερθούν, επειδή τα όρια του ρυθμού μεταφοράς και το *downtime-limit* δεν μπορούν να ρυθμιστούν βέλτιστα χωρίς να είναι γνωστή η συμπεριφορά της εφαρμογής. Προκειμένου να αντιμετωπιστούν οι προκλήσεις που προκύπτουν σχετικά με το πρόβλημα της σύγκλισης της pre-copy live migration τεχνικής στα σύγχρονα κέντρα δεδομένων, αναπτύσσουμε ένα framework για την παρακολούθηση των διαθέσιμων VMs όπου λαμβάνονται δυναμικές αποφάσεις και ενέργειες με βάση το αποτύπωμά τους στη μνήμη προτού ξεκινήσει το live migration. Υλοποιούμε ένα μηχανισμό που ονομάζεται BitmapTrace και ενσωματώνεται στο QEMU/KVM, ο οποίος καταγράφει τον αριθμό των dirty σελίδων της εικονικής μηχανής για μια συγκεκριμένη χρονική περίοδο με overhead μόλις λίγα επιπλέον δευτερόλεπτα στο χρόνο εκτέλεσης της εφαρμογής. Χρησιμοποιούμε αυτόν τον μηχανισμό σε ένα σενάριο χρονοδρομολόγησης του migration ενός υποσυνόλου VMs από όσα εκτελούνται στο ίδιο φυσικό μηχάνημα. Παρακολουθώντας τη συμπεριφορά της μνήμης τους και χρησιμοποιώντας ένα μοντέλο πρόβλεψης, επιλέγουμε για live migration τις εικονικές μηχανές χωρίς να παραβιάζεται η διαθεσιμότητα υπηρεσιών με βάση το συμφωνηθέν SLA. Με τον τρόπο αυτό επιτυγχάνεται ένας συμβιβασμός μεταξύ των στόχων του Cloud παρόχου και των αναμενόμενων απαιτήσεων QoS των πελατών.

Λέξεις κλειδιά

Υπολογιστικό Νέφος, Εικονικές Μηχανές, Ζωντανή Μεταφορά, αποτύπωμα μνήμης, τεχνικές πρόβλεψης

Abstract

One of the key virtualization technologies of Infrastructure-as-a-Service in the era of Cloud Computing is the live migration of running Virtual Machines (VMs). Through VM live migration, issues such as VM consolidation and service performance degradation can be coordinated and balanced. However, the unavoidable short downtime/service unavailability during VM live migration may be substantial with regard to the customers' expectations on responsiveness as well as the requested Quality of Service (QoS) levels. These metrics are declared in established Service Level Agreements (SLAs). Especially for pre-copy live migration algorithm the main risk is when the migration process cannot converge to an optimal point for the final stop-and-copy phase. This situation happens when the VM dirties its memory pages faster than the migration bandwidth. Since Cloud Service Providers (CSPs) are not able to tune the dirty page rate of the workload running on a Virtual Machine, they need to configure the migration termination conditions. In case of QEMU/KVM, the system admin configurable parameters are the maximum migration transfer rate (*max-bandwidth*) and the maximum tolerated downtime (*downtime-limit*). Due to physical network, bandwidth has an upper limit and operators do not want to make full utilization of it. Thus, *downtime-limit* should be configured to make VM live migration converge and complete successfully.

Memory intensive applications are difficult to migrate because rate limits and downtime cannot be optimally set without detailed knowledge of the application behavior. In order to address the concerns that arise regarding the convergence problem of pre-copy live migration in modern data centers, we employ a framework for monitoring the available VMs and make dynamic decisions and actions based on their memory footprint right before migration is triggered. We implement a memory profiling module in QEMU/KVM called BitmapTrace which tracks the dirty pages of the workload for a certain profiling period with an execution time overhead of just a few seconds. We utilize this mechanism in a migration scheduling scheme where given a set of real-life workloads, each of them running in a separate Virtual Machine on the same host, we monitor their memory behavior and we select the best VM candidates using a prediction model without violating the agreed service availability. In this way, a trade-off between the CSP's objectives and the customers' expected QoS requirements can be achieved successfully.

Key words

Virtual Machines, Live Migration, pre-copy, QEMU/KVM, memory footprint, SLA, prediction techniques, modeling

Ευχαριστίες

Με την παρούσα διπλωματική εργασία ολοκληρώνεται ένα σημαντικό κεφάλαιο της ακαδημαϊκής μου πορείας. Θα ήθελα στο σημείο αυτό να ευχαριστήσω όλα τα άτομα που με βοήθησαν στη διαδρομή αυτή και συντέλεσαν στην ολοκλήρωσή του.

Αρχικά θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Γεώργιο Γκούμα για τις γνώσεις που μου προσέφερε με τη διδασκαλία του και την ευκαιρία που μου έδωσε να δουλέψω στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Θα ήθελα, επίσης, να ευχαριστήσω τον καθηγητή κ. Νεκτάριο Κοζύρη, που με τις διαλέξεις του ενίσχυσε σημαντικά το ενδιαφέρον μου σε αυτό τον τομέα της επιστήμης των υπολογιστών. Οφείλω ένα ιδιαίτερο ευχαριστώ στον υποψήφιο διδάκτορα και μέλος του εργαστηρίου Στέφανο Γεράγγελο για την εποπτεία του κατά την εκπόνηση της διπλωματικής μου εργασίας. Η συμβολή του ήταν καθοριστική για την ολοκλήρωση της και τον ευχαριστώ για την άρτια συνεργασία μας και το χρόνο που αφιέρωσε.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους μου και ιδιαίτερα τους γονείς μου, Δήμο και Ρούλα, και τον αδερφό μου, Τάσο, για τη συνεχή υποστήριξη και συμπαράσταση που μου προσέφεραν μέχρι σήμερα.

Δημήτριος Καλογερόπουλος,

Αθήνα, 8η Ιουλίου 2019

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Περιεχόμενα 11

Κατάλογος πινάκων 15

Κατάλογος σχημάτων 17

1. Εισαγωγή 19
1.1 Κίνητρο 19
1.2 Συνεισφορά 20

2. Θεωρητικό Υπόβαθρο 21
2.1 Υπολογιστικό Νέφος 21
2.2 Εικονικοποίηση 22
2.2.1 Hypervisor 23
2.2.2 Τεχνικές Εικονικοποίησης 23
2.2.3 Διαφοροποίηση Εικονικών Μηχανών και Containers 25
2.2.4 QEMU/KVM 26
2.2.5 XEN 28
2.3 Ζωντανή Μεταφορά (Live Migration) Εικονικών Μηχανών 28
2.3.1 Τεχνικές του Live Migration 29
2.3.2 Μετρικές απόδοσης του Live Migration 32
2.3.3 Προαπαιτούμενα του Migration 33
2.3.4 Live Migration με χρήση QEMU/KVM 33
2.4 Κατηγορίες Εφαρμογών 36
2.4.1 Έντονη δραστηριότητα CPU (CPU-Intensive) 36
2.4.2 Έντονη δραστηριότητα μνήμης (Memory-intensive) 37
2.4.3 Έντονη δραστηριότητα Εισόδου/Εξόδου (I/O-Intensive) 37

3. Αξιολόγηση του Pre-Copy Live Migration 39
3.1 Τοπολογία 39

3.2 Benchmarks . . . 40
3.2.1 Microbenchmarks . . . 40
3.2.2 SPEC2006 . . . 40
3.2.3 SPECjbb2005 . . . 40
3.2.4 Idle . . . 41
3.3 Αξιολόγηση των χαρακτηριστικών του QEMU/KVM Live Migration . . . 41
3.3.1 Δέσμευση πόρων μνήμης (*VM_Size*) . . . 42
3.3.2 Αξιολόγηση της παραμέτρου *downtime-limit* . . . 42
3.3.3 Αξιολόγηση της παραμέτρου *max-bandwidth* . . . 43
3.3.4 Αξιολόγηση της χρονικής στιγμής εκκίνησης του migration . . . 44
3.3.5 Ελάχιστη τιμή του *downtime-limit* για επιτυχές migration . . . 45
3.4 Χαρακτηριστικά του Live Migration σχετικά με τη μνήμη . . . 47
3.5 Εργαλεία παρακολούθησης της μνήμης . . . 47
3.5.1 Cache misses και MPKI με χρήση των Performance Counters . . . 47
3.5.2 IOwait . . . 51

4. Σχεδιασμός και Υλοποίηση του BitmapTrace μηχανισμού στο QEMU/KVM . . . 53
4.1 Σχεδιασμός . . . 53
4.2 Υλοποίηση . . . 54
4.3 Χρήση του BitmapTrace μηχανισμού . . . 55
4.4 Αξιολόγηση του BitmapTrace μηχανισμού . . . 57
4.4.1 Εκτίμηση του WWS . . . 57
4.4.2 Πειραματική Ρύθμιση της παραμέτρου *period* . . . 57
4.4.3 Πειραματική Ρύθμιση της παραμέτρου *iterations* . . . 60
4.4.4 Performance Overhead . . . 65

5. Προσέγγιση Μηχανικής Μάθησης για Migration Υποψήφιων VM . . . 67
5.1 Κατασκευή Μοντέλου . . . 67
5.1.1 Κατασκευή Συνόλου Δεδομένων . . . 67
5.1.2 Δημιουργία Μοντέλου . . . 72
5.2 Αξιολόγηση Μοντέλου . . . 73
5.3 Πολιτικές Λειτουργίας για Επιλογή Υποψήφιων VMs για Live Migration . . . 77
5.4 Αξιολόγηση της Επιλογής Υποψήφιων VMs για Live Migration . . . 78
5.4.1 VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις . . . 79
5.4.2 VMs με ίδια εφαρμογή και με SLA παραβιάσεις . . . 81
5.4.3 VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις . . . 82

6. Σχετική Έρευνα . . . 85

7. Επίλογος και Μελλοντικές Επεκτάσεις . . . 89

Συντομογραφίες . . . 91

Γλωσσάριο . . . 93

Βιβλιογραφία 95

Κατάλογος πινάκων

3.1 Ελάχιστη τιμή του *downtime-limit* για επιτυχές Live Migration κάθε εφαρμογής (*max-bandwidth*=800Mbps) 46

4.1 Runtime overhead του BitmapTrace με παραμέτρους (*"iterations"*, *"period"*)=(10,1000) σε εφαρμογή που πραγματοποιεί συνεχόμενα writes σε 512MB μνήμης 65
4.2 Runtime overhead του BitmapTrace με διάφορες τιμές των παραμέτρων (*"iterations"*, *"period"*) σε εφαρμογή που πραγματοποιεί συνεχόμενα writes σε 512MB μνήμης . 66
4.3 Performance overhead του BitmapTrace σε διαφορετικές εφαρμογές (execution time=3 min) 66

5.1 Ελάχιστη τιμή του *downtime-limit* για επιτυχές migration κάθε εφαρμογής (*max-bandwidth*=800Mbps) 71
5.2 Χρόνος εκπαίδευσης μοντέλου για (iterations,pediod) = (10,1000) 74
5.3 Accuracy και MAE των *min_success_downtime-limit*, *downtime* και *total_time* μοντέλων πρόβλεψης 75
5.4 Live Migration σενάριο χρονοδρομολόγησης για VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις 79
5.5 Aggregated total time για VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις . . . 80
5.6 Live Migration σενάριο χρονοδρομολόγησης για VMs με ίδια εφαρμογή και με SLA παραβιάσεις 81
5.7 Live Migration σενάριο χρονοδρομολόγησης για VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις 82
5.8 Aggregated total time για VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις 83

Κατάλογος σχημάτων

2.1 Μοντέλα υπηρεσιών Cloud . . . 22
2.2 Τύποι Λογισμικών Ελέγχου (hypervisors) . . . 23
2.3 Τεχνικές Εικονικοποίησης . . . 25
2.4 Virtual Machines vs Containers . . . 26
2.5 Επισκόπηση του QEMU/KVM περιβάλλοντος εικονικοποίησης . . . 27
2.6 Pre-copy Live Migration . . . 31
2.7 Post-copy Live Migration . . . 31
2.8 Hybrid Live Migration . . . 32
2.9 Κατηγορίες εφαρμογών . . . 36

3.1 Επίδραση του *VM_Size* στο Live Migration . . . 42
3.2 Επίδραση της παραμέτρου *downtime-limit* στο Live Migration . . . 43
3.3 Επίδραση της παραμέτρου *max-bandwidth* στο Live Migration . . . 44
3.4 Επίδραση της χρονικής στιγμής εκκίνησης του migration . . . 45
3.5 Ελάχιστη τιμή του *downtime-limit* για επιτυχές Live Migration κάθε εφαρμογής . . . 48
3.6 MPKI με χρήση του *cache-misses* counter του εργαλείου *perf* . . . 51

4.1 Πειραματική αξιολόγηση του WWS . . . 58
4.2 Πειραματική ρύθμιση της παραμέτρου *"period"* του BitmapTrace . . . 61
4.2 Πειραματική ρύθμιση της παραμέτρου *"period"* του BitmapTrace (συνέχεια) . . . 62
4.3 Πειραματική ρύθμιση της παραμέτρου *"iterations"* του BitmapTrace . . . 63
4.3 Πειραματική ρύθμιση της παραμέτρου *"iterations"* του BitmapTrace (συνέχεια) . . . 64

5.1 AVG και STDEV Dirty Pages κατά την profiling περίοδο . . . 69
5.1 AVG και STDEV Dirty Pages κατά την profiling περίοδο (συνέχεια) . . . 70
5.2 Διάγραμμα ροής του K-fold cross-validation . . . 74
5.3 Διάγραμμα ροής εργασιών των μοντέλων πρόβλεψης . . . 76
5.4 *max_downtime* και *προβλεπόμενο min_success_downtime-limit* για VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις . . . 80
5.5 *max_downtime* και *προβλεπόμενο min_success_downtime-limit* για VMs με ίδια εφαρμογή και με SLA παραβιάσεις . . . 82
5.6 *max_downtime* και *προβλεπόμενο min_success_downtime-limit* για VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις . . . 84

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Το live migration είναι ένα απαραίτητο εργαλείο για την αποδοτική διαχείριση των πόρων ενός υπολογιστικού κέντρου και χρησιμεύει τόσο στην ενοποίηση των εικονικών μηχανών σε έναν κόμβο όσο και στην εξισορρόπηση του φόρτου μεταξύ των φυσικών μηχανημάτων. Στην περίπτωση του pre-copy live migration η μνήμη μεταφέρεται επαναληπτικά από τον αρχικό κόμβο στον κόμβο προορισμού μέχρι το υπολειπόμενο κομμάτι της dirty μνήμης να είναι αρκετά μικρό ώστε να μπορεί να μεταφερθεί σε μια μόνο επανάληψη. Έτσι, στο τελευταίο iteration μεταφέρονται το κομμάτι αυτό της μνήμης μαζί με την κατάσταση των συσκευών του VM σε χρονικό διάστημα το πολύ ίσο με το μέγιστο επιτρεπτό χρόνο μη λειτουργίας (downtime), ο οποίος έχει οριστεί από συμφωνίες σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου (SLAs) [1, 2]. Επιπλέον, οι cloud πάροχοι χρειάζεται να διαχειριστούν με προσοχή τα workloads που είναι απαιτητικά σε πόρους, όπως για παράδειγμα εφαρμογές με έντονη δραστηριότητα μνήμης (memory-intensive), λόγω της απρόβλεπτης συμπεριφοράς τους ως προς τη μνήμη και της πιθανής ανάγκης για μεγάλο downtime κατά το migration. Οι παράγοντες αυτοί οδηγούν σε μη σύγκλιση του pre-copy live migration αλγορίθμου, όπως και στην περίπτωση του QEMU/KVM όπου το migration δεν ολοκληρώνεται.

Συχνά οι cloud πάροχοι υπηρεσιών περιμένουν την ολοκλήρωση ενός pre-copy live migration για αρκετό χρόνο έως ότου το ακυρώσουν ή εφαρμόσουν κάποια τεχνική βελτιστοποίησης με σκοπό να μειώσουν το πλήθος των μεταφερόμενων δεδομένων (total transferred data), το συνολικό χρόνο του migration (total time) και το χρόνο μη λειτουργίας (downtime). Όμως, οι μέθοδοι αυτοί οδηγούν σε αρνητική επίδραση στην απόδοση της προσφερόμενης υπηρεσίας (QoS) και πιθανό φόρτο στο σύστημα και στο δίκτυο παραβιάζοντας το SLA.

Κατά την εκτέλεση διαχειριστικών εργασιών που υλοποιούν το live migration πρέπει να αποφασίζεται όχι μόνο οι υποψήφιες εικονικές μηχανές για migration αλλά και και η σειρά της μεταφοράς τους. Υποθέτοντας ένα σενάριο χρονοδρομολόγησης όπου 8 VMs συνυπάρχουν σε έναν host και ο cloud διαχειριστής πρέπει να μετακινήσει 4 από αυτά, τότε υπάρχουν 70 πιθανοί συνδυασμοί. Πρέπει, λοιπόν, να επιλεχθούν τα 4 ”καλύτερα” υποψήφια VMs λαμβάνοντας υπόψη τα χαρακτηριστικά του SLA με τον κάθε πελάτη αλλά και τις πολιτικές λειτουργίας του υπολογιστικού κέντρου. Επίσης, ο system administrator πρέπει να αποφασίσει αν η μεταφορά των εικονικών θα γίνει σειριακά ή παράλληλα. Με την παράλληλη μεταφορά υπάρχει ο κίνδυνος να μην ολοκληρωθεί επιτυχώς κανένα VM live migration λόγω του από κοινού μοιραζόμενου εύρους ζώνης του δικτύου. Για τους λόγους αυτούς και προκειμένου το data center να παρέχει υψηλή διαθεσιμότητα υπηρεσιών αποφεύγοντας πιθανές παραβιάσεις του SLA συνίσταται η παρακολούθηση της

λειτουργίας των υποψήφιων για migration VMs.

Στην εργασία αυτή προτείνεται ένα μοντέλο επιβλεπόμενης μάθησης (supervised learning model) το οποίο προβλέπει με μεγάλη ακρίβεια την ελάχιστη τιμή του downtime-limit που απαιτείται ώστε το pre-copy live migration να ολοκληρωθεί επιτυχώς εντός ενός χρονικού παραθύρου. Με τον όρο *downtime-limit* αναφερόμαστε στο μέγιστο επιτρεπτό χρόνο μη λειτουργίας του VM στο τελικό στάδιο του pre-copy live migration. Οι παράμετροι που δίνονται ως input στο μοντέλο συλλέγονται με τη βοήθεια ενός εργαλείου παρακολούθησης (monitoring tool), το οποίο ονομάζεται BitmapTrace και ενσωματώνεται στον υπάρχοντα migration κώδικα του QEMU/KVM. Το συγκεκριμένο εργαλείο πραγματοποιεί profiling του VM για ένα συγκεκριμένο χρονικό διάστημα κατά το οποίο αποθηκεύει στιγμιότυπα της συμπεριφοράς του VM ως προς τις εγγραφές στη μνήμη. Η εκτέλεση του workload στο VM επιβαρύνεται μόνο με μια καθυστέρηση μερικών δευτερολέπτων, μικρότερη του profiling χρονικού παραθύρου. Επιπλέον, υλοποιούμε ένα framework στο οποίο οι εικονικές μηχανές που εκτελούνται σε ένα φυσικό μηχάνημα ταξινομούνται με βάση το αποτύπωμα μνήμης τους και τις πολιτικές λειτουργίας του υπολογιστικού κέντρου. Τα επιλεχθέντα VMs μεταφέρονται σειριακά στον host προορισμού ικανοποιώντας τους SLA όρους του κάθε πελάτη στο μέγιστο δυνατό βαθμό. Ταυτόχρονα, η απόφαση του μοντέλου για την επιλογή των υποψήφιων VMs συμβάλλει στη μείωση του αθροιστικού συνολικού χρόνου για τη μεταφορά του επιθυμητού αριθμού εικονικών μηχανών.

1.2 Συνεισφορά

Η συνεισφορά της διπλωματικής εργασίας συνοψίζεται ως εξής:

(i) Οι κύριες παράμετροι του QEMU/KVM pre-copy live migration αξιολογούνται με διαφορετικά workloads. Από τα αποτελέσματα που προκύπτουν, αναλύεται η επίδραση των παραμέτρων *max-bandwidth* και *downtime-limit* στην απόδοση του VM live migration ως προς τα χαρακτηριστικά total time και downtime.

(ii) Ο σχεδιασμός και η υλοποίηση ενός εργαλείου παρακολούθησης του αποτυπώματος μνήμης (memory footprint) ενός VM πριν την εκκίνηση του migration. Το module αυτό, το οποίο ονομάζεται BitmapTrace και ενσωματώνεται στον υπάρχων migration κώδικα του QEMU/KVM, είναι υπεύθυνο για την αποθήκευση των νέων dirty pages που παράγονται ανά περιοδικά διαστήματα καθώς το VM εκτελείται. Επίσης, παραθέτουμε μια ανάλυση για τις προτεινόμενες τιμές των παραμέτρων του BitmapTrace οι οποίες εξασφαλίζουν ένα επαρκές διάστημα για profiling της μνήμης.

(iii) Η υλοποίηση τεχνικών πρόβλεψης σε VM live migration framework με σκοπό τη μοντελοποίηση της συσχέτισης της συμπεριφοράς ενός VM ως προς το πλήθος εγγραφών στη μνήμη με την ελάχιστη τιμή της παραμέτρου *downtime-limit* που εξασφαλίζει την επιτυχή ολοκλήρωση του pre-copy live migration. Υποθέτοντας ότι σε έναν κόμβο εκτελούνται πολλά VMs, επιτυγχάνεται η επιλογή για migration ενός υποσυνόλου VMs τα οποία ικανοποιούν τόσο τους κανόνες λειτουργίας του cloud computing περιβάλλοντος όσο και τους περιορισμούς που υπάρχουν λόγω των SLAs (π.χ. μέγιστο downtime κατά το migration).

Κεφάλαιο 2

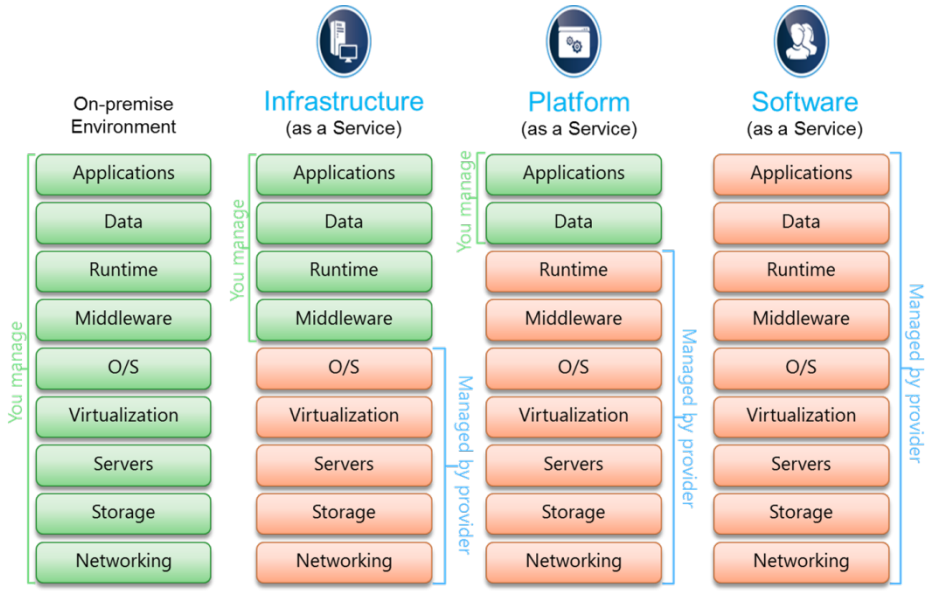
Θεωρητικό Υπόβαθρο

Σε αυτό το κεφάλαιο παρουσιάζεται μια αναλυτική περιγραφή των εννοιών που σχετίζονται με το Υπολογιστικό Νέφος, οι οποίες είναι σημαντικές για την κατανόηση της παρούσας εργασίας. Αφού αναφερθεί ο ορισμός του cloud computing, αξιολογούνται οι τεχνικές εικονικοποίησης που αποτελούν την βασική τεχνολογία του και αναλύονται τα διαθέσιμα λογισμικά ελέγχου. Στις υπόλοιπες ενότητες, περιγράφεται αναλυτικά η έννοια του live migration των εικονικών μηχανών με έμφαση στην pre-copy τεχνική αντιγραφή της μνήμης του QEMU/KVM hypervisor. Τέλος, παρατίθενται οι κατηγορίες εφαρμογών που εκτελούνται σε μια εικονική μηχανή και τα χαρακτηριστικά τους.

2.1 Υπολογιστικό Νέφος

Σύμφωνα με το National Institute of Standards and Technology (NIST) [3], το Υπολογιστικό Νέφος (Cloud Computing) ορίζεται ως το μοντέλο που επιτρέπει την ευρεία, βολική, κατ' απαίτηση δικτυακή πρόσβαση σε μία κοινόχρηστη δεξαμενή από υπολογιστικούς πόρους (π.χ. δίκτυα, διακομιστές (server), αποθηκευτικοί χώροι, εφαρμογές και υπηρεσίες) που μπορούν άμεσα να παρέχονται και να απελευθερωθούν με ελάχιστη διαχειριστική προσπάθεια από τους παρόχους υπηρεσιών. Αυτό το μοντέλο Cloud αποτελείται από πέντε σημαντικά χαρακτηριστικά (αυτοεξυπηρέτηση κατ' απαίτηση, ευρεία δικτυακή πρόσβαση, διάθεση πόρων, γρήγορη ελαστικότητα, μετρήσιμη υπηρεσία). Σύμφωνα με τον ορισμό του NIST, τα τρία μοντέλα υπηρεσιών είναι τα "Υποδομή ως Υπηρεσία" (IaaS), "Λογισμικό ως Υπηρεσία" (SaaS) και "Πλατφόρμα ως Υπηρεσία" (PaaS), όπως φαίνονται και στο σχήμα 2.1.

- *"Λογισμικό ως Υπηρεσία" (SaaS)*: Η υπηρεσία αυτή δίνει τη δυνατότητα στον χρήστη να χρησιμοποιήσει τις εφαρμογές λογισμικού του παρόχου που υπάρχουν στο Cloud.
- *"Πλατφόρμα ως Υπηρεσία" (PaaS)*: Σε αυτό το μοντέλο υπηρεσιών ο πελάτης έχει τον έλεγχο των εφαρμογών που ο ίδιος ανέπτυξε ή απέκτησε χρησιμοποιώντας εργαλεία ή γλώσσες προγραμματισμού που παρέχονται από τον πάροχο μέσω της υποδομής του νέφους.
- *"Υποδομή ως Υπηρεσία" (IaaS)*: Σε αυτή την υπηρεσία παρέχεται στον πελάτη το υλικό. Δηλαδή δίνεται η δυνατότητα στο χρήστη να χρησιμοποιήσει αποθηκευτικά μέσα, επεξεργαστική ισχύ και άλλους υπολογιστικούς όρους για να "τρέξει" δικό του λογισμικό (λειτουργικό σύστημα ή εφαρμογές).



Σχήμα 2.1: Μοντέλα υπηρεσιών Cloud

Όταν το Cloud είναι διαθέσιμο στο κοινό ως υπηρεσία τύπου "Πληρώνεις όσο χρησιμοποιείς" (Pay-As-You-Use), τότε αποκαλείται "Δημόσιο Νέφος" (Public Cloud); η υπηρεσία πωλείται για παροχή υπολογιστικών υπηρεσιών (Utility Computing). Παραδείγματα αυτού του τύπου είναι τα Amazon EC2 [4], GoogleAppEngine [5] και Microsoft Azure [6]. Ο όρος "Ιδιωτικό Νέφος" (Private Cloud) αναφέρεται στα ιδιωτικά υπολογιστικά κέντρα (data centers) μιας επιχείρησης ή κάποιου οργανισμού τα οποία δεν είναι διαθέσιμα στο ευρύ κοινό. Το "Νέφος κοινότητας" (Community Cloud) είναι η υποδομή που παρέχεται σε μια συγκεκριμένη κοινότητα χρηστών από οργανισμούς με κοινά ενδιαφέροντα και κοινές ανάγκες. Τέλος, το "Υβριδικό νέφος" (Hybrid Cloud) αποτελείται από 2 ή περισσότερα νέφη διαφορετικού είδους.

Ολοένα και περισσότεροι οργανισμοί "μετακινούν" τα συστήματα, τις εφαρμογές και τα δεδομένα τους στο Cloud. Για αυτό το λόγο είναι σημαντικό να παρέχονται καθαρές εγγυήσεις από τους Cloud παρόχους για τις υπηρεσίες που προσφέρουν οι οποίες διασαφηνίζονται σε συμφωνίες σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου (Service-level Agreements - SLAs). Οι οργανισμοί πρέπει να ορίσουν χρονικά περιθώρια πλήρους λειτουργίας ή μερικής λειτουργίας, και κάθε εφαρμογή που λειτουργεί σε περιβάλλον νέφους πρέπει να καλύπτει (στο ελάχιστο) τις προϋποθέσεις για διαθεσιμότητα και ανάκτηση δεδομένων. Οι επιχειρήσεις, λοιπόν, οφείλουν να σταθμίσουν τα υπέρ και τα κατά για τη χρήση των Cloud παρόχων λαμβάνοντας υπόψη τα κόστη μιας πιθανής μη λειτουργίας.

2.2 Εικονικοποίηση

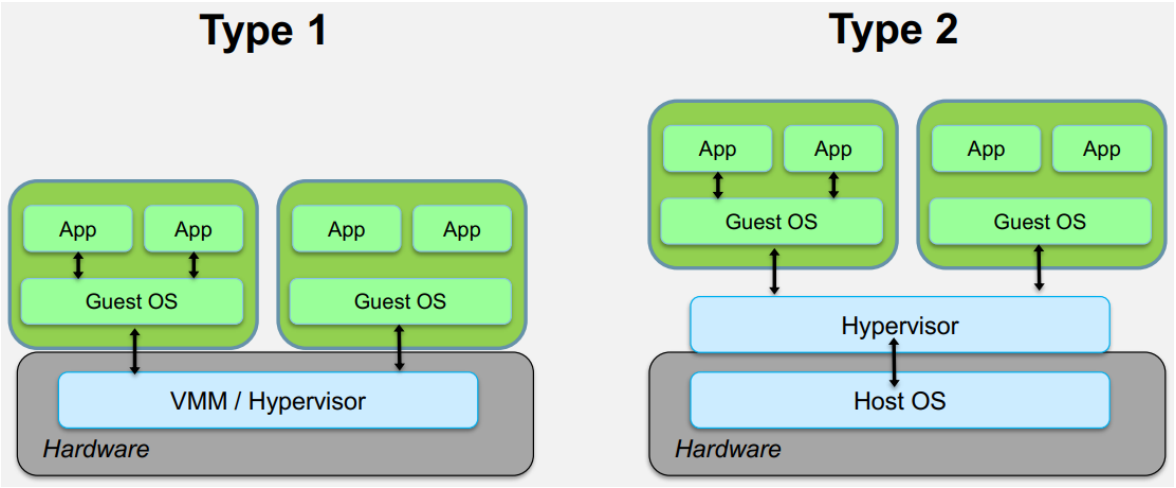
Σε περιβάλλοντα Υπολογιστικού Νέφους, η Υποδομή-ως-Υπηρεσία (IaaS) παρέχει κατά παραγγελία εικονικές μηχανές με τεχνολογίες εικονικοποίησης. Εικονικοποίηση είναι η διαδικασία της δημιουργίας μιας εικονικής συσκευή ή πόρου, όπως εικονικές πλατφόρμες υλικού, συσκευές αποθηκευτικού χώρου ή πόροι δικτύων υπολογιστών. Η τεχνολογία αυτή επιτρέπει τη δημιουργία

πολλαπλών περιβαλλόντων προσομοίωσης ή εξειδικευμένους πόρους από ένα μοναδικό, φυσικό μηχάνημα υλικού. Συγκεκριμένα, η προσομοίωση λογισμικού ενός υπολογιστικού συστήματος ονομάζεται Εικονική Μηχανή (*VM* - Virtual Machine). Το πραγματικό υλικό πάνω στο οποίο τρέχει το VM είναι ευρέως γνωστό ως φυσικό μηχάνημα (”host”), και το εικονικό μηχάνημα που προσομοιώνει αναφέρεται ως φιλοξενούμενος/εικονικό σύστημα (”guest”).

Η εικονικοποίηση εμφανίστηκε αρχικά τη δεκαετία του 1960 ως μια μέθοδος λογικού διαχωρισμού των υπολογιστικών πόρων που παρέχονταν μεγάλα, συγκεντρωτικά συστήματα (mainframes) σε διαφορετικές εφαρμογές και έχει γίνει πλέον κοινή πρακτική λόγω της εξέλιξης των υπολογιστικών κέντρων και του υπολογιστικού νέφους. Μέσω της εικονικοποίησης, οι Cloud πάροχοι είναι δυνατόν να προσομοιώνουν ταυτόχρονα πολλαπλές εικονικές μηχανές, εντελώς απομονωμένες μεταξύ τους, στον ίδιο host.

2.2.1 Hypervisor

Ο επόπτης ή λογισμικό ελέγχου (hypervisor) είναι ένα λογισμικό που είναι υπεύθυνο για τη δημιουργία, τον έλεγχο και την παρακολούθηση των εικονικών μηχανών. Ο κύριος ρόλος του επόπτη είναι η κατανομή των υπολογιστικών πόρων στους guests για την εικονικοποίηση υλικού. Αυτό επιτυγχάνεται με ένα μηχανισμό αφαίρεσης, στοχευμένο στην απόκρυψη λεπτομερειών της υλοποίησης και της κατάστασης των υπολογιστικών πόρων από πελάτες των πόρων αυτών. Οι επόπτες κατηγοριοποιούνται σε τύπου-1 και τύπου-2 επόπτες. Οι τύπου-1 επόπτες τρέχουν απευθείας πάνω στο υλικό του συστήματος, για αυτό και αποκαλούνται και ως ”bare-metal hypervisors”. Οι τύπου-2 επόπτες τρέχουν πάνω σε ένα σύστημα που ήδη έχει εγκατεστημένο ένα λειτουργικό. Οι τύπου-1 hypervisors χειρίζονται διαχειριστικές εργασίες τόσο στο υλικό όσο και στο λογισμικό του φυσικού μηχανήματος, ενώ στους τύπου-2 hypervisors το λειτουργικό σύστημα αναλαμβάνει τις εργασίες αυτές.



Σχήμα 2.2: Τύποι Λογισμικών Ελέγχου (hypervisors)

2.2.2 Τεχνικές Εικονικοποίησης

Ανάλογα με τον τύπο των εφαρμογών που προσπαθούν να προσομοιώσουν και τη χρησιμοποίηση του υλικού, τα είδη εικονικοποίησης είναι τα εξής:

- **Εικονικοποίηση υλικού**
  Η κύρια τεχνική εικονικοποίησης είναι η εικονικοποίηση υλικού (server/hardware virtualization), η οποία σχετίζεται και με τη διαδικασία της ζωντανής μεταφοράς (live migration) των εικονικών μηχανών, όπως περιγράφεται και στην ενότητα 2.3. Στην εικονικοποίηση υλικού οι χρήστες μπορούν να προσομοιώνουν διαφορετικά λειτουργικά συστήματα στο ίδιο πραγματικό σύστημα την ίδια χρονική στιγμή. Με άλλα λόγια, ένα ή περισσότερα εικονικά μηχανήματα (guests) δημιουργούνται και μοιράζονται τους πόρους ενός φυσικού μηχανήματος (host). Ανάλογα με το πόσο το φιλοξενούμενο λειτουργικό σύστημα γνωρίζει ότι εικονικοποιείται, υπάρχουν τρεις διαφορετικά είδη εικονικοποίησης υλικού:

  - **Πλήρης εικονικοποίηση (Full Virtualization)**
    Με αυτή την τεχνική εικονικοποιούνται όλα τα υπολογιστικά εξαρτήματα ενός μηχανήματος για κάθε φιλοξενούμενο, όπως η CPU, η μνήμη, μονάδες εισόδου-εξόδου (I/O) και ο δίσκος. Το εικονικό μηχάνημα δεν έχει γνώση ότι βρίσκεται σε ένα εικονικό περιβάλλον.
  - **Παραεικονικοποίηση (Paravirtualization)**
    Με της τεχνική της παραεικονικοποίησης ο πυρήνας του φιλοξενούμενου λειτουργικού συστήματος έχει επίγνωση ότι εικονικοποιείται και έχει ειδικά διαμορφωμένους οδηγούς (drivers) ώστε οι εντολές να εκτελούνται από το λογισμικό ελέγχου και όχι απευθείας από το υλικό. Έτσι, πραγματοποιούνται κλήσεις συστήματος προς τον επόπτη, οι οποίες ονομάζονται υπερκλήσεις (hypercalls), με σκοπό την ταχύτερη εκτέλεση ορισμένων λειτουργιών εκ μέρους του φιλοξενούμενου λειτουργικού συστήματος.
  - **Εικονικοποίηση υποβοηθούμενη από το υλικό (Hardware Assisted Virtualization)**
    Η τεχνική αυτή στοχεύει στην αποτελεσματική χρήση της πλήρους εικονικοποίησης με βοήθεια από τις δυνατότητες υλικού. Έτσι, το guest φιλοξενούμενο λειτουργικό σύστημα χρησιμοποιεί το ίδιο σύνολο εντολών όπως το host μηχάνημα βελτιώνοντας αισθητά την απόδοση του για διάφορες προκλήσεις εικονικοποίησης, όπως η μετάφραση εντολών και διευθύνσεων μνήμης. Η εικονικοποίηση υποβοηθούμενη από το υλικό προστέθηκε στους επεξεργαστές x86 Intel (**Intel VT-x**) and AMD (**AMD-V**).

- **Εικονικοποίηση εφαρμογών (Application Virtualization)**
  Εικονικοποίηση εφαρμογών είναι ο διαχωρισμός της εγκατάστασης μιας εφαρμογής από το σύστημα που θα έχει πρόσβαση σε αυτή. Οι εικονικές εφαρμογές περικλείονται σε απομονωμένα εικονικά περιβάλλοντα (sandboxes) με ενσωματωμένα αρχεία εκτελέσιμου κώδικα, με αποτέλεσμα οι εφαρμογές αυτές να είναι φορητές και εύκολες να εγκατασταθούν σε διάφορες πλατφόρμες υλικού και λειτουργικών συστημάτων.

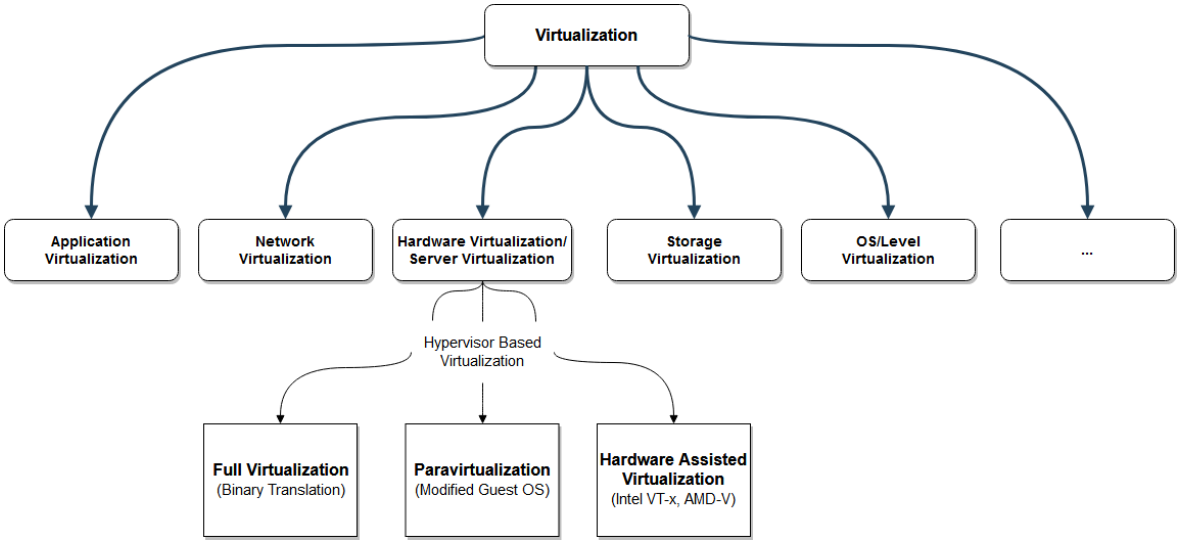
- **Εικονικοποίηση λειτουργικού συστήματος (Operating System(OS) Virtualization)**
  Με την τεχνική αυτή επιτρέπεται στο ίδιο φυσικό μηχάνημα να τρέχουν διαφορετικές εφαρμογές πάνω από ένα κοινό Λειτουργικό Σύστημα. Ο host, λοιπόν, τρέχει τις εφαρμογές σε απομονωμένα περιβάλλοντα εκτέλεσης (containers). Αν και τα containers τρέχουν πάνω από τον ίδιο πυρήνα, κάθε ένα έχει δικό του σύστημα αρχείων (file system), διεργασίες, μνήμη κλπ.

- **Εικονικοποίηση δικτύου (Network Virtualization)**

Η εικονικοποίηση δικτύου είναι μια μέθοδος που συνδυάζει τους πόρους του δικτύου υπολογιστών σε μία ενιαία πλατφόρμα, γνωστή ως εικονικό δίκτυο (Virtual network). Αυτό επιτυγχάνεται με την βοήθεια λογισμικού και υπηρεσιών που επιτρέπουν την κοινή χρήση δικτυακών πόρων.

- **Εικονικοποίηση αποθήκευσης (Storage Virtualization)**
  Με την τεχνική της εικονικοποίησης αποθήκευσης, οι πραγματικοί αποθηκευτικοί χώροι από πολλαπλά δίκτυα αποθηκευτικών συσκευών ομαδοποιούνται ώστε να εμφανίζονται σαν μια ενιαία αποθηκευτική συσκευή.

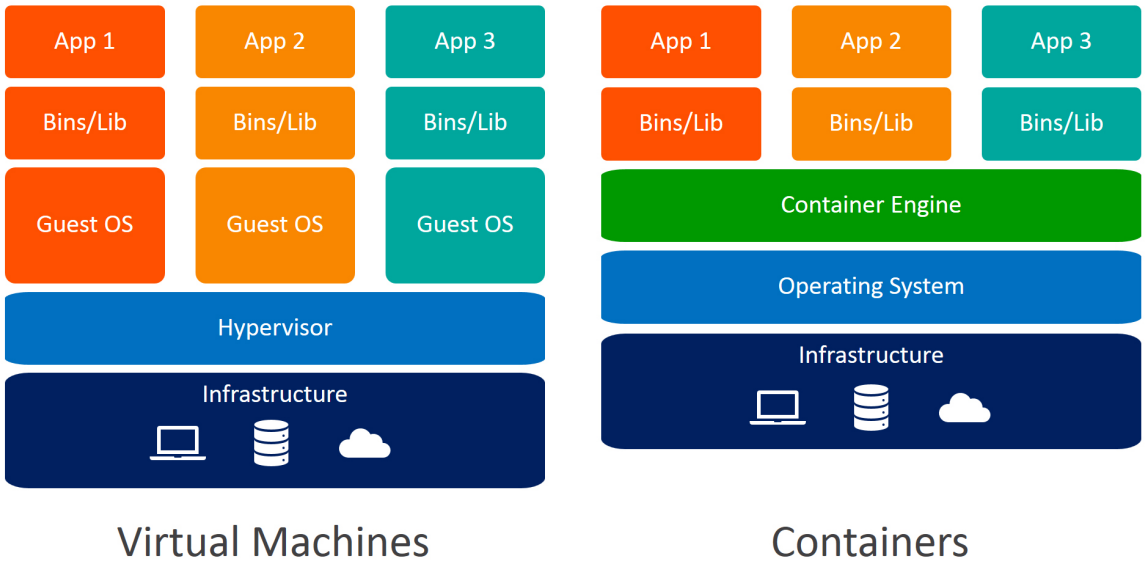
  Οι παραπάνω τεχνικές εικονικοποίησης συνοψίζονται στο ακόλουθο σχήμα :



Σχήμα 2.3: Τεχνικές Εικονικοποίησης

2.2.3 Διαφοροποίηση Εικονικών Μηχανών και Containers

Η κύρια διαφορά των containers και των εικονικών μηχανών έγκειται ότι στην πρώτη τεχνολογία εικονικοποιείται το λειτουργικό σύστημα ώστε πολλαπλές εφαρμογές να εκτελούνται στο ίδιο λειτουργικό σύστημα, ενώ στις εικονικές μηχανές το υλικό προσομποιώνεται ώστε να εκτελούνται πολλαπλά λειτουργικά συστήματα. Στην περίπτωση των Linux containers, το λογισμικό χρησιμοποιεί τεχνολογίες του πυρήνα του όπως τα cgroups και οι χώροι ονομάτων πυρήνα (kernel namespaces), το οποίο στην περίπτωση των εικονικών μηχανών δεν γίνεται, όπως π.χ. στο KVM. Επίσης, ο διαμοιρασμός αρχείων μεταξύ container και host έχει ως αποτέλεσμα τα containers να αποτελούν μια ”ελαφριά” τεχνολογία, αφού το μέγεθος τους είναι πολύ μικρότερο (μερικά Megabyte) σε σύγκριση με αυτό μιας εικονικής μηχανής (μερικά Gigabyte). Έτσι, η ταχύτητα, η ευελιξία και η φορητότητα τους καθιστούν τα containers ένα ακόμα ισχυρό εργαλείο στην ανάπτυξη λογισμικού τα τελευταία χρόνια.



Σχήμα 2.4: Virtual Machines vs Containers

2.2.4 QEMU/KVM

Το QEMU είναι ένας τύπου-2 λογισμικό ελέγχου (hypervisor) το οποίο εκτελείται σε περιβάλλον χρήστη (user space) και πραγματοποιεί εικονικοποίηση υλικού. Το QEMU μπορεί να εκτελεστεί αυτόνομα, αλλά επειδή η προσομοίωση γίνεται εξ' ολοκλήρου από το λογισμικό, είναι πολύ αργό. Αυτό μπορεί να αποφευχθεί με τη βοήθεια του KVM, το οποίο λειτουργεί ως επιταχυντής έχοντας πρόσβαση στις επεκτάσεις εικονικοποίησης του επεξεργαστή. Στην τελευταία περίπτωση, η εικονική μηχανή είναι απαραίτητο να είναι της ίδιας αρχιτεκτονικής με το μηχάνημα στο οποίο εκτελείται.

QEMU

Το **QEMU(Quick Emulator)** είναι μια πλατφόρμα γενικού σκοπού και ανοιχτού κώδικα (open source) για την προσομοίωση (machine emulator) και την εικονικοποίηση συστημάτων (virtualizer) [7].

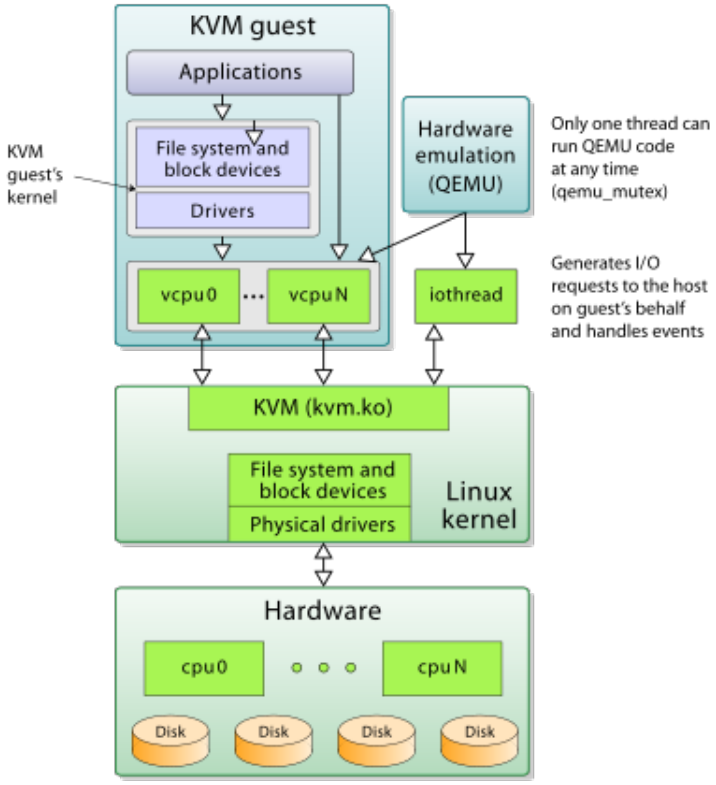
Όταν χρησιμοποιείται ως machine emulator, το QEMU μπορεί να εκτελέσει λειτουργικά συστήματα και προγράμματα τα οποία έχουν δημιουργηθεί για κάποιο συγκεκριμένο είδος μηχανημάτων (π.χ. μια πλακέτα με ARM επεξεργαστή) σε κάποιο άλλο μηχάνημα. Το QEMU προσομοιώνει τις CPUs μέσω τεχνικών δυαδικής μετάφρασης και παρέχει ένα σύνολο από μοντέλα συσκευών. Το *Tiny Code Generator (TCG)* [8] είναι υπεύθυνο για τη δυαδική μετάφραση, δηλαδή την μετατροπή του δυαδικού κώδικα που έχει γραφτεί για έναν συγκεκριμένο επεξεργαστή σε κάποιον άλλον. Όταν το QEMU χρησιμοποιείται ως virtualizer τότε εκτελεί τον κώδικα του φιλοξενούμενου στη CPU του φυσικού μηχανήματος, όπως για παράδειγμα όταν λειτουργεί με τη βοήθεια των Xen/KVM hypervisors.

Συνοπτικά, το QEMU μπορεί να εκτελεστεί χωρίς το KVM χρησιμοποιώντας τη μέθοδο της δυαδικής μετάφρασης. Όταν εκτελείται με αυτό τον τρόπο θα είναι πιο αργό σε σχέση με τη χρήση επεκτάσεων εικονικοποίησης του επεξεργαστή από το KVM. Σε κάθε τρόπο λειτουργίας (emulator ή virtualizer) το QEMU προσομοιώνει τις συσκευές υλικού και τις περιφερειακές συσκευές, όπως

δίσκοι, κάρτες δικτύου, δίαυλοι, VGA, σειριακές πόρτες, USB. Επιτυγχάνει λοιπόν, με τη βοήθεια του KVM, να δημιουργεί και να αρχικοποιεί εικονικές μηχανές, όπου κάθε εικονική μηχανή νοείται ως μία διεργασία και χρονοδρομολογείται μαζί με τις υπόλοιπες διεργασίες του λειτουργικού.

Μια κονσόλα παρακολούθησης, η οποία ονομάζεται *Human Monitor Interface (HMP)* [9], παρέχεται για την αλληλεπίδραση με το χρήστη καθώς το QEMU εκτελείται. Επιπλέον, το QEMU παρέχει ένα JSON πρωτόκολλο επικοινωνίας με το χρήστη, το οποίο ονομάζεται *QEMU Machine Protocol (QMP)* [10].

Στη συγκεκριμένη διπλωματική, χρησιμοποιήθηκε η έκδοση 2.10.1 (stable version) του QEMU και ο πηγαίος κώδικας υπάρχει στο https://www.qemu.org/.



Σχήμα 2.5: Επισκόπηση του QEMU/KVM περιβάλλοντος εικονικοποίησης

KVM

Το **Kernel-based Virtual Machine (KVM)** [11] είναι ένα υποσύστημα εικονικοποίησης που έχει ενσωματωθεί στον πηγαίο κώδικα του πυρήνα του Linux από την έκδοση 2.6.20 και προσφέρει έναν ενιαίο τρόπο για χρήση των επεκτάσεων εικονικοποίησης του επεξεργαστή (Intel VT or AMD-V). Χρησιμοποιώντας την τεχνική πλήρους εικονικοποίησης υποστηρίζει αρχιτεκτονικές x86. Κατά τη φόρτωση της μονάδας KVM, το Linux συνεχίζει τη λειτουργία του ως λειτουργικό σύστημα, αλλά ταυτόχρονα αποκτά και το ρόλο του ελεγκτή εικονικών μηχανών. Αποτελείται από δύο modules, το *kvm.ko* και το *kvm-intel.ko* ή *kvm-amd.ko* ανάλογα με το μοντέλο του επεξεργαστή του φυσικού μηχανήματος. Κατά τη φόρτωση της μονάδας του KVM, εμφανίζεται στο σύστημα αρχείων η συσκευή χαρακτήρων */dev/kvm*, που αναπαριστά τη διεπαφή του KVM με το user space. Μέσω ενός συνόλου κλήσεων ioctls() επιτρέπει τον έλεγχο του εικονικού περιβάλλοντος. Το QEMU, ως εργαλείο virtualization επιπέδου χρήστη, προσομοιώνει εικονικά συστήματα και όπως φαίνεται

στην εικόνα 2.5 ο εικονικός επεξεργαστής (vCPU) αποτελεί τη χαμηλότερη μονάδα εικονικοποίησης πάνω από την οποία ”χτίζεται” η εικονική μηχανή. Στο KVM, κάθε εικονική μηχανή νοείται ως μία διεργασία και χρονοδρομολογείται μαζί με τις υπόλοιπες διεργασίες του λειτουργικού. Όταν μια εικονική μηχανή εκτελεί μια εντολή Ε/Ε, λαμβάνεται από το KVM και ανακατευθύνεται στην αντίστοιχη διεργασία QEMU.

Η εκκίνηση του QEMU με χρήση του KVM γίνεται με τη παράμετρο *–enable-kvm* και αναφέρεται ως QEMU/KVM hypervisor. Με αυτό τον τρόπο η εικονική μηχανή έχει καλύτερη απόδοση σε σύγκριση με άλλες λειτουργίες του QEMU, όπως για παράδειγμα ως machine emulator όπου γίνεται χρήση του TCG.

2.2.5 XEN

Το **Xen** [12, 13] είναι ανοιχτού κώδικα και είναι τύπου-1 hypervisor, δηλαδή λειτουργεί ακριβώς πάνω από το υλικό του φυσικού μηχανήματος και μπορεί να εκτελέσει παράλληλα εικονικές μηχανές πάνω από ένα φυσικό μηχάνημα (host). Το Xen υποστηρίζει δύο τεχνικές εικονικοποίησης: παρα-εικονικοποίηση (paravirtualization - PV) και εικονικοποίηση βοηθούμενη από το υλικό (Hardware Virtualized Machine – HVM) ή πλήρη εικονικοποίηση (Full Virtualization). Η HVM τεχνική υποστηρίζεται μέσω επεκτάσεων εικονικοποίησης στην CPU. Και οι δύο αυτές τεχνικές μπορούν να χρησιμοποιηθούν ταυτόχρονα από ένα μηχάνημα με Xen επόπτη. Επίσης, είναι δυνατό να χρησιμοποιηθούν τεχνικές παραεικονικοποίησης σε έναν HVM guest. Τα φιλοξενούμενα λειτουργικά συστήματα που ”τρέχουν” σε ένα Xen επόπτη ονομάζονται Domains. Το domain0 (“*dom0*”) ενεργοποιείται από τον hypervisor του Xen κατά την αρχική εκκίνηση του συστήματος και έχει πρόσβαση στο υλικό και διαχειριστικά προνόμια. Μόλις εκκινήσει το *dom0*, τα υπόλοιπα φιλοξενούμενα domain (“*domU*”) ελέγχονται από το *dom0* και λειτουργούν ξεχωριστά στο σύστημα. Ο hypervisor έχει άμεση επαφή με το υλικό και η πρόσβαση των *domU* στις φυσικές συσκευές γίνεται μέσω του dom0.

2.3 Ζωντανή Μεταφορά (Live Migration) Εικονικών Μηχανών

Μεταφορά (Migration) μιας εικονικής μηχανής είναι η διαδικασία της μετακίνησης του φιλοξενούμενου (guest) από ένα φυσικό μηχάνημα σε ένα άλλο. Με αυτή τη διαδικασία μεταφέρονται όλα οι πόροι που σχετίζονται με το εικονικό μηχάνημα, αποτελούμενοι από την κατάσταση των συσκευές και την εικονική CPU, καθώς και από τη μνήμη, η οποία απαιτεί και τον περισσότερο χρόνο για τη μεταφορά της. Η μετακίνηση του τοπικού συστήματος αρχείων μπορεί να αποφευχθεί με τη χρήση ενός NFS δομής αποθήκευσης [14] στην οποία τόσο ο αρχικός host (source) όσο και ο host προορισμού (destination) έχουν πρόσβαση.

Υπάρχουν δύο τύποι μεταφοράς της μνήμης: ψυχρή (offline) και ζωντανή (live). Κατά την ψυχρή μεταφορά το φιλοξενούμενο μηχάνημα είτε έχει διακόψει τη λειτουργία του είτε είναι απενεργοποιημένο. Έτσι, μια εικόνα της μνήμης του guest μεταφέρεται στον host προορισμού όπου συνεχίζεται η λειτουργία του φιλοξενούμενου μηχανήματος αποδεσμεύοντας τη μνήμη που χρησιμοποιούνταν στον source host. Αντίθετα, κατά τη ζωντανή μεταφορά η εικονική μηχανή μεταφέρεται από τον ένα φυσικό μηχάνημα σε κάποιο άλλο καθώς το φιλοξενούμενο μηχάνημα εκτελείται. Οι υπηρεσίες που εκτελούνται στο VM δεν γνωρίζουν για το migration, εκτός από ένα μικρό χρονικό παράθυρο

όπου ίσως χρειαστεί να σταματήσουν τη λειτουργία τους ώστε ο νέος επόπτης στον destination host να εκκινήσει το φιλοξενούμενο μηχάνημα.
Η χρησιμότητα του live migration έγκειται στους παρακάτω λόγους :

- **Εξισορρόπηση φόρτου μεταξύ φυσικών μηχανημάτων**: Οι πόροι των φυσικών servers μπορεί να υπερφορτωθούν ή να μην επαρκούν αν στο φυσικό μηχάνημα "τρέχουν" πολλά VMs. Το live migration μερικών εικονικών μηχανών σε άλλους hosts με λιγότερο ή πιο ελαφρύ φορτίο μπορεί να αντιμετωπίσει αποτελεσματικά αυτό το πρόβλημα της ασυμφωνίας εκμετάλλευσης των φυσικών πόρων του data center.
- **Ενοποίηση διακομιστών**: Οι πάροχοι οφείλουν να χρησιμοποιούν βέλτιστα τους διαθέσιμους πόρους αποφεύγοντας την ύπαρξη κόμβων με ανεκμετάλλευτους πόρους. Με το live migration των guests από αυτούς τους κόμβους σε κάποιο άλλο φυσικό μηχάνημα, οι "ελεύθεροι" (από φορτίο) πλέον κόμβοι μπορούν είτε να απενεργοποιηθούν είτε να χρησιμοποιηθούν για απαιτητικές εφαρμογές.
- **Συντήρηση του φυσικού μηχανήματος (λογισμικό ή υλικό)**: Τα εξαρτήματα των φυσικών πόρων συχνά χρειάζονται επισκευή, αντικατάσταση ή αναβάθμιση με καινούριο εξοπλισμό. Επίσης, τα λογισμικά ελέγχου πρέπει να αναβαθμίζονται περιστασιακά. Και στις δύο περιπτώσεις συντήρησης, στο φυσικό μηχάνημα δεν πρέπει να εκτελούνται εικονικές μηχανές.
- **Διαχείριση ενέργειας**: Οι Cloud πάροχοι οφείλουν να εφαρμόζουν πολιτικές για αποδοτική διαχείριση της ενέργειας των συστοιχιών τους. Κατ' αυτόν τον τρόπο, κάποιοι hosts μπορούν να απενεργοποιηθούν εφόσον τα φιλοξενούμενα μηχανήματα σε αυτούς ανακατανεμηθούν σε άλλα φυσικά μηχανήματα. Έτσι, οι πάροχοι εξοικονομούν ενέργεια ενώ παράλληλα τα έξοδά τους μπορούν να μειωθούν σε περιόδους χαμηλής χρησιμοποίησης των πόρων.

Σύμφωνα με τον Clark [15], η μεταφορά της μνήμης ενός VM μπορεί να διαχωριστεί σε τρεις βασικές φάσεις:

- **Push φάση**: Η εικονική μηχανή εκτελείται στον source host και συγκεκριμένες σελίδες επαναληπτικά μεταφέρονται στον destination host. Προκειμένου να διασφαλισθεί η συνοχή, αποστέλλονται επανειλημμένα οι σελίδες που έχουν αλλάξει περιεχόμενο (dirty) κατά τη διαδικασία αυτή.
- **Stop-and-Copy φάση**: Η εκτέλεση του VM στον αρχικό host σταματάει. Οι dirty σελίδες μεταφέρονται στον host προορισμού μαζί με την κατάσταση των συσκευών. Έπειτα, το VM συνεχίζει τη λειτουργία του στον host προορισμού.
- **Pull φάση**: Καθώς το VM εκτελείται στον host προορισμού, ίσως αναφερθεί σε μια σελίδα που δεν έχει αντιγραφεί ακόμα από το αρχικό φυσικό μηχάνημα. Προκαλείται, λοιπόν, ένα σφάλμα σελίδας (page fault) ώστε η σελίδα αυτή να προσκομιστεί από τον απομακρυσμένο source host.

2.3.1 Τεχνικές του Live Migration

Ανάλογα με τον τρόπο μεταφοράς της μνήμης του εικονικού μηχανήματος από τον source host στον destination host, οι κύριες προσεγγίσεις της ζωντανής μεταφοράς [16] περιγράφονται στις παρακάτω ενότητες.

Live Migration με pre-copy τεχνική αντιγραφής της μνήμης

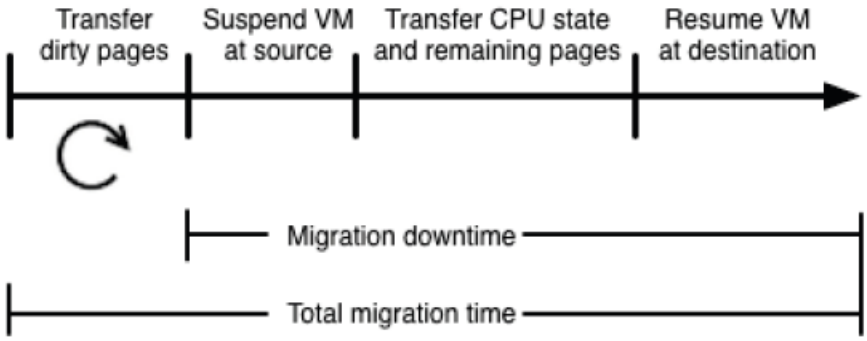
Το live migration με pre-copy τεχνική αντιγραφής της μνήμης (εικόνα 2.6) είναι η προεπιλεγμένη τεχνική στις περισσότερες υπάρχοντες πλατφόρμες εικονικοποίησης, όπως QEMU/KVM [17], XEN [18] and VMware [19]. Με αυτόν τον δημοφιλή αλγόριθμο του live migration, πρώτα στέλνεται η μνήμη του VM στο φυσικό μηχάνημα προορισμού και στη συνέχεια το VM συνεχίζει τη λειτουργία του. Η προσέγγιση αυτή αποτελείται από τα πρώτα δύο στάδια του VM migration. Συγκεκριμένα, στην *Αρχική φάση (Initial phase)* όλη η μνήμη της εικονικής μηχανής θεωρείται dirty ώστε να σταλεί στον host προορισμού. Κατά τη *Επαναληπτική φάση (Iterative phase)*, το λογισμικό ελέγχου χρειάζεται να στείλει ξανά τις σελίδες που άλλαξαν την κατάστασή τους από την προηγούμενη επανάληψη (iteration) του αλγορίθμου. Κατά το migration, το VM στον source host συνεχίζει να εκτελείται και σελίδες που έχουν μεταφερθεί ίσως εγγραφούν ξανά. Όταν ο επόπτης αποφασίζει να μεταφέρει μια σελίδα, τότε την σημειώνει ως ”καθαρή” (”clean”). Όμως, η σελίδα αυτή όμως θεωρείται ”βρώμικη” (”dirty”) όταν η εικονική μηχανή πραγματοποιεί εγγραφή στην αντίστοιχη σελίδα του φυσικού μηχανήματος αφού προηγηθεί, με τη βοήθεια του επόπτη, το page fault και η χαρτογράφηση από την εικονική στη φυσική μνήμη. Μια σελίδα πρέπει να επανασταλεί ακόμα και αν ένα μικρό τμήμα της (από τα 4KB που είναι το συνηθέστερο μέγεθος σελίδας) αλλάξει κατάσταση από τη τελευταία φορά που αντιγράφτηκε στον host προορισμού. Όταν το migration ολοκληρωθεί, το VM στον προορισμό πρέπει να έχει την πιο πρόσφατη έκδοση της μνήμης της εικονικής μηχανής. Ανάλογα με την πλατφόρμα εικονικοποίησης, η *Επαναληπτική φάση* τελειώνει όταν οι σελίδες που πρέπει να μεταφερθούν είναι κάτω από ένα κατώφλι (threshold) ή αν φτάσει σε κάποιο μέγιστο αριθμό επαναλήψεων. Τέλος, κατά την *Stop-and-copy φάση* το source VM διακόπτει τη λειτουργία του και οι υπόλοιπες σελίδες μνήμης που διαφέρουν μεταξύ source-destination, καθώς και η CPU και η κατάσταση των συσκευών μεταφέρονται στον προορισμό ώστε το VM να συνεχίσει την εκτέλεσή του. Εφαρμογές με έντονη δραστηριότητα μνήμης (memory-intensive applications) που συνεχώς αλλάζουν την κατάσταση σελίδων μνήμης σε dirty οδηγούν σε αύξηση του συνολικού χρόνου του migration ή ακόμα χειρότερα σε μη σύγκλιση του αλγορίθμου, αφού οι dirty σελίδες στέλνονται επαναληπτικά κατά την iterative φάση. Στην εργασίας αυτή εστιάζουμε στον pre-copy αλγόριθμο του QEMU/KVM, ο οποίος περιγράφεται στην ενότητα 2.3.4.

Live Migration με post-copy τεχνική αντιγραφής της μνήμης

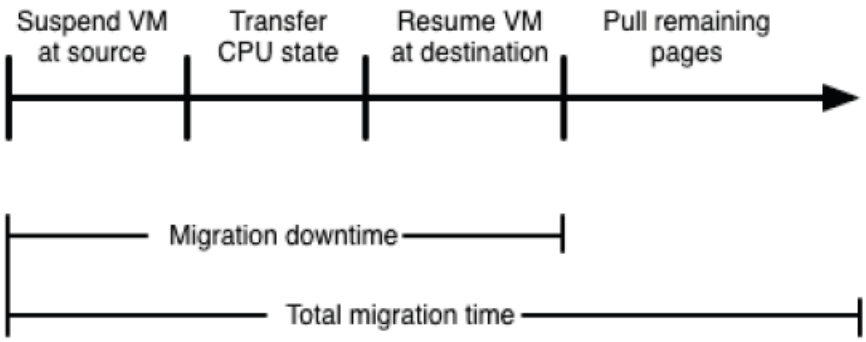
Το live migration με post-copy τεχνική αντιγραφής της μνήμης (εικόνα 2.7) περιλαμβάνει την stop-and-copy και pull φάσεις. Η κατάσταση της CPU και των συσκευών μεταφέρονται στον host προορισμού κατά την εκκίνηση του live migration και το VM ξεκινά την εκτέλεσή του. Οι σελίδες μνήμης μεταφέρονται κατά παραγγελία όταν το VM για πρώτη φορά θέλει να αποκτήσει πρόσβαση σε αυτές. Τα κύρια πλεονεκτήματα του post-copy αλγορίθμου είναι ο μικρός χρόνος μη λειτουργίας (downtime) και η μεταφορά κάθε σελίδας μνήμης μία φορά μόνο σε σύγκριση με τον pre-copy αλγόριθμο. Η απόδοση όμως της εικονική μηχανής και των εφαρμογών επηρεάζονται σημαντικά, αφού κάθε πρώτη πρόσβαση σε μια σελίδα μνήμης δημιουργεί ένα page fault και το VM πρέπει να περιμένει μέχρι η σελίδα που ζητήθηκε να μεταφερθεί μέσω του δικτύου από τον source host. Ένα επιπλέον μειονέκτημα της τεχνικής αυτής, σε σύγκριση με το pre-copy live migration, είναι η αδυναμία ανάκτησης του VM σε περίπτωση σφάλματος του destination κατά το migration.

Live Migration με hybrid τεχνική αντιγραφής της μνήμης

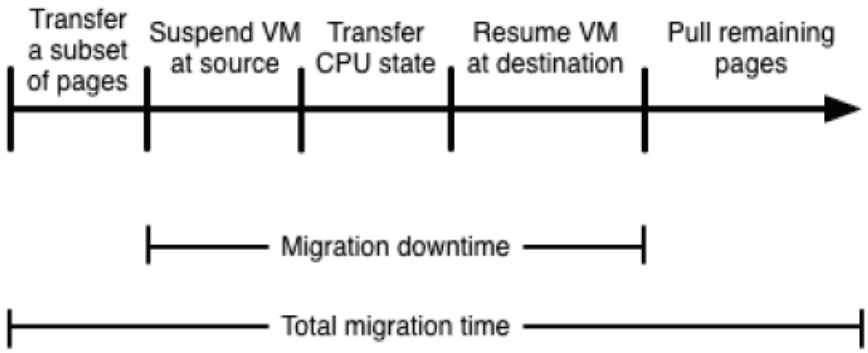
Με σκοπό να αντιμετωπιστούν τα μειονεκτήματα του post-copy live migration με τις σελίδες που προκαλούν page fault, προτάθηκε η υβριδική (hybrid) προσέγγιση (εικόνα 2.8). Η τεχνική αυτή προσπαθεί να μειώσει το συνολικό χρόνο του migration και το συνολικό αριθμό των μεταφερόμενων δεδομένων σε σύγκριση με το pre-copy live migration. Έτσι, το hybrid live migration περιλαμβάνει και τις τρεις φάσεις της διαδικασίας του migration. Κατά την push φάση, οι συχνά προσπελάσιμες σελίδες μνήμης μεταφέρονται στον destination host. Έπειτα, το VM στον source host σταματάει τη λειτουργία του και οι καταστάσεις των συσκευών μεταφέρονται (*stop-and-copy φάση*). Το VM συνεχίζει να εκτελείται στο φυσικό μηχάνημα προορισμού και οι υπόλοιπες σελίδες μνήμης ανακτώνται από το source δυναμικά με τη χρήση page fault (*pull φάση*). Παρόμοια με την post-copy τεχνική, η προσέγγιση αυτή δεν είναι αξιόπιστη.



Σχήμα 2.6: Pre-copy Live Migration



Σχήμα 2.7: Post-copy Live Migration



Σχήμα 2.8: Hybrid Live Migration

2.3.2 Μετρικές απόδοσης του Live Migration

Τα κυριότερα χαρακτηριστικά της διαδικασίας του live migration [20] είναι:

- **Συνολικός χρόνος (Total Time)**
  Ο χρόνος που απαιτείται για την ολοκλήρωση της διαδικασίας του migration, ο οποίος περιλαμβάνει και τις τρεις φάσεις που αναφέρθηκαν. Πρόκειται για το χρονικό διάστημα που πέρασε από την αρχικοποίηση του migration μέχρι τη χρονική στιγμή που η εικονική μηχανή συνεχίζει τη λειτουργία της στον host προορισμού και όλοι οι χρησιμοποιούμενοι πόροι στον αρχικό host αποδεσμεύονται.
- **Χρόνος μη λειτουργίας (Downtime)**
  Αναφέρεται στο χρονικό διάστημα μεταξύ της προσωρινής διακοπής της λειτουργίας του VM στον source host και τη συνέχισή του στον destination host. Σε αυτό το διάστημα οι υπηρεσίες δεν είναι προσβάσιμες από το χρήστη. Το downtime αποτελεί μία από τις πιο σημαντικές παραμέτρους που περιλαμβάνονται στη συμφωνία σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου (SLAs).
- **Μεταφερόμενα Δεδομένα (Data transferred)**
  Είναι τα συνολικά μεταφερόμενα δεδομένα προς τον host προορισμού κατά τη διάρκεια του migration. Αν και το μέγεθος της εικονικής μηχανής και των συσκευών επηρεάζουν αυτή τη μετρική, η επιλογή της τεχνικής του migration καθορίζει κυρίως το συνολικό αριθμό των μεταφερόμενων δεδομένων. Για παράδειγμα, με την pre-copy τεχνική όλες οι σελίδες που αλλάζουν κατάσταση, δηλαδή επαναγράφονται, κατά τη διάρκεια του migration πρέπει να σταλούν ξανά στον destination host.

Αξίζει να αναφέρουμε ότι για το offline migration ο συνολικός χρόνος είναι ίσος με το χρόνο μη λειτουργίας. Ο χρόνος που χρειάζεται για την ολοκλήρωση του migration εξαρτάται από το εύρος ζώνης και την καθυστέρηση του δικτύου. Επίσης, το πλήθος των δεδομένων που πρέπει να μεταφερθεί ισούται με το συνολικό μέγεθος της εικονικής μηχανής.
Οι σημαντικοί παράγοντες που επηρεάζουν την απόδοση του migration είναι:

- **Εύρος ζώνης της ζεύξης (Migration Link Bandwidth)**
  Πρόκειται για την χωρητικότητα δικτύου μεταξύ των source και destination φυσικών μη-

χανημάτων. Το εύρος ζώνης είναι αντιστρόφως ανάλογο τόσο του συνολικού χρόνου όσο και του downtime του migration. Σε περίπτωση που το δίκτυο είναι επιβαρυμένο ή το εύρος ζώνης είναι μικρό, το migration θα διαρκέσει πολύ περισσότερο. Όπως περιγράφεται στην ενότητα 2.3.4, στο QEMU οι διαχειριστές μπορούν να ορίσουν το μέγιστο ρυθμό μετάδοσης για το migration (παράμετρος *max-bandwidth*), οποίος σχετίζεται άμεσα και με το φυσικό εύρος ζώνης της ζεύξης.

◇ **Ρυθμός "βρώμικων" σελίδων (Dirty Page Rate)**
Πρόκειται για το ρυθμό με τον οποίο οι σελίδες μνήμης της εικονικής μηχανής αλλάζουν κατάσταση. Ο αριθμός των σελίδων που μεταφέρονται σε κάθε επανάληψη του pre-copy migration επηρεάζεται από το κομμάτι της μνήμης που έγινε dirty μεταξύ δύο διαδοχικών επαναλήψεων. Σε εφαρμογές με μεγάλο dirty page rate χρειάζεται να μεταφερθούν περισσότερα δεδομένα στον host προορισμού κατά τις Iterative και Stop-And-Copy φάσεις. Επομένως, τα total time και downtime διαρκούν περισσότερο.

Συνδυάζοντας τους παραπάνω παράγοντες, όταν το εύρος ζώνης για το migration είναι σημαντικά χαμηλότερο από το dirty page rate, τότε το live migration μπορεί να αποτύχει, αφού η iterative φάση της pre-copy τεχνικής δεν θα ικανοποιήσει ποτέ τις συνθήκες σύγκλισης του αλγορίθμου.

2.3.3 Προαπαιτούμενα του Migration

Πριν την εκκίνηση του migration, τα εμπλεκόμενα φυσικά μηχανήματα πρέπει να έχουν κοινό setup. Τα προαπαιτούμενα σχετικά με τις ρυθμίσεις και την τοπολογία δικτύου [21] συνοψίζονται παρακάτω:

- **Μοιραζόμενος αποθηκευτικός χώρος (Shared storage) [22]**
- **Συγχρονισμός των ρολογιών των hosts (Host time sync)**
- **Ρυθμίσεις δικτύου (Network configuration)**
- **Κοινοί τύποι CPU των hosts (Host CPU types)**
- **Κοινοί τύποι των φιλοξενούμενων μηχανημάτων (Guest machine types)**

2.3.4 Live Migration με χρήση QEMU/KVM

Το QEMU/KVM λογισμικό ελέγχου επιτρέπει είτε το offline είτε το live migration εικονικών μηχανών μεταξύ των hosts, Με τον όρο live migration εννοούμε την μεταφορά ενός εικονικού συστήματος από μια QEMU διεργασία στον source φυσικό μηχάνημα σε μια άλλη διεργασία στο destination φυσικό μηχάνημα καθώς το εικονικό σύστημα συνεχίζει να εκτελείται στο source φυσικό μηχάνημα.

Παρόμοια με την ανάλυση της pre-copy τεχνικής αντιγραφής της μνήμης που περιγράφηκε στην ενότητα 2.3.1, το Live Migration με χρήση του QEMU/KVM πραγματοποιείται σε 3 στάδια [23]:

- Στάδιο 1: Όλες οι σελίδες της RAM μνήμης αλλάζουν την κατάστασή τους σε dirty *<ram_save_setup()>*

- Στάδιο 2: Οι dirty σελίδες μεταφέρονται μέχρι την τελευταία επανάληψη *<ram_save_iterate()>*
    - το στάδιο αυτό σταμάταει όταν ικανοποιηθεί μια συνθήκη
- Stage 3: Η λειτουργία του εικονικού συστήματος σταματάει, η υπόλοιπη dirty μνήμη και οι καταστάσεις συσκευών μεταφέρονται στον destination host *<migration_thread()>*

Σε κάθε στάδιο αναφέρεται το όνομα της συνάρτησης του πηγαίου κώδικα του QEMU που χειρίζεται το αντίστοιχο βήμα του αλγορίθμου. Η συνάρτηση *migration_thread()* είναι υπεύθυνη για το χειρισμό όλης της διαδικασίας του migration καλώντας τις υπόλοιπες συναρτήσεις.

Σχετικά με τη δομή που χειρίζεται τη μεταφορά της μνήμης και των καταστάσεων των συσκευών, το QEMU χρησιμοποιεί το *QEMUFile* προκειμένου να πραγματοποιήσει το migration [24]. Οι σημαντικότερες συναρτήσεις είναι οι *qemu_put_buffer()*/*qemu_get_buffer()* που επιτρέπουν την εγγραφή/ανάγνωση του buffer στο QEMUFile. Με την ολοκλήρωση της μεταφοράς όλης της dirty RAM μνήμης στον destination host, η μετακίνηση της κατάστασης των συσκευών πραγματοποιείται στο στάδιο 3. Η δομή *VMstate* προστέθηκε πρόσφατα στον κώδικα του QEMU ώστε η αποστολή και η λήψη των συσκευών να είναι ανεξάρτητη από τον τύπο της συσκευής. Σε παλιότερες εκδόσεις του QEMU, κάθε συσκευή χειριζόταν την διαδικασία αυτή μόνη της, με αποτέλεσμα τη μη ύπαρξη ενιαίου κώδικα στο QEMU για όλες τις συσκευές.

Η ταχύτητα μεταφοράς εξαρτάται από το εύρος ζώνης του δικτύου. Ως προεπιλογή, το QEMU θα χρησιμοποιήσει όλο το διαθέσιμο bandwidth. Μέσω της κονσόλας του QEMU ή της QMP εντολής, ο χρήστης με την παράμετρο ***max-bandwidth*** έχει τη δυνατότητα να ορίσει το μέγιστο εύρος ζώνης που θα είναι διαθέσιμο για το migration.

Ο αλγόριθμος του QEMU για το live migration σε κάθε επανάληψη της iterative φάσης υπολογίζει το εκτιμώμενο downtime. Ο υπολογισμός αυτός γίνεται με βάση το πλήθος των δεδομένων που στάλθηκαν στην προηγούμενη επανάληψη, το οποίο είναι γνωστό ως εύρος ζώνης δεδομένων (data bandwidth). Αν στο VM εκτελείται μια απαιτητική σε πόρους εφαρμογή, το πλήθος των dirty σελίδων μνήμης είναι μεγάλο και η διαδικασία για τη μεταφορά τους είναι χρονοβόρα. Είναι σημαντικό να έχουμε υπόψη ότι dirty σελίδες μεταξύ των iterations στον pre-copy αλγόριθμο θα υπάρχουν πάντα σε μια εικονική μηχανή που εκτελείται. Επομένως, το QEMU επαναληπτικά μεταφέρει τις dirty σελίδες μέχρι το πλήθος των dirty σελίδων που δεν έχει μεταφερθεί ακόμα να είναι σχετικά μικρό, ώστε να μπορεί να μεταφερθεί μέσα σε ένα μέγιστο χρονικά επιτρεπτό όριο όπου το VM θα είναι εκτός λειτουργίας. Αυτή η τιμή του downtime μπορεί να προσδιοριστεί από την παράμετρο ***downtime-limit*** στον source host είτε πριν την εκκίνηση του migration είτε δυναμικά κατά τη διάρκεια του live migration. Η προεπιλεγμένη τιμή για την παράμετρο αυτή είναι 300ms.

Όταν το migration δεν τερματίζει, δεν πρέπει να θεωρείται ελαττωματικό. Ο πιο πιθανός λόγος είναι ότι το VM γράφει στη μνήμη με γρηγορότερο ρυθμό σε σχέση με το ρυθμό αποστολής των δεδομένων στον προορισμό μέσω του δικτύου. Με τη βοήθεια της εντολής '*info migrate*' στον source host ο χρήστης μπορεί να πληροφορηθεί με διάφορα στατιστικά της τρέχουσας διαδικασίας μεταφοράς της μνήμης καθώς και για την κατάσταση του live migration. Στην προαναφερθείσα περίπτωση όπου το dirty page rate είναι υψηλό, η κατάσταση του migration είναι 'ενεργή (active)' και το throughput (μεταβλητή '*mbps*') πιθανώς θα είναι αρκετά μεγάλο, ίσο περίπου με την τιμή *max-bandwith* που ορίστηκε πριν την εκκίνηση του migration. Αν το διαθέσιμο εύρος ζώνης του δικτύου είναι στη μέγιστη τιμή του, τότε μια πιθανή λύση στο πρόβλημα της μη σύγκλισης του

pre-copy live migration αλγορίθμου είναι η αύξηση της τιμής του *downtime-limit*. Διαφορετικά, άλλοι μέθοδοι ζωντανής μεταφοράς, όπως η τεχνική post-copy, ή οι διαθέσιμες βελτιστοποιήσεις του pre-copy live migration μπορούν να χρησιμοποιηθούν. Στην παρούσα διπλωματική οι τελευταίες λύσεις που αναφέραμε δεν προτείνονται, αφού ασχοληθήκαμε μόνο με τον κλασσικό pre-copy αλγόριθμο.

Επικοινωνία μεταξύ QEMU και KVM κατά τη διάρκεια του Live Migration

Η επικοινωνία του QEMU με το KVM γίνεται με τη βοήθεια των ioctl() κλήσεων συστήματος στο */dev/kvm* αρχείο συσκευής, το οποίο είναι διαθέσιμο από το KVM kernel module. Το βασικό αρχείο του QEMU για την επικοινωνία αυτή είναι το *kvm-all.c*. Το KVM API αποτελείται από ioctl() κλήσεις συστήματος και μία βασική κλάση αυτών είναι οι VM ioctls που εκτελούν ερωτήματα ή αναθέσεις σχετικά με την εικονική μηχανή, όπως π.χ. για τη διάταξη της μνήμης. Σε επίπεδο πυρήνα (kernel space), το KVM module διατηρεί ένα kernel dirty bitmap (ένα bit ανά σελίδα μνήμης), το οποίο είναι γνωστό ως *kvm_dirty_log*, και ανανεώνεται σε κάθε εγγραφή. Έτσι, κάποιο bit παίρνει την τιμή "1" όταν η αντίστοιχη σελίδα γίνει dirty. Κατά την iterative φάση του live migration, το QEMU παίρνει ένα αντίγραφο του dirty bitmap από τον πυρήνα στο χώρο χρήστη (user space) ώστε να ανιχνεύει τις σελίδες που άλλαξαν την κατάστασή τους σε σχέση με την προηγούμενη φορά που ζητήθηκε ένα αντίστοιχο στιγμιότυπο μνήμης [23]. Το αποτύπωμα της μνήμης αποθηκεύεται σε ένα struct στο χώρο χρήστη που ονομάζεται kvm_dirty_log. Η συνάρτηση *kvm_physical_sync_dirty_bitmap()* του αρχείου *kvm-all.c* είναι υπεύθυνη για τη λειτουργία αυτή χρησιμοποιώντας την ***kvm_vm_ioctl(*KVM_GET_DIRTY_LOG*)*** κλήση συστήματος.

Βελτιστοποιήσεις του Live Migration

Αν και στην παρούσα διπλωματική χρησιμοποιούμε την κλασσική τεχνική του pre-copy live migration, το QEMU/KVM υποστηρίζει μερικές λειτουργίες που το καθιστούν ένα ισχυρό εργαλείο για migration εικονικών μηχανών:

(i) Autoconverge: Η λειτουργία αυτή δυναμικά περιορίζει την απόδοση της εικονικής CPU προκειμένου οι εικονικές μηχανές να μειώσουν την ταχύτητα εγγραφής και ο pre-copy αλγόριθμος να συγκλίνει.

(ii) XBZRLE [25]: Η τεχνική συμπίεσης Xor Binary Zero Run-Length-Encoding (XBZRLE) μειώνει σημαντικά τον όγκο των μεταφερόμενων δεδομένων στο δίκτυο κατά το migration εικονικών μηχανών με έντονη δραστηριότητα μνήμης. Με την τεχνική αυτή μεταφέρονται συμπιεσμένα οι διαφορές μιας σελίδας μεταξύ δύο διαδοχικών επαναλήψεων της Iterarive φάσης.

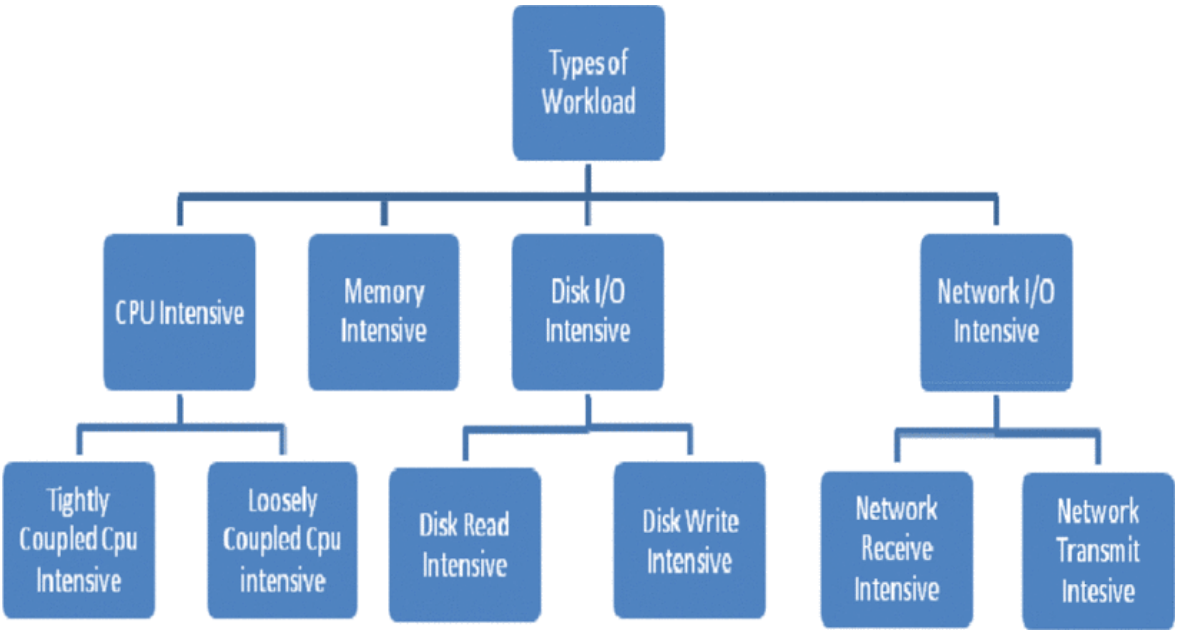
(iii) RDMA: Σε σύγκριση με το migration σε δίκτυα Ethernet τα δεδομένα μεταφέρονται με γρηγορότερο ρυθμό με χρήση της τεχνικής Remote Direct Memory Access (RDMA) σε μοντέρνα δίκτυα υψηλών ταχυτήτων, όπως Infiniband.

(iv) Block Migration: Όταν ο αποθηκευτικός χώρος δεν είναι μοιραζόμενος, η μεταφορά αποθηκευτικού χώρου μαζί με την κατάσταση του guest μπορεί να επιτευχθεί με την τεχνική αυτή. Επειδή στον host προορισμού μεταφέρονται μεγάλα blocks δεδομένων, το δίκτυο και τα υποσυστήματα αποθήκευσης επιβαρύνονται αρκετά.

(v) Multiple thread (de)compression: Με την τεχνική αυτή κάθε σελίδα μνήμης συμπιέζεται πριν σταλεί στον host προορισμού. Κατά τη λήψη της στο destination, τα δεδομένα θα αποσυμπιεστούν. Η τεχνική αυτή προτείνεται όταν το εύρος ζώνης του δικτύου είναι περιορισμένο και υπάρχουν επαρκείς επεξεργαστικοί πόροι στα φυσικά μηχανήματα, αφού η συμπίεση και αποσυμπίεση των σελίδων απαιτεί υπολογιστικούς κύκλους της CPU.

2.4 Κατηγορίες Εφαρμογών

Πολλές έρευνες που σχετίζονται με το Cloud Computing [26, 27, 28] έχουν αναφερθεί στην κατηγοριοποίηση των εφαρμογών. Συχνά, μια εφαρμογή που εκτελείται σε ένα VM αναφέρεται ως workload. Το live migration των εικονικών μηχανών είναι άμεσα συνδεδεμένο με το τον αριθμό των προσβάσεων στη μνήμη που πραγματοποιούνται από την εφαρμογή που "τρέχει" στην εικονική μηχανή. Παρόμοια με την προσέγγιση των Sliwko, L. και Getov, V. [29], διακρίνουμε τις κατηγορίες των εφαρμογών που παρουσιάζονται στην εικόνα 2.9. Η κατηγοριοποίηση γίνεται με κριτήριο τις απαιτήσεις πόρων και εστιάζουμε κυρίως στις εφαρμογές με έντονη δραστηριότητα μνήμης (memory-intensive applications).



Σχήμα 2.9: Κατηγορίες εφαρμογών

2.4.1 Έντονη δραστηριότητα CPU (CPU-Intensive)

Όταν η ταχύτητα του επεξεργαστή καθορίζει το χρόνο ολοκλήρωσης μιας εφαρμογής, τότε μπορεί να χαρακτηριστεί ως εφαρμογή έντονης δραστηριότητας CPU. Σε αυτήν την περίπτωση, ο αριθμός των εντολών που εκτελούνται από τον επεξεργαστή σε ένα συγκεκριμένο χρονικό διάστημα είναι μεγάλος και η χρήση της CPU μπορεί να φτάσει το 100%.

2.4.2 Έντονη δραστηριότητα μνήμης (Memory-intensive)

Οι εφαρμογές αυτής της κατηγορίας πραγματοποιούν εκτεταμένο αριθμό προσβάσεων στη μνήμη και τον περισσότερο χρόνο της εκτέλεσής τους είτε διαβάζουν είτε γράφουν δεδομένα. Η έλλειψη περισσότερης μνήμης είναι ο κύριος λόγος που η απόδοση της εφαρμογής παρουσιάζει bottleneck.

2.4.3 Έντονη δραστηριότητα Εισόδου/Εξόδου (I/O-Intensive)

Οι εφαρμογές που ”σπαταλούν” μεγάλο χρονικό διάστημα περιμένοντας την ολοκλήρωση λειτουργιών I/O ανήκουν στην κατηγορία αυτή. Για παράδειγμα, ένα πρόγραμμα που ψάχνει ένα μεγάλο αρχείο για κάποια δεδομένα, μπορεί να χαρακτηριστεί έντονης δραστηριότητας Εισόδου-/Εξόδου, αφού η CPU καθυστερεί περιμένοντας τη φόρτωση/εκφόρτωση των δεδομένων από την κύρια μνήμη στις συσκευές αποθήκευσης.

Κεφάλαιο 3

Αξιολόγηση του Pre-Copy Live Migration

Τα χαρακτηριστικά του live migration των εικονικών μηχανών έχουν μελετηθεί σε αρκετές έρευνες [29, 30, 31, 32] και στην ενότητα αυτή θα γίνει μια προσπάθεια να επαληθευτούν οι προσεγγίσεις αυτές. Παρουσιάζονται τα αποτελέσματα του live migration με την pre-copy τεχνική αντιγραφή μνήμης και χρήση του QEMU/KVM. Στο μεταφερόμενο VM κάθε φορά εκτελείται διαφορετική εφαρμογή και η μετακίνησή του γίνεται μεταξύ hosts σε Gigabit Ethernet δίκτυο διασύνδεσης. Παρόμοια με τους Li et al. [31], ερευνούμε την επίδραση των παραμέτρων *max-bandwidth* και *downtime-limit* στην απόδοση του live migration του VM ως προς το total migration time και downtime. Βασιζόμενοι στα πειραματικά αποτελέσματα, προσδιορίζουμε το ελάχιστο *downtime-limit* για κάθε εφαρμογή, το οποίο εξασφαλίζει την επιτυχή ολοκλήρωση του migration. Επίσης, μελετώντας τους παράγοντες που επηρεάζουν την απόδοση του live migration, επιχειρούμε να διακρίνουμε τις memory-intensive εφαρμογές από τις υπόλοιπες. Όπως παρουσίασαν οι Ibrahim et al. [33], οι εφαρμογές αυτής της κατηγορίας είναι δύσκολο να μετακινηθούν, επειδή οι τιμές των bandwidth και downtime πρέπει να ρυθμιστούν βέλτιστα σε συνδυασμό με τους όρους των SLAs που πρέπει να ικανοποιούνται. Τέλος, αναλύουμε τη διαδικασία που ακολουθήσαμε και τις προκλήσεις που αντιμετωπίσαμε στην προσπάθειά μας να καταγράψουμε τα χαρακτηριστικά της συμπεριφοράς της μνήμης της εφαρμογής που σχετίζονται με το live migration. Σε κάθε βήμα περιγράφουμε το στόχο μας και παρουσιάζουμε την πληροφορία που συλλέχθηκε από τα εργαλεία, αλλά και τους περιορισμούς της χρήσης τους.

3.1 Τοπολογία

Η τοπολογία που χρησιμοποιείται για τα πειράματα της διπλωματικής αυτής αποτελείται από δύο hosts, όπου ο καθένας αποτελείται από 2 CPUs με 4 πυρήνες, άρα 8 πυρήνες συνολικά (μοντέλο: Intel(R) Xeon(R) CPU E5405 @ 2.00GHz) και 8 GB RAM. Η διαδικασία του live migration πραγματοποιείται μέσω του QEMU (έκδοση 2.10.1) που εκτελείται σε KVM επόπτες, αφού οι κόμβοι υποστηρίζουν επεκτάσεις εικονικοποίησης του επεξεργαστή. Επιπλέον, τα προαπαιτούμενα για το migration που αναφέρονται στην ενότητα 2.3.3 ικανοποιούνται. Τα δύο μηχανήματα συνδέονται μέσω Gigabit Ethernet(GbE) στο ίδιο δίκτυο LAN. Στα tests που εκτελούνται, τα VMs έχουν είτε 1 vCPU και 4 GB RAM είτε 1 vCPU and 4 GB RAM. Σε όλες τις περιπτώσεις, ο εικονικός επεξεργαστής (vCPU), και επομένως το κάθε VM, αντιστοιχίζεται σε ένα συγκεκριμένο πυρήνα του φυσικού μηχανήματος.

3.2 Benchmarks

Με σκοπό να αποκτήσουμε μια εκτίμηση για τη συμπεριφορά των εφαρμογών ως προς τη μνήμη αναπτύσσουμε μερικά απλά προγράμματα και χρησιμοποιούμε ένα σύνολο από μονονηματικά benchmarks της Spec2006 σουίτας και την πολυνηματική εφαρμογή SPECjbb2005. Τα workloads που χρησιμοποιούνται για την αξιολόγηση των χαρακτηριστικών του migration περιγράφονται παρακάτω:

3.2.1 Microbenchmarks

Με σκοπό την μελέτη του live migration αναπτύσσουμε ορισμένες απλές εφαρμογές στις οποίες γνωρίζουμε εκ των προτέρων τη συμπεριφορά τους ως προς τη μνήμη. Η πρώτη εφαρμογή λέγεται *write_memory_loop* και αποτελεί ένα microbenchmark που σειριακά και κυκλικά γράφει (αλλάζει) συνεχώς το πρώτο byte κάθε σελίδας της μνήμης. Το μέγεθος κάθε σελίδας είναι 4KB. Η μνήμη που δεσμεύεται κατά την εκτέλεση του προγράμματος δίνεται ως input παράμετρος και ισούται με το Writable Working Set (WWS). Το microbenchmark αυτό που δεσμεύει και συνεχώς κάνει dirty x MB της μνήμης αναφέρεται ως ***write_memory_loop_xMB***.

Επίσης, αναπτύσσουμε ένα I/O-intensive πρόγραμμα, το οποίο ονομάζεται ***write_file*** και συνεχώς γράφει σε ένα αρχείο του δίσκου. Προκειμένου να διασφαλιστεί ότι κάθε εγγραφή αποθηκεύεται στο filesystem και όχι σε κάποια ενδιάμεση cache ή μνήμη, χρησιμοποιείται η συνάρτηση sync() μετά από κάθε write.

3.2.2 SPEC2006

Η σουίτα SPEC2006 [34] είναι ένα σύνολο από μονονηματικές εφαρμογές που χρησιμοποιούνται ευρέως για την αξιολόγηση της απόδοσης υπολογιστικών συστημάτων. Πρόκειται για πραγματικές εφαρμογές που είναι γραμμένες σε C, C++ ή FORTRAN με μικρές αλλαγές στον κώδικα για να περιοριστούν οι input/output λειτουργίες και να χρησιμοποιηθούν κυρίως για την αξιολόγηση του επεξεργαστή, της μνήμης ή του compiler. Στην εργασία αυτή, χρησιμοποιήθηκαν τα παρακάτω benchmarks που έχουν διαφορετικά dataset και χαρακτηριστικά ώστε να καλύψουμε όσο το δυνατόν μεγαλύτερο εύρος εφαρμογών:

436.cactusADM, ***450.soplex***, ***459.GemsFDTD***, ***471.omnetpp***, ***473.astar***

3.2.3 SPECjbb2005

Το SPECjbb2005 (Java Server Benchmark) [35] είναι ένα benchmark για την αξιολόγηση της απόδοσης ενός Java server. Αυτό επιτυγχάνεται με την προσομοίωση ενός τριών επιπέδων client/server συστήματος (με έμφαση στο μεσαίο επίπεδο). Το SPECjbb2005 εκτελείται σε ένα JVM του οποίου τα threads αναπαριστούν τερματικούς σταθμούς που αποκαλούνται “warehouses”. Κάθε thread παράγει το δικό του τυχαίο input και πραγματοποιεί συναλλαγές με ένα warehouse. Στην εργασία αυτή, το benchmark αυτό χρησιμοποιείται είτε με 8 threads (και αναφέρεται ως ***SPECjbb*** ή ***SPECjbb*_8thread**) είτε με 1 thread (και αναφέρεται ως ***SPECjbb*_1thread**). Οι δύο αυτές περιπτώσεις χειρίζονται ως δύο διαφορετικές εφαρμογές στα πειράματά μας.

3.2.4 Idle

Τα ***Idle*** VMs είναι εικονικές μηχανές που είναι σε running κατάσταση χωρίς να εκτελείται σε αυτές καμία εφαρμογή. Οι εικονικές αυτές μηχανές καταναλώνουν CPU, μνήμη και αποθηκευτικό χώρο από τους πόρους του host. Το live migration των idle VMs θεωρείται ως το σημείο αναφοράς, αφού αποτελεί το απλούστερο VM για την μετακίνηση από έναν κόμβο σε κάποιον άλλον.

3.3 Αξιολόγηση των χαρακτηριστικών του QEMU/KVM Live Migration

Στην ενότητα αυτή, παρουσιάζονται τα αποτελέσματα που συλλέχθηκαν από τα migration πολλαπλών guests με σκοπό την κατανόηση του pre-copy αλγορίθμου σε συνδυασμό με τις παραμέτρους του QEMU που καθορίζουν τη ολοκλήρωση του migration. Αρχικά, αξιολογούμε την επίδραση του μεγέθους της μνήμης που δεσμεύεται από το VM (*VM_Size*). Έπειτα, ρυθμίζουμε τις παραμέτρους *downtime-limit* και *max-bandwidth* με διάφορες τιμές ώστε να κατανοήσουμε την βαρύτητά τους όταν σε κάθε migration στο VM εκτελείται διαφορετική εφαρμογή. Επίσης, διερευνάμε τη σημασία της χρονικής στιγμής ενεργοποίησης του migration ως συνέπεια της δυναμικής κατανάλωσης της μνήμης κατά τη διάρκεια εκτέλεσης της εφαρμογής στο VM. Τέλος, για κάθε workload προσπαθούμε να υπολογίσουμε το ελάχιστο *downtime-limit* με το οποίο το migration της εικονικής μηχανής μέσω QEMU/KVM ολοκληρώνεται επιτυχώς.

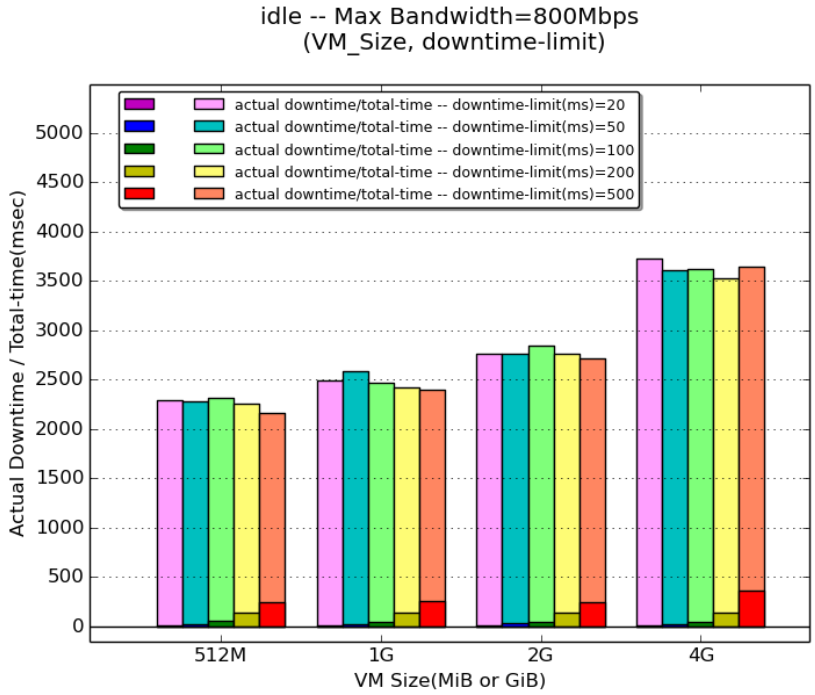
Όταν η τιμή του *max-bandwidth* φτάνει το φυσικό bandwidth (1 Gbps), το throughput παρουσιάζει διακυμάνσεις σε μερικές περιπτώσεις. Αυτό οφείλεται στη κοινή χρήση του δικτύου και από άλλους κόμβους που δεν συμμετέχουν στο migration. Επομένως, προτείνεται να μην καταναλώνεται όλο το bandwidth κατά τη διάρκεια του migration, για αυτό το λόγο στα πειράματά μας η τιμή που θέτουμε στο *max-bandwidth* είναι 800Mbps, δηλαδή το 80% του φυσικού εύρους ζώνης του δικτύου.

Όπως αναφέρεται και στο προηγούμενο κεφάλαιο (2.3.1), ο pre-copy αλγόριθμος ίσως να μην συγκλίνει ποτέ όταν το φιλοξενούμενο μηχάνημα γράφει στη μνήμη με ρυθμό μεγαλύτερο από αυτόν που μεταφέρονται οι dirty pages από τον source host στον destination host κατά τη διάρκεια της *επαναληπτικής φάσης*. Προκειμένου να αποφευχθούν τέτοιες καταστάσεις, οι οποίες δεν φτάνουν ποτέ στην *stop-and-copy* φάση, ακυρώνουμε το migration όταν δεν έχει συγκλίνει σε διάστημα 40 δευτερολέπτων. Η τιμή αυτή επιλέγεται αυθαίρετα και καθορίζει το μέγιστο επιθυμητό total time του migration. Στις γραφικές παραστάσεις που παραθέτονται στο κεφάλαιο αυτό, η κατάσταση της μη σύγκλισης, και συνεπώς αποτυχίας του migration, παρουσιάζεται είτε με μηδενική τιμή είτε με αρνητική μπάρα.

Με σκοπό να διασφαλίσουμε την ακρίβεια των αποτελεσμάτων μας, τα πειράματα εκτελέστηκαν τρεις φορές με το ίδιο configuration και παρουσιάζεται η μέση τιμή των τιμών. Σε όλα τα πειράματα, η εφαρμογή εκτελείται για ένα καθορισμένο χρονικό διάστημα (*running_time*) στην εικονική μηχανή πριν δοθεί η εντολή για το migration. Εκτός από τα πειράματα για την αξιολόγηση του *VM_Size*, το μέγεθος του VM που χρησιμοποιείται είναι ίσο με 1G.

3.3.1 Δέσμευση πόρων μνήμης (*VM_Size*)

Αρχικά, μελετάμε την επίδραση του μεγέθους της μνήμης των φυσικών πόρων που δεσμεύεται για την εικονική μηχανή. Το χαρακτηριστικό αυτό αναφέρεται ως *VM_Size* και πραγματοποιούνται πειράματα με τη χρήση ενός *idle* VM του οποίου το μέγεθος της μνήμης αυξάνεται από 512M[1] σε 4G[2]. Όπως παρουσιάζεται στο διάγραμμα 3.1 , το total migration time και το downtime αυξάνονται καθώς το μέγεθος της μνήμης του VM αυξάνεται. Η αύξηση αυτή είναι αναμενόμενη, αφού η ύπαρξη μεγαλύτερης μνήμης συνεπάγεται μεγαλύτερη ποσότητα δεδομένων που χρειάζεται να μεταφερθούν από τον αρχικό κόμβο στον κόμβο προορισμού. Το *idle* VM αποτελεί το σημείο αναφοράς των πειραμάτων μας, επομένως, κάθε benchmark που πραγματοποιεί εγγραφές στη μνήμη αναμένεται να έχει παρόμοια συμπεριφορά καθώς η μνήμη του VM στο οποίο εκτελείται αυξάνεται. Το total time και downtime του migration αναμένεται, βέβαια, να είναι μεγαλύτερο σε σύγκριση με το idle VM.



Σχήμα 3.1: Επίδραση του *VM_Size* στο Live Migration

3.3.2 Αξιολόγηση της παραμέτρου *downtime-limit*

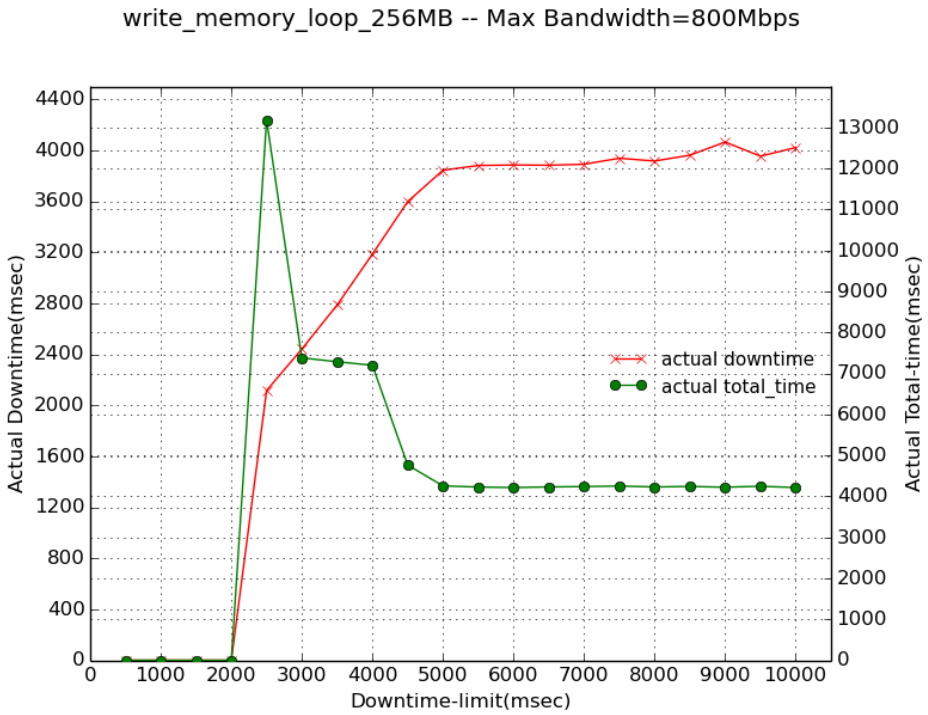
Όπως ήδη αναφέρθηκε στο κεφάλαιο 2.3.4, η παράμετρος *downtime-limit* καθορίζει τη μέγιστη επιτρεπτή διάρκεια του χρόνου μη λειτουργίας κατά τη μετάβαση του VM από ένα κόμβο σε έναν άλλο. Ο pre-copy αλγόριθμος του QEMU καταγράφει τον εφικτό ρυθμό μεταφοράς των δεδομένων στο δίκτυο και τον συγκρίνει με το μέγεθος της μνήμης RAM που απομένει να μεταφερθεί. Ο έλεγχος αυτός γίνεται σε κάθε επανάληψη της *iterative φάσης* με σκοπό να αποφασίσει αν η μνήμη αυτή μπορεί να μεταφερθεί σε χρονικό παράθυρο το πολύ ίσο με το μέγιστο επιτρεπτό downtime. Επομένως, υποθέτοντας ότι ένα VM workload μεταφέρεται με ένα σταθερό network bandwidth,

[1] M ή MiB για mebibytes (2^{20} ή blocks των 1,048,576 bytes)
[2] G ή GiB για gibibytes (2^{30} ή blocks των 1,073,741,824 bytes)

οι χαμηλές τιμές του *downtime-limit* οδηγούν στη μη σύγκλιση του pre-copy αλγορίθμου, ενώ οι υψηλές τιμές του οδηγούν το downtime να έχει τη μέγιστη διάρκεια.

Στη γραφική παράσταση 3.2 απεικονίζονται τα αποτελέσματα του live migration ενός VM στο οποίο εκτελείται το *write_memory_loop_256MB* microbenchmark. Όταν το *downtime-limit* λαμβάνει τιμές μικρότερες από 2500 msec και ο μέγιστος ρυθμός μεταφοράς δικτύου (*max-bandwidth*) είναι 800Mbps, τότε το migration δεν ολοκληρώνεται. Τιμές στο διάστημα 2000 και 2500 msec ίσως οδηγούν σε επιτυχή ολοκλήρωση του migration, αλλά αυτό δεν διασφαλίζεται αφού το εύρος ζώνης μεταφοράς δεδομένων μπορεί να παρουσιάζει διακυμάνσεις γύρω από το όριο του *max-bandwidth*.

Μια σημαντική παρατήρηση όσον αναφορά τη λειτουργία του QEMU/KVM pre-copy live migration είναι ότι στοχεύει στη βελτιστοποίηση του total time του migration τηρώντας το όριο του *downtime-limit*. Έτσι, στα πειράματά μας παρατηρείται ότι αυξάνοντας το μέγιστο επιθυμητό downtime από 2500ms μέχρι 5000ms, τότε το total time του migration μειώνεται ενώ το χρονικό διάστημα του downtime αυξάνεται, παραμένοντας όμως κάτω από το όριο του *downtime-limit*. Περαιτέρω αύξηση της *downtime-limit* τιμής δεν οδηγεί σε βελτίωση του συνολικού χρόνου της μεταφοράς του VM, αφού το downtime παραμένει περίπου 4000msec και είναι περίπου ίσο με το total time. Στην περίπτωση αυτή, η υψηλή τιμή του επιτρεπτού χρόνου μη λειτουργίας οδηγεί στην άμεση μεταφορά της εικονικής μηχανής.

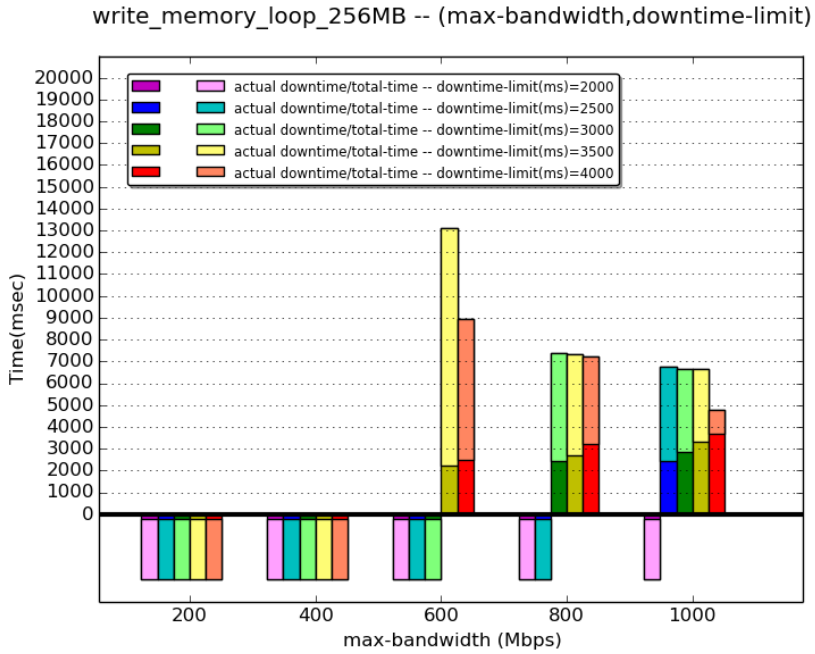


Σχήμα 3.2: Επίδραση της παραμέτρου *downtime-limit* στο Live Migration

3.3.3 Αξιολόγηση της παραμέτρου *max-bandwidth*

Στο πείραμα αυτό αξιολογήθηκε η επίδραση του διαθέσιμου bandwidth στη μεταφορά των απαιτούμενων δεδομένων για το migration. Αρχικά θέτουμε στην παράμετρο *max-bandwidth* τιμή ίση με 200 Mbps και την αυξάνουμε με βήμα 200 Mbps μέχρι το όριο των 1000 Mbps.

Τα αποτελέσματα των μετρήσεων παρουσιάζονται στο διάγραμμα 3.3, όπου το VM που χρησιμοποιήθηκε για το migration εκτελεί το *write_memory_loop_256MB* microbenchmark. Για κάθε τιμή του *max-bandwidth*, δίνουμε και διαφορετικές τιμές στην παράμετρο *downtime-limit* και, όπως φαίνεται στο διάγραμμα, το ζευγάρι (*max-bandwidth*, *downtime-limit*) καθορίζει σε μεγάλο βαθμό την σύγκλιση ή μη του migration. Στην περίπτωση όπου το *max-bandwidth* παίρνει μικρές τιμές, όπως 200 ή 400 Mbps, το migration αποτυγχάνει ακόμα και με μεγάλες τιμές του μέγιστου επιτρεπτού downtime. Σε αυτές τις περιπτώσεις, ο ρυθμός των παραγόμενων dirty pages είναι μεγαλύτερος από τον επιθυμητό ρυθμό μεταφοράς των δεδομένων στο δίκτυο. Με την αύξηση του διαθέσιμου bandwidth, παρατηρείται ότι για μερικές τιμές του *downtime-limit* το dirty page rate μπορεί να καλυφθεί από την ταχύτητα μεταφοράς στο δίκτυο. Συνεπώς, όταν ο Cloud πάροχος επιθυμεί να αυξήσει τις πιθανότητες ενός επιτυχούς pre-copy live migration, προτείνεται, ως πρώτη προσέγγιση, να αυξήσει το διαθέσιμο bandwidth για τη διαδικασία του migration. Στη συνέχεια, όπως παρουσιάζεται και στο διάγραμμα 3.2, τηρώντας το SLA με τον πελάτη, μπορεί να ρυθμίσει την τιμή του *downtime-limit* ώστε να πετύχει το βέλτιστο total migration time.



Σχήμα 3.3: Επίδραση της παραμέτρου *max-bandwidth* στο Live Migration

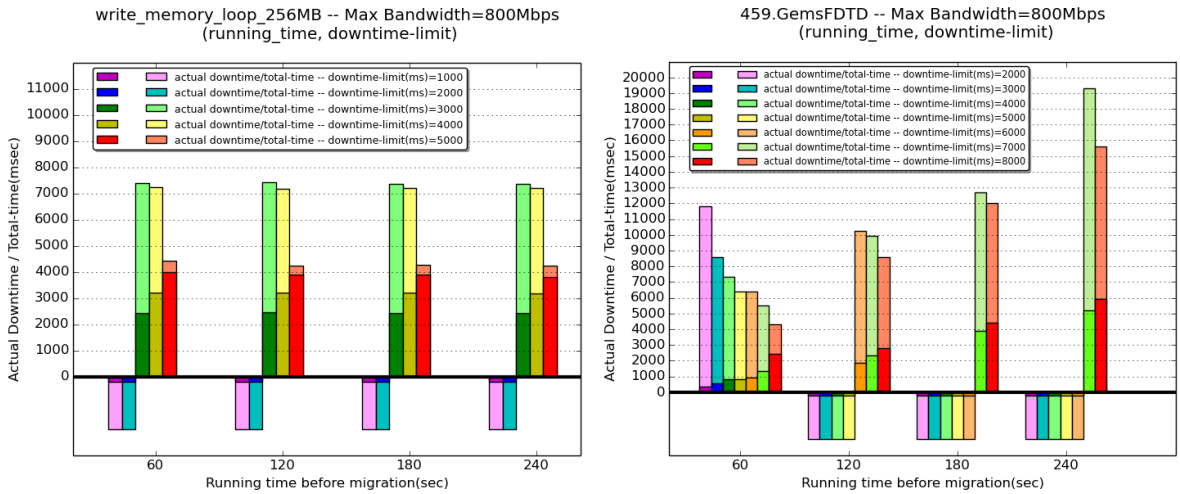
3.3.4 Αξιολόγηση της χρονικής στιγμής εκκίνησης του migration

Η χρήση των πόρων από μια εφαρμογή μπορεί να ποικίλλει κατά το runtime της ανάλογα με την φάση της εκτέλεσης ή τον φόρτο (για παράδειγμα στην περίπτωση μιας web υπηρεσίας). Ως αποτέλεσμα, εκτελούμε κάποια πειράματα για την αξιολόγηση της επίδρασης της χρονικής στιγμή όπου το migration ενεργοποιείται αφού στις περισσότερες πραγματικές εφαρμογές ο αριθμός των προσβάσεων στη μνήμη αλλάζει δυναμικά κατά την διάρκεια εκτέλεσης. Σε αυτά τα tests ξεκινάμε το migration μετά από 60, 120, 180 και 240 δευτερόλεπτα runtime της εφαρμογής. Αν το migration δεν ολοκληρωθεί μετά από 40 δευτερόλεπτα, τότε ακυρώνεται από τον host και θεωρείται ανεπιτυχές.

Στο διάγραμμα 3.4a παρατηρείται ότι η χρονική στιγμή εκκίνησης του migration για το *write_memory_loop*

microbenchmark δεν επηρεάζει το total time και το downtime για όλα τις *downtime-limit* που εξετάσαμε. Τα αποτελέσματα είναι αναμενόμενα μιας και το πρόγραμμα αυτό έχει σταθερό dirty page rate και, επομένως, το ίδιο πλήθος δεδομένων πρέπει να μεταφερθεί στον destination host σε κάθε επανάληψη. Αντίθετα, σύμφωνα με το διάγραμμα 3.4b όπου το *459.GemsFDTD* αποτελεί παράδειγμα real-world εφαρμογής, τα αποτελέσματα για διαφορετικές χρονικές στιγμές εκκίνησης του migration διαφέρουν μεταξύ τους. Για παράδειγμα, όταν η εντολή για migration δοθεί μετά από 240 sec εκτέλεσης της εφαρμογής, 5 από τις 7 μεταφορές είναι ανεπιτυχείς σε σύγκριση με το migration μετά από 60 sec. Επίσης, στην περίπτωση των ολοκληρωμένων migration όπου το *downtime-limit* είναι 7000msec και 8000msec, ο συνολικός χρόνος της διαδικασίας είναι 3,5 φορές μεγαλύτερος, ενώ το downtime είναι 2,5-3,5 φορές μεγαλύτερο. Μπορούμε, λοιπόν, να συμπεράνουμε ότι υπάρχουν χρονικές περίοδοι που η εφαρμογή γράφει σε διαφορετικές σελίδες της μνήμης με ταχύτερο ρυθμό, ενώ άλλες στιγμές μπορεί να επεξεργάζεται δεδομένα που βρίσκονται στην ίδια σελίδα ή να ελαττώνεται ο ρυθμός των εγγραφών. Στις τελευταίες περιπτώσεις όπου μειώνεται το πλήθος των dirty pages που χρειάζεται να μεταφερθούν στον host προορισμού μπορεί να εξασφαλιστεί η ικανοποίηση της συνθήκης σύγκλισης του αλγορίθμου του pre-copy live migration.

Επειδή τα αποτελέσματα του *running_time*=60 και *running_time*=120 διαφέρουν σημαντικά σε ορισμένες περιπτώσεις, στα πειράματά μας θεωρούνται διαφορετικά workloads που παράγουν τα δικά τους migration αποτελέσματα.



(a) Microbenchmark **(b)** Real-world application

Σχήμα 3.4: Επίδραση της χρονικής στιγμής εκκίνησης του migration

3.3.5 Ελάχιστη τιμή του *downtime-limit* για επιτυχές migration

Ως τελευταίο στάδιο της αξιολόγησης της διαδικασίας του migration, επιχειρούμε να υπολογίσουμε την ελάχιστη τιμή για την *downtime-limit* παράμετρο που διασφαλίζει την επιτυχή ολοκλήρωση του live migration για τις εφαρμογές που χρησιμοποιούμε. Συνεπώς, κάθε workload εκτελείται για *running_time*=60 και *running_time*=120 δευτερόλεπτα πριν ενεργοποιήσουμε το migration προκειμένου να βρούμε την βέλτιστη τιμή του *downtime-limit*. Τα πειράματα εκτελούνται 5 φορές και καταγράφουμε το μέσο όρο των χρόνων του total time και downtime από τα ολοκληρωμένα migration.

Όπως φαίνεται και στα διαγράμματα 3.5, για κάθε workload υπάρχουν 3 διαφορετικές κατηγορίες αποτελεσμάτων:

- Ανεπιτυχές migration: Για τις τιμές αυτές του *downtime-limit* που φαίνονται στο άξονα y, το migration πάντα ακυρώνεται από τον host μετά από 40 δευτερόλεπτα λόγω μη σύγκλισης[3].
- Μερικώς επιτυχές migration: Για τις τιμές αυτές του *downtime-limit* που φαίνονται στο άξονα y, το migration δεν ολοκληρώνεται εντός του χρονικού παραθύρου των 40 δευτερολέπτων σε τουλάχιστον 1 από τις 5 προσπάθειες.
- Επιτυχές migration: Για τις τιμές αυτές του *downtime-limit* που φαίνονται στο άξονα y, το migration ολοκληρώνεται εντός του χρονικού παραθύρου των 40 δευτερολέπτων σε όλες(5/5) τις προσπάθειες. Στα σημεία που ανήκουν σε αυτή την κατηγορία, σημειώνεται ως X / Y οι τιμές των total-time / downtime.

Στον παρακάτω πίνακα 3.1 παρουσιάζουμε την ελάχιστη τιμή του *downtime-limit* που διασφαλίζει την ολοκλήρωση του migration για όλα τα benchmarks. Παρατηρείται ότι για τα περισσότερα benchmarks της SPEC σουίτας, η ελάχιστη αποδεκτή τιμή του downtime για επιτυχές migration διαφέρει μεταξύ των runtimes που ελέγχθηκαν.

Benchmark	***downtime-limit*(msec)**	
	rt60	**rt120**
idle	20	20
write_file	20	20
write_memory_loop_16MB	200	200
write_memory_loop_32MB	400	400
write_memory_loop_64MB	700	700
write_memory_loop_128MB	1400	1400
write_memory_loop_256MB	2600	2600
write_memory_loop_512MB	5200	5200
SPECjbb (1 thread)	600	600
SPECjbb (8 threads)	2400	2400
436.cactusADM	20	20
450.soplex	80	150
459.GemsFDTD	2800	6000
471.omnetpp	900	900
473.astar	1800	200

Πίνακας 3.1: Ελάχιστη τιμή του *downtime-limit* για επιτυχές Live Migration κάθε εφαρμογής (*max-bandwidth*=800Mbps)

Βασιζόμενοι στα αποτελέσματα που παρουσιάζονται στο κεφάλαιο αυτό καθώς και στα χαρακτηριστικά των κατηγοριών των workloads που αναφέρονται στην ενότητα 2.4, ορισμένες εφαρμογές που χρησιμοποιήσαμε μπορούν να χαρακτηριστούν ως εφαρμογές με έντονη δραστηριότητα μνήμης (memory-intensive). Λόγω των εκτεταμένων προσβάσεων στη μνήμη μερικά workloads χρειάζονται σχετικά μεγάλη τιμή για το *downtime-limit* ώστε το migration να ολοκληρωθεί με επιτυχία. Επίσης, συγκριτικά με την default τιμή του QEMU/KVM που είναι 300ms, benchmarks όπως

[3] Η *Επαναληπτική φάση* συγκλίνει όταν το τμήμα της μνήμης που δεν έχει μεταφερθεί ακόμα από τον source στον destination host είναι μικρό αρκετά ώστε να μπορεί να μεταφερθεί σε μια επανάληψη με downtime το πολύ ίσο με το χρονικό όριο μη λειτουργίας που θέτει ο Cloud πάροχος

τα *write_memory_loop* (κυρίως για memory size μεγαλύτερα από 128MB), *SPECjbb* και *459.GemsFDTD* μπορούν να χαρακτηριστούν ως **memory-intensive**.

3.4 Χαρακτηριστικά του Live Migration σχετικά με τη μνήμη

Μετά την εκτέλεση ενός μεγάλου πλήθους live migrations με διαφορετικές εφαρμογές, έχουμε πλέον αποκτήσει μια εκτίμηση για τα χαρακτηριστικά που επηρεάζουν τη διαδικασία της ζωντανής μεταφοράς ενός VM και κυρίως αυτά που σχετίζονται με την συμπεριφορά του workload ως προς τη μνήμη. Τα κυριότερα χαρακτηριστικά είναι τα εξής:

(i) Το μέγεθος της μνήμης του VM (**VM_Size**) επηρεάζει το migration, αφού όλο το VM μεταφέρεται από τον ένα κόμβο σε έναν άλλο.

(ii) Το **Dirty Page Rate**, το οποίο ορίζεται ως το μέγεθος της μνήμης που αλλάζει η κατάστασή του από clean σε dirty ανά δευτερόλεπτο. Στον pre-copy αλγόριθμο ο source host επαναληπτικά στέλνει στον destination host τις σελίδες του VM που άλλαξαν κατάσταση. Επομένως, το χαρακτηριστικό αυτό καθορίζει σε μεγάλο βαθμό τη σύγκλιση ή μη του αλγορίθμου κατά την *Επαναληπτική φάση* του live migration.

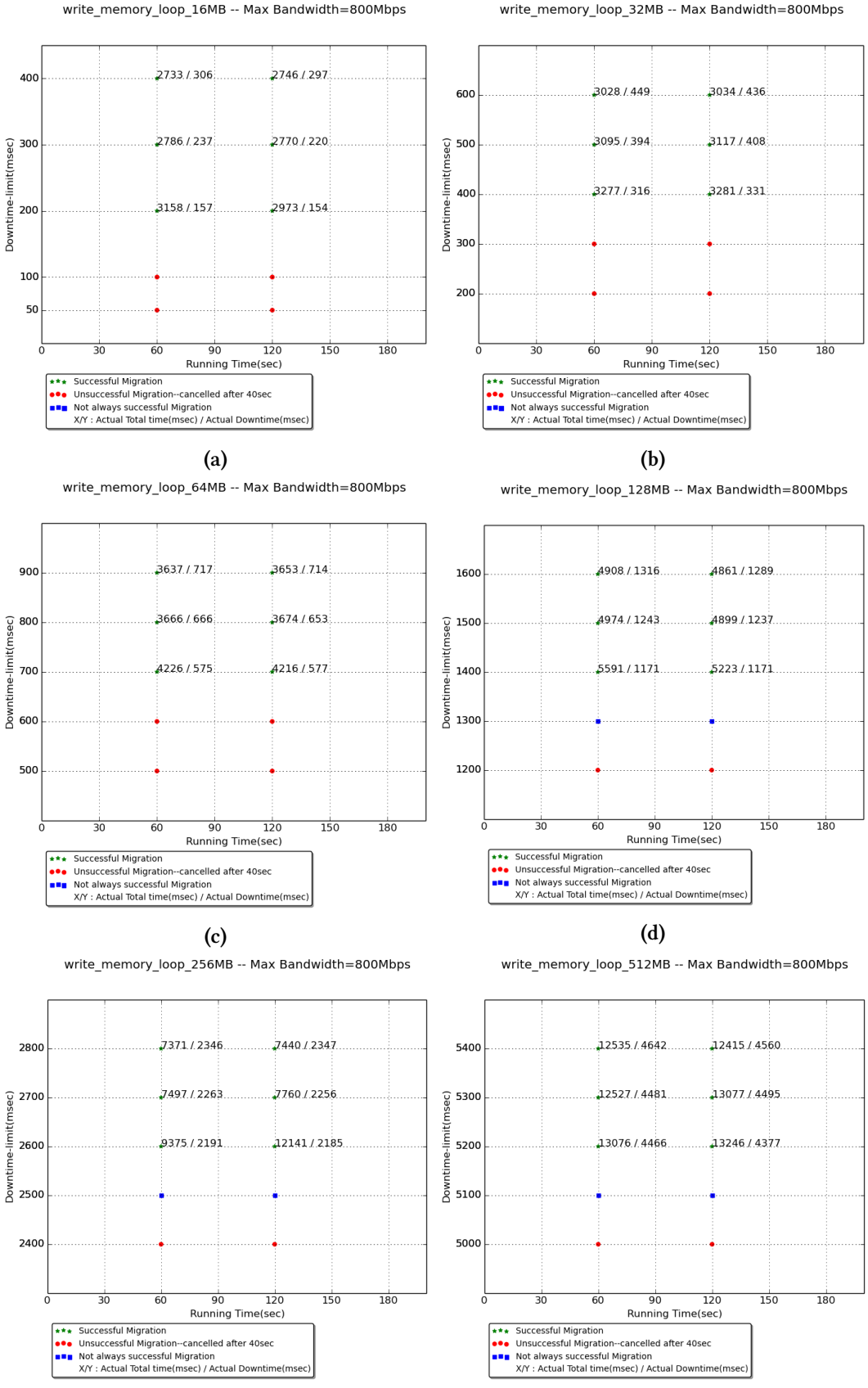
(iii) Ο συνολικός αριθμός των σελίδων που έγιναν εγγραφές, δηλαδή έγιναν dirty, κατά το migration ορίζεται ως Writable Working Size (**WWS**). Το χαρακτηριστικό αυτό επηρεάζει το live migration, αφού έστω και μια μόνο εγγραφή σε ένα τμήμα μιας σελίδας αρκεί ώστε η σελίδα αυτή να σταλεί ξανά στον κόμβο προορισμού.

3.5 Εργαλεία παρακολούθησης της μνήμης

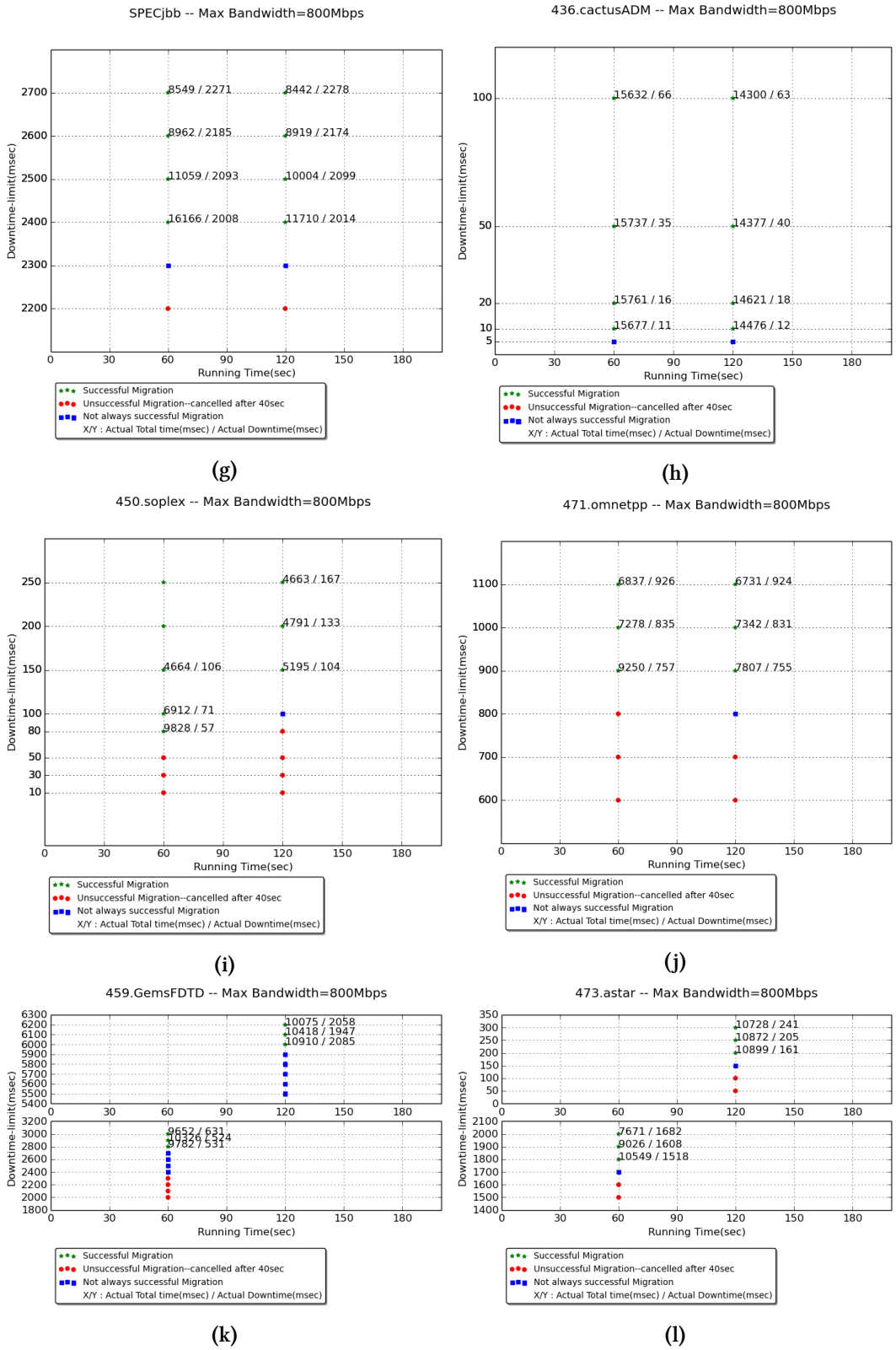
Προσπαθώντας να εντοπίσουμε τα χαρακτηριστικά του live migration που σχετίζονται με τη μνήμη, και κυρίως το Dirty Page Rate, το WWS και το *VM_Size*, ως πρώτη προσπάθεια χρησιμοποιήσαμε κάποια εργαλεία που είναι ενσωματωμένα στο Linux, όπως το *perf* και *iowait*. Με αυτό τον τρόπο, καταφέρνουμε επιτυχώς να διακρίνουμε τις εφαρμογές με έντονη δραστηριότητα σε I/O λειτουργίες. Όμως, με τα δεδομένα από αυτά τα εργαλεία δεν καταφέρνουμε να ταξινομήσουμε τις εφαρμογές ως προς τις απαιτήσεις τους σε μνήμη.

3.5.1 Cache misses και MPKI με χρήση των Performance Counters

Η πρώτη προσέγγιση είναι η συλλογή και ανάλυση της δεδομένων που καταγράφουμε με τη χρήση των Linux Performance Counters. Το *Perf* [36, 37, 38] είναι ένα εργαλείο καταγραφής δεδομένων που συλλέγονται με τη βοήθεια των performance counters. Τα δεδομένα αυτά είναι είτε **software events** όπως context switches και page faults, ή **hardware events** που προέρχονται από τον επεξεργαστή, όπως cache misses (από όλα τα επίπεδα), ή κύκλοι ρολογιού (clock cycles). Το πλήθος και το είδος των hardware events καταγράφονται με τη βοήθεια των Performance Monitoring Units (PMUs). Οι performance counters υποστηρίζουν δύο είδη καταγραφής δεδομένων, καταμέτρηση (counting) και δειγματοληψία (sampling). Κατά το counting mode, οι μετρητές αρχικοποιούνται και καταγράφουν τον αριθμό των γεγονότων που συνέβησαν σε ένα χρονικό διάστημα (εντολή "*perf stat*"). Κατά το sampling mode, οι μετρητές παρουσιάζονται ως άθροισμα δειγμάτων



Σχήμα 3.5: Ελάχιστη τιμή του *downtime-limit* για επιτυχές Live Migration κάθε εφαρμογής



Σχήμα 3.5: Ελάχιστη τιμή του *downtime-limit* για επιτυχές Live Migration κάθε εφαρμογής (συνέχεια)

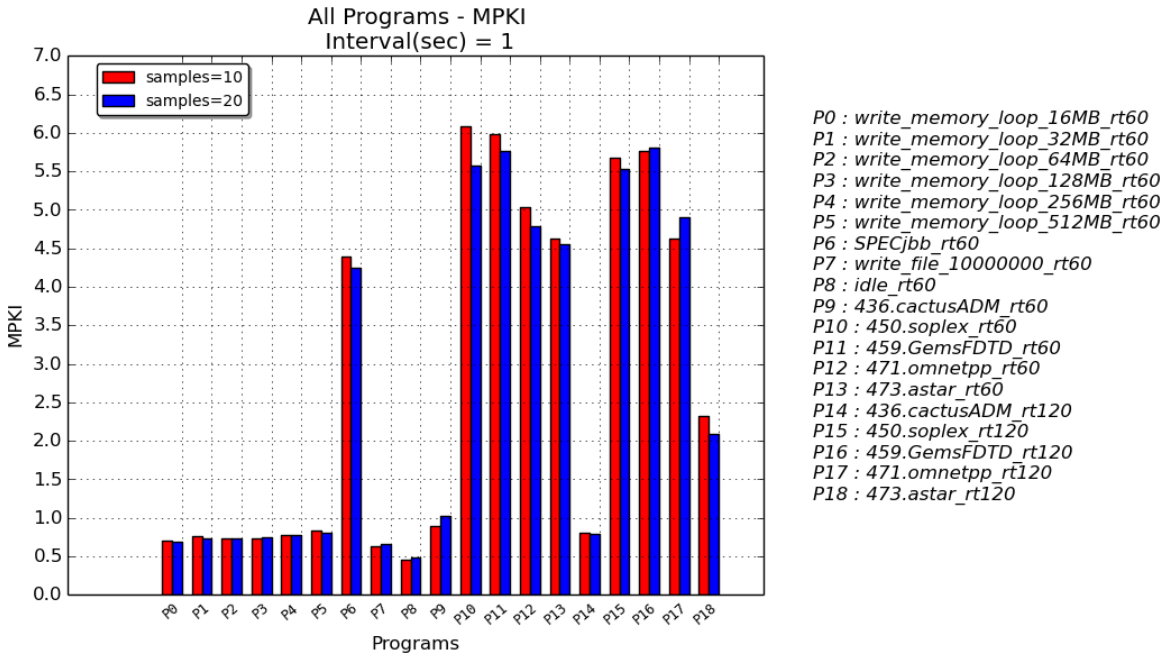
όπου κάθε δείγμα καταγράφεται ανά συγκεκριμένο πλήθος γεγονότων (εντολή "*perf report*"). Το βασικό μειονέκτημα της χρήσης των performance counters είναι το γεγονός ότι το είδος των διαθέσιμων μετρητών μπορεί να διαφέρει ανάμεσα σε διαφορετικούς τύπους επεξεργαστών.

Στην εργασία αυτή, ερευνούμε αν ο *page-fault* counter μπορεί να δώσει μια εκτίμηση του αριθμού των εγγραφών που πραγματοποιούνται από το workload του VM. Δεδομένου ότι οι εγγραφές αυτές στη μνήμη του VM αντιστοιχούν σε εγγραφές στην πραγματική μνήμη του host, ο hypervisor χειρίζεται τα page faults που προκύπτουν ώστε τα shadow page tables να παραμένουν ενημερωμένα. Χρησιμοποιούμε το *perf* σε counting mode, επειδή επιθυμούμε την πλήρη καταγραφή όλων των γεγονότων και όχι κάποια δείγματα αυτών. Επίσης, στην εντολή "*perf stat*" χρησιμοποιούμε το flag *-I msecs, –interval-print msecs*, ώστε η τιμή των counters να επιστρέφεται ανά συγκεκριμένα χρονικά διαστήματα. Από τα αποτελέσματα που συλλέγονται σχετικά με το πλήθος των page-fault events δεν προκύπτει κάποια πληροφορία σχετικά με τη συμπεριφορά του workload κατά τη διάρκεια της εκτέλεσής του. Η μοναδική χρήσιμη πληροφορία προκύπτει όταν η εντολή "*perf stat*" εκτελεστεί μαζί με την εφαρμογή εντός του VM. Με αυτό τον τρόπο αποκτούμε μια εκτίμηση του WWS της εφαρμογής με τη βοήθεια των page-fault counters. Όμως, σκοπός μας είναι η δυναμική καταγραφή του πλήθους εγγραφών του workload κατά την εκτέλεσή του. Η διαδικασία αυτή επιθυμούμε να γίνει χωρίς επίγνωση του VM, οπότε ως λύση προτείνεται η καταγραφή των εγγραφών από την πλευρά του host.

Ως επόμενο βήμα εξετάζεται η συσχέτιση του πλήθους των εγγραφών στη μνήμη με τον αριθμό των προσβάσεων στη μνήμη που δεν μπορούν να εξυπηρετηθούν από καμία cache του επεξεργαστή. Υποθέτοντας ότι οι σελίδες αυτές υπάρχουν στη μνήμη, καταγράφουμε με τη βοήθεια του εργαλείου *perf* την τιμή του *cache-misses* Performance Counter (PC). Επίσης, μέσω του "*perf stat*" καταγράφουμε και τον "*instructions*" counter. Συνδυάζοντας τις τιμές των δύο αυτών hardware events και συλλέγοντας τα δεδομένα κάθε ένα δευτερόλεπτο, υπολογίζουμε το MPKI (Misses per Kilo Instructions) με το παρακάτω τύπο:

$$MPKI = \frac{\sum_i^n cache\text{-}misses[i]}{\sum_i^n instructions[i]} \cdot 1000$$

Όπως παρατηρείται στο διάγραμμα 3.6, δεν μπορούμε να βγάλουμε κάποιο ασφαλές συμπέρασμα σχετικά με την συσχέτιση του MPKI και του αριθμού των εγγραφών στη μνήμη, αφού memory-intensive εφαρμογές μπορεί να έχουν χαμηλή MPKI τιμή (π.χ. *write_memory_loop_xMB* εφαρμογές) και αντιστρόφως, μεγάλες MPKI τιμές δεν σημαίνει workload με χαμηλό dirty page rate (π.χ. *450.soplex_rt60* και *450.soplex_rt120*). Μπορούμε να συμπεράνουμε ότι το MPKI μπορεί να μας δώσει πληροφορία σχετικά με τη λειτουργία της cache, δηλαδή όσο χαμηλότερες τιμές λαμβάνει τόσο λιγότερο επηρεάζεται η εφαρμογή από την cache.



Σχήμα 3.6: MPKI με χρήση του *cache-misses* counter του εργαλείου *perf*

3.5.2 IOwait

Το IOwait, σύμφωνα με το man page της εντολής *iostat* [39], είναι το ποσοστό του χρόνου κατά το οποίο η CPU παραμένει ανενεργή περιμένοντας να εξυπηρετηθεί ένα disk I/O. Επομένως, από την πλευρά της CPU, iowait σημαίνει ότι δεν εκτελείται κανένα task, ενώ τουλάχιστον μια διεργασία I/O εκτελείται. Γενικά, η CPU μπορεί να βρίσκεται σε μια από τις εξής τέσσερις καταστάσεις: user, sys, idle, ή iowait. Με τη χρήση μετρητών ο πυρήνας καταγράφει το χρόνο για κάθε μία κατάσταση όταν συμβαίνει μια διακοπή ρολογιού (clock interrupt). Οι τιμές καταγράφονται στο αρχείο */proc/stat* ώστε να είναι προσβάσιμες προς το χρήστη.

Το dirty page rate είναι το βασικό χαρακτηριστικό που καθορίζει τη σύγκλιση ή όχι του pre-copy live migration. Επομένως, τα I/O-intensive workloads έχουν σχεδόν μηδενικό downtime. Με τη βοήθεια του IOwait μπορούμε εύκολα να εντοπίσουμε τις εφαρμογές αυτής της κατηγορίας και να προβλέψουμε τη συμπεριφορά τους κατά το migration. Όμως, με το εργαλείο αυτό δεν αποκτούμε καμία πληροφορία για τα memory-intensive workloads που αποτελούν τη βασική κατηγορία εφαρμογών που επηρεάζουν το migration.

Κεφάλαιο 4

Σχεδιασμός και Υλοποίηση του BitmapTrace μηχανισμού στο QEMU/KVM

Με σκοπό να αποφασιστεί η βέλτιστη σειρά για migration σε ένα σενάριο χρονοδρομολόγησης, οι προαναφερθείσες προσεγγίσεις για την εύρεση του αποτυπώματος μνήμης των VMs που αναλύονται στην ενότητα 3.5 δεν οδηγούν σε σαφή συμπεράσματα για εφαρμογές με έντονη δραστηριότητα μνήμης. Συνεπώς, υλοποιούμε ένα μηχανισμό που ονομάζεται BitmapTrace και πραγματοποιεί profiling ενός VM με βάση τον pre-copy migration αλγόριθμο του QEMU/KVM. Στο κεφάλαιο αυτό αναλύουμε το σχεδιασμό και την υλοποίηση του μηχανισμού και, έπειτα, αξιολογούμε τις παραμέτρους του με την εκτέλεση πολλαπλών workloads (3.2). Επιπλέον, ρυθμίζουμε τις παραμέτρους με στόχο την εύρεση των βέλτιστων τιμών που διασφαλίζουν ένας επαρκές profiling χρονικό διάστημα. Τέλος, διερευνάμε το performance overhead που εισάγεται λόγω της χρήσης του μηχανισμού BitmapTrace.

4.1 Σχεδιασμός

Τα εργαλεία παρακολούθησης της μνήμης που παρέχει το Linux δεν καλύπτουν τις ανάγκες μας για την απόκτηση των απαραίτητων πληροφοριών σχετικά με τη συμπεριφορά του workload. Επομένως, υλοποιούμε ένα μηχανισμό, ο οποίος ονομάζεται BitmapTrace και ενσωματώνεται στον κώδικα του QEMU (έκδοση 2.10.1). Βασιζόμενοι στις δομές και τις συναρτήσεις του QEMU/KVM migration module, ο μηχανισμός αυτός προσομοιώνει τον αλγόριθμο καταγραφής των dirty pages του migration ο οποίος καλεί την *KVM_GET_DIRTY_LOG* ioctl() πριν την εκκίνηση του migration και αποθηκεύει τα στιγμιότυπα του dirty bitmap. Το BitmapTrace module είναι υπεύθυνο για την αποθήκευση του αποτυπώματος μνήμης του VM καθώς αυτό εκτελείται. Το profiling αποτελείται από διαδοχικές επαναλήψεις και οι καινούριες dirty σελίδες (ως προς την προηγούμενη επανάληψη) αποθηκεύονται σε ένα πίνακα. Ο χρήστης ορίζει το συνολικό αριθμό επαναλήψεων (παράμετρος *"iterations"*) καθώς και τη χρονική διάρκεια του κάθε iteration (παράμετρος *"period"*). Κατά τον τερματισμό του BitmapTrace, τα αποτελέσματα αποθηκεύονται σε ένα αρχείο για μελλοντική χρήση.

Για τους σκοπούς της έρευνάς μας, η λειτουργία του BitmapTrace module ξεπερνά τα υπάρχοντα εργαλεία του Linux που αναφέρθηκαν στην ενότητα 3.5. Συγκεκριμένα, στο iteration zero των αποτελεσμάτων φαίνεται ο συνολικός αριθμός σελίδων που δεσμεύεται για την εικονική μηχανή (***VM_Size***). Η τιμή αυτή δείχνει ότι στην πρώτη επανάληψη ολόκληρη η μνήμη του VM θεωρείται dirty. Επίσης, οι I/O-intensive εφαρμογές μπορούν να ανιχνευτούν εύκολα, αφού οι καινούριες dirty σελίδες ανά iteration παραμένουν στην ελάχιστη δυνατή τιμή και παρατηρείται παρόμοια

συμπεριφορά με το profiling ενός idle VM. Τα logs, λοιπόν, που προκύπτουν από το BitmapTrace μηχανισμό παρέχουν τα στιγμιότυπα της συμπεριφοράς της μνήμης του VM ανά iteration, δηλαδή το **Dirty Page Rate**, για κάποιο συγκεκριμένο profiling χρονικό διάστημα. Όταν στο BitmapTrace μηχανισμό δοθεί ως είσοδος μεγάλη τιμή της παραμέτρου *"period"*, π.χ. μεγαλύτερη ή ίση των 3000 msec, τότε μπορεί να υπολογιστεί και το Writable Working Set (**WWS**) του workload. Όπως θα αναλύσουμε σε επόμενη ενότητα, στην περίπτωση αυτή το profiling χρονικό παράθυρο ανά iteration είναι αρκετά μεγάλο ώστε το VM να κάνει dirty όσες σελίδες χρησιμοποιεί. Έτσι, οι σελίδες αυτές αποθηκεύονται προτού το dirty bitmap καθαριστεί και ο μηχανισμός συνεχίσει με την επόμενη επανάληψη.

4.2 Υλοποίηση

Η διαδικασία του live migration στηρίζεται στην καταγραφή των dirty σελίδων της μνήμης ώστε να σταλούν στον κόμβο προορισμού κάθε φορά που αλλάζουν κατάσταση κατά τη διάρκεια του migration. Για αυτό το λόγο στην υλοποίηση του BitmapTrace εκμεταλλευόμαστε την κλήση συστήματος *KVM_GET_DIRTY_LOG* του KVM ώστε να καταγράφονται οι νέες dirty σελίδες που προκύπτουν στο διάστημα μεταξύ δύο διαδοχικών στιγμιοτύπων μνήμης, το οποίο ισούται με τη χρονική διάρκεια μιας περιόδου (*"period"*). Ο συνολικός χρόνος του profiling υπολογίζεται ως το γινόμενο του αριθμού των *"iterations"* με την παράμετρο *"period"*. Οι τιμές των δύο αυτών παραμέτρων προσδιορίζονται κατά την εκκίνηση του BitmapTrace με την αντίστοιχη HMP ή QMP εντολή. Γενικά, το BitmapTrace module χρησιμοποιεί αρκετά χαρακτηριστικά του κώδικα του QEMU migration και κυρίως τις λειτουργίες της δημιουργίας, καθαρισμού και συγχρονισμού του migration dirty bitmap που υπάρχουν στο αρχείο *ram.c*. Για αυτό το λόγο, το κύριο μέρος του κώδικα του BitmapTrace module υλοποιείται στο αρχείο αυτό.

Το QEMU migration χρησιμοποιεί το RAMState struct για την καταγραφή της κατάστασης της RAM μνήμης κατά τη διάρκεια του migration και πολλά μέλη της δομής αυτής σχετίζονται με το QEMUFile και τα RAMBlocks. Όπως έχουμε ήδη αναφέρει στην ενότητα 2.3.4, η δομή QEMUFile αναπαριστά έναν buffer στον οποίο αποθηκεύονται οι dirty σελίδες πριν αποσταλούν στον host προορισμό μέσω ενός socket. Τα RAMBlocks είναι δομές που υπάρχουν σε μια global λίστα που ονομάζεται RAMList, η οποία εκτός από τα RAMBlocks περιέχει και τα dirty memory bitmaps [40]. Εφόσον ο μηχανισμός BitmapTrace ασχολείται με την καταγραφή του πλήθους των εγγραφών στη μνήμη για ένα δοθέν ζεύγος (iterations, period) τιμών, δημιουργούμε το BitmapTraceState struct που είναι παρόμοιο με το RAMState, αλλά τα μέλη του είναι μόνο όσα σχετίζονται με τα dirty pages και bitmap. Επίσης, τα μέλη της RAMState δομής που αφορούν τις βελτιστοποιήσεις του migration είναι εκτός ερευνητικού σκοπού της παρούσας υλοποίησης. Ο κώδικας της δομής BitMapTraceState απεικονίζεται στο 4.1.

Ο BitmapTrace μηχανισμός εκτελείται από ένα ξεχωριστό thread το οποίο είναι υπεύθυνο για την αποθήκευση των αποτελεσμάτων σε ένα αρχείο. Η συνολική διαδικασία του profiling αποτελείται από τρεις φάσεις - ***bitmapTrace_init***, ***bitmapTrace_dirty_bitmap_sync*** (στη συνάρτηση *bitmap_trace_thread*) και ***bitmap_trace_thread_end***. Μόλις δοθεί η HMP ή QMP εντολή εκκίνησης, η συνάρτηση *bitmapTrace_init()* δημιουργεί τη δομή BitmapTraceState και αρχικοποιεί τα μέλη της. Σε ένα νέο thread εκτελείται η συνάρτηση *bitmap_trace_thread()* η οποία αρχικά μαρκάρει ως dirty όλη τη RAM μνήμη του VM, παρόμοια με το αρχικό στάδιο του pre-copy migration

αλγορίθμου. Έπειτα, η συνολική profiling περίοδος διασπάται σε ίσης χρονικής διάρκειας διαστήματα, όπου σε κάθε διάστημα καλείται η συνάρτηση *bitmap_trace_thread()* κατά την οποία γίνεται χρήση της ioctl(*KVM_GET_DIRTY_LOG*) κλήσης συστήματος ώστε να επιστρέψει το αντίγραφο του kernel dirty bitmap μέσω του KVM. Με αυτό τον τρόπο σε κάθε επανάληψη ανιχνεύονται τα dirty τμήματα της μνήμης και ο αριθμός των dirty σελίδων αποθηκεύεται στον πίνακα *dirtyPages_array*. Κατά το τελευταίο στάδιο του *bitmap_trace_thread()*, ο πίνακας αυτός τυπώνεται στο QEMU Monitor και αποθηκεύεται στο αρχείο του οποίου το path δίνεται από τον χρήστη (παράμετρος *"filename"*). Τέλος, το *bitmap_trace_thread_end* σηματοδοτεί την ολοκλήρωση της profiling περιόδου και αποδεσμεύει τη μνήμη που χρησιμοποιήθηκε από τη δομή BitmapTraceState.

```
1        struct BitmapTraceState {
2            int state;
3            FILE *pFile;
4            int64_t current_period;
5            int64_t current_iteration;
6            int64_t iterations;
7            int64_t time_last_bitmap_sync;     /*last time we did a full
    bitmap_sync*/
8            uint64_t num_dirty_pages_period;  /*number of dirty pages since
    start_time(per iteration)*/
9            uint64_t bitmapTrace_dirty_pages; /*number of dirty bits in the
    bitmap*/
10           uint64_t *dirtyPages_array;
11           QemuMutex bitmap_mutex;           /*protects modification of the
    bitmap*/
12           QemuThread thread;
13
14           bool bitmapTrace_thread_running;
15
16           /* The last error that occurred */
17           Error *error;
18         };
19
```

Listing 4.1: BitmapTraceState Struct

Ο κώδικας του BitmapTrace είναι διαθέσιμος στη διεύθυνση https://github.com/dmtrskal/QEMU-BitmapTrace.

4.3 Χρήση του BitmapTrace μηχανισμού

Οι παράμετροι που χρησιμοποιούνται στις εντολές του BitmapTrace's είναι οι εξής:

- state: ενεργοποίηση/απενεργοποίηση του μηχανισμού (on/off) (**υποχρεωτικό**)
- filename: όνομα αρχείου (πλήρες path) όπου αποθηκεύονται τα αποτελέσματα με τις τις dirty σελίδες ανά iteration (**υποχρεωτικό**)
- iterations: συνολικός αριθμός καταγραφής των dirty σελίδων. Η ελάχιστη και η μέγιστη τιμή είναι 3 και 100000 αντίστοιχα. Η προκαθορισμένη (default) τιμή είναι ίση με 3, αν η παράμετρος αυτή δεν προσδιοριστεί κατά την ενεργοποίηση του μηχανισμού.

- period: χρονικό διάστημα σε milliseconds μεταξύ δύο διαδοχικών επαναλήψεων. Η ελάχιστη και η μέγιστη τιμή είναι 10 και 100000 αντίστοιχα. Η προκαθορισμένη (default) τιμή είναι ίση με 10, αν η παράμετρος αυτή δεν προσδιοριστεί κατά την ενεργοποίηση του μηχανισμού.

Παράδειγμα χρήσης QMP εντολής:

(i) Επαληθεύουμε ότι το QEMU ξεκινάει με ενεργοποιημένο το KVM και το QMP έχει δεθεί με ένα TCP socket ώστε να χρησιμοποιηθεί μέσω telnet:
$ qemu [...] -enable-kvm [...] -qmp tcp:localhost:4444,server,nowait

(ii) Εκτελούμε την telnet εντολή :
$ telnet localhost 4444

(iii) Το QMP επιστρέφει μήνυμα χαιρετισμού με πληροφορίες:
{"QMP": {"version": {"qemu": {"micro": 0, "minor": 6, "major": 1}, "package": ""}, "capabilities": []}}

(iv) Εκτελούμε την εντολή *qmp_capabilities*, ώστε το QMP να έλθει σε λειτουργία εντολών (command mode) :
{ "execute": "qmp_capabilities" }

(v) Μπορούμε πλέον να εκτελέσουμε εντολές. Ενεργοποιούμε το BitmapTrace μηχανισμό μέσω της εντολής *bitmap-trace* μαζί με τα απαραίτητα ορίσματα, όπως φαίνεται παρακάτω:
{ "execute": "bitmap-trace", "arguments": { "state": true, "filename": "/home/user/dirty_pages_profiling.log", "iterations": 10, "period": 1000 } }

(vi) Μόλις το BitmapTrace τερματιστεί επιτυχώς, τα αποτελέσματα τυπώνονται στο QEMU Monitor και είναι αποθηκευμένα στο αρχείο που δόθηκε ως όρισμα.

Παράδειγμα χρήσης QEMU Monitor (ή HMP) εντολής:

(i) Επαληθεύουμε ότι το QEMU ξεκινάει με ενεργοποιημένο το KVM:
$ qemu [...] -enable-kvm

(ii) Με την εντολή *"help bitmap-trace"*, μπορούμε να δούμε λεπτομέρειες για τη χρήση της εντολής:
(qemu) help bitmap-trace
bitmap-trace state filename iterations period – Enable/Disable the bitmap tracking mechanism

filename: file where the dirty pages per iteration will be logged

iterations: number of times the dirty pages will be logged

period: time interval in milliseconds between two consecutive iterations

(iii) Ενεργοποιούμε το BitmapTrace μηχανισμό εκτελώντας την εντολή *"bitmap-trace"* μαζί με τα απαραίτητα ορίσματα:
(qemu) bitmap-trace on "/home/user/dirty_pages_profiling.log" 10 1000

(iv) Μόλις το BitmapTrace τερματιστεί επιτυχώς, τα αποτελέσματα τυπώνονται στο QEMU Monitor και είναι αποθηκευμένα στο αρχείο που δόθηκε ως όρισμα.

4.4 Αξιολόγηση του BitmapTrace μηχανισμού

Στις ενότητες αυτές αξιολογούμε το μηχανισμό BitmapTrace και συγκεκριμένα υπολογίζουμε το WWS κάθε εφαρμογής. Στη συνέχεια προσπαθούμε να εντοπίσουμε τις βέλτιστες τιμές των παραμέτρων του μηχανισμού ώστε το profiling να είναι χρονικά επαρκές. Τα πειράματά μας υλοποιούνται με όλα τα benchmarks που περιγράφονται στο κεφάλαιο 3.2. Το BitmapTrace ενεργοποιείται στο QEMU ύστερα από 60 δευτερόλεπτα διάρκεια εκτέλεσης κάθε εφαρμογής. Για τα SPEC benchmarks ο μηχανισμός ενεργοποιείται, εκτός από 60 δευτερόλεπτα runtime, και ύστερα από 120 δευτερόλεπτα και κάθε περίπτωση αναφέρεται ως *rt60* ή *rt120*.

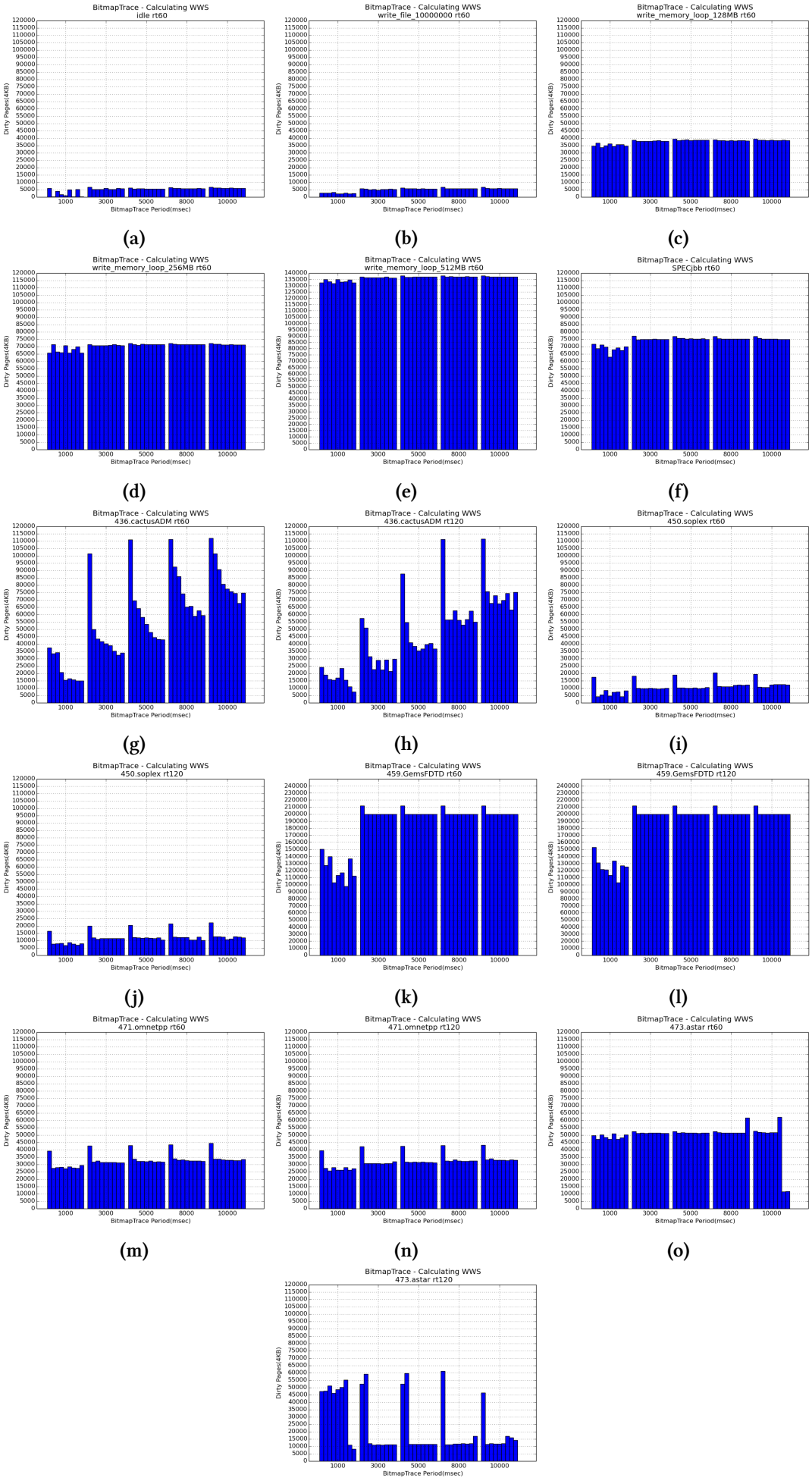
Το μέγεθος της μνήμης κάθε εικονικής μηχανής (*VM_Size*) είναι ίσο με 1G, οπότε 266450 σελίδες είναι dirty στο τέλος του πρώτου iteration του BitmapTrace, παρόμοια με το πρώτο στάδιο του pre-copy live migration αλγόριθμο όπου όλη η μνήμη θεωρείται dirty. Η αρχική αυτή τιμή των dirty pages δεν εμφανίζεται στα διαγράμματα της ενότητας αυτής λόγω της μεγάλης απόκλισης με τις τιμές των υπόλοιπων επαναλήψεων.

4.4.1 Εκτίμηση του WWS

Η ρύθμιση της παραμέτρου *"period"* του BitmapTrace μηχανισμού επιτρέπει στον host να εκτιμήσει το WWS της εφαρμογής που εκτελείται στο VM. Εκτελούμε διάφορα πειράματα θέτοντας σχετικά μεγάλες τιμές στην παράμετρο *"period"*(1000,3000,5000,8000,10000 msec) ενώ η παράμετρος *"iterations"* είναι ίση με 10. Στο διάγραμμα 4.1 απεικονίζονται οι νέες dirty σελίδες ανά iteration για κάθε *"period"*. Παρά το γεγονός ότι ο συνολικός χρόνος του profiling διαφέρει σε κάθε περίπτωση, το πείραμα αυτό σκοπεύει στην εκτίμηση του WWS του workload. Σύμφωνα με τα αποτελέσματα, τιμές του *"period"* μεγαλύτερες ή ίσες από 3000 msec δίνουν την απαραίτητη πληροφορία για τη δυναμική δέσμευση μνήμης από το VM, αφού τα αποτελέσματα είναι παρόμοια καθώς μεγαλώνουμε την τιμή της παραμέτρου. Επίσης, μπορούμε εύκολα να διακρίνουμε ότι το WWS των *write_memory_loop_xMB* microbenchmarks διπλασιάζεται από 128MB σε 256MB και από 256MB σε 512MB. Η I/O-intensive εφαρμογή, όπως αναμενόταν, δεν καταναλώνει σχεδόν καθόλου μνήμη, αφού τα αποτελέσματα είναι ίδια με το *idle* VM. Τέλος, μπορούμε να διακρίνουμε την επίδραση της χρονικής στιγμής που θα εκκινήσει το migration (*running_time*) αφού το WWS διαφέρει σημαντικά σε ορισμένα benchmark, όπως π.χ. τα *436.cactusADM* και *473.astar*.

4.4.2 Πειραματική Ρύθμιση της παραμέτρου *period*

Ως δεύτερη αξιολόγηση του μηχανισμού παρακολούθησης της μνήμης επιχειρούμε να ρυθμίσουμε την *"period"* παράμετρο ώστε να βρούμε μια βέλτιστη τιμή. Λαμβάνοντας υπόψη τα αποτελέσματα που προκύπτουν από την προηγούμενη ενότητα για την αξιολόγηση του WWS, δεν θέλουμε μια μεγάλη τιμή για την παράμετρο *"period"*, αφού οι νέες dirty σελίδες ανά iteration θα έχουν παρόμοιες τιμές στο τέλος κάθε επανάληψης. Έτσι, ως πρώτη προσέγγιση, οι τιμές που θα χρησιμοποιήσουμε θα είναι μικρότερες του 3000 msec. Συγκεκριμένα, στην παράμετρο *"period"* θέτουμε τις τιμές: 10,50,100,200,500,1000,1500,2000 ή 2500 msec και η παράμετρος *"iterations"* ρυθμίζεται αντίστοιχα ώστε ο συνολικός χρόνος του profiling να είναι 20000 msec σε κάθε πείραμα. Τα αποτελέσματα για κάθε εφαρμογή φαίνονται στο διάγραμμα 4.2. Μπορούμε να διακρίνουμε ότι οι μικρές τιμές του *"period"*, όπως 10,20,100,200 msec, περιορίζουν τη δυνατότητα του workload



Σχήμα 4.1: Πειραματική αξιολόγηση του WWS

ως προς το ρυθμό εγγραφής σελίδων, ενώ μεγαλύτερα χρονικά διαστήματα ανά επανάληψη επιτρέπουν μεγαλύτερο αριθμό εγγραφών στη μνήμη. Από την άλλη πλευρά, για τις απαιτητικές σε μνήμη εφαρμογές όπως οι *write_memory_loop_xMB* και *459.GemsFDTD*, η επιλογή μεγάλων τιμών της *"period"* παραμέτρου (2000 ή 2500 msec) ίσως οδηγήσει σε αποτελέσματα παρόμοια με το WWS, όπως φαίνεται στο διάγραμμα 4.1. Με βάση τα αποτελέσματα προτείνεται η χρήση τιμών που ούτε αποκρύπτουν την πληροφορία σχετικά με τη συμπεριφορά της εφαρμογής ως προς τη μνήμη ούτε επιστρέφουν το WWS μετά από κάθε επανάληψη.

Δεδομένου ότι ο μηχανισμός BitmapTrace προσομοιώνει τον αλγόριθμο καταγραφής των dirty pages του migration, λαμβάνουμε υπόψη ότι το QEMU μηδενίζει τους counters των dirty pages ύστερα από κάθε bitmap sync και εφόσον ο συγχρονισμός αυτός έχει χρονική διάρκεια μεγαλύτερη ή ίση από 1000 msec. Το αντίστοιχο κομμάτι κώδικα του QEMU *migration/ram.c* αρχείου παρατίθεται στο listing 4.2.

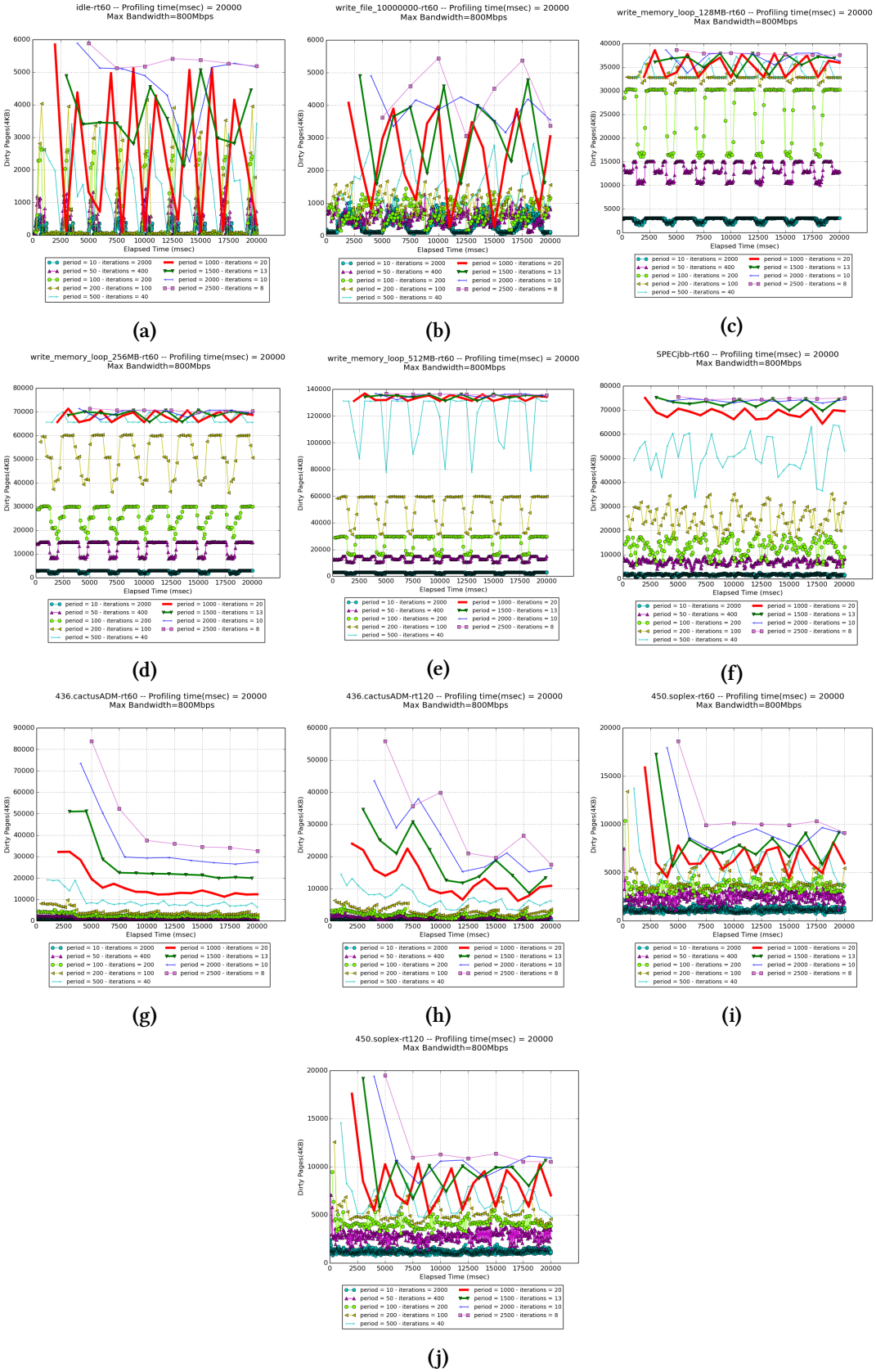
```
1   static void migration_bitmap_sync(RAMState *rs)
2   {
3       RAMBlock *block;
4       int64_t end_time;
5       uint64_t bytes_xfer_now;
6
7       ram_counters.dirty_sync_count++;
8
9       if (!rs->time_last_bitmap_sync) {
10          rs->time_last_bitmap_sync = qemu_clock_get_ms(QEMU_CLOCK_REALTIME);
11      }
12
13      trace_migration_bitmap_sync_start();
14      memory_global_dirty_log_sync();
15
16      qemu_mutex_lock(&rs->bitmap_mutex);
17      rcu_read_lock();
18      RAMBLOCK_FOREACH(block) {
19          migration_bitmap_sync_range(rs, block, 0, block->used_length);
20      }
21      rcu_read_unlock();
22      qemu_mutex_unlock(&rs->bitmap_mutex);
23
24      trace_migration_bitmap_sync_end(rs->num_dirty_pages_period);
25
26      end_time = qemu_clock_get_ms(QEMU_CLOCK_REALTIME);
27
28      /* more than 1 second = 1000 millisecons */
29      if (end_time > rs->time_last_bitmap_sync + 1000) {
30          /* calculate period counters */
31          ram_counters.dirty_pages_rate = rs->num_dirty_pages_period * 1000
32              / (end_time - rs->time_last_bitmap_sync);
33          bytes_xfer_now = ram_counters.transferred;
34          ...
35
36          /* reset period counters */
37          rs->time_last_bitmap_sync = end_time;
38          rs->num_dirty_pages_period = 0;
39          rs->bytes_xfer_prev = bytes_xfer_now;
40
41          ...
42  }
43
```

Listing 4.2: *migration_bitmap_sync* συνάρτηση του QEMU migration κώδικα

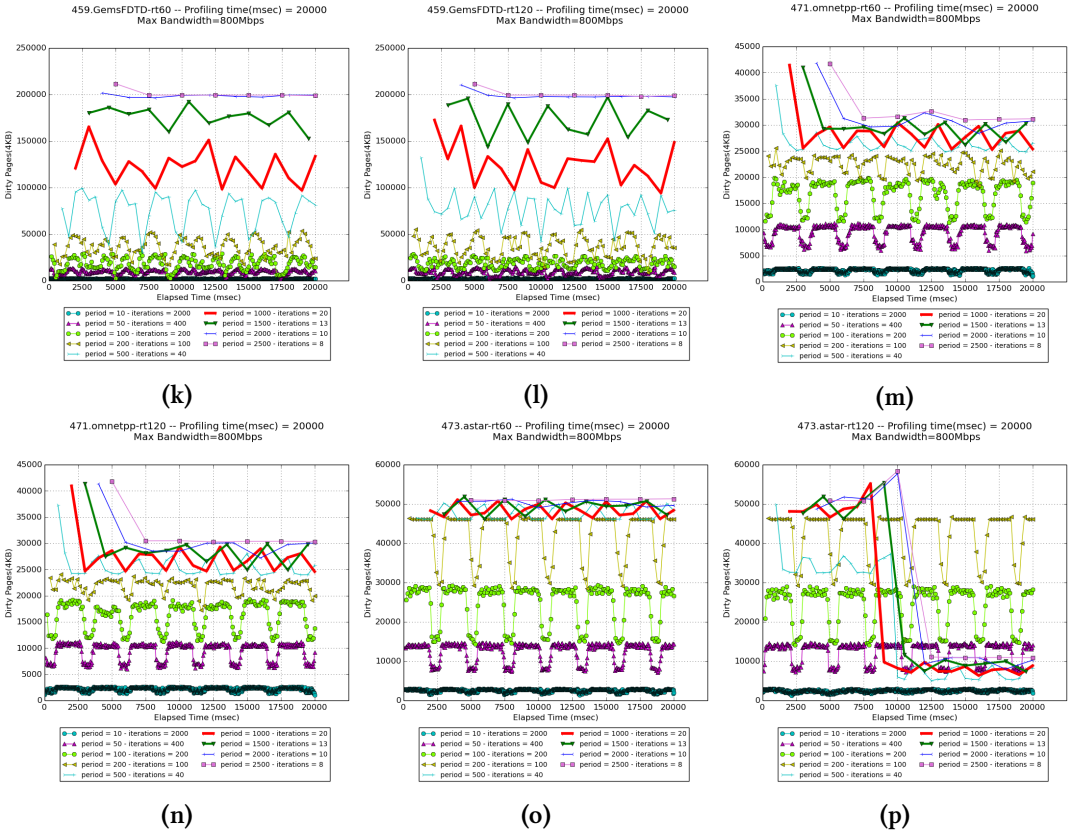
Επομένως, οι προτεινόμενες τιμές για την παράμετρο *"period"* του BitmapTrace μηχανισμού είναι 1000 ή 1500 msec. Οι τιμές αυτές θα χρησιμοποιηθούν στις επόμενες ενότητες καθώς και στα πειράματα του κεφαλαίου 5.

4.4.3 Πειραματική Ρύθμιση της παραμέτρου *iterations*

Στο κομμάτι αυτό της αξιολόγησης του BitmapTrace μηχανισμού ερευνούμε την επίδραση της παραμέτρου *"iterations"*. Σύμφωνα με τα διαγράμματα 4.3 ο αριθμός των επαναλήψεων καθορίζει το πλήθος των δειγμάτων που θα συλλεχθούν καθώς και το συνολικό χρονικό διάστημα του profiling για μία συγκεκριμένη τιμή του *"period"*. Αυξάνοντας τον αριθμό των *"iterations"* μπορούμε να αποκτήσουμε ένα μεγαλύτερο στιγμιότυπο της μνήμης. Παρατηρούμε ότι ακόμα και με 5 επαναλήψεις



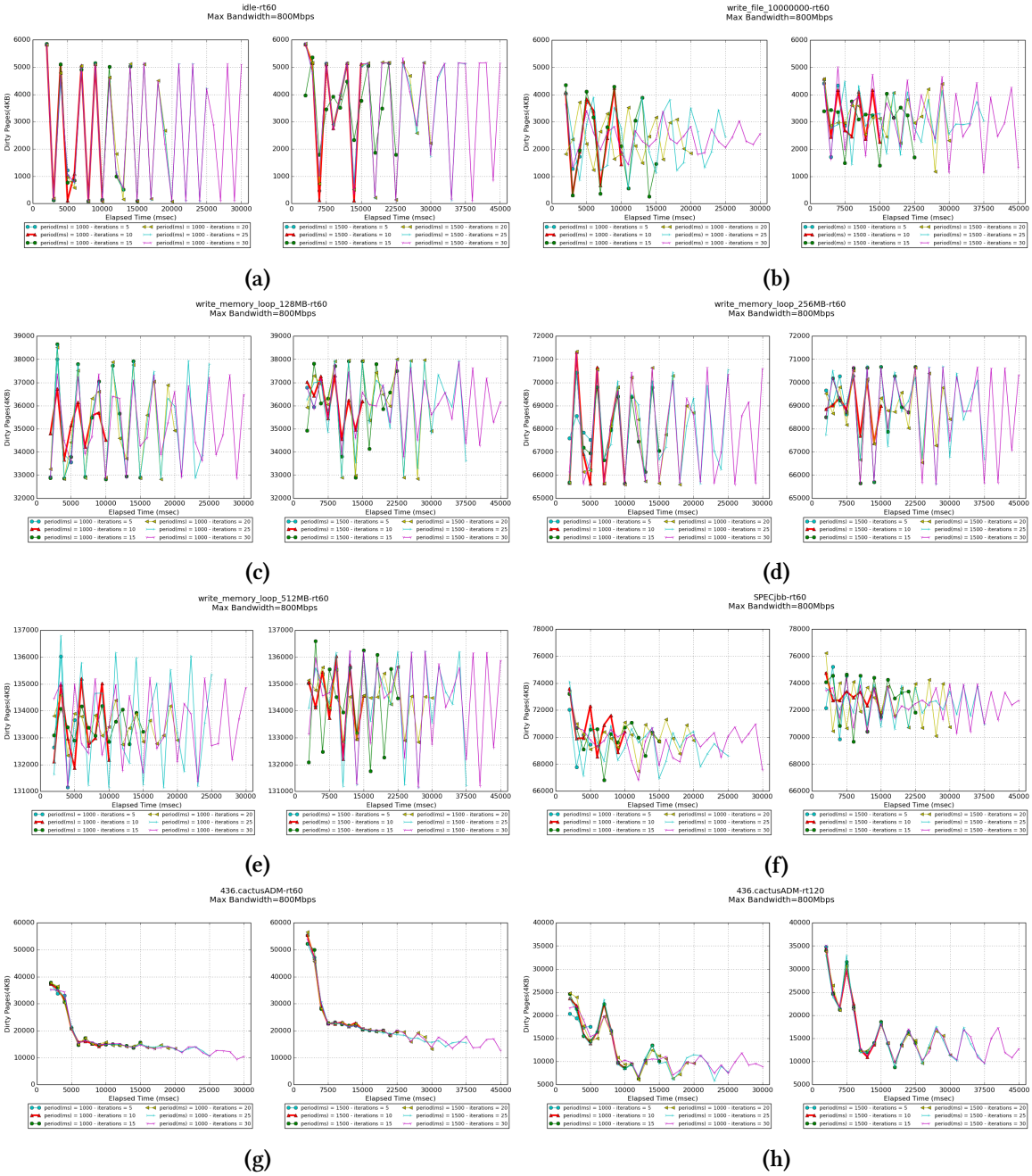
Σχήμα 4.2: Πειραματική ρύθμιση της παραμέτρου *"period"* του BitmapTrace



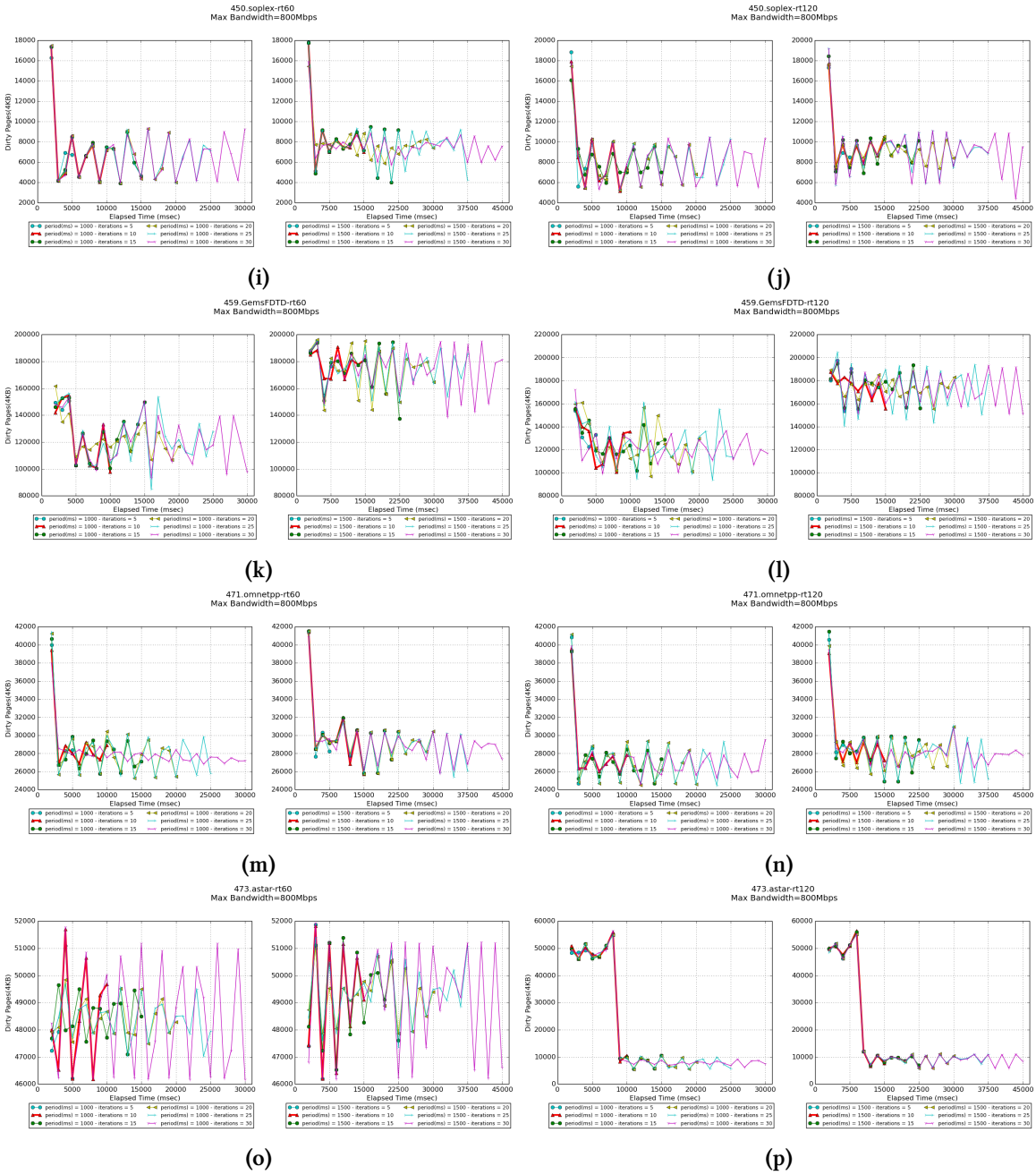
Σχήμα 4.2: Πειραματική ρύθμιση της παραμέτρου *"period"* του BitmapTrace (συνέχεια)

ο διαχειριστής μπορεί να αποκτήσει μια άποψη για τη συμπεριφορά του guest. Όμως, η επιλογή ενός μικρού αριθμού δειγμάτων μπορεί να μην οδηγήσει σε ασφαλή συμπεράσματα για το αποτύπωμα της μνήμης πραγματικών εφαρμογών που αλλάζουν δυναμικά το dirty page rate τους. Αυτό φαίνεται και από την εφαρμογή *473.astar_rt120* (διάγραμμα 4.3p), η οποία ξαφνικά μειώνει το ρυθμό εγγραφών. Επομένως, η επιλογή 10 *"iterations"* με *"period"* 1000msec, δηλαδή συνολικό profiling διάστημα ίσο με 10 δευτερόλεπτα, προτείνεται για την καταγραφή της μνήμης της εφαρμογής. Σε περιπτώσεις που το *VM_Size* είναι μεγαλύτερο από 1GB, τότε πιθανώς ο συνολικός χρόνος του profiling να πρέπει να αυξηθεί.

Με γνώμονα την ανάλυση για την παράμετρο *"period"* στην προηγούμενη ενότητα, επιλέγουμε τις τιμές 1000ms και 1500ms για τα πειράματα σχετικά με την παράμετρο *"iterations"*. Ειδικότερα, στα γραφήματα με *"period"* 1000ms μπορούμε να παρατηρήσουμε το **Dirty Page Rate** κάθε workload κατά τη διάρκεια του profiling, αφού απεικονίζονται οι καινούριες dirty σελίδες κάθε 1000ms. Επιπλέον, το πλήθος των εγγραφών στη μνήμη έχει παρόμοιο μοτίβο για *"period"* 1000ms και 1500ms και σε ορισμένες περιπτώσεις, όπως για παράδειγμα για τα benchmarks *SPECjbb*, *436.cactusADM* και *459.GemsFDTD*, η γραμμή για την τιμή 1500ms είναι κάθετα μετατοπισμένη σε σύγκριση με την γραφική της τιμής 1000ms.



Σχήμα 4.3: Πειραματική ρύθμιση της παραμέτρου *"iterations"* του BitmapTrace



Σχήμα 4.3: Πειραματική ρύθμιση της παραμέτρου *"iterations"* του BitmapTrace (συνέχεια)

4.4.4 Performance Overhead

Για την ανάλυση της επίδρασης του BitmapTrace μηχανισμού στην απόδοση του VM αξιολογούμε τη συμπεριφορά του στην περίπτωση μιας απαιτητικής εφαρμογής. Τροποποιούμε το *write_memory_loop_xMB* microbenchmark ώστε ο χρήστης να ρυθμίζει τη διάρκεια εκτέλεσής του καθώς και το πλήθος των εγγραφών σε ολόκληρο το δεσμευμένο τμήμα μνήμης (xMB). Στο πείραμα αυτό, η εφαρμογή μας δεσμεύει 512MB μνήμης. Χρησιμοποιούμε, επίσης, ένα microbenchmark το οποίο δεν πραγματοποιεί εγγραφές στη μνήμη. Πραγματοποιούνται τα παρακάτω πειράματα με ενεργοποιημένο ή απενεργοποιημένο το BitmapTrace μηχανισμό:

(i) Για το προτεινόμενο ζεύγος τιμών (*iterations,period*)=(10,1000) εκτελούμε το memory-intensive benchmark για διαφορετικό αριθμό εγγραφών στη μνήμη, άρα και διαφορετική χρονική διάρκεια εκτέλεσης. (Πίνακας 4.1)

(ii) Για διάφορα ζεύγη τιμών (*iterations,period*) εκτελούμε το memory-intensive benchmark για συγκεκριμένο αριθμό εγγραφών στη μνήμη, άρα και συγκεκριμένη χρονική διάρκεια εκτέλεσης. (Πίνακας 4.2)

(iii) Για το προτεινόμενο ζεύγος τιμών (*iterations,period*)=(10,1000) εκτελούμε το memory-intensive benchmark και το απλό (χωρίς εγγραφές) benchmark για συγκεκριμένο αριθμό εγγραφών στη μνήμη, άρα και για συγκεκριμένη χρονική διάρκεια εκτέλεσης. (Πίνακας 4.3)

Για τον υπολογισμό της συνολικής χρονικής διάρκειας εκτέλεσης των εφαρμογών χρησιμοποιούμε την εντολή *time* [41] του Linux. Όπως παρατηρείται στους παρακάτω πίνακας, ο μηχανισμός BitmapTrace μπορεί να θεωρηθεί ένα ”ελαφρύ” εργαλείο παρακολούθησης αφού προσθέτει ένα offset 5-7 δευτερόλεπτα στο χρόνο ολοκλήρωσης της memory-intensive εφαρμογής για τις προτεινόμενες τιμές των *”iterations”* και *”period”*, σύμφωνα με τις ενότητες 4.4.2 και 4.4.3. Αυτός ο επιπρόσθετος χρόνος εκτέλεσης μπορεί να θεωρηθεί αμελητέος αν λάβουμε υπόψη ότι το workload του VM μπορεί να εκτελείται για ώρες. Επίσης, τα αποτελέσματα αυτά προκύπτουν για την εφαρμογή με την πιο έντονη δραστηριότητα μνήμης, αφού πραγματοποιεί συνεχώς εγγραφές. Σε περιπτώσεις εφαρμογών με λιγότερη επικοινωνία με τη μνήμη, ο χρόνος του overhead μειώνεται αισθητά, ενώ για εφαρμογές που δεν αλληλεπιδρούν καθόλου με τη μνήμη το overhead που εισάγεται είναι σχεδόν μηδενικό, όπως φαίνεται στον πίνακα 4.3.

Σύμφωνα με τον πίνακα 4.2, η αύξηση του πλήθους των επαναλήψεων οδηγεί σε μικρή αύξηση του overhead. Αυτό είναι αναμενόμενο, αφού το KVM διαβάζει το kernel dirty bitmap, μέσω της ioctl(*KVM_GET_DIRTY_LOG*), περισσότερες φορές προκειμένου να υπολογιστούν οι dirty σελίδες.

BitmapTrace (*iterations,period*)	Number of memory writes	Execution Time (msec)	BitmapTrace Overhead (msec)
No BitmapTrace	(10000)	175556	-
	(20000)	348883	-
	(30000)	522520	-
(10,1000)	(10000)	180570	5014
	(20000)	353771	4888
	(30000)	529954	7434

Πίνακας 4.1: Runtime overhead του BitmapTrace με παραμέτρους (*”iterations”*, *”period”*)=(10,1000) σε εφαρμογή που πραγματοποιεί συνεχόμενα writes σε 512MB μνήμης

BitmapTrace (*iterations*,*period*)	Execution Time (msec)	BitmapTrace Overhead (msec)
No BitmapTrace	184200	-
(5,1000)	185845	1645
(10,1000)	188708	4508
(15,1000)	190597	6397
(5,1500)	186120	1920
(10,1500)	188952	4752
(15,1500)	190675	6475

Πίνακας 4.2: Runtime overhead του BitmapTrace με διάφορες τιμές των παραμέτρων (*"iterations"*, *"period"*) σε εφαρμογή που πραγματοποιεί συνεχόμενα writes σε 512MB μνήμης

BitmapTrace (*iterations*,*period*)	Application type	Number of completed loops	BitmapTrace Performance Overhead (%)
No BitmapTrace	memory-intensive(512MB)	10248	-
	simple(without memory writes)	2523475150	-
(10,1000)	memory-intensive(512MB)	9982	2,59
	simple(without memory writes)	2522351699	0,04

Πίνακας 4.3: Performance overhead του BitmapTrace σε διαφορετικές εφαρμογές (execution time=3 min)

Κεφάλαιο 5

Προσέγγιση Μηχανικής Μάθησης για Migration Υποψήφιων VM

Οι αλγόριθμοι μηχανικής μάθησης χτίζουν ένα μαθηματικό μοντέλο από δεδομένα δειγματοληψίας, τα οποία είναι γνωστά ως ”δεδομένα εκπαίδευσης” (training data), ώστε να κάνουν προβλέψεις για νέα δεδομένα εισόδου. Χρησιμοποιώντας την προσέγγιση της επιβλεπόμενης μάθησης (supervised learning) προσπαθούμε να δημιουργήσουμε μοντέλα που βασίζονται στο αποτύπωμα μνήμης των εφαρμογών πριν την εκκίνηση του migration και προβλέπουν α) την ελάχιστη τιμή της παραμέτρου *downtime-limit* που διασφαλίζει την επιτυχή ολοκλήρωση του live migration και β) τους migration χρόνους (downtime, total-time). Οι cloud πάροχοι υπηρεσιών επιθυμούν την εύρεση των βέλτιστων αποφάσεων για migration ανάμεσα στο πλήθος υποψήφιων VMs με σκοπό τη μείωση του κόστους του migration τηρώντας, όμως, τις συμφωνίες σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου (SLAs). Αφού, λοιπόν, ορίσουμε κάποιες πολιτικές λειτουργίας εφαρμόζουμε τα μοντέλα μας σε ένα live migration framework που ταξινομεί τις εικονικές μηχανές που εκτελούνται σε ένα φυσικό μηχάνημα και επιλέγει για migration τα ”καλύτερα” υποψήφια VMs με βάση τη συμπεροφορά τους ως προς το πλήθος εγγραφών στη μνήμη.

5.1 Κατασκευή Μοντέλου

Ο βασικός στόχος της εργασίας αυτής είναι η πρόβλεψη της σειράς με την οποία πρέπει να μεταφερθούν σε έναν άλλον host οι εικονικές μηχανές. Η προσέγγιση της μοντελοποίησης πρέπει να γίνει από το φυσικό μηχάνημα και χωρίς επίγνωση της διαδικασίας αυτής από το VM. Η μετρική-στόχος (target metric) του μοντέλου μας είναι η ελάχιστη τιμή του *downtime-limit* που διασφαλίζει τη σύγκλιση του pre-copy live migration αλγορίθμου και συνεπώς την επιτυχή ολοκλήρωση του migration. Ως επόμενο βήμα στο πρόβλημά μας, χτίζουμε ξεχωριστά μοντέλα που προσπαθούν να προβλέψουν το downtime και το total time αντίστοιχα. Αν και το accuracy των μοντέλων αυτών είναι χαμηλότερο των προσδοκιών μας, μπορούν να χρησιμοποιηθούν ως μια εκτίμηση για αυτές τις μετρικές απόδοσης του migration.

5.1.1 Κατασκευή Συνόλου Δεδομένων

Η επιλογή των χαρακτηριστικών ως training data του μοντέλου είναι σημαντική για την ακρίβεια της πρόβλεψης. Για αυτό το λόγο, τα δεδομένα που επιλέγουμε για την εκπαίδευση των μοντέλων μηχανικής μάθησης βασίζονται στην παρακολούθηση των VMs μέσω του BitmapTrace μηχανισμού καθώς εκτελούν διαφορετικές εφαρμογές (3.2). Επίσης, πραγματοποιούμε ένα μεγάλο

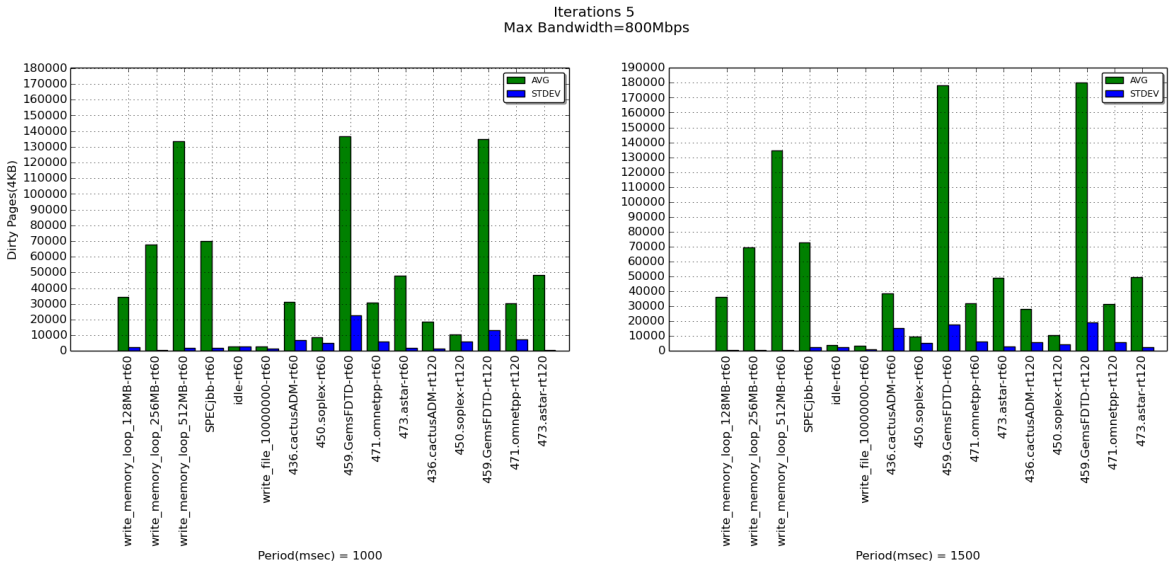
πλήθος live migration εικονικών μηχανών με διαφορετικές ρυθμίσεις των configuration παραμέτρων. Η τοπολογία που χρησιμοποιείται είναι ίδια με αυτή που αναφέρεται στο κεφάλαιο 3.1 όπου το μέγιστο διαθέσιμο εύρος ζώνης (*max-bandwidth*) ρυθμίζεται στα 800Mbps και το μέγεθος της μνήμης των VMs είναι 1G.

Σχετικά με το πρώτο μοντέλο (*min_success_downtime-limit* μοντέλο), ο μέσος όρος (AVG)[1] και η τυπική απόκλιση (STDEV)[2] των dirty σελίδων του guest για ένα συγκεκριμένο profiling χρονικό διάστημα θεωρούνται ως χαρακτηριστικά εισόδου του μοντέλου. Οι dirty σελίδες προκύπτουν από το μηχανισμό BitmapTrace με τις προτεινόμενες τιμές για το ζεύγος (*iterations,period*), όπως αναλύεται στην ενότητα 4.4.

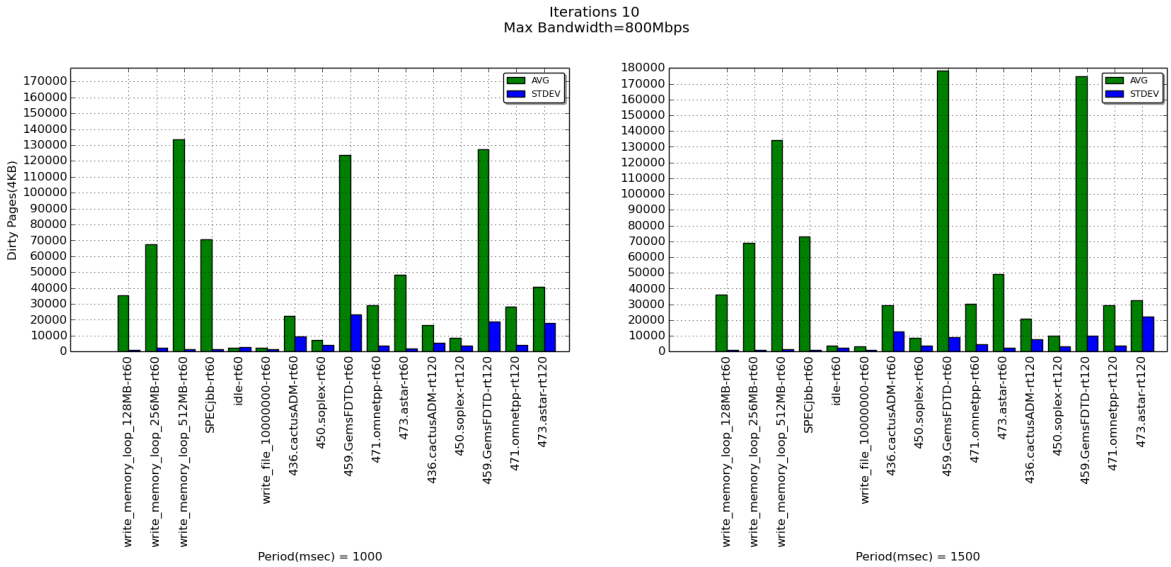
Ο μέσος όρος (AVG) και η τυπική απόκλιση (STDEV) από τα δείγματα που συλλέγουμε υπολογίζονται με τη χρήση του πακέτου NumPy [42] της Python και τα αποτελέσματα απεικονίζονται στο διάγραμμα 5.1.

[1] Η μέση αριθμητική τιμή ή Μέσος όρος προκύπτει από το πηλίκο διαίρεσης του αθροίσματος των τιμών μιας μεταβλητής δια του συνολικού πλήθους τους

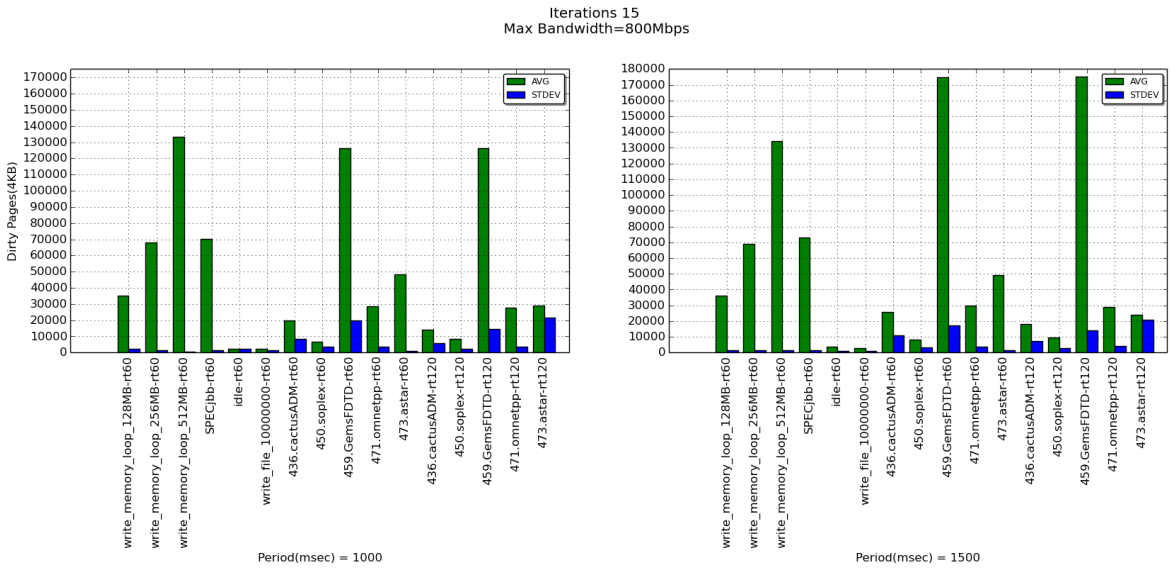
[2] Η τυπική απόκλιση ενός συνόλου δεδομένων είναι ένα μέτρο που χρησιμοποιείται για να υπολογιστεί το ποσό της μεταβολής ή της διασποράς ενός συνόλου τιμών δεδομένων από το μέσο όρο του και υπολογίζεται από την τετραγωνική ρίζα της διακύμανσης του.



(a)

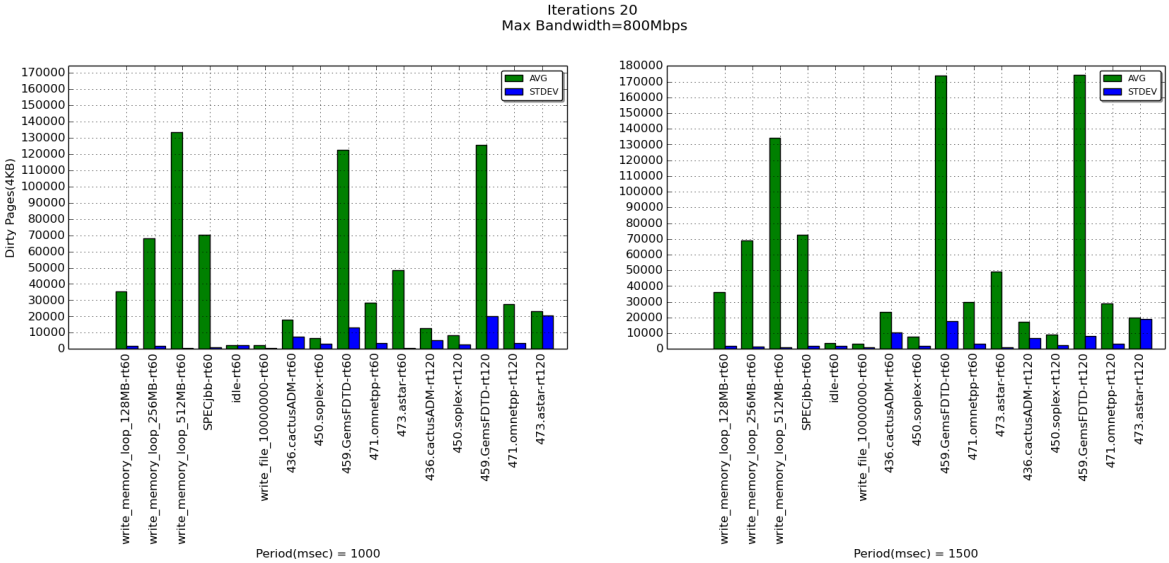


(b)

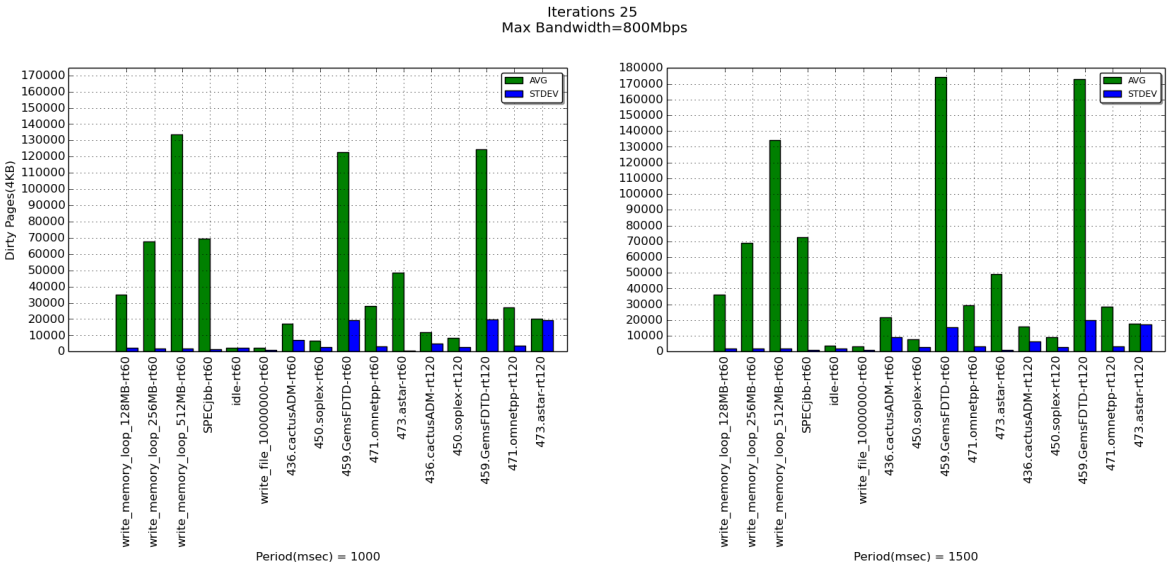


(c)

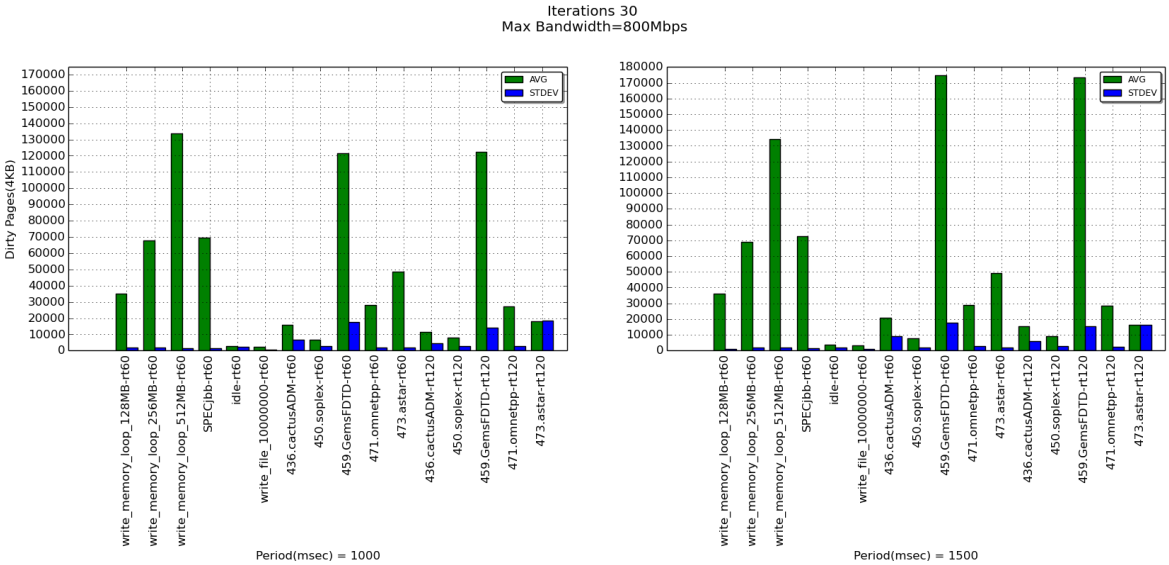
Σχήμα 5.1: AVG και STDEV Dirty Pages κατά την profiling περίοδο



(d)



(e)



(f)

Σχήμα 5.1: AVG και STDEV Dirty Pages κατά την profiling περίοδο (συνέχεια)

Η target metric του μοντέλου είναι η ελάχιστη τιμή του *downtime-limit* που οδηγεί σε επιτυχές live migration με χρήση του QEMU/KVM. Χρησιμοποιώντας το μοντέλο αυτό επιθυμούμε να αποφασίσουμε αν ο pre-copy αλγόριθμος θα συγκλίνει με ένα συγκεκριμένο configuration του migration και για δεδομένη συμπεριφορά της εφαρμογής ως προς τη μνήμη. Στον παρακάτω πίνακα παρουσιάζονται ξανά τα αποτελέσματα που προέκυψαν από την ανάλυση του κεφαλαίου 3.3.5 για την ελάχιστη τιμή του *downtime-limit* για επιτυχές migration κάθε εφαρμογής.

Benchmark	***downtime-limit*(msec)**	
	rt60	**rt120**
idle	20	20
write_file	20	20
write_memory_loop_16MB	200	200
write_memory_loop_32MB	400	400
write_memory_loop_64MB	700	700
write_memory_loop_128MB	1400	1400
write_memory_loop_256MB	2600	2600
write_memory_loop_512MB	5200	5200
SPECjbb (1 thread)	600	600
SPECjbb (8 threads)	2400	2400
436.cactusADM	20	20
450.soplex	80	150
459.GemsFDTD	2800	6000
471.omnetpp	900	900
473.astar	1800	200

Πίνακας 5.1: Ελάχιστη τιμή του *downtime-limit* για επιτυχές migration κάθε εφαρμογής (*max-bandwidth*=800Mbps)

Συνδυάζοντας τα παραπάνω αποτελέσματα καταλήγουμε στα εξής συμπεράσματα:

- Σχετικά με τις memory-intensive εφαρμογές παρατηρούμε ότι η τιμή του AVG Dirty Pages είναι αρκετά υψηλή και το STDEV είναι είτε σχεδόν μηδενικό (π.χ. *write_memory_loop_xMB_rt60*, *SPECjbb* και *473.astar_rt60*) είτε αντιστοιχεί σε ένα μικρό ποσοστό του AVG (π.χ. *459.GemsFDTD_rt60* και *459.GemsFDTD_rt120*).
- Η I/O intensive εφαρμογή *write_file* έχει τις ελάχιστες AVG και STDEV τιμές που είναι ίσες με τις τιμές του *idle* VM. Αποδεικνύεται, λοιπόν, ότι στις περιπτώσεις αυτές δεν έχουν πραγματοποιηθεί προσβάσεις στη μνήμη.
- Όταν ο λόγος STDEV/ AVG είναι μεγαλύτερος από 0.5 τότε φαίνεται ότι το STDEV των dirty σελίδων υπερτερεί και είναι ο κύριος παράγοντας που καθορίζει το ελάχιστο *downtime-limit* για την επιτυχή ολοκλήρωση του migration, όπως φαίνεται από τις τιμές των *436.cactusADM_rt60*, *436.cactusADM_rt120* και *473.astar_rt120*. Η υψηλή τιμή της τυπικής απόκλισης δείχνει ότι τα δείγματα απλώνονται σε ένα μεγαλύτερο εύρος τιμών. Έτσι, ο αλγόριθμος του migration μπορεί να επωφεληθεί από το χαμηλό dirty page rate που προκύπτει ανά φάσεις και το κριτήριο σύγκλισης να ικανοποιηθεί.
- Όταν οι AVG τιμές διαφορετικών εφαρμογών είναι σχεδόν ίσες αλλά οι STDEV τιμές διαφέρουν, οι εφαρμογές με το μεγαλύτερο STDEV έχουν μικρότερο *downtime-limit* για επιτυχές

migration. Για παράδειγμα, στο διάγραμμα 5.1b όταν το period είναι 1000ms οι AVG τιμές του *436.cactusADM_rt60* και του *SPECjbb-1thread* είναι περίπου 20000 dirty σελίδες. Όμως, το STDEV του *436.cactusADM_rt60* είναι πολύ υψηλότερο για αυτό και η ελάχιστη *downtime-limit* τιμή του είναι 20ms ενώ το *SPECjbb-1thread* έχει ελάχιστο *downtime-limit* 600ms.

- Τα SPEC benchmarks που αποτελούν πραγματικές εφαρμογές έχουν υψηλότερες STDEV τιμές σε σύγκριση με τα *write_memory_loop_xMB* microbenchmarks των οποίων το STDEV είναι σχεδόν μηδέν λόγω των συνεχών εγγραφών στη μνήμη. Αυτή η απόκλιση ανάμεσα στις STDEV τιμές των real-world benchmarks δείχνει το διαφορετικό μοτίβο προσπέλασης της μνήμης από τα διάφορα benchmarks.

Για τα υπόλοιπα μοντέλα πρόβλεψης (*downtime* και *total_time* μοντέλα) δίνεται ως χαρακτηριστικό εισόδου και το *downtime-limit* εκτός από το AVG και STDEV των dirty σελίδων. Όπως παρουσιάζεται στο διάγραμμα 3.3.2, ο pre-copy live migration αλγόριθμος του QEMU/KVM στοχεύει στην ελαχιστοποίηση του total migration time. Επομένως, η παράμετρος *downtime-limit* είναι ιδιαίτερα σημαντική για τη σύγκλιση του αλγορίθμου και το downtime είναι αντιστρόφως ανάλογο του total time. Συγκεκριμένα, το σύνολο δεδομένων, με το οποίο τα μοντέλα αυτά εκπαιδεύονται, αποτελείται από αποτελέσματα που προέκυψαν με την εκτέλεση μεγάλου πλήθους live migration όπου η παράμετρος *downtime-limit* λαμβάνει διαφορετικές τιμές. Πριν την εκκίνηση του migration, εκτελούμε το μηχανισμό BitmapTrace για την καταγραφή των εγγραφών στη μνήμη και οι τιμές των AVG και STDEV υπολογίζονται. Με την επιτυχή ολοκλήρωση κάθε μεταφοράς του VM, καταγράφουμε τις τιμές downtime και total time. Οι τιμές του *downtime-limit* που θέτονται στα πειράματά μας ξεκινούν από την τιμή *min_success_downtime-limit* και φτάνουν μέχρι αρκετά υψηλές τιμές που οδηγούν σε άμεση μεταφορά του VM με total time σχεδόν ίσο με το downtime.

Η συλλογή των παραπάνω τιμών για τα χαρακτηριστικά και τις μετρικές-στόχους που χρησιμοποιούνται για την εκπαίδευση (training) του κάθε μοντέλου αναφέρονται ως Ιστορικά Δεδομένα (Historical Data).

5.1.2 Δημιουργία Μοντέλου

Με βάση την ανάλυση της ενότητας 5.1.1, τα χαρακτηριστικά AVG και STDEV των dirty σελίδων υποδεικνύουν ότι συσχετίζονται γραμμικά με το *min_success_downtime-limit*. Επίσης, είναι φανερή και η συσχέτιση αυτών των χαρακτηριστικών με το downtime. Αν και το total time του migration υποθέτουμε ότι συσχετίζεται με τα AVG και STDEV dirty pages, υπάρχουν κι άλλοι παράμετροι που καθορίζουν τη συσχέτιση αυτή, όπως για παράδειγμα το bandwidth της ζεύξης που στα πειράματά μας θεωρείται σταθερό. Παρ' όλα αυτά εξετάζουμε τη γραμμική συσχέτιση των μετρικών αυτών και ο βαθμός της εξάρτησής τους θα εκτιμηθεί σε επόμενες ενότητες. Έτσι, η τεχνική του Linear Regression επιλέγεται για την εφαρμογή των δεδομένων μας.

Για την κατασκευή και την αξιολόγηση των regression μοντέλων χρησιμοποιείται η εργαλειοθήκη scikit-learn v0.20 [43] της Python. Χρησιμοποιούμε το Support Vector Regression (SVR) [44] που αποτελεί μια τεχνική regression για σύνθετα δεδομένα που έχουν γραμμική ή μη-γραμμική συσχέτιση μεταξύ των χαρακτηριστικών εισόδου και του target metric. Η συσχέτιση αυτή προσδιορίζεται από τον τύπο του kernel που θα χρησιμοποιήσει ο αλγόριθμος. Προκειμένου να ρυθμίσουμε τις παραμέτρους της SVR εκτιμήτριας συνάρτησης χρησιμοποιούμε το GridSearchCV [45]

του scikit-learn. Μετά από εξαντλητική αναζήτηση με διάφορες τιμές των παραμέτρων επιλέγουμε τις παρακάτω τιμές που πετυχαίνουν και το καλύτερο score:

- Για το *min_success_downtime-limit* μοντέλο επιλέγουμε τον 'linear' kernel με τιμή C=1.0 για την penalty παράμετρο και epsilon=0.1 (default τιμή).
- Για τα *downtime* και *total_time* μοντέλα επιλέγουμε τον 'linear' kernel με τιμή C=0.1 για την penalty παράμετρο και epsilon=0.1 (default τιμή).

Η penalty παράμετρος C χρησιμοποιείται για να διατηρεί την κανονικοποίηση (regularization). Για αυτό το λόγο αναφέρεται και ως regularization παράμετρος και καθορίζει το περιθώριο της εσφαλμένης ταξινόμησης (misclassification or error term). Όταν το C λάβει χαμηλές τιμές, δηλαδή το penalty είναι χαμηλό, τότε το περιθώριο εσφαλμένης ταξινόμησης μεταξύ των σημείων αυξάνεται. Στην περίπτωση αυτή, επιταχύνεται η σύγκλιση του αλγορίθμου ταξινόμησης.

5.2 Αξιολόγηση Μοντέλου

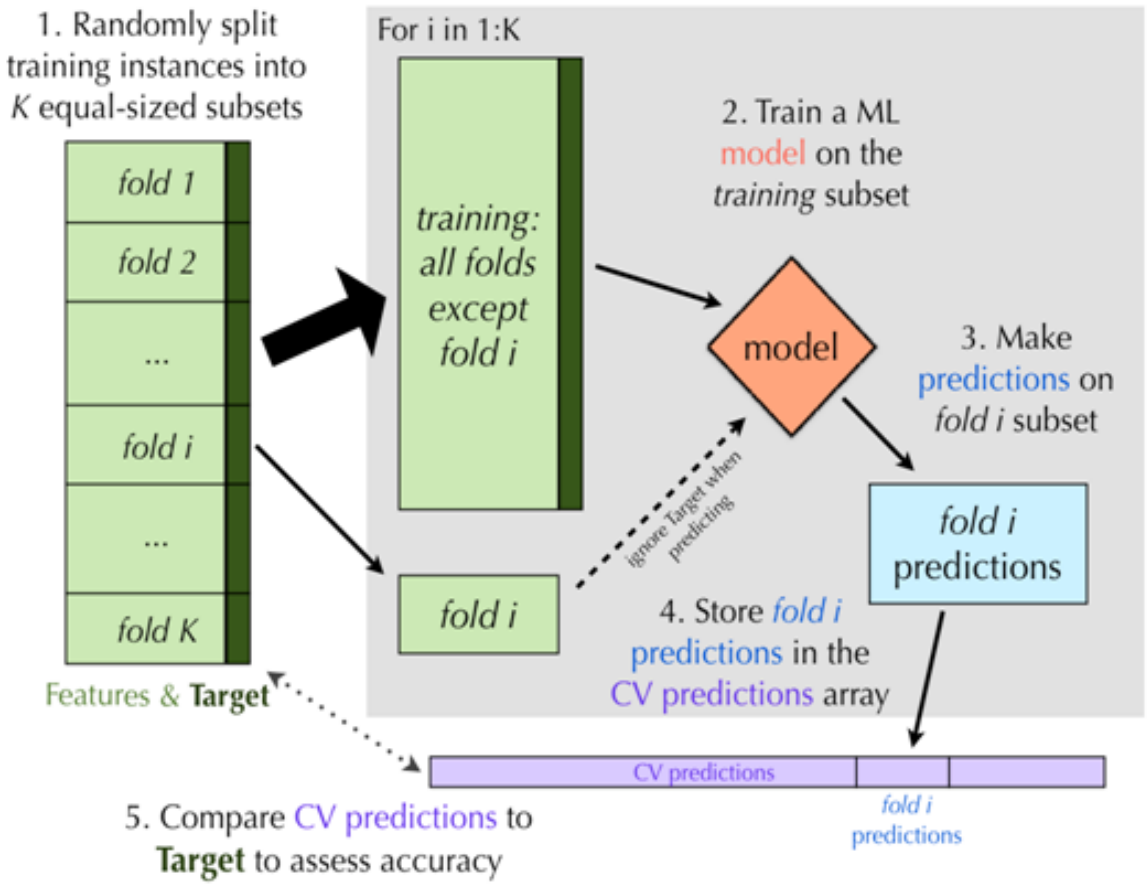
Στην ενότητα αυτή αναλύουμε τον τρόπο που εκπαιδεύουμε τα μοντέλα μας με τη μέθοδο του supervised learning και τα αξιολογούμε με βάση το accuracy τους. Τα μοντέλα μηχανικής μάθησης επιθυμούμε να είναι όσο το δυνατό πιο ακριβή στην πρόβλεψη τους όταν δοθούν σε αυτά νέα δεδομένα ως είσοδος. Τα Historical Data χωρίζονται σε δεδομένα εκπαίδευσης (training data) και δεδομένα ελέγχου (test data). Το training σύνολο θεωρούμε ότι περιέχει μια γνωστή τιμή ως έξοδο και το μοντέλο εκπαιδεύεται στα δεδομένα του ώστε να γενικευτεί και για άλλα δεδομένα στο μέλλον. Χρησιμοποιώντας τα test data, ή ένα υποσύνολό τους, ελέγχουμε την πρόβλεψη του μοντέλου ως προς το training set. Κατά τη διαδικασία αυτή, ο κύριος στόχος είναι να αποφευχθεί το φαινόμενο της υπερπροσαρμογής[3] (overfitting) και το φαινόμενο της υποπροσαρμογής[4] (underfitting).

Για την αξιολόγηση των μοντέλων μηχανικής μάθησης χρησιμοποιούμε το K-fold cross-validation (CV). Όπως φαίνεται στο διάγραμμα 5.2, κατά το K-fold cross-validation τα δείγματα διαχωρίζονται τυχαία (shuffle=True) σε K υποσύνολα. Από τα K αυτά υποσύνολα, ένα υποσύνολο θεωρείται ως δεδομένα επικύρωσης (validation data) για τον έλεγχο του μοντέλου και τα υπόλοιπα K-1 υποσύνολα χρησιμοποιούνται ως δεδομένα εκπαίδευσης. Η διαδικασία του cross-validation επαναλαμβάνεται K φορές, όπου κάθε ένα από τα K υποσύνολα μία μόνο φορά χρησιμοποιείται ως δεδομένα επικύρωσης. Έπειτα, υπολογίζουμε το μέσο όρο των K αποτελεσμάτων-προβλέψεων για να λάβουμε μία εκτίμηση. Η μέθοδος αυτή πλεονεκτεί στο γεγονός ότι όλα τα δείγματα χρησιμοποιούνται τόσο για την εκπαίδευση όσο και για την επικύρωση του μοντέλου, όπου κάθε δείγμα χρησιμοποιείται για επικύρωση μία μόνο φορά.

Με χρήση του GridsearchCV πραγματοποιούμε πειράματα για τη ρύθμιση την τιμή του K σε συνδυασμό με τη ρύθμιση των παραμέτρων της SVR εκτιμήτριας συνάρτησης. Τα βέλτιστα αποτελέσματα προκύπτουν για K=3 (3-fold cross-validation) στην περίπτωση του *min_success_downtime-limit* μοντέλου και K=5 (5-fold cross-validation) για τα μοντέλα *downtime* και *total_time*. Αφού ολοκληρωθούν οι K επαναλήψεις πλήρως, τότε για κάθε μοντέλο παίρνουμε το μέσο όρο των K

[3] Κατά το overfitting, το μοντέλο προσαρμόζεται υπερβολικά στα δεδομένα του συνόλου εκπαίδευσης και αδυνατεί να λειτουργήσει το ίδιο αποδοτικά κατά τη γενίκευση σε νέα δεδομένα, με τα οποία δεν έχει εκπαιδευτεί

[4] Κατά το underfitting, το λάθος κατά τη γενίκευση του μοντέλου σε νέα δεδομένα είναι πολύ μεγάλο και υπάρχουν σφάλματα ακόμα και κατά την εφαρμογή δεδομένων από το training set



Σχήμα 5.2: Διάγραμμα ροής του K-fold cross-validation

βαθμολογιών που έχει. Οι τελικές βαθμολογίες του accuracy κάθε μοντέλου παρουσιάζονται στον πίνακα 5.3.

Στον πίνακα 5.2 απεικονίζονται οι χρόνοι που απαιτούνται για την εκπαίδευση του κάθε μοντέλου όταν ως είσοδος δοθούν τα αποτελέσματα του BitmapTrace με παραμέτρους (iterations,period) = (10,1000). Σε ένα υπολογιστικό κέντρο, η συχνή επανεκπαίδευση των μοντέλων είναι επιθυμητή, αφού τα μοντέλα οφείλουν να ανανεώνονται ώστε να εφαρμόζονται αποδοτικά για μεγάλο πλήθος διαφορετικών workload των VMs. Το υπολογιστικό overhead λίγων λεπτών που απαιτείται για την εκπαίδευσή τους ανά ημέρες είναι αποδεκτό.

Model	***min_success_downtime-limit***	***downtime***	***total_time***
Training Time(msec)	384	1686	543

Πίνακας 5.2: Χρόνος εκπαίδευσης μοντέλου για (iterations,pediod) = (10,1000)

Για την αξιολόγηση της ακρίβειας πρόβλεψης (prediction accuracy) των SVR μοντέλων μας χρησιμοποιούνται οι συναρτήσεις r2_score [46] και mean_absolute_error [47] από το πακέτο sklearn.metrics [48] της Python και τα αποτελέσματα παρουσιάζονται στον πίνακα 5.3. Ο συντελεστής προσδιορισμού (coefficient of determination (CoD) ή R^2) δείχνει πόσο ακριβείς είναι οι προβλέψεις του μοντέλου ως προς την μετρική-στόχος. Η μέγιστη τιμή R^2 ισούται με 1 και υποδεικνύει τη βέλτιστη ακρίβεια πρόβλεψης του μοντέλου. Στην περίπτωση του *total_time* μοντέλου, η ύπαρξη μεγάλων τιμών του *downtime-limit* που οδηγεί σε total time σχεδόν ίσο με το downtime οφείλεται για τις χα-

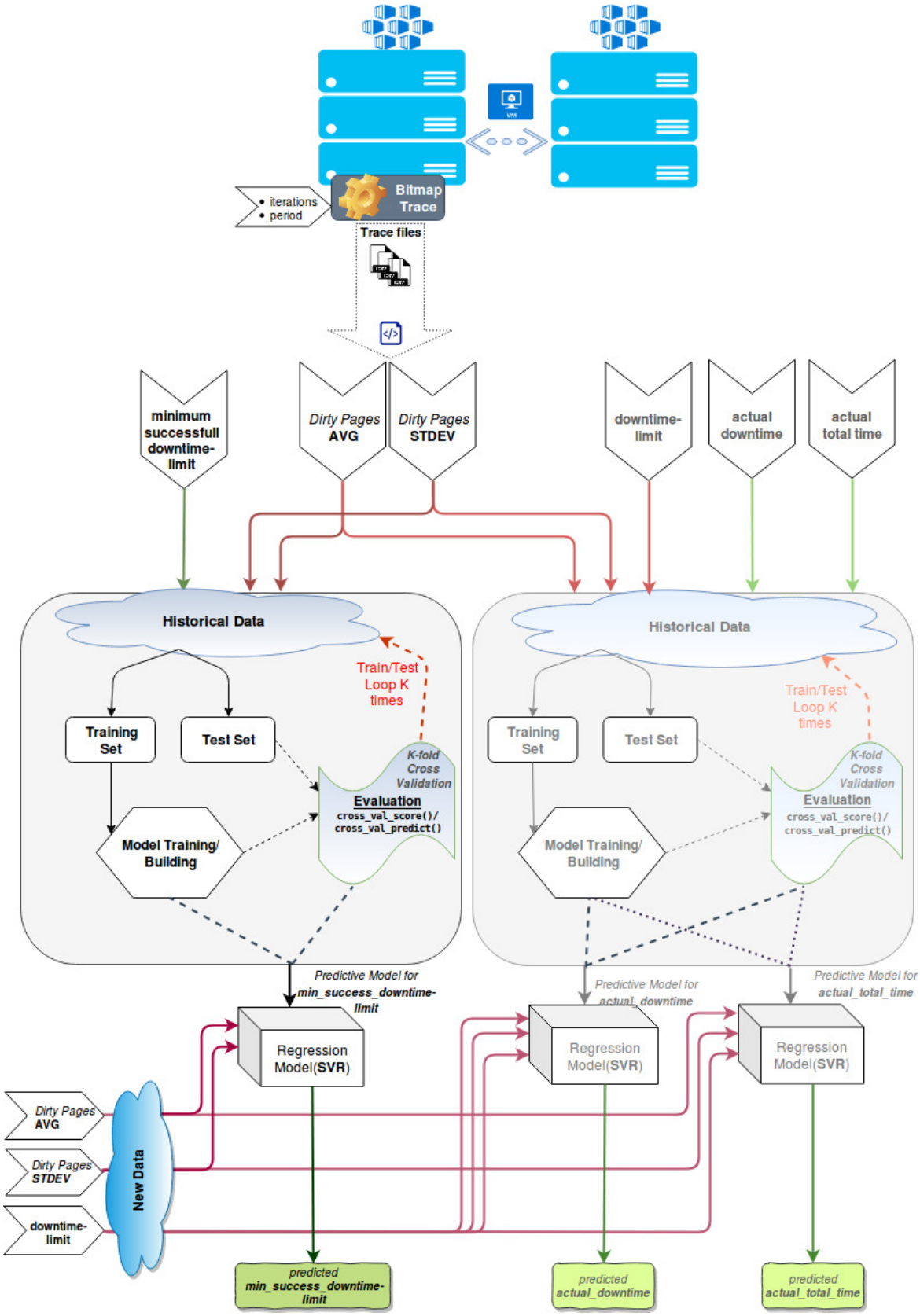
μηλές τιμές του R^2, όπως π.χ. 20% και 25%. Η περίπτωση αυτή αποδεικνύει την πιθανή μη-γραμμική συσχέτιση των δεδομένων εισόδου (*downtime-limit*, AVG, STDEV) του *total_time* μοντέλου με την μετρική-στόχος (total time). Το Μέσο Απόλυτο Σφάλμα (Mean Absolute Error (MAE)) αναπαριστά την απόκλιση της προβλεπόμενης τιμής από την πραγματική, σε απόλυτη τιμή της μετρικής (msec). Σε κάθε βήμα του cross-validation, η predicted τιμή του test set συγκρίνεται με την τιμή της μετρικής-στόχου του training set και υπολογίζεται ο μέσος όρος των MAE τιμών αυτών από τις K επαναλήψεις.

Οι χαμηλές τιμές του R^2, όπως 20% και 25% για το *total_time* μοντέλο εμφανίζονται λόγω των μεγάλων τιμών του *downtime-limit*

Model	**(iterations,period)**	**Model Accuracy(%)**	**Prediction Accuracy(%)**	**MAE(ms)**
min_success_downtime-limit	(10,1000)	88	89	271
	(10,1500)	87	88	222
downtime	(10,1000)	85	86	301
	(10,1500)	69	70	403
total_time	(10,1000)	26	25	1828
	(10,1500)	20	20	2036

Πίνακας 5.3: Accuracy και MAE των *min_success_downtime-limit*, *downtime* και *total_time* μοντέλων πρόβλεψης

Η διαδικασία που ακολουθούμε για την παραγωγή και αξιολόγηση των μοντέλων μας συνοψίζεται στο διάγραμμα 5.3.



Σχήμα 5.3: Διάγραμμα ροής εργασιών των μοντέλων πρόβλεψης

5.3 Πολιτικές Λειτουργίας για Επιλογή Υποψήφιων VMs για Live Migration

Με βάση τα χαρακτηριστικά του αποτυπώματος μνήμης του VM, τα οποία συλλέγονται μέσω του BitmapTrace μηχανισμού, και των υπαρχόντων SLA, ταξινομούμε τις εικονικές μηχανές που εκτελούνται στον source host ώστε να αποφασίσουμε το υποσύνολο αυτών (k από τις N) που θα μεταφερθούν στον destination host. Διαμορφώνουμε, λοιπόν, τις παρακάτω πολιτικές λειτουργίας με σκοπό την επιτυχή ολοκλήρωση του live migration των επιλεχθέντων VMs.

(i) Κάθε live migration πρέπει να ολοκληρώνεται εντός ενός συγκεκριμένου χρονικού διαστήματος. Για αυτό το λόγο, τα live migrations με κάποιο συγκεκριμένο *downtime-limit* που δεν συγκλίνουν μέσα σε 40 δευτερόλεπτα (timeout) ακυρώνονται.

(ii) Δεν επιθυμούμε την μη ολοκλήρωση ενός live migration. Μετά την καταγραφή του αποτυπώματος μνήμης των υποψήφιων VMs, με βάση τεχνικές πρόβλεψης προτείνεται για κάθε guest η ελάχιστη τιμή του *downtime-limit* που διασφαλίζει την επιτυχή ολοκλήρωση του migration εντός του χρονικού timeout.

(iii) Οι SLA περιορισμοί πρέπει να ικανοποιούνται κατά το μέγιστο δυνατό βαθμό. Ένα υπολογιστικό κέντρο οφείλει να διασφαλίζει τη συνεχή διαθεσιμότητα των υπηρεσιών. Αυτό σημαίνει ότι ο χρόνος μη λειτουργίας (downtime) της εικονικής μηχανής κατά το migration δεν πρέπει να ξεπερνά το *max_downtime* που είναι συμφωνημένο κατά το SLA.

(iv) Το αθροιστικό total time (aggregated total time) των live migration από το αρχικό φυσικό μηχάνημα σε ένα άλλο για τα k από τα N VMs πρέπει να είναι το ελάχιστο. Ο χρόνος αυτός ισούται με το χρονικό διάστημα από τη στιγμή που ξεκινάει η μεταφορά του πρώτου VM μέχρι και τη στιγμή που θα ολοκληρωθεί το migration του τελευταίου VM.

(v) Οι υποψήφιες εικονικές μηχανές ταξινομούνται με βάση το *Relative Downtime Error*. Η μετρική αυτή ορίζεται από τον παρακάτω τύπο:

$$\textit{Relative Downtime Error} = \frac{\textit{max\_downtime} - \overbrace{\textit{min\_success\_downtime-limit}}^{predicted}}{\textit{max\_downtime}}$$

Οι εικονικές μηχανές με τις υψηλές τιμές του *Relative Downtime Error* είναι οι "καλύτεροι" υποψήφιοι για migration. Αρνητικές τιμές της μετρικής αυτής υποδεικνύουν ότι το δοθέν *max_downtime*(βάσει SLA) δεν εξασφαλίζει την επιτυχή ολοκλήρωση του live migration με βάση την πρόσφατη συμπεριφορά του workload ως προς τη μνήμη.

Όταν το *Relative Downtime Error* λάβει μεγάλη τιμή, το VM αναμένεται να μετακινηθεί αρκετά γρήγορα στον host προορισμού, δηλαδή η τιμή total time να είναι χαμηλή. Στην περίπτωση της σειριακής μεταφοράς των k επιλεχθέντων VMs, πρώτα μεταφέρονται οι εικονικές μηχανές με τις χαμηλότερες *Relative Downtime Error* τιμές. Οι τιμές αυτές δείχνουν ότι το *max_downtime* αυτών των VMs είναι κοντά στην τιμή του *min_success_downtime-limit* που προέβλεψε το μοντέλο. Επειδή η συμπεριφορά των εφαρμογών αλλάζει δυναμικά από στιγμή

σε στιγμή, επιθυμούμε την επιτυχή μεταφορά αυτών των ”οριακών” VMs αμέσως μετά την ολοκλήρωση του profiiling ώστε να αποφευχθεί πιθανή αύξηση του dirty page rate που θα επηρεάσει το αποτέλεσμα του migration. Στην περίπτωση του παράλληλου migration, τα k επιλεχθέντα VMs μεταφέρονται ταυτόχρονα στον destination host οπότε η σειρά δεν επηρεάζει τη διαδικασία μεταφοράς.

Αν δύο ή περισσότερα VMs έχουν την ίδια *Relative Downtime Error* τιμή, τότε πρώτα επιλέγουμε το VM με την ελάχιστη AVG/STDEV τιμή. Σύμφωνα με την ανάλυση της ενότητας 5.1.1, ο λόγος αυτός καθορίζει τη σύγκλιση του pre-copy live migration αλγορίθμου.

(vi) Αν η *Relative Downtime Error* τιμή είναι θετική τότε η τιμή της παραμέτρου *downtime-limit* που ορίζεται για το migration των k επιλεχθέντων VMs είναι ίση με το *max_downtime*, ενώ αν η τιμή του *Relative Downtime Error* είναι αρνητική τότε στην παράμετρο *downtime-limit* θέτουμε τιμή ίση με το predicted *min_success_downtime-limit*. Με βάση τα κριτήρια ταξινόμησης που περιγράφονται παραπάνω, επιλέγουμε τις εικονικές μηχανές με το ελάχιστο penalty που προκύπτει αν χρειαστεί να παραβιάσουμε το μέγιστο επιτρεπτό dowtime που είναι συμφωνημένο βάσει SLA.

5.4 Αξιολόγηση της Επιλογής Υποψήφιων VMs για Live Migration

Για την αξιολόγηση του *min_success_downtime-limit* μοντέλου υλοποιούμε ένα VM live migration framework χρονοδρομολόγησης (scheduling) που αυτόματα ταξινομεί τα υποψήφια για migration VMs με βάση το αποτύπωμα μνήμης τους και δοθέντος του *max_downtime* που είναι γνωστό από το SLA. Επίσης, το framework εκτελεί το live migration των k από τα N VMs με βάση τις πολιτικές λειτουργίας που περιγράφονται στην ενότητα 5.3. Για την αξιολόγησή του εφαρμόζουμε τρία διαφορετικά σενάρια. Σε όλες τις περιπτώσεις θεωρούμε ότι 8 VMs συνυπάρχουν στον source host και επιθυμούμε να μεταφέρουμε 4 από αυτά. Χρησιμοποιούμε την τοπολογία που περιγράφεται στο κεφάλαιο 3.1 και η εκτέλεση κάθε VM (μεγέθους 1G) αντιστοιχίζεται σε ξεχωριστό πυρήνα του φυσικού μηχανήματος.

Βασική απόφαση ως πολιτική λειτουργίας που πρέπει να λάβουμε είναι η σειριακή ή παράλληλη μεταφορά των επιλεχθέντων VMs. Με το παράλληλο live migration, το διαθέσιμο φυσικό bandwidth μοιράζεται ανάμεσα στους μεταφερόμενους guests. Υπάρχει, λοιπόν, ο κίνδυνος της μη σύγκλισης του pre-copy live migration αλγορίθμου για καμία εικονική μηχανή. Επειδή η επιτυχή ολοκλήρωση του migration κάθε VM είναι πολύ σημαντική και δεδομένου του περιορισμένου εύρους ζώνης (1 Gbps) του δικτύου της τοπολογίας μας, χρησιμοποιούμε το σειριακό τρόπο μεταφοράς των εικονικών μηχανών. Το μέγιστο επιτρεπτό bandwidth (*max-bandwidth*) για κάθε live migration ορίζεται στα 800Mbps. Επιπλέον, για την παρούσα εργασία η σειρά μεταφοράς των επιλεχθέντων VMs δε λαμβάνεται υπόψη. Για παράδειγμα, η σειριακή μεταφορά των εικονικών μηχανών VM1, VM2, VM3, VM4 θεωρείται ίδια με τη μεταφορά των VM4, VM3, VM2, VM1. Παρ’ όλα αυτά, όπως αναφέρεται στο 5.3(v), οι εικονικές μηχανές με χαμηλή τιμή του *Relative Downtime Error*, δηλαδή το *max_downtime* είναι κοντά στο προβλέπομενο ελάχιστο *downtime-limit* για επιτυχές migration, μεταφέρονται πρώτα ώστε να αποφευχθεί πιθανή αύξηση του dirty page rate του workload τους.

Μιας και η σειρά μεταφοράς δε λαμβάνεται υπόψη, οι πιθανοί συνδυασμοί για την επιλογή k από τα N υποψήφια VMs δίνεται από τον τύπο C(N,k) = $\frac{n!}{k!(n-k)!}$. Στα σενάρια που υλοποιούμε N=8 και k=4, οπότε υπάρχουν C(N,k)=70 πιθανοί συνδυασμοί. Συγκρίνουμε, λοιπόν, τα αποτελέσματα του scheduling framework με την τυχαία επιλογή εικονικών μηχανών για migration, η οποία αποτελεί και τον απλούστερο τρόπο εξισορρόπησης φόρτου ενός υπερφορτωμένου host. Η μέθοδος αυτή, όμως, αυξάνει τις πιθανότητες για παραβίαση των εκάστοτε SLAs και ίσως οδηγήσει σε άσκοπη χρήση του εύρους ζώνης του δικτύου αν ο pre-copy live migration αλγόριθμος δεν συγκλίνει.

Στα πειράματά μας, κάθε σενάριο εκτελείται 10 φορές και υπολογίζουμε το μέσο όρο των aggregated total time τιμών. Σε όλα τα σενάρια, το profiling ενεργοποιείται ύστερα από 30 δευτερόλεπτα διάρκεια εκτέλεσης κάθε εφαρμογής και διαρκεί 10 δευτερόλεπτα ((*iterations,period*)=(10,1000)). Έπειτα, το migration ξεκινάει αυτόματα ακριβώς μετά το τέλος του profiling. Η λήψη της απόφασης για την επιλογή των υποψήφιων VMs για live migration βασίζεται στο *min_success_downtime-limit* μοντέλο. Επίσης, σε κάθε σενάριο παρουσιάζουμε το predicted aggregated total time που βασίζεται στο *total_time* μοντέλο πρόβλεψης. Αν και η πραγματική τιμή διαφέρει μερικά δευτερόλεπτα από την τιμή πρόβλεψης, ο cloud πάροχος μπορεί να αξιοποιήσει την τιμή πρόβλεψης σε περίπτωση που επιθυμεί μια εκτίμηση του συνολικού χρόνου του migration των k VMs προτού εκκινήσει την διαδικασία της μεταφοράς.

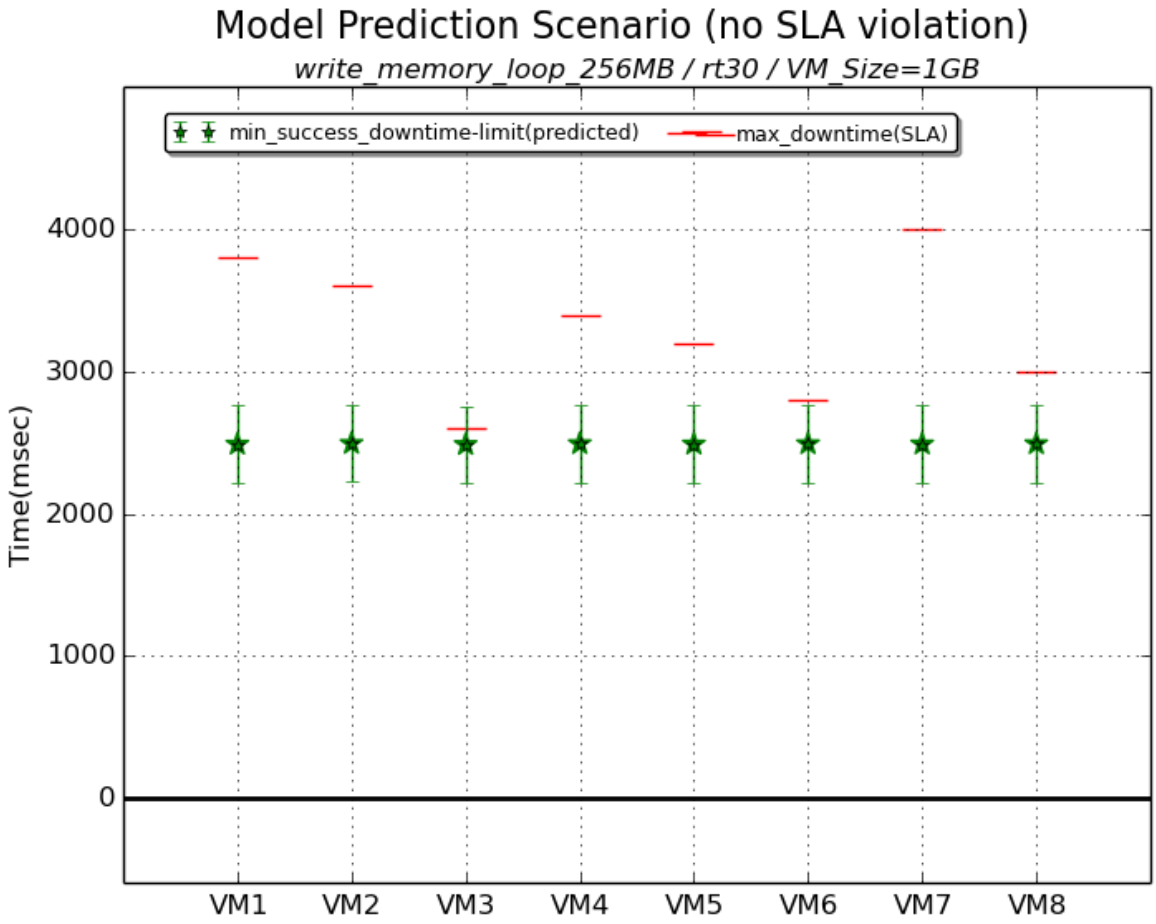
5.4.1 VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις

VM: Application	**max_downtime(msec) (SLA)**
VM1: *write_memory_loop_256MB*	3800
VM2: *write_memory_loop_256MB*	3600
VM3: *write_memory_loop_256MB*	2600
VM4: *write_memory_loop_256MB*	3400
VM5: *write_memory_loop_256MB*	3200
VM6: *write_memory_loop_256MB*	2800
VM7: *write_memory_loop_256MB*	4000
VM8: *write_memory_loop_256MB*	3000

Πίνακας 5.4: Live Migration σενάριο χρονοδρομολόγησης για VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις

Στο σενάριο αυτό, σε κάθε VM εκτελείται το *write_memory_loop_256MB* microbenchmark το οποίο σειριακά και κυκλικά γράφει (αλλάζει) συνεχώς σε 256MB μνήμης. Η μοναδική διαφοροποίηση ανάμεσα στις εικονικές μηχανές είναι η τιμή του *max_downtime*, όπως φαίνεται και στον πίνακα 5.4. Η τιμή του *min_success_downtime-limit* για τη συγκεκριμένη εφαρμογή είναι 2600msec όταν το migration ξεκινάει ύστερα από 60 ή 120 δευτερόλεπτα runtime της εφαρμογής. Οι τιμές, λοιπόν, που επιλέγουμε για το *max_downtime* είναι μεγαλύτερες ή ίσες της *min_success_downtime-limit*=2600msec τιμής. Στην περίπτωση της χρονοδρομολόγησης με βάση το μοντέλο πρόβλεψης, η predicted *min_success_downtime-limit* τιμή είναι περίπου 2490msec για κάθε VM. Οι τιμές αυτές μαζί με το περιθώριο σφάλματος (ίσο με την τιμή MAE του *min_success_downtime-limit* μοντέλου) απεικονίζονται στο διάγραμμα 5.4. Η επιλογή των VMs και η προτεινόμενη σειρά για migration με βάση το framework είναι: VM4, VM2, VM1, VM7. Είναι φανερό ότι αυτές οι εικονικές μηχανές είναι

οι καλύτεροι υποψήφιοι για migration με το ελάχιστο κόστος. Στον πίνακα 5.5 παρουσιάζουμε το aggregated total time σε σύγκριση με την τυχαία επιλογή από τα διαθέσιμα VMs, όπου ο χρόνος αυτός μειώνεται κατά 3,75% περίπου (~1 δευτερόλεπτο λιγότερο).



Σχήμα 5.4: *max_downtime* και *προβλεπόμενο min_success_downtime-limit* για VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις

Scheduling Scenario	Actual Aggregated Total Time	Predicted Aggregated Total Time
Model Prediction	29487	28840
Random	30636	-

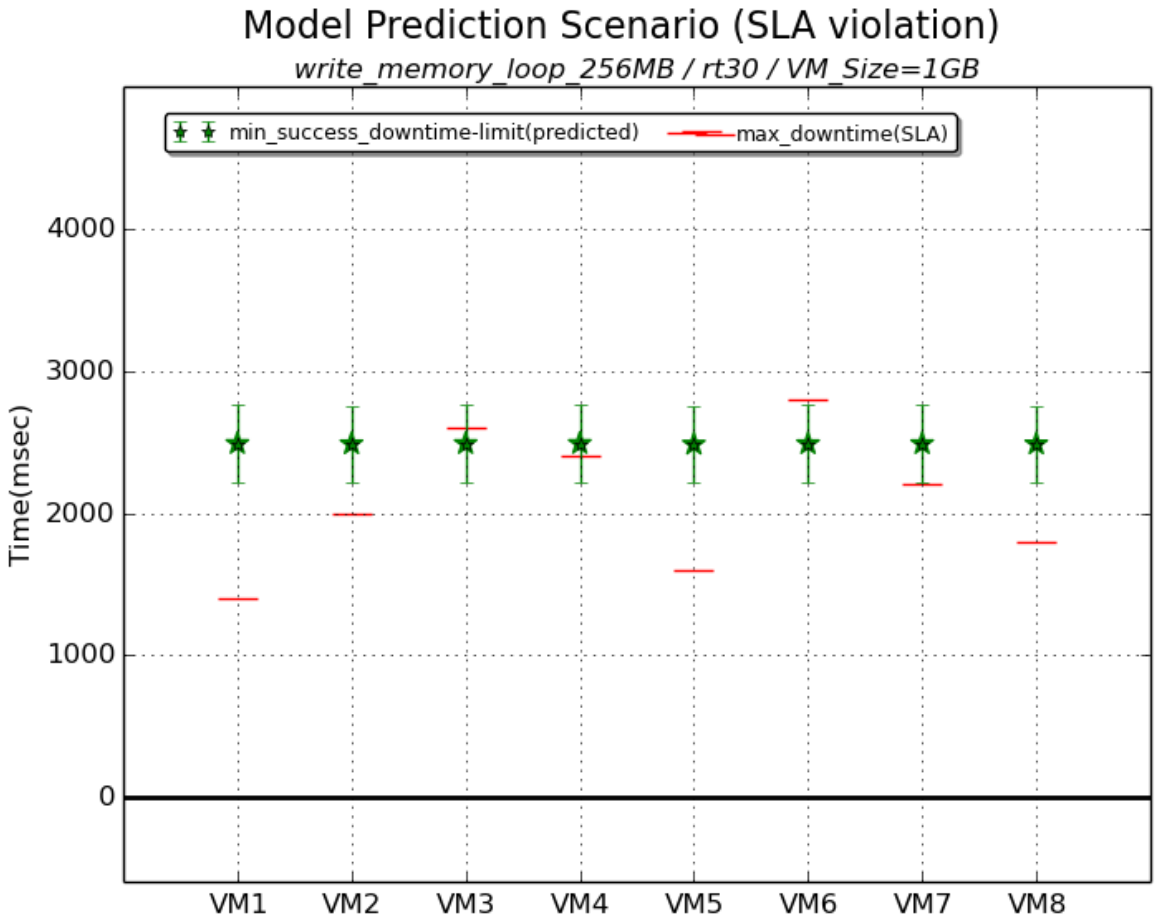
Πίνακας 5.5: Aggregated total time για VMs με ίδια εφαρμογή και χωρίς SLA παραβιάσεις

5.4.2 VMs με ίδια εφαρμογή και με SLA παραβιάσεις

VM: Application	**max_downtime(msec) (SLA)**
VM1: *write_memory_loop_256MB*	1400
VM2: *write_memory_loop_256MB*	2000
VM3: *write_memory_loop_256MB*	2600
VM4: *write_memory_loop_256MB*	2400
VM5: *write_memory_loop_256MB*	1600
VM6: *write_memory_loop_256MB*	2800
VM7: *write_memory_loop_256MB*	2200
VM8: *write_memory_loop_256MB*	1800

Πίνακας 5.6: Live Migration σενάριο χρονοδρομολόγησης για VMs με ίδια εφαρμογή και με SLA παραβιάσεις

Παρόμοια με το προηγούμενο σενάριο, κάθε VM εκτελεί το *write_memory_loop_256MB* micro-benchmark το οποίο σειριακά και κυκλικά γράφει (αλλάζει) συνεχώς σε 256MB μνήμης. Η τιμή του *min_success_downtime-limit* για τη συγκεκριμένη εφαρμογή είναι 2600msec όταν το migration ξεκινάει ύστερα από 60 ή 120 δευτερόλεπτα runtime της εφαρμογής. Η μοναδική διαφοροποίηση ανάμεσα στις εικονικές μηχανές είναι η τιμή του *max_downtime*, όπως στην περίπτωση αυτή έξι από τις οκτώ τιμές του *max_downtime* είναι μικρότερες του 2600msec, όπως φαίνεται και στον πίνακα 5.6. Με αυτό τον τρόπο, κάποια VMs αναμένουμε να παραβιάσουν κατά τη διάρκεια του migration το *max_downtime* του SLA. Με βάση το μοντέλο πρόβλεψης, η predicted *min_success_downtime-limit* τιμή είναι περίπου 2490msec για κάθε VM. Οι τιμές αυτές μαζί με το περιθώριο σφάλματος (ίσο με την τιμή MAE του *min_success_downtime-limit* μοντέλου) απεικονίζονται στο διάγραμμα 5.5. Επειδή επιθυμούμε να μετακινήσουμε 4 VMs, το framework επιλέγει τα δύο VMs με τιμή του *max_downtime* μεγαλύτερη ή ίση της τιμής του *min_success_downtime-limit* και τα άλλα δύο επιλεχθέντα VMs είναι εκείνα με το ελάχιστο κόστος παραβίασης του SLA. Επομένως, τα VM7, VM4, VM3, VM6 μεταφέρονται σειριακά στον destination host. Σε ένα σενάριο χρονοδρομολόγησης με τυχαία επιλογή, ίσως και οι τέσσερις επιλεχθέντες εικονικές μηχανές παραβιάζουν τους SLA περιορισμούς. Επιπλέον, στο σενάριο αυτό η επιλογή της τιμής της παραμέτρου *downtime-limit* πρέπει να γίνει τυχαία χωρίς ο διαχειριστής να έχει κάποια προηγούμενη γνώση για την εφαρμογή του κάθε VM.



Σχήμα 5.5: *max_downtime* και *προβλεπόμενο min_success_downtime-limit* για VMs με ίδια εφαρμογή και με SLA παραβιάσεις

5.4.3 VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις

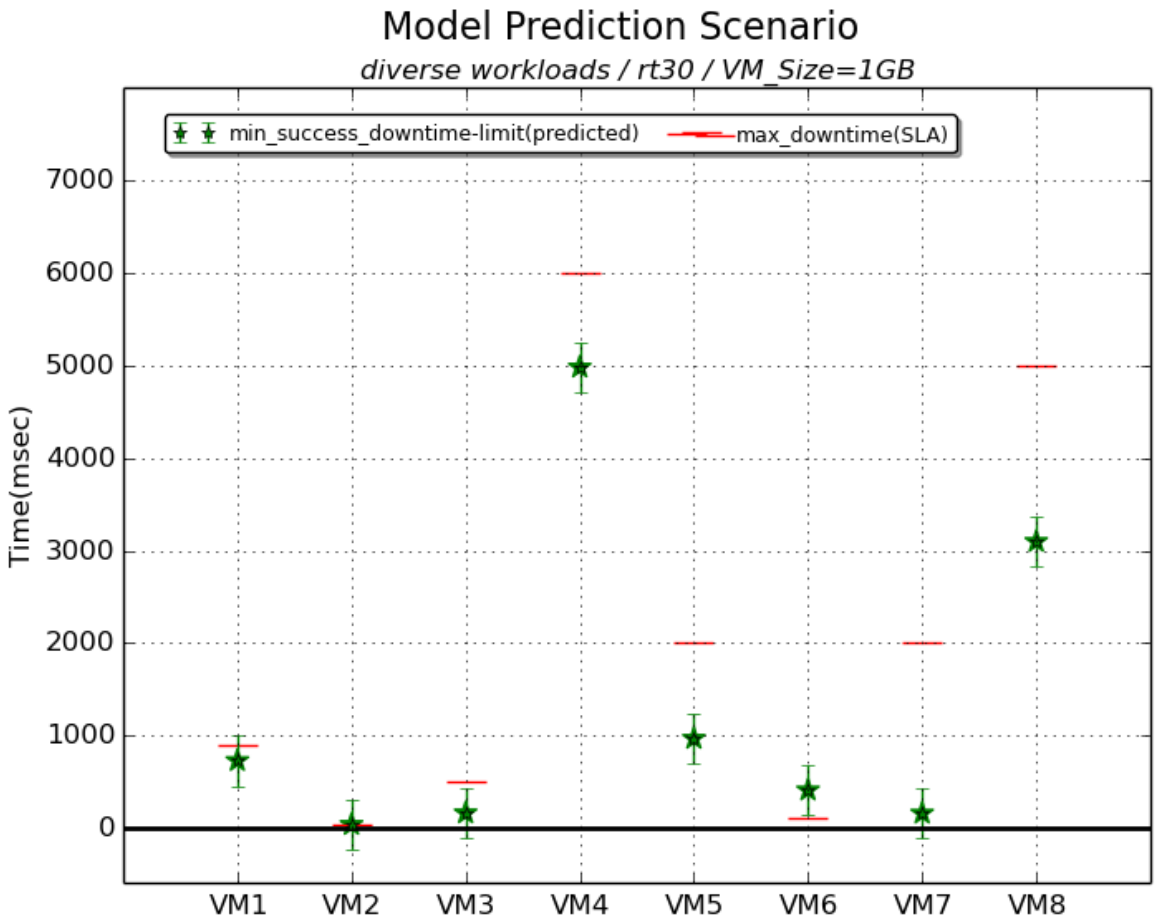
VM: Application	**max_downtime(msec) (SLA)**
VM1: *SPECjbb* (1 thread)	900
VM2: *write_file*	40
VM3: *450.soplex*	500
VM4: *write_memory_loop_512MB*	6000
VM5: *471.omnetpp*	2000
VM6: *436.cactusADM*	100
VM7: *473.astar*	2000
VM8: *459.GemsFDTD*	5000

Πίνακας 5.7: Live Migration σενάριο χρονοδρομολόγησης για VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις

Scheduling Scenario	Actual Aggregated Total Time	Predicted Aggregated Total Time
Model Prediction	23368	21513
Random	29440	-

Πίνακας 5.8: Aggregated total time για VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις

Τα migration σενάρια χρονοδρομολόγησης που περιγράφονται στις ενότητες 5.4.2, 5.4.1 δεν αναπαριστούν real-world παραδείγματα. Σε ένα υπολογιστικό κέντρο, κάθε VM εκτελεί διαφορετική εφαρμογή. Έτσι, προσομοιώνουμε ένα σενάριο όπου σε κάθε εικονική μηχανή τρέχει διαφορετικό workload, όπως φαίνεται στον πίνακα 5.7. Οι τιμές του *max_downtime* που επιλέγουμε για κάθε VM είναι μεγαλύτερες από την τιμή του *min_success_downtime-limit* που υπολογίζεται όταν η αντίστοιχη εφαρμογή εκτελείται για 60 ή 120 δευτερόλεπτα (Πίνακας 5.1) προτού εκκινήσει το migration. Με βάση το μοντέλο πρόβλεψης, οι predicted *min_success_downtime-limit* τιμές για runtime 30 δευτερόλεπτα μαζί με το περιθώριο σφάλματος (ίσο με την τιμή MAE του *min_success_downtime-limit* μοντέλου) απεικονίζονται στο διάγραμμα 5.6. Όπως είναι αναμενόμενο, οι τιμές αυτές πρόβλεψης διαφέρουν μεταξύ τους, αφού κάθε VM εκτελεί διαφορετικό workload. Με βάση την τιμή *Relative Downtime Error*, οι καλύτεροι υποψήφιοι για migration και η προτεινόμενη σειρά μεταφοράς είναι: VM8, VM5, VM3, VM7. Όπως φαίνεται στον πίνακα 5.8, το κόστος του migration μειώνεται αισθητά με την καθοδήγηση του μοντέλου μας. Συγκεκριμένα, το aggregated total time μειώνεται κατά 20,63% (~6 δευτερόλεπτα λιγότερο) σε σχέση με την τυχαία επιλογή από τα υποψήφια VMs. Αυτό ισοδυναμεί με σημαντική μείωση της κίνησης στο δίκτυο του data center καθώς και χαμηλότερη κατανάλωση ενέργειας.



Σχήμα 5.6: *max_downtime* και *προβλεπόμενο min_success_downtime-limit* για VMs με διαφορετική εφαρμογή και χωρίς SLA παραβιάσεις

Κεφάλαιο 6

Σχετική Έρευνα

Στην ενότητα αυτή σχολιάζουμε τις υπάρχουσες εργασίες που έχουν πραγματοποιηθεί γύρω από την τεχνολογία του live migration με έμφαση στην πρόβλεψη των παραγόντων που επηρεάζουν τη διαδικασία του migration καθώς και των τεχνικών πρόβλεψης για μεταφορά πολλαπλών εικονικών μηχανών. Οι Zhang et al. [49] παρουσιάζουν μια συγκεντρωτική έρευνα για τις υπάρχουσες τεχνικές του migration και τους μηχανισμούς βελτιστοποίησης που έχουν προταθεί. Συγκεκριμένα, εστιάζουν σε τρεις προκλήσεις του migration: migration δεδομένων μνήμης, migration δεδομένων αποθηκευτικού χώρου και αδιάλειπτη συνδεσιμότητα δικτύου. Επίσης συνοψίζουν τις έρευνες που αναφέρονται στη σχέση της απόδοσης του migration και των παραγόντων που το επηρεάζουν.

Το live migration παρουσιάστηκε για πρώτη φορά από τους Clark et al. [15] με στόχο την βελτίωση του pre-copy live migration αλγορίθμου μέσω της καταγραφής των dirty σελίδων μνήμης που αλλάζουν κατάσταση συχνά, δηλαδή το WWS. Εκμεταλλευόμενοι τη δυνατότητα της παραεικονικοποίησης του Xen για την καταγραφή του WWS κάθε διεργασίας στο migrated VM, προτείνουν μια λύση για μείωση του πλήθους των dirty σελίδων που στέλνονται στον destination host κατά την επαναληπτική φάση. Επίσης, προτείνουν ένα μηχανισμό για δυναμική ρύθμιση του migration bandwidth ώστε να ικανοποιήσουν τις συνθήκες σύγκλισης σε συνδυασμό με την αποδοτική χρήση του network bandwidth. Οι Zhang et al. [50] σχεδιάζουν ένα σύστημα παρακολούθησης των εικονικών μηχανών που ονομάζεται MigVisor. Ο μηχανισμός αυτός πραγματοποιεί προβλέψεις με βάση τον Pattern-Collector (PC) σε πρώτο στάδιο. Αν η πρόβλεψη του PC επιστρέψει αρνητικά αποτελέσματα, τότε χρησιμοποιείται το Dry Run (DR) για την πρόβλεψη του αποτελέσματος του VM live migration με χρήση του τεχνική συμπίεσης μνήμης. Τα modules αυτά ενσωματώνονται στο QEMU/KVM και η πληροφορία που επιστρέφουν για το μοτίβο προσβάσεων του VM στη μνήμη χρησιμοποιείται από ένα μοντέλο πρόβλεψης (working-set pattern (WSP) prediction model). Παρόμοια με την έρευνα του Clark, το μοντέλο αυτό προβλέπει το ελάχιστο bandwidth που απαιτείται για σύγκλιση του pre-copy live migration αλγορίθμου. Παρόμοια με το PC module, στη δική μας εργασία σχεδιάζουμε και υλοποιούμε για το QEMU/KVM το BitmapTrace μηχανισμό που περιοδικά καλεί τη συνάρτηση *KVM_GET_DIRTY_LOG* για την παρακολούθηση της κατάστασης των σελίδων μνήμης του VM όπως καταγράφεται στο kernel dirty bitmap. Σε αντίθεση με τους Zhang et al. , εμείς διατηρούμε σταθερή την τιμή του μέγιστου επιτρεπτού bandwidth ρυθμίζοντας μόνο την τιμή της παραμέτρου *downtime-limit* και τηρούμε κατά το μέγιστο δυνατό το *max_downtime* του SLA. Επιπλέον, ο μηχανισμός μας ενσωματώνεται σε ένα live migration framework στοχεύοντας στη λήψη απόφασης για live migration ανάμεσα σε ένα σύνολο από διαθέσιμα VMs. Με αυτό τον τρόπο επιλέγονται οι εικονικές μηχανές για τις οποίες διασφαλίζεται ότι ο αλγόριθμος

του pre-copy live migration θα συγκλίνει, ενώ κάποια υποψήφια VMs με *max_downtime* που δεν καλύπτεται από το υψηλό dirty page rate των εφαρμογών τους θα παραμείνουν στον source host.

Αρκετά μοντέλα πρόβλεψης έχουν αναπτυχθεί με στόχο την πρόβλεψη των μετρικών απόδοσης του migration. Οι Liu et al. [51] παρουσιάζουν ένα μοντέλο που προβλέπει το χρόνο για την ολοκλήρωση του επαναληπτικού pre-copy live migration αλγορίθμου με βάση το page transfer rate και το dirty page rate. Η πληροφορία για τις dirty σελίδες λαμβάνεται από τον Xen hypervisor. Η έρευνα περιλαμβάνει το migration πολλών διαφορετικών εφαρμογών όπου αξιολογούνται μετρικές όπως η καθυστέρηση του migration, το downtime, κίνηση στο δίκτυο και κατανάλωση ενέργειας. Όμως, δεν ασχολούνται με την αξιολόγηση μετρικών του QoS, π.χ. χρόνοι απόκρισης και επίπεδα υπηρεσιών. Οι Akoush et al. [52] αναλύουν τις σχέσεις μεταξύ των σημαντικών παραμέτρων του migration με έμφαση το Xen και επισημαίνουν τη μεγάλη εξάρτηση της απόδοσης του migration με το running workload. Επιπρόσθετα, σχεδιάζουν δύο μοντέλα προσομοίωσης για την πρόβλεψη του pre-copy migration μοτίβου εγγραφών στη μνήμη, τα AVG (average page dirty rate) και HIST (history based page dirty). Το AVG μοντέλο χρησιμοποιείται για την πρόβλεψη του migration εφαρμογών με σταθερό dirty page rate, ενώ το HIST μοντέλο χρησιμοποιείται για εικονικές μηχανές που παρουσιάζουν παρόμοια συμπεριφορά μεταξύ διαφορετικών στιγμών εκκίνησης του migration. Το τελευταίο μοντέλο χρησιμοποιείται, επίσης, από τους Patel et al. [53] σε ένα framework όπου γίνονται προβλέψεις με βάση την ανάλυση παλαιότερων δεδομένων του migration. Προτείνουν δύο regression μοντέλα χρονολογικών σειρών, ARIMA (autoregressive integrated moving average) και SVR (support vector regression). Σε αντίθεση με τις προαναφερθείσες εργασίες, το μοντέλο μας δέχεται ως στοιχεία εισόδου τα AVG και STDEV των dirty σελίδων που προκύπτουν μετά από παρακολούθηση του VM πριν την εκκίνηση του migration. Αποφεύγουμε την χρήση ιστορικών δεδομένων για το μοντέλο μας, αφού ο pre-copy live migration αλγόριθμος εξαρτάται σε μεγάλο βαθμό από τη δυναμική συμπεριφορά του workload στη μνήμη και το μοτίβο με το οποίο πραγματοποιεί εγγραφές κατά τη διάρκεια του live migration. Οι Nathan et al. [54] αναλύουν σε βάθος δώδεκα υπάρχοντα μοντέλα για την pre-copy τεχνική και προτείνουν ένα νέο μοντέλο για pre-copy live migration με τη χρήση του page-skipping optimization. Καταγράφουν, λοιπόν, τις σελίδες που θα μεταφερθούν σε κάθε iteration καθώς και τις σελίδες που θα γίνουν skip. Τα πειράματα με όλα τα παραπάνω μοντέλα γίνονται σε Xen λογισμικό ελέγχου και υπολογίζουν το downtime, το total time και το συνολικό αριθμό των σελίδων που θα μεταφερθούν. Η διαφορά με το μοντέλο της δικής μας εργασίας είναι η μετρική-στόχος, αφού εμείς προβλέπουμε την ελάχιστη τιμή του *downtime-limit* που διασφαλίζει την επιτυχή ολοκλήρωση του QEMU/KVM pre-copy live migration χωρίς χρήση βελτιστοποιήσεων.

Το migration πολλαπλών εικονικών μηχανών είναι αντικείμενο πολλαπλών εργασιών που στοχεύουν στην αποδοτικότερη χρησιμοποίηση των πόρων του υπολογιστικού κέντρου. Οι Lu et al. [1] παρουσιάζουν έναν Generic SLA Manager όπου με χρήση του OpenStack λαμβάνονται αποφάσεις για την διαχείριση του VM live migration και στρατηγικές τοποθέτησης των VMs όταν υπάρχουν πολυεπίπεδα SLAs. Οι Ye et al. [55] προτείνουν τη δέσμευση των resources (CPU cycles και memory space) για το migration και τρία μοντέλα βελτιστοποίησης, optimization στον source host, παράλληλη μεταφορά πολλαπλών εικονικών μηχανών και migration στρατηγικές όπου το workload θεωρείται γνωστό. Παρόμοια με τις παρατηρήσεις της δικής μας έρευνας, αναφέρουν ότι η παράλληλη μεταφορά των VMs είναι προτιμότερη από τη σειριακή όταν υπάρχουν αρκετοί

πόροι, αλλιώς τα αποτελέσματα δεν είναι ικανοποιητικά.

Τέλος, οι Jo et al. [56] προτείνουν ένα live migration framework όπου το μοντέλο πρόβλεψης στοχεύει στη μείωση των παραβιάσεων του SLA και στην τήρηση των πολιτικών λειτουργίας που θέτει ο διαχειριστής. Το προτεινόμενο μοντέλο μηχανικής μάθησης προβλέπει βασικά χαρακτηριστικά του migration, όπως total time, downtime, πλήθος μεταφερόμενων δεδομένων, μείωση της απόδοσης του VM και ποσοστά χρήσης της CPU και της μνήμης στα φυσικά μηχανήματα, για διάφορες τεχνικές του live migration (pre-copy, post-copy, CPU throttling, delta compression και data compression). Σε σύγκριση με το μεγάλο πλήθος στοιχείων που δέχεται ως είσοδο το μοντέλο των Jo et al., το μοντέλο που προτείνουμε στην παρούσα εργασία χρησιμοποιεί ως είσοδο μόνο τα AVG και STDEV των dirty σελίδων και προβλέπει την τιμή του ελάχιστου *downtime-limit* που οδηγεί σε σύγκλιση του vanilla pre-copy αλγορίθμου. Επίσης, το framework που υλοποιούμε επιλέγει αυτόματα για migration τις ”καλύτερες” εικονικές μηχανές ανάμεσα σε ένα σύνολο υποψήφιων VMs που εκτελούνται στο ίδιο φυσικό μηχάνημα. Αντίθετα, οι Jo et al. στοχεύουν στην αυτόματη επιλογή της κατάλληλης live migration τεχνικής για τη μεταφορά ενός VM σε κάποιον άλλον host.

Κεφάλαιο 7

Επίλογος και Μελλοντικές Επεκτάσεις

Στην παρούσα διπλωματική εργασία προτείνουμε ένα VM live migration framework που βασίζεται σε τεχνικές πρόβλεψης με σκοπό την βέλτιστη επιλογή για migration ανάμεσα σε ένα σύνολο εικονικών μηχανών που συνυπάρχουν στο ίδιο φυσικό μηχάνημα. Με βάση το αποτύπωμα μνήμης του workload κάθε VM και την τιμή του *max_downtime* που πρέπει να τηρήσουμε βάσει SLA, επιλέγουμε δυναμικά τις εικονικές μηχανές για τις οποίες το aggregated total time του migration είναι το ελάχιστο δυνατό. Για την καταγραφή ενός στιγμιότυπου της μνήμης με τις dirty σελίδες του VM για ένα συγκεκριμένο profiling χρονικό διάστημα, υλοποιούμε το BitmapTrace μηχανισμό που ενσωματώνεται στον κώδικα του QEMU/KVM migration. Το module αυτό εκτελείται πριν την εκκίνηση του migration για ένα συγκεκριμένο αριθμό επαναλήψεων, όπου σε κάθε iteration καλείται η *KVM_GET_DIRTY_LOG* ioctl() κλήση συστήματος και με βάση το kernel dirty bitmap καταγράφουμε τις νέες dirty σελίδες ανά iteration. Ο μέσος όρος (AVG) και η τυπική απόκλιση (STDEV) των δειγμάτων δίνονται ως είσοδος σε ένα supervised learning μοντέλο το οποίο προβλέπει την ελάχιστη τιμή του *downtime-limit* που διασφαλίζει την επιτυχή σύγκλιση του pre-copy live migration αλγορίθμου. Συγκεκριμένα, το live migration εικονικών εφαρμογών που εκτελούν memory-intensive εφαρμογές εμπεριέχει ρίσκο όσο αναφορά την επιλογή της *downtime-limit* τιμής που πρέπει να ρυθμιστεί κατά το migration. Με χρήση, λοιπόν, του μοντέλου πρόβλεψης αντιμετωπίζουμε το πρόβλημα του live migration με τιμές του *downtime-limit* που δεν οδηγούν σε σύγκλιση της pre-copy τεχνικής ενώ παράλληλα ικανοποιούμε τις πολιτικές λειτουργίας ενός υπολογιστικού κέντρου. Επίσης, αποφεύγεται η άσκοπη χρήση του δικτύου για VM live migration σε συνδυασμό με την μείωση της απόδοσης των εμπλεκόμενων συστημάτων από τη δέσμευση πόρων.

Τέλος, παρατίθενται μερικές λειτουργίες με τις οποίες ο BitmapTrace μηχανισμός και το προτεινόμενο live migration framework μπορούν να επεκταθούν. Οι μελλοντικές κατευθύνσεις που μπορούμε να ακολουθήσουμε είναι οι εξής:

- **Υποστήριξη περισσότερων live migration τεχνικών**: Η υπάρχουσα υλοποίηση του BitmapTrace μηχανισμού υποστηρίζει μόνο τον pre-copy live migration αλγόριθμο. Όπως αναφέρεται και στο κεφάλαιο 2, το QEMU/KVM υποστηρίζει και την post-copy τεχνική καθώς και μερικές βελτιστοποιήσεις, όπως auto-converge (CPU throttling) και XBZRLE (delta-compression) που μπορούν να ενσωματωθούν στο module μας.

- **Επέκταση του συνόλου δεδομένων με περισσότερες *max-bandwidth VM_Size* τιμές**: Τα χαρακτηριστικά εισόδου (AVG και STDEV των dirty σελίδων) των μοντέλων μας υπολογίζονται όταν το μέγιστο διαθέσιμο bandwidth (*max-bandwidth*) είναι 800Mbps και το μέγεθος μνήμης του VM είναι 1G. Προκειμένου να μην περιοριζόμαστε μόνο σε αυτές τις τιμές, σκοπός

μας είναι η παρακολούθηση εικονικών μηχανών με διαφορετικά μεγέθη μνήμης και υπολογισμός του *min_success_downtime-limit* για μεγαλύτερο εύρος τιμών του *max-bandwidth*. Για κάθε (*max-bandwidth*, *VM_Size*) ζεύγος τιμών, τα μοντέλα χρειάζεται να επανεκπαιδευτούν για την ακριβή πρόβλεψη του *min_success_downtime-limit*.

- **Βελτίωση του accuracy του *total_time* μοντέλου και πρόβλεψη επιπρόσθετων μετρικών-στόχων**: Η ακριβή πρόβλεψη των migration χρόνων (downtime και total-time) είναι δευτερεύον στόχος της έρευνάς μας. Για αυτό το λόγο δεν ερευνήθηκε περαιτέρω το prediction accuracy του *total_time* μοντέλου. Σκοπεύουμε, λοιπόν, να εκτελέσουμε μεγαλύτερο αριθμό πειραμάτων και να δοκιμάσουμε άλλα regression μοντέλα με σκοπό την βελτίωση των βαθμολογιών πρόβλεψης. Επίσης, τα υπάρχοντα μοντέλα μπορούν να επεκταθούν ώστε να προβλέπονται επιπρόσθετες μετρικές-στόχοι, όπως για παράδειγμα η ποσότητα των μεταφερόμενων δεδομένων στο δίκτυο.

- **Υποστήριξη παράλληλου VM live migration**: Η τοπολογία που χρησιμοποιήθηκε για την αξιολόγηση του framework περιορίζεται από τη χωρητικότητα του network bandwidth αφού το διαθέσιμο φυσικό εύρος ζώνης του δικτύου είναι 1Gbps. Για αυτό το λόγο, κατά τη διάρκεια του παράλληλου migration το bandwidth μοιράζεται ανάμεσα στα επιλεχθέντα VMs, αφού δεν μπορεί να υποστηριχθεί η μέγιστη τιμή του *max-bandwidth*=800Mbps που θέτουμε κατά το migration. Οι πιθανότητες, λοιπόν, για μη σύγκλιση κανενός pre-copy live migration αλγορίθμου αυξάνονται. Μελλοντική πρόθεσή μας είναι η αξιολόγηση του framework σε high throughput δίκτυα όπως 10Gbps ή InfiniBand, τα οποία χρησιμοποιούνται σε προηγμένα υπολογιστικά κέντρα.

- **Πρόβλεψη για επιλογή του destination host**: Η υπάρχουσα υλοποίηση του live migration framework υποθέτει ότι στον destination host δεν εκτελείται κανένα VM. Σε ένα πραγματικό περιβάλλον υπολογιστικού νέφους, οι πιθανοί προορισμοί ενός VM είναι πολλοί και το framework πρέπει να επιλέξει, με βάση τη διαθεσιμότητα των πόρων, το καταλληλότερο φυσικό μηχάνημα στο οποίο το VM θα μεταφερθεί.

Συντομογραφίες

AVG	Average
CLI	Command-Line Interface
CV	Cross-Validation
HMP	Human Monitor Interface
I/O(or IO)	Input/Output
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
MAE	Mean Absolute Error
MPKI	Misses Per Kilo Instructions
OS	Operating System
PaaS	Platform as a Service
PC	Performance Counter
PMU	Performance Monitoring Unit
QEMU	Quick Emulator
QMP	QEMU Machine Protocol
SaaS	Software as a Service
SLA	Service-Level Agreement
STDEV	Standard Deviation
VM	Virtual Machine
VMM	Virtual Machine Monitor
WWS	Writable Working Set
XBZRLE	Xor Binary Zero Run-Length-Encoding

Γλωσσάριο

Ελληνικός όρος	Αγγλικός όρος
αποτύπωμα μνήμης	memory footprint
”βρώμικες” σελίδες/σελίδες που έχουν αλλάξει περιεχόμενο	dirty pages
δεδομένα εκπαίδευσης	training data
δεδομένα ελέγχου	test data
δεδομένα επικύρωσης	validation data
διακομιστής/εξυπηρετητής	server
διαχειριστής πόρων	resource manager
Είσοδος/Έξοδος	I/O
εύρος ζώνης δεδομένων	data bandwidth
εφαρμογή με έντονη δραστηριότητα μνήμης	memory-intensive application
ζωντανή μεταφορά	live migration
κατώφλι	threshold
λογισμικό ελέγχου/επόπτης	hypervisor
Λογισμικό ως Υπηρεσία”	SaaS
Μέσο Απόλυτο Σφάλμα	Mean Absolute Error (MAE)
μετρική-στόχος	target metric
μοντέλο επιβλεπόμενης μάθησης	supervised learning model
Πλατφόρμα ως Υπηρεσία	PaaS
ποιότητα υπηρεσιών	Quality of Service (QoS)
συμφωνία σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου	Service Level Agreement (SLA)
συντελεστής προσδιορισμού	coefficient of determination (CoD) ή R^2)
σφάλμα σελίδας	page fault
υπερπροσαρμογή	overfitting
Υποδομή ως Υπηρεσία	IaaS
υπολογιστικό κέντρο	data center
υπολογιστικό νέφος	cloud computing
φιλοξενούμενος/εικονικό σύστημα	guest
φυσικό μηχάνημα	host
χρονοδρομολογητής εργασιών	job scheduler
χρόνος μη λειτουργίας	downtime

Βιβλιογραφία

[1] K. Lu, R. Yahyapour, P. Wieder, C. Kotsokalis, E. Yaqub, and A. I. Jehangiri. QoS-Aware VM Placement in Multi-domain Service Level Agreements Scenarios. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 661–668, June 2013.

[2] Stefan Frey, Christoph Reich, and Claudia Lüthje. Key Performance Indicators for Cloud Computing SLAs. 2013.

[3] Peter M. Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, Gaithersburg, MD, United States, 2011.

[4] Amazon, Amazon Elastic Compute Cloud (Amazon EC2). https://aws.amazon.com/ec2/. [Online, accessed April 2019].

[5] Google, Google App Engine. https://cloud.google.com/. [Online, accessed April 2019].

[6] Microsoft, Microsoft Azure. https://azure.microsoft.com/. [Online, accessed April 2019].

[7] QEMU Wiki. https://wiki.qemu.org/Main_Page. [Online, accessed April 2019].

[8] Wikipedia: Tiny Code Generator(TCG). https://en.wikipedia.org/wiki/QEMU#Tiny_Code_Generator. [Online, accessed April 2019].

[9] QEMU Machine Protocol(QMP) Wiki. https://wiki.qemu.org/Documentation/QMP. [Online, accessed April 2019].

[10] QEMU Monitor. https://en.wikibooks.org/wiki/QEMU/Monitor. [Online, accessed April 2019].

[11] Avi Kivity Qumranet, Yaniv Kamay Qumranet, Dor Laor Qumranet, Uri Lublin Qumranet, and Anthony Liguori. KVM: The Linux virtual machine monitor. *Proceedings Linux Symposium*, 15, 01 2007.

[12] XEN Wiki. https://wiki.xen.org/wiki/Xen_Project_Software_Overview. [Online, accessed April 2019].

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. XEN and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[14] Wikipedia: Network File System(NFS). https://en.wikipedia.org/wiki/Network_File_System. [Online, accessed April 2019].

[15] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, page 273–286. USENIX Association, 2005.

[16] Petter Svärd, Benoit Hudzia, Steve Walsh, Johan Tordsson, and Erik Elmroth. Principles and Performance Characteristics of Algorithms for Live VM Migration. *SIGOPS Oper. Syst. Rev.*, 49(1):142–155, January 2015.

[17] QEMU. https://www.qemu.org/. [Online, accessed April 2019].

[18] XEN Project. https://www.xenproject.org/. [Online, accessed April 2019].

[19] VMware. https://www.vmware.com/. [Online, accessed April 2019].

[20] Jeffrey M. Galloway, Gabriel Loewen, and Susan V. Vrbsky. Performance Metrics of Virtual Machine Live Migration. *2015 IEEE 8th International Conference on Cloud Computing*, pages 637–644, 2015.

[21] Red Hat: KVM Live Migration. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/virtualization/chap-virtualization-kvm_live_migration. [Online, accessed April 2019].

[22] QEMU Wiki: Migration with shared storage. https://wiki.qemu.org/Documentation/Migration_with_shared_storage. [Online, accessed April 2019].

[23] Red Hat Blog: Live Migrating QEMU-KVM Virtual Machines. https://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines/. [Online, accessed April 2019].

[24] QEMU 2.10.1 Migration Documentation. https://git.qemu.org/?p=qemu.git;a=blob;f=docs/devel/migration.txt;h=1b940a829b14504f28de043426eff50843dff35f;hb=7851197b812b383ae1208c5d86391c5179c8209d. [Online, accessed April 2019].

[25] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. volume 46, pages 111–120, 07 2011.

[26] Pooja and Asmita Pandey. Impact of memory intensive applications on performance of cloud virtual machine. pages 1–6, 03 2014.

[27] Xinkui Zhao, Jianwei Yin, Zuoning Chen, and Sheng He. Workload Classification Model for Specializing Virtual Machine Operating System. pages 343–350, 06 2013.

[28] Lars Cromley. Introduction to High Performance Computing (HPC). https://www.2ndwatch.com/blog/introduction-to-high-performance-computing-hpc/, July 2017. [Online, accessed April 2019].

[29] L. Sliwko and Vladimir Getov. Transfer Cost of Virtual Machine Live Migration in Cloud Systems. Technical report, 15 New Cavendish St., London W1W 6UW, U.K., 2017.

[30] Osama Alrajeh, Matt Forshaw, and Nigel Thomas. Performance of Virtual Machine Live Migration with Various Workloads. September 2016.

[31] Jianxin Li, Jieyu Zhao, Yi Li, Lei Cui, Bo Li, Lu Liu, and John Panneerselvam. iMIG: Toward an Adaptive Live Migration Method for KVM Virtual Machines. *Comput. J.*, 58:1227–1242, 2015.

[32] Daniel Berrange. Analysis of techniques for ensuring migration completion with KVM. https://www.berrange.com/posts/2016/05/12/analysis-of-techniques-for-ensuring-migration-completion-with-kvm/, May 2016. [Online, accessed April 2019].

[33] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. Optimized pre-copy live migration for memory intensive applications. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2011.

[34] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[35] SPECjbb2005. https://www.spec.org/jbb2005/index.html. [Online, accessed April 2019].

[36] Perf Wiki. https://perf.wiki.kernel.org/index.php. [Online, accessed April 2019].

[37] Saurabh Badhwar. Performance profiling with perf. https://fedoramagazine.org/performance-profiling-perf/, August 2016. [Online, accessed April 2019].

[38] Paul Drongowski. PERF. https://sandsoftwaresound.net/perf/, May 2015. [Online, accessed April 2019].

[39] iostat(1) Linux man page. https://linux.die.net/man/1/iostat. [Online, accessed April 2019].

[40] Stefan Hajnoczi. QEMU Internals: How guest physical RAM works. http://blog.vmsplice.net/2016/01/qemu-internals-how-guest-physical-ram.html, January 2016. [Online, accessed April 2019].

[41] time(1) Linux man page. https://linux.die.net/man/1/time. [Online, accessed April 2019].

[42] Travis Oliphant. NumPy: A guide to NumPy. http://www.numpy.org/. [Online, accessed April 2019].

[43] scikit-learn - Machine Learning in Python. https://scikit-learn.org/stable/. [Online, accessed April 2019].

[44] scikit-learn - Support Vector Regression(SVR. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html. [Online, accessed April 2019].

[45] scikit-learn - GridSearchCV. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. [Online, accessed April 2019].

[46] scikit-learn - $R\hat{2}$ (coefficient of determination) regression score function. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html. [Online, accessed April 2019].

[47] scikit-learn - Mean absolute error regression loss. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html. [Online, accessed April 2019].

[48] API Reference - scikit-learn : sklearn.metrics: Metrics. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics. [Online, accessed April 2019].

[49] Fei Zhang, Guang Ming Liu, Xiaoming Fu, and Ramin Yahyapour. A Survey on Virtual Machine Migration: Challenges, Techniques, and Open Issues. *IEEE Communications Surveys & Tutorials*, 20:1206–1243, 2018.

[50] Jinshi Zhang, Eddie Dong, Jian Li, and Haibing Guan. MigVisor Accurate Prediction of VM Live Migration Behavior Using a Working-Set Pattern Model. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 30–43, New York, NY, USA, 2017. ACM.

[51] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and Energy Modeling for Live Migration of Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 171–182, New York, NY, USA, 2011. ACM.

[52] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper. Predicting the Performance of Virtual Machine Migration. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 37–46, Aug 2010.

[53] Minal Patel, Sanjay Chaudhary, and Sanjay Garg. Machine Learning Based Statistical Prediction Model for Improving Performance of Live Virtual Machine Migration. *Journal of Engineering*, 2016:Article ID 3061674, 9 pages, 04 2016.

[54] Senthil Nathan, Umesh Bellur, and Purushottam Kulkarni. Towards a Comprehensive Performance Model of Virtual Machine Live Migration. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 288–301, New York, NY, USA, 2015. ACM.

[55] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang. Live Migration of Multiple Virtual Machines with Resource Reservation in Cloud Computing Environments. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 267–274, July 2011.

[56] Changyeon Jo, Youngsu Cho, and Bernhard Egger. A Machine Learning Approach to Live Migration Modeling. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 351–364, New York, NY, USA, 2017. ACM.

[57] Wikipedia: Java Virtual Machine. https://en.wikipedia.org/wiki/Java_virtual_machine. [Online, accessed April 2019].

[58] libvirt: The virtualization API. https://libvirt.org/. [Online, accessed April 2019].

[59] KVM. https://www.linux-kvm.org/page/Main_Page. [Online, accessed April 2019].

[60] VirtIO. https://www.linux-kvm.org/page/Virtio. [Online, accessed April 2019].

[61] Wikipedia: VMware ESXi. https://en.wikipedia.org/wiki/VMware_ESXi. [Online, accessed April 2019].

[62] Wikipedia: Network Time Protocol(NTP). https://en.wikipedia.org/wiki/Network_Time_Protocol. [Online, accessed April 2019].

National Technical University
of Athens
School of Electrical
and Computer Engineering
Department of Computer Science

Prediction Techniques for Live Migration of Virtual Machines in Cloud Computing Environments

DIPLOMA THESIS

DIMITRIOS KALOGEROPOULOS

Supervisor : Georgios Goumas
Assistant Professor N.T.U.A.

Athens, July 2019

National Technical University
of Athens
School of Electrical
and Computer Engineering
Department of Computer Science

Prediction Techniques for Live Migration of Virtual Machines in Cloud Computing Environments

DIPLOMA THESIS

DIMITRIOS KALOGEROPOULOS

Supervisor : Georgios Goumas
Assistant Professor N.T.U.A.

Approved by the committee on the July 8, 2019.

Georgios Goumas Assistant Professor N.T.U.A.	Nectarios Koziris Professor N.T.U.A.	Nikolaos Papaspyrou Associate Professor N.T.U.A.

Athens, July 2019

Dimitrios Kalogeropoulos
Electrical and Computer Engineer

Περίληψη

Μια από τις βασικές τεχνολογίες εικονικοποίησης της Υποδομής-ως-Υπηρεσία (IaaS) στην εποχή του Cloud Computing είναι η ζωντανή μεταφορά (live migration) των εικονικών μηχανών (VMs). Μέσω του live migration, προβλήματα όπως η ενοποίηση των servers και η εξισορρόπηση φόρτου μεταξύ των φυσικών μηχανημάτων μπορούν να συντονιστούν. Ωστόσο, η μη διαθεσιμότητα της υπηρεσίας κατά τη διάρκεια του VM live migration μπορεί να είναι σημαντική σε σχέση με τις προσδοκίες των πελατών για την απόκριση των υπηρεσιών καθώς και τα επίπεδα ποιότητας της υπηρεσίας (QoS). Αυτές οι μετρικές δηλώνονται σε συμφωνίες σε επίπεδο υπηρεσιών μεταξύ πελάτη-παρόχου (SLAs). Συγκεκριμένα, για το live migration με pre-copy τεχνική αντιγραφής της μνήμης, υπάρχει το ρίσκο της μη σύγκλισης του αλγορίθμου και επομένως τη μη μετάβασή του στο stop-and-copy στάδιο. Αυτή η κατάσταση συμβαίνει όταν το VM γράφει στις σελίδες μνήμης ταχύτερα από το ρυθμό μεταφοράς των σελίδων αυτών από τον αρχικό host στον host προορισμού. Καθώς οι Cloud πάροχοι υπηρεσιών δεν μπορούν να ρυθμίσουν το ρυθμό "βρώμικων" σελίδων (dirty page rate) της εφαρμογής που εκτελείται σε μια εικονική μηχανή, πρέπει να διαμορφώσουν τις συνθήκες τερματισμού του migration. Στην περίπτωση του QEMU/KVM λογισμικού ελέγχου, οι τροποποιήσιμες παράμετροι είναι η μέγιστη ταχύτητα μεταφοράς (*max-bandwidth*) και ο μέγιστος ανεκτός χρόνος μη λειτουργίας (*downtime-limit*). Λόγω του φυσικού δικτύου, το εύρος ζώνης έχει περιορισμένο άνω όριο και οι διαχειριστές δεν θέλουν να το εκμεταλλευτούν πλήρως. Επομένως, η παράμετρος *downtime-limit* θα πρέπει να διαμορφωθεί ώστε το pre-copy live migration να συγκλίνει και να ολοκληρωθεί με επιτυχία.

Οι εφαρμογές με εκτεταμένες εγγραφές στη μνήμη είναι δύσκολο να μεταφερθούν, επειδή τα όρια του ρυθμού μεταφοράς και το *downtime-limit* δεν μπορούν να ρυθμιστούν βέλτιστα χωρίς να είναι γνωστή η συμπεριφορά της εφαρμογής. Προκειμένου να αντιμετωπιστούν οι προκλήσεις που προκύπτουν σχετικά με το πρόβλημα της σύγκλισης της pre-copy live migration τεχνικής στα σύγχρονα κέντρα δεδομένων, αναπτύσσουμε ένα framework για την παρακολούθηση των διαθέσιμων VMs όπου λαμβάνονται δυναμικές αποφάσεις και ενέργειες με βάση το αποτύπωμά τους στη μνήμη προτού ξεκινήσει το live migration. Υλοποιούμε ένα μηχανισμό που ονομάζεται BitmapTrace και ενσωματώνεται στο QEMU/KVM, ο οποίος καταγράφει τον αριθμό των dirty σελίδων της εικονικής μηχανής για μια συγκεκριμένη χρονική περίοδο με overhead μόλις λίγα επιπλέον δευτερόλεπτα στο χρόνο εκτέλεσης της εφαρμογής. Χρησιμοποιούμε αυτόν τον μηχανισμό σε ένα σενάριο χρονοδρομολόγησης του migration ενός υποσυνόλου VMs από όσα εκτελούνται στο ίδιο φυσικό μηχάνημα. Παρακολουθώντας τη συμπεριφορά της μνήμης τους και χρησιμοποιώντας ένα μοντέλο πρόβλεψης, επιλέγουμε για live migration τις εικονικές μηχανές χωρίς να παραβιάζεται η διαθεσιμότητα υπηρεσιών με βάση το συμφωνηθέν SLA. Με τον τρόπο αυτό επιτυγχάνεται ένας συμβιβασμός μεταξύ των στόχων του Cloud παρόχου και των αναμενόμενων απαιτήσεων QoS των πελατών.

Λέξεις κλειδιά

Υπολογιστικό Νέφος, Εικονικές Μηχανές, Ζωντανή Μεταφορά, αποτύπωμα μνήμης, τεχνικές πρόβλεψης

Abstract

One of the key virtualization technologies of Infrastructure-as-a-Service in the era of Cloud Computing is the live migration of running Virtual Machines (VMs). Through VM live migration, issues such as VM consolidation and service performance degradation can be coordinated and balanced. However, the unavoidable short downtime/service unavailability during VM live migration may be substantial with regard to the customers' expectations on responsiveness as well as the requested Quality of Service (QoS) levels. These metrics are declared in established Service Level Agreements (SLAs). Especially for pre-copy live migration algorithm the main risk is when the migration process cannot converge to an optimal point for the final stop-and-copy phase. This situation happens when the VM dirties its memory pages faster than the migration bandwidth. Since Cloud Service Providers are not able to tune the dirty page rate of the workload running on a Virtual Machine, they need to configure the migration termination conditions. In case of QEMU/KVM, the system admin configurable parameters are the maximum migration transfer rate (*max-bandwidth*) and the maximum tolerated downtime (*downtime-limit*). Due to physical network, bandwidth has an upper limit and operators do not want to make full utilization of it. Thus, *downtime-limit* should be configured to make VM live migration converge and complete successfully.

Memory intensive applications are difficult to migrate because rate limits and downtime cannot be optimally set without detailed knowledge of the application behavior. In order to address the concerns that arise regarding the convergence problem of pre-copy live migration in modern data centers, we employ a framework for monitoring the available VMs and make dynamic decisions and actions based on their memory footprint right before migration is triggered. We implement a memory profiling module in QEMU/KVM called BitmapTrace which tracks the dirty pages of the workload for a certain profiling period with an execution time overhead of just a few seconds. We utilize this mechanism in a migration scheduling scheme where given a set of real-life workloads, each of them running in a separate Virtual Machine on the same host, we monitor their memory behavior and we select the best VM candidates using a prediction model without violating the agreed service availability. In this way, a trade-off between the Cloud Service Provider's objectives and the customers' expected QoS requirements can be achieved successfully.

Key words

Virtual Machines, Live Migration, pre-copy, QEMU/KVM, memory footprint, SLA, prediction techniques, modeling

Contents

Περίληψη . . . 5

Abstract . . . 7

Contents . . . 9

List of Tables . . . 13

List of Figures . . . 15

1. Introduction . . . 17
- 1.1 Motivation . . . 17
- 1.2 Thesis Contribution . . . 18

2. Background . . . 19
- 2.1 Cloud Computing . . . 19
- 2.2 Virtualization . . . 20
  - 2.2.1 Hypervisor . . . 21
  - 2.2.2 Types of Virtualization . . . 22
  - 2.2.3 What is the difference between a container and a virtual machine? . . . 24
  - 2.2.4 QEMU/KVM . . . 24
  - 2.2.5 XEN . . . 26
  - 2.2.6 VMWare ESX/ESXi . . . 26
- 2.3 Live Migration of Virtual Machines . . . 27
  - 2.3.1 Live Migration Techniques . . . 28
  - 2.3.2 Performance Metrics of Live Migration . . . 30
  - 2.3.3 Migration requirements . . . 30
  - 2.3.4 Live Migration using QEMU/KVM . . . 31
- 2.4 Workload Characterization . . . 33
  - 2.4.1 CPU-Intensive . . . 34
  - 2.4.2 Memory-Intensive . . . 34
  - 2.4.3 I/O-Intensive . . . 34

3. Evaluation of Pre-Copy Live Migration . . . 35
- 3.1 Testbed . . . 35
- 3.2 Benchmarks . . . 35

3.2.1 Microbenchmarks . . . 36
3.2.2 SPEC2006 . . . 36
3.2.3 SPECjbb2005 . . . 36
3.2.4 Idle . . . 36
3.3 Evaluation of QEMU/KVM Live Migration Features . . . 36
3.3.1 Memory Resource Reservation (*VM_Size*) . . . 37
3.3.2 Maximum acceptable downtime (*downtime-limit*) evaluation . . . 38
3.3.3 Maximum available bandwidth (*max-bandwidth*) evaluation . . . 39
3.3.4 Evaluation of the moment triggering live migration . . . 40
3.3.5 Minimum *downtime-limit* for successful migration . . . 41
3.4 Memory-related Live Migration Features . . . 42
3.5 Memory Monitoring Tools . . . 45
3.5.1 Cache misses and MPKI using Performance Counters . . . 45
3.5.2 IOwait . . . 46
3.5.3 Tuning dirty page rate with microbenchmarks . . . 47

4. Design and Implementation of BitmapTrace mechanism in QEMU/KVM . . . 49
4.1 Design . . . 49
4.2 Implementation . . . 50
4.3 Basic Usage . . . 51
4.4 Evaluation of BitmapTrace mechanism . . . 52
4.4.1 WWS Estimation . . . 52
4.4.2 Tuning *period* parameter . . . 53
4.4.3 Tuning *iterations* parameter . . . 57
4.4.4 Performance Overhead . . . 60

5. Machine Learning Approach for VM Migration Candidates . . . 63
5.1 Model Building . . . 63
5.1.1 Building the Data Set . . . 63
5.1.2 Model Generation . . . 67
5.2 Model Evaluation . . . 68
5.3 Operation Policies for VM Live Migration Candidate Selection . . . 71
5.4 Evaluation of VM Live Migration Candidate Selection . . . 72
5.4.1 VMs with same application and without SLA violations . . . 73
5.4.2 VMs with same application and SLA violations . . . 74
5.4.3 VMs with different application and without SLA violations . . . 76

6. Related Work . . . 79

7. Conclusions and Future Work . . . 83

Abbreviations . . . 85

Bibliography 87

List of Tables

3.1 Minimum *downtime-limit* for successful migration (*max-bandwidth*=800Mbps) . . . 42

4.1 Runtime overhead of application dirtying 512MB of memory when triggering Bitmap-Trace mechanism for (*"iterations"*, *"period"*)=(10,1000) . . . 60
4.2 Runtime overhead of application dirtying 512MB of memory when triggering Bitmap-Trace mechanism for different (*"iterations"*, *"period"*) values . . . 61
4.3 Performance overhead of different applications with execution time=3min when triggering BitmapTrace mechanism . . . 61

5.1 Minimum *downtime-limit* for successful migration (*max-bandwidth*=800Mbps) . . . 66
5.2 Model Training time in case of (iterations,pediod) = (10,1000) . . . 68
5.3 Accuracy and MAE of the *min_success_downtime-limit*, *downtime* and *total_time* prediction models . . . 69
5.4 Live Migration Scheduling scenario with VMs running the same application and without SLA violations . . . 73
5.5 Aggregated total time with VMs running the same application and without SLA violations . . . 73
5.6 Live Migration Scheduling scenario with VMs running the same application and SLA violations . . . 74
5.7 Live Migration Scheduling scenario with VMs running different application and without SLA violations . . . 76
5.8 Aggregated total time with VMs running different application and without SLA violations . . . 76

List of Figures

2.1 Cloud Service Models . . . 20
2.2 Hypervisor types . . . 22
2.3 Virtualization techniques . . . 23
2.4 Virtual Machines vs Containers . . . 24
2.5 Overview of the QEMU/KVM virtualization environment . . . 25
2.6 Pre-copy Live Migration . . . 29
2.7 Post-copy Live Migration . . . 29
2.8 Hybrid Live Migration . . . 29
2.9 Types of Workload . . . 34

3.1 Impact of *VM_Size* in Live Migration . . . 38
3.2 Impact of *downtime-limit* in Live Migration . . . 39
3.3 Impact of *max-bandwidth* in Live Migration . . . 40
3.4 Impact of the exact moment migration is triggered . . . 41
3.5 Minimum *downtime-limit* for successful Live Migration . . . 43
3.6 MPKI using *cache-misses* from *perf* tool . . . 46

4.1 WWS Evaluation . . . 54
4.2 Tuning *"period"* parameter of BitmapTrace . . . 56
4.2 Tuning *"period"* parameter of BitmapTrace (cont.) . . . 57
4.3 Tuning *"iterations"* parameter of BitmapTrace . . . 58
4.3 Tuning *"iterations"* parameter of BitmapTrace (cont.) . . . 59

5.1 AVG and STDEV Dirty Pages during profiling period . . . 64
5.1 AVG and STDEV Dirty Pages during profiling period (cont.) . . . 65
5.2 Flowchart of the K-fold cross-validation . . . 69
5.3 General prediction models workflow . . . 70
5.4 *max_downtime* and *predicted min_success_downtime-limit* in case of VMs with same application and without SLA violations . . . 74
5.5 *max_downtime* and *predicted min_success_downtime-limit* in case of VMs with same application and SLA violations . . . 75
5.6 *max_downtime* and *predicted min_success_downtime-limit* in case of Ms with different application and without SLA violations . . . 77

Chapter 1

Introduction

1.1 Motivation

Live migration has become an essential tool for efficient management of resources in a data center by enabling VM consolidation and load balancing. In the case of pre-copy live migration, the memory is iteratively transferred from the source to destination until the remaining dirty memory along with the device states are left to be transferred in the last iteration within a maximum acceptable downtime that is agreed in SLA contracts [1, 2]. However, demanding workloads, i.e. workloads that are memory-intensive, are a major concern for cloud providers because of high downtime or unpredictable memory behavior issues which may result in convergence problem. In particular, the standard (vanilla) pre-copy migration process with short configured downtimes, as implemented in QEMU/KVM, may iterate indefinitely for many real-world applications.

Traditionally, cloud providers wait for the pre-copy migration process to execute for a long time until they simply cancel the migration or employ optimization techniques to minimize the key performance metrics of total transferred data, total migration time and downtime. However, these remedies lead to significant influence on service performance, system load and network which affect the Service Level Agreements (SLAs) and Quality of Service (QoS).

Management tasks that use live migration should wisely decide not only the VM candidates for migration from source to destination host, but also the transfer order. Consider a migration scheduling scenario where eight (8) Virtual Machines run on a single host and the cloud provider should migrate four (4) of them. There are seventy (70) possible combinations and the best four (4) VM candidates should be chosen according to SLAs and operation policies of the data center. Moreover, the system administrator should decide if the VMs will be migrated in parallel or sequentially. With a parallel migration the chances that neither guest will ever be able to complete are increased as the bandwidth is shared between the selected VMs. Thus, in order to provide high availability, avoid SLA violations and the subsequent penalties, the VM candidates for migration should be monitored and controlled.

In this thesis, we propose a supervised learning model which predicts with high accuracy the minimum downtime-limit required to successfully complete a pre-copy live migration within a specific time window. The term *downtime-limit* is used by QEMU/KVM as the maximum acceptable downtime during the last phase of pre-copy live migration. The input features of this model are gathered with a monitoring tool, called BitmapTrace, which has been integrated in the existing QEMU/KVM migration module and takes a snapshot of the memory dirtying behavior of the VM for a certain profiling period with an interference in the execution time of the workload of just a few

seconds. Furthermore, we implement a framework where the available VM candidates running on a single host are sorted based on the memory footprint of the workload running on them and the operation policies of a cloud computing environment. The selected VMs are sequentially live migrated to the destination host by satisfying the SLA constraints for each workload to maximum possible level as well as minimizing the aggregated total migration time.

1.2 Thesis Contribution

The major contributions of our thesis are the following:

(i) The impacts of the key parameters affecting the QEMU/KVM pre-copy live migration have been evaluated with diverse synthetic and real-world workloads. By analyzing the results, we explore the influences of the configured parameters *max-bandwidth* and *downtime-limit* along with the VM live migration performance metrics (total migration time and downtime).

(ii) We design a monitoring tool, called BitmapTrace, to track the memory footprint of a VM before its migration. This module is responsible for storing the new dirty pages that are produced at periodic intervals while the VM is running and is integrated in the existing live migration mechanism of QEMU/KVM. Moreover, we suggest the values of the BitmapTrace parameters that ensure an adequate profiling period.

(iii) We implement prediction techniques in a VM live migration framework to model the relationship between the memory dirtying behavior and the minimum *downtime-limit* that guarantees the successful completion of pre-copy live migration for all running VMs on a single host. The proposed VM candidates for migration meet the objectives of the operation policies in a cloud computing environment and satisfy the existing SLA constraints.

Chapter 2

Background

In this chapter, we provide a detailed overview about the concepts of cloud computing which are required to understand the hardware and software environment targeted by this work. Initially, we explain what exactly cloud computing is, then we review the basic principles of Virtualization as the core technology of cloud computing. More specifically, we present the available virtualization types and provide an inclusive description of hypervisors. In the next section, we thoroughly analyze the concept of Live Migration of Virtual Machines and its features focusing on the Pre-Copy algorithm of QEMU/KVM hypervisor. Finally, we discuss the different types of workload and their characteristics.

2.1 Cloud Computing

According to the National Institute of Standards and Technology (NIST) [3], Cloud Computing is defined as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This Cloud model is composed of five essential characteristics (On-demand self-service, Broad network access, Resource pooling, Rapid elasticity, Measured service), three service models, and four deployment models. The three service models mentioned in the NIST definition are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS), as shown in figure 2.1.

- *Software as a Service (SaaS)*: This service provides the capability to the consumer to use the provider's applications running on the cloud.
- *Platform as a Service (PaaS)*: In this type of service, the consumer can deploy the consumer-created or acquired applications created by using programming languages or tools provided by provider, on the cloud infrastructure.
- *Infrastructure as a Service (IaaS)*: This is a capability provided to the consumer which can provision processing, storage, networks and other fundamental computing resources where the consumers can deploy and run the software (i.e. operating systems, applications)

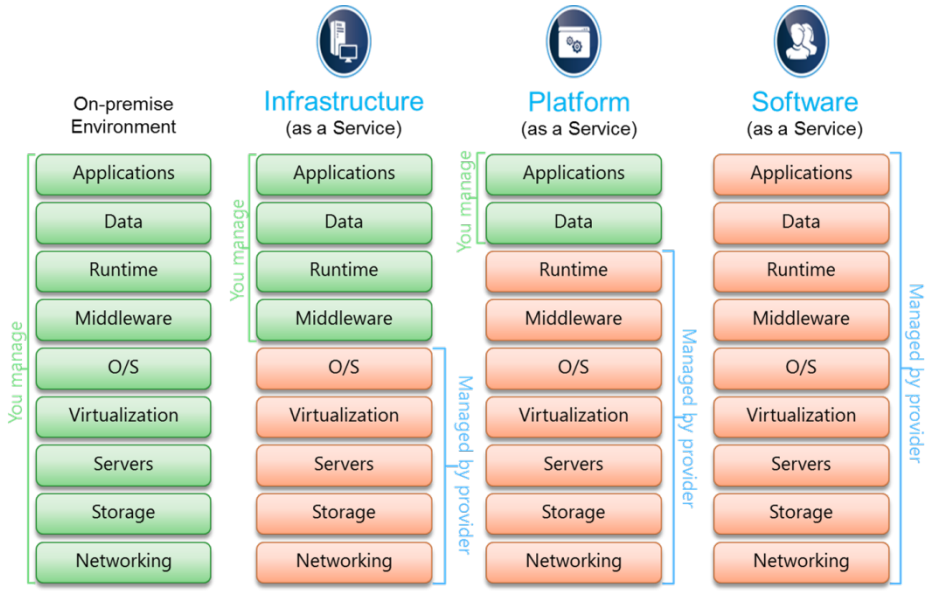


Figure 2.1: Cloud Service Models

When a Cloud is made available in a Pay-As-You-Use manner to the public, we call it *Public Cloud*; the service being sold is Utility Computing. Some common examples of public Utility Computing include Amazon EC2 [4], GoogleAppEngine [5] and Microsoft Azure [6]. We use the term *Private Cloud* to refer to the internal data centers of a business or other organization that are not made available to the public. Community Cloud refers to the cloud infrastructure that is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). Finally, when the cloud infrastructure is composed of two or more distinct cloud infrastructures (private, community, or public) we call it *Hybrid Cloud*.

Service-level agreements (SLAs) have become more important as organizations move their systems, applications and data to the cloud. A cloud SLA ensures that cloud providers meet certain enterprise-level requirements and provide customers with a clearly defined set of deliverables. Organizations must define time windows in terms of full functionality and partial functionality, and each application should (at minimum) meet its on-premise standards for availability and data recovery when operating within the Cloud environment. It is possible to mitigate against the risk of a major failure at a cloud provider by using a second cloud provider for disaster recovery. The costs of using multiple cloud providers in this fashion are high and this should be weighed up against the costs of downtime to the business.

2.2 Virtualization

In cloud computing, Infrastructure-as-a-Service (IaaS) provides on-demand virtual machine instances with virtualization technologies. Virtualization refers to the process of creating a virtual version of a device or resource, such as virtual computer hardware platforms, storage devices or computer network resources. This technology allows the creation of multiple simulated environ-

ments or dedicated resources from a single, physical hardware system. Specifically, the software emulation of a computer system is called *Virtual Machine* (VM). The physical, "real-world" hardware that runs the VM is generally known as 'host', and the virtual machine emulated on that machine is generally referred to as 'guest'. Depending on the use and level of correspondence to any physical computer, virtual machines can be divided into two categories:

(i) System Virtual Machines: A system platform that supports the sharing of the host computer's physical resources (CPU, memory, disk, I/O, etc.) between multiple virtual machines, each one running with its own copy of the operating system. The virtualization technique is provided by a software layer known as hypervisor, which can run either on bare hardware or on top of an operating system. More details regarding hypervisors can be found on section 2.2.1

(ii) Process Virtual Machine: Designed to provide a platform-independent programming environment that masks the information of the underlying hardware or operating system and allows program execution to take place in the same way on any given platform. This type of VM has become popular with the Java programming language, which is implemented using the Java virtual machine [7].

Despite the fact that virtualization has its roots since early 1960's as a method of logically dividing the system resources provided by mainframe computers between different applications, it has gained a huge popularity recently due to the evolution of Cloud Computing and Data Centers. Offloading hardware requirements and utility costs can rapidly transform a company's infrastructure and improve its efficiency by itself. Virtualization in cloud computing allows cloud providers to run multiple applications and operating systems on the same server, thereby providing a solution for efficient resource utilization and reduced costs. Companies that host their applications on the cloud can benefit from switching to a virtualized IT environment in case of disaster recovery and failover protection as well as benefit from improved system reliability and security.

2.2.1 Hypervisor

The hypervisor or Virtual Machine Monitor (VMM) is a software that is responsible for creating, controlling and monitoring virtual machines. The primary role of the hypervisor is the scheduling and allocation of computing resources to the guests in case of hardware/server virtualization. This is achieved by abstraction of the underlying host's physical resources into isolated virtualized environments that create to the guests the illusion of actual hardware. Thereby, a single host may execute multiple instances of a variety of operating systems concurrently. Hypervisors are classified into type-1 and type-2 hypervisors. Type-1 hypervisors run directly on top of the host's physical hardware, hence they are also called bare-metal hypervisors. Type-2 hypervisors operate on top of a host OS as an application, and hence they are also known as hosted hypervisors. Type-1 hypervisors are required to handle hardware and software management tasks on the host, while in type-2 hypervisors the underlying host OS kernel is responsible them.

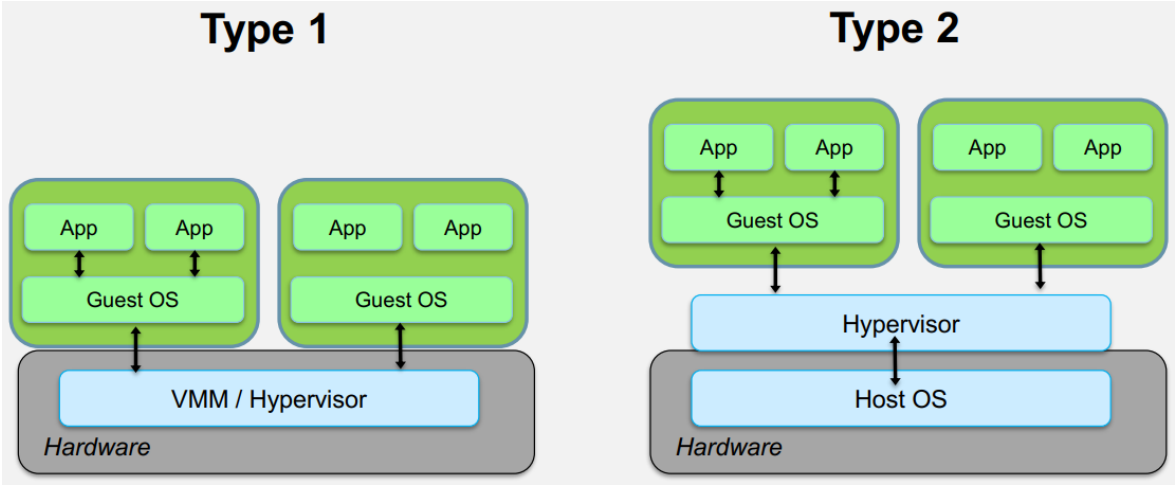


Figure 2.2: Hypervisor types

2.2.2 Types of Virtualization

Virtualization can take many forms depending on the type of application use and hardware utilization. The main types are listed below:

- **Hardware Virtualization/Server Virtualization**
  The main form of virtualization is the server/hardware virtualization, which is also the related to the live migration process, as we describe in detail in section 2.3. Hardware virtualization means that users can run different operating systems on the same machine and system at the same time. In other words, one or more virtual machines(guests) are created to share the hardware resources of one physical computer(host).

  Regarding the level of the awareness of the guest operating system of the fact that is being virtualized there are three basic different techniques of hardware virtualization:

  - **Full Virtualization**
    This technique involves virtualization of all the computing units of a typical machine including the CPU, memory, I/O and disk for each guest which is unaware that it is in a virtualized environment. Unmodified operating systems [1] as guests can be executed on the host. When the guest kernel executes privileged instructions, the underlying hypervisor performs a binary translation of them into non-privileged instructions that will be executed on the host hardware. This can incur a large performance overhead compared to the other modes of hardware virtualization.
  - **Paravirtualization**
    Under this technique the kernel of the guest operating system is aware that it is being virtualized and modified specifically to run on the hypervisor. This typically involves replacing any privileged operations with system calls to the hypervisor, known as *hypercalls*. The hypercalls invoke the VMM to perform the task on behalf of the guest kernel

[1] The term *unmodified* refers to operating system kernels which have not been altered to run on a hypervisor and therefore still execute privileged operations.

which results in performance enhancement for some operations performed by the guest OS.

  - **Hardware Assisted Virtualization**
    This approach aims to efficiently use full virtualization with the hardware capabilities. Thus, the performance of the guest environment for common virtualization challenges like translating instructions and memory addresses is improved. Hardware-assisted virtualization was added to x86 processors by Intel (**Intel VT-x**) and AMD (**AMD-V**) .

- **Application Virtualization**
  Application Virtualization (also known as Process Virtualization) is the separation of the installation of an application from the client computer accessing it. The application behaves at runtime like it is directly interfacing with the original operating system and all the resources managed by it, but can be isolated or sandboxed to varying degrees.

- **Operating System(OS)-Level Virtualization**
  This technique allows the same physical host to serve different workloads that operate independently on a shared OS. This allows a physical server to run multiple isolated operating system instances, called containers. Even though all the virtual machines/containers are running under the same kernel, they have their own file system, processes, memory, devices etc. This technique is also known as Shared Kernel Virtualization.

- **Network Virtualization**
  Network virtualization combines all physical networking equipment as well as software network resources and network functionality into a single software-based administrative resource, known as virtual network.

- **Storage Virtualization**
  Storage virtualization is the process of grouping the physical storage from multiple network storage devices so that it looks like a single storage device.

  The above virtualization techniques can be summarized in the following figure :

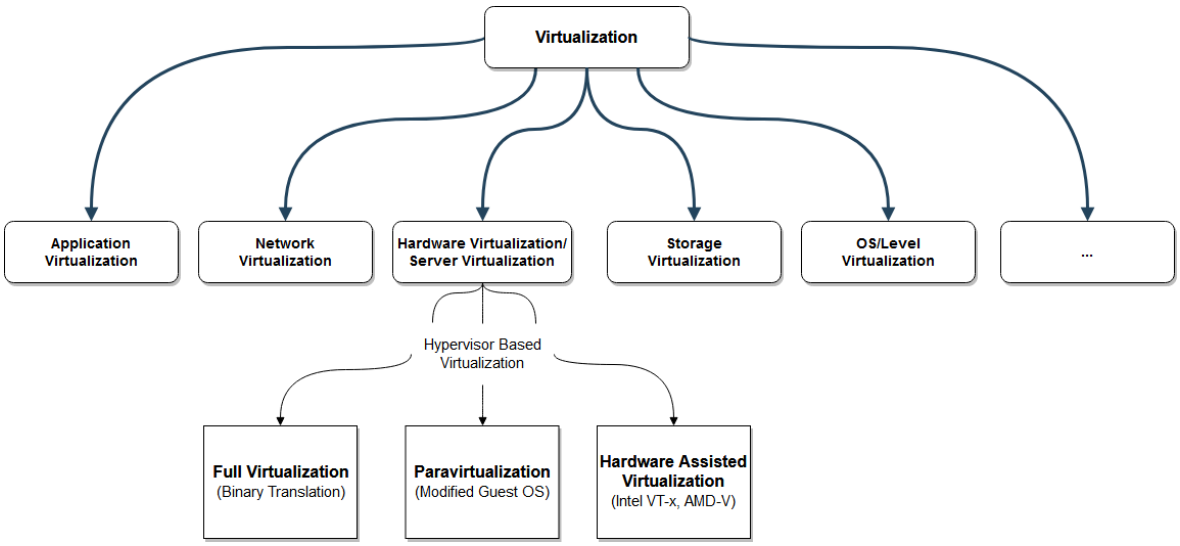


Figure 2.3: Virtualization techniques

2.2.3 What is the difference between a container and a virtual machine?

What makes containers so popular nowadays, is what separates them from Virtual Machines. The main difference is that containers provide a way to virtualize an OS so that multiple workloads run on a single OS instance, whereas in VMs the hardware is being virtualized to run multiple OS instances. Container oriented implementations share the kernel of the host operating system which is not the case in Virtual Machine technologies such as KVM. Typically, containers are designed to run a single program, as opposed to emulating a full multi-purpose server. Therefore, containers' speed, agility, and portability make them yet another tool to help streamline software development.

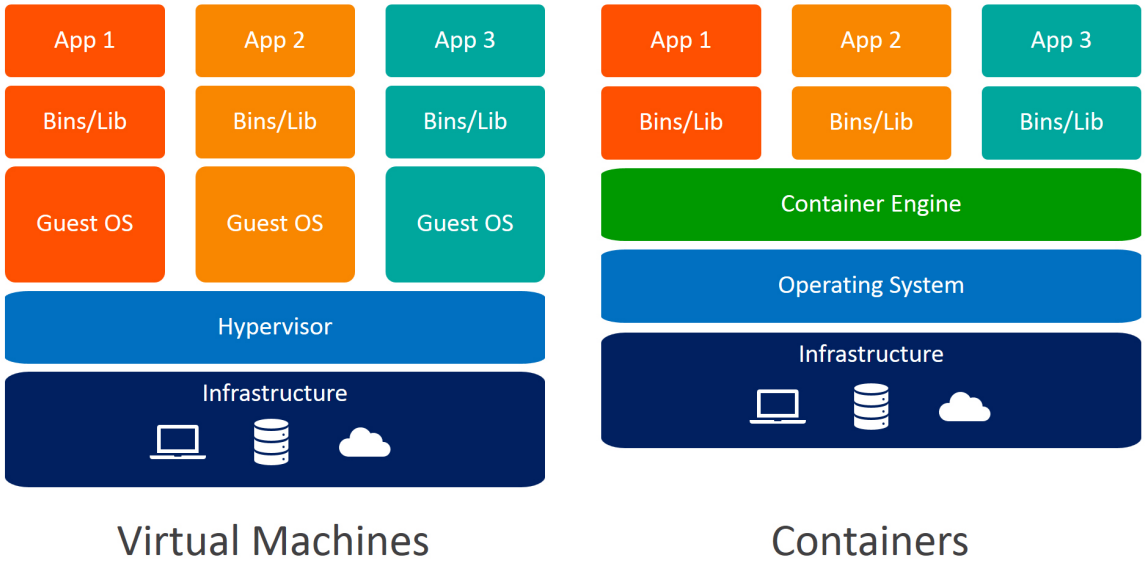


Figure 2.4: Virtual Machines vs Containers

2.2.4 QEMU/KVM

QEMU is a type 2 hypervisor/VMM that that runs within user space and performs hardware virtualization. QEMU can run independently, but due to the emulation being performed entirely in software, it is extremely slow. To overcome this, QEMU can make use of KVM as an accelerator and as a result the physical CPU virtualization extensions can be used.

QEMU

QEMU(Quick Emulator) is a generic and open source machine emulator and virtualizer [8]. It was written by Fabrice Bellard, it is free software and mainly licensed under the GNU General Public License (GPL).

When is used as a machine emulator, QEMU can run OSs and programs made for one machine (e.g. an ARM board) on a different machine. QEMU emulates CPUs through dynamic binary translation techniques and provides a set of device models. The binary translator that does this job is known as *Tiny Code Generator (TCG)* [9]. It transforms the binary code written for a given processor to another one. When used as a virtualizer, QEMU achieves near native performance by executing the

guest code directly on the host CPU. For example, when working under Xen/KVM hypervisors, QEMU can operate in this mode.

In short, QEMU is capable of running without KVM using the previously mentioned binary translation method. This execution will be slower compared to the hardware-accelerated virtualization enabled by KVM. In any mode (either as a virtualizer or emulator), QEMU is involved in the emulation of hardware devices and different peripherals, such as disks, network cards, VGA, buses, serial and parallel ports, USB. Apart from this I/O device emulation, when working with KVM, QEMU/KVM creates and initializes virtual machines. It also initializes different POSIX threads for each virtual CPU of a guest. Also, it provides a framework to emulate the virtual machine's physical address space within the user mode address space of QEMU/KVM.

When QEMU is running, a monitor console, called *Human Monitor Interface (HMP)* [10], is provided for interaction with the user. Furthermore, QEMU provides a stable JSON-based protocol called *QEMU Machine Protocol (QMP)* [11]. It allows higher level tools, such as libvirt [12], to communicate with a running QEMU instance and easily parse the commands sent to QEMU as well as the returned output in case of query commands or asynchronous events. Users can also connect the QMP interface to a network socket and interact with QEMU via a specified port.

In this thesis, the version 2.10.1 of QEMU was used, which was released as stable version. Source code can be found at https://www.qemu.org/.

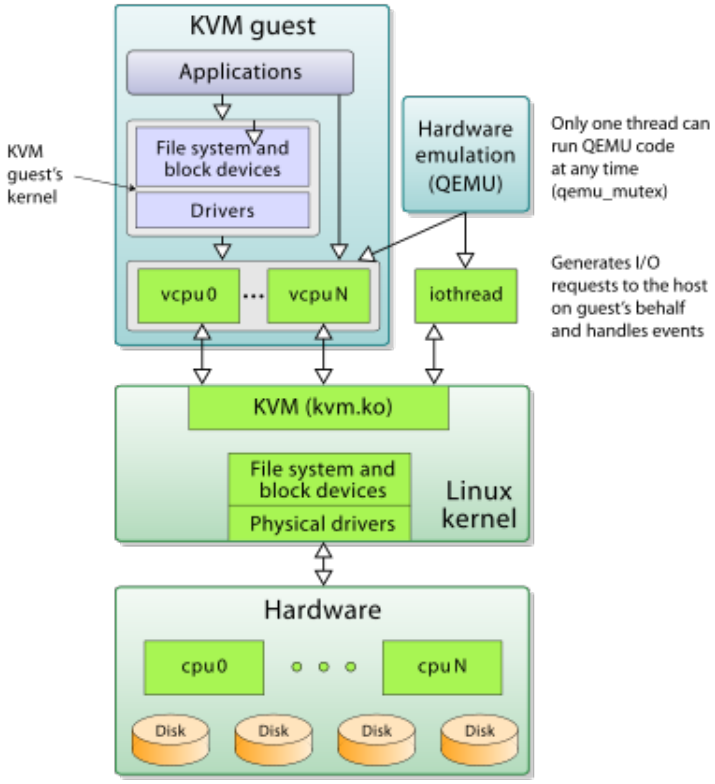


Figure 2.5: Overview of the QEMU/KVM virtualization environment

KVM

Kernel-based Virtual Machine (KVM) [13] is a set of Linux kernel modules that when loaded, the Linux kernel is turned into hypervisor. It is a full virtualization solution for Linux on x86 hard-

ware that supports the Intel VT or AMD-V Virtualization extensions. Linux continues its normal operations as OS but also provides hypervisor facilities to user space. KVM consists of a loadable kernel module, known as *kvm.ko*, that provides the core virtualization infrastructure and a processor specific module, named *kvm-intel.ko* or *kvm-amd.ko*, depending on the host machine CPU. Previously, KVM was a fork of QEMU, named qemu-kvm. However, now the kernel component of KVM is included in mainline Linux, as of 2.6.20. The userspace component of KVM is included in mainline QEMU, as of 1.3 [14].

The architecture of KVM consists mainly of two components, a kernel and a user-space. The kernel component is the KVM kernel module itself which exposes the virtualization extensions to the user-space via a character device */dev/kvm*. It supports a set of ioctls() that allow the guest code, which in our case is QEMU, to be executed on the host hardware via this device node. The lowest unit of virtualization in QEMU/KVM is a vCPU and everything builds on top of it, as we can also see in figure 2.5.

KVM can be used when launching a QEMU process with *–enable-kvm* option. Once enabled, it will give better performance than other techniques, such as binary translation.

KVM also provides paravirtualization for I/O devices through the VirtIO API [15]. VirtIO was introduced to achieve a common framework for hypervisors for IO virtualization. Rather than having a variety of device emulation mechanisms (for network, block, and other drivers), VirtIO provides a common front-end for these device emulations to standardize the interface and increase the reuse of code across the platforms. The Virtio API relies on a simple buffer abstraction to encapsulate the command and data needs of the guest.

2.2.5 XEN

Xen [16, 17] is an open-source type-1 or bare-metal hypervisor, which makes it possible to run many instances of an operating system or indeed different operating systems in parallel on a single machine (or host). Xen supports two primary types of virtualization: paravirtualization (PV) and Hardware Virtualized Machine (HVM) also known as Full Virtualization. HVMs are supported through virtualization extensions in the CPU. Both guest types can be used at the same time on a single Xen system. It is also possible to use techniques used for Paravirtualization in an HVM guest. Guest Operating Systems that run in a Xen hypervisor are called Domains. A special domain, called domain 0 ("*dom0*") has hardware access and management privileges. Once the *dom0* has started, additional unprivileged domains ("*domU*") can be started and controlled from the *dom0* for accessing the hardware or I/O functionality.

2.2.6 VMWare ESX/ESXi

VMWare ESXi, formerly ESX, is an enterprise class, Type-1 hypervisor developed by VMWare for deploying and serving virtual computers [18]. In ESX (Elastic Sky X), a linux kernel, called *vmkernel*, is the virtualization kernel which is managed by a console operating system (Service console). Its main purpose is to provide a Management interface for the host for monitoring the ESX hypervisor. After version 4.1, VMWare renamed ESX to ESXi. In ESXi (Elastic Sky X Integrated) the Service Console is removed. All the management and monitoring agents run directly on the vmkernel. Thus,

ESXi can be considered as an ultra-thin architecture that is highly reliable and its small code-base allows it to be more secure with fewer codes to patch.

2.3 Live Migration of Virtual Machines

Migration of a Virtual Machine(VM) is the process of moving a guest from one physical host to another. This activity involves transferring all the resources associated with the Virtual Machine, comprising the internal state of the devices and the virtual CPU and, of course, the memory which is the most time-consuming. The transfer of the local file system can be avoided by keeping it on a NFS storage volume accessible to both the source and the destination.

There are two types of migration: *offline* and *live migration*. During an *offline migration* the guest will be either suspended or shut down. An image of the guest's memory is moved to the destination host. Then, the guest is resumed on the destination host and the memory used on the source host can now be freed. On the other hand, *live migration* means moving a guest VM from one host to another while the guest is still running. Services running in the VM are supposed to be unaware of the migration, except from a small window when they may need to stop so that the new hypervisor start running the guest.
Live migration is useful for various reasons:

- **Load balancing**: Physical server's resources may become overloaded on some congested hosts. Live Migration of some Virtual Machines to hosts with light load can facilitate this problem of utilization discrepancy.
- **Server Consolidation**: Providers should avoid nodes with underutilized resources which are distributed in a data-center. After migrating the guests of such nodes to an existing physical machine, hosts can either shut down the offloaded nodes or execute new demanding workloads.
- **System maintenance (hardware or software)**: Hardware resources often need to be physically repaired, replaced or upgraded with new equipment. Furthermore, hypervisors regularly should be software upgraded. In both cases, the physical machine must be free of the Virtual Machines for maintenance.
- **Energy Management**: Clusters should adhere to policies regarding power efficiency. In order to save energy and reduce provider's expenses in low usage periods, some hosts can be powered off after redistributing their guests to other physical machines.

As described by Clark [19], VM memory migration can be divided into three basic phases:

- **Push Phase**: The source Virtual Machine continues running while certain pages are iteratively transferred to the destination. To ensure consistency, pages modified during this process must be resent.
- **Stop-and-Copy Phase**: The execution of source VM is stopped. The dirtied pages are transferred to the destination along with the device states. Then, the VM is resumed on destination host.
- **Pull Phase**: While VM is running on the destination, it may access a page that has not been copied yet. It remotely "pulls" this page from the source causing a page fault.

2.3.1 Live Migration Techniques

Depending on how virtual machine's memory state is transferred from the source to the destination host, the main approaches of live migration [20] are described below.

Pre-copy Live Migration

Pre-copy live migration (figure 2.6) is the default live migration technique of the current mainstream virtualization platforms such as QEMU/KVM [21], XEN [22] and VMware [23]. It is the most popular algorithm of live migration, which first sends VM's memory to the destination host and then resumes VM in it. This approach consists of the first two phases of VM migration. Specifically, in the *Initial Phase* all memory's pages are considered dirty in order to be transferred to the destination. During the *Iterative Phase*, the hypervisor needs to retransfer the pages that were modified in the previous iteration. As the source VM continues its execution during migration, already transmitted pages might be rewritten. When the VMM decides to transfer a page, it is marked as "clean," but it is considered again "dirty" when the supervisory system observes the VM remapping/faulting the page for write. It is typically irrelevant how many times a source VM writes to a given page. All that matters is the retransmission of a page even if a fraction of the page[2] has changed since the last time it was copied to the destination. When migration finishes, the VM in the destination must have the most updated version of the memory. Depending on the virtualization platform, the *Iterative phase* finishes when the pages that need to be transferred are below a certain threshold or a maximum number of iterations is reached. Finally, at the *Stop-and-copy Phase* the source VM is suspended and the remaining inconsistent memory pages, the CPU and device states are transferred to the destination in order for the VM to continue its execution. Memory-intensive applications that continuously dirty memory pages result in increasing the total migration time, as dirtied pages are retransmitted during iterative phase, and reduce pre-copy's agility, since the algorithm may never converge. The vanilla pre-copy algorithm of QEMU/KVM is our research focus and it is described in section 2.3.4.

Post-copy Live Migration

Post-copy live migration (figure 2.7) incorporates the stop-and-copy phase and the pull phase of migration process. The CPU state and device states are transferred to the destination host at the beginning of live migration and then the VM starts its execution. The memory pages are transferred on demand when they are accessed for the first time. Main advantages of post-copy algorithm are the short downtime and the transfer of each memory page only once compared to pre-copy algorithm. However, the performance of the VM and its applications may be severely affected as each memory page access generates a page fault causing the VM to wait until the requested page is transferred over the network. Moreover, another drawback is that if the destination fails during migration, pre-copy can recover the VM, whereas post-copy cannot.

Hybrid Live Migration

Hybrid approach (figure 2.8) seeks to tackle the disadvantages of post-copy approach occurred from the faulted pages. It also tries to decrease the total migration time and the total data transferred

[2] 4KB is the most used page size

compared to pre-copy approach. Thus, hybrid live migration includes all the three phases of the migration process. During the *push phase*, the most frequently accessed memory pages are transferred to the destination. Then, the source VM is suspended and all the device states are transmitted to the destination (*stop-and-copy phase*). The remaining memory pages are retrieved from source host after the VM is resumed at the destination host (*pull phase*). Similar to the post-copy algorithm, this approach is not reliable.

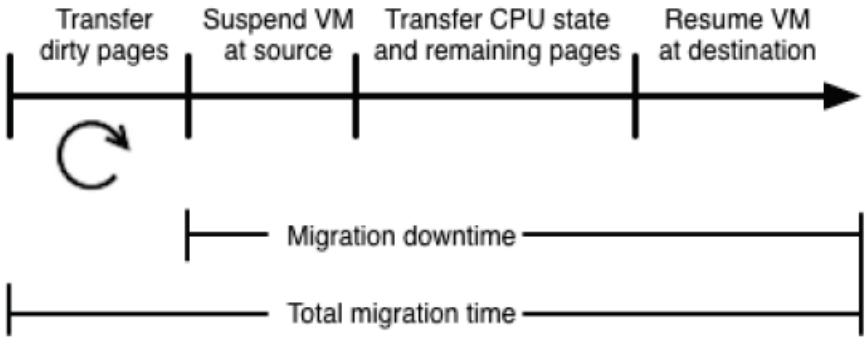


Figure 2.6: Pre-copy Live Migration

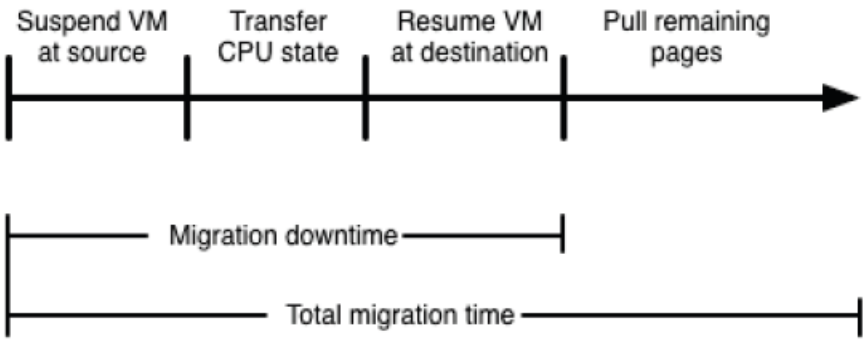


Figure 2.7: Post-copy Live Migration

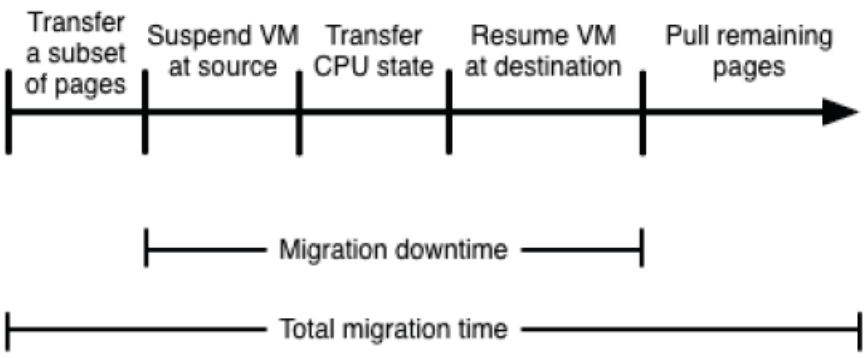


Figure 2.8: Hybrid Live Migration

2.3.2 Performance Metrics of Live Migration

The most important aspects of live migration process [24] are:

- **Total Time**
  The time taken to completely finish the entire migration process, which includes all the three phases mentioned. It is the overall time elapsed from the initiation of migration until the time when the migrated VM is resumed on the destination host and all the resources on the source are released.
- **Downtime**
  The time between suspending the execution of VM on the source host and resuming it on destination. It is equal to the Stop-and-copy phase's execution time. During this interval, the services are unavailable to the user. Downtime is one of the crucial variables included in the Service-Level Agreement (SLA) between providers and customers.
- **Data transferred**
  It is the overall transferred data to the destination host during migration process. Although the size of the Virtual Machine and the device states affect this metric, the selection of migration technique primarily determines the total data transferred. For example, in the pre-copy algorithm all pages that are modified during migration need to be resent to the destination.

It worth mentioning that for offline migration the total-time is equal to downtime. The time taken for its completion depends on network bandwidth and latency. Moreover, the data transferred is the total size of the Virtual Machine.
Furthermore, the major factors that affect migration's performance are:

- **Migration Link Bandwidth**
  The available network capacity between source and destination physical machines. Link capacity is inversely proportional to total migration time and downtime. If the network is in heavy use or in low bandwidth, the migration will take much longer. As described in section 2.3.4, QEMU hosts can configure the maximum available bandwidth for migration(*max-bandwidth* parameter).
- **Dirty Page Rate**
  Dirty Page Rate is the rate at which memory pages in the Virtual Machine are modified. The number of pages that are transferred in each pre-copy iteration are directly affected by the amount of memory that dynamically becomes dirty. Higher dirty rates result in more data that need to be sent to destination host during Iterative and Stop-And-Copy Phases. Therefore, migration total time and VM downtime last longer.

Combining the above factors, when Migration Link Bandwidth is considerably lower than the dirty page rate, the live migration process may fail, since the iterative pre-copy stage will never finish.

2.3.3 Migration requirements

Before migrating the guests, it is essential that both the source and the destination machines have identical setup. The following prerequisites regarding configuration and network topology [25] are required:

– **Shared storage**
The Virtual Machines must use a shared storage with one of the protocols Fibre Channel, iSCSI, NFS or GFS2 [26]. We use the NFS [27] storage, which allows to access files on both source and destination host in the same way local files are accessed by the users. Shared storage must mount at the same location on source and destination systems. The mounted directory names must be identical.

– **Host time sync**
Both servers must synchronize their wall-clocks timers. Using NTP [28] we can tackle this issue, but it is important that the time zones be rightly configured.

– **Network configuration**
Both hosts must have identical network configurations. Bridging issues, such as firewall configuration, and the appropriate open TCP or UDP ports must be configured prior to the migration process.

– **Host CPU types**
Host CPU types must match. Migrating from a newer-generation processor to an older-generation processor may not work due to restricted instruction set.

– **Guest machine types**
The guests involved in migration should be systems of the same version with the same updates. This requirement applies especially when migrating across QEMU versions. QEMU maintains a list of machine types that are exposed to the guest. This is made up of a lot of compatibility code for the various devices that are exposed to guests.

2.3.4 Live Migration using QEMU/KVM

The QEMU/KVM hypervisor allows either live or offline migration of Virtual Machines across hosts. Live migration in this model means moving a guest from one QEMU process to the top of another at the destination host while the guest is still running on the source host.

Similarly to pre-copy analysis mentioned in chapter 2.3.1, QEMU/KVM Live Migration happens in 3 stages [29]:

- Stage 1: Mark all RAM dirty *<ram_save_setup()>*
- Stage 2: Keep sending dirty RAM pages since last iteration *<ram_save_iterate()>*
    - stop when some low watermark or condition reached
- Stage 3: Stop guest, transfer remaining dirty RAM, device state *<migration_thread()>*

The name of the function in the QEMU source code that handles each stage is mentioned in each step. *migration_thread()* is the main function that handles the entire migration process and calls the other functions.

Regarding the infrastructure that deals with the transfer of memory and device state, QEMU uses a *QEMUFile* abstraction to be able to do migration [30]. The most important functions are the *qemu_put_buffer()*/*qemu_get_buffer()* which allow to write/read a buffer into the QEMUFile. After sending to destination host all the dirty RAM, the migration of device state happens in stage 3.

VMstate is the latest infrastructure added to QEMU code that makes the sending and receiving part device-independent. In previous versions, each device had to handle the sending and receiving part of its state itself, leaving a lot of duplicate code across all the devices in QEMU.

The speed of transfer depends on the network bandwidth. In default mode, it will use the full bandwidth. Via QEMU monitor or QMP command we can set the ***max-bandwidth*** that will be available in migration process.

Live migration algorithm in QEMU estimates the migration downtime in each iteration of Iterative Phase. The calculation is performed with the amount of data sent in the previous iteration, which is the data bandwidth. If the VM runs a resource-demanding workload, the number of dirty memory pages is high and the process of moving them is time-consuming. We should also have in mind that dirty pages always exist and there is no state of zero dirty pages on a running VM. Hence, QEMU iteratively transfers the dirty pages until the set of memory pages that has not been transferred yet is small enough that it can be transferred all at once within a chosen maximum downtime threshold. This downtime value can be determined by the ***downtime-limit*** (default 300ms) parameter on the source host before migration is triggered or can be dynamically configured throughout live migration.

A migration that doesn't finish should not be considered as faulty. The most probable reason is that the VM is changing its memory too quickly for the migration to send the data across the network connection. Command '*info migrate*' on the source returns with the current migration status and other statistics about migration process. In the aforementioned case of high dirty page rate, the status is 'active' and the throughput value ('*mbps*') most likely will be high, almost equal to the *max-bandwith* value set before migration. Given the fact that the available network bandwidth is the maximum, the possible workaround to this issue is to increase the configured *downtime-limit* value, otherwise other methods like post-copy migration or available optimizations can be used. The last solutions mentioned are out of the scope of this thesis since we cope only with the naive pre-copy algorithm.

Communication between QEMU and KVM during Live Migration

QEMU talks to the KVM via ioctl() system call through the */dev/kvm* device file exposed by the KVM kernel module. *kvm-all.c* is one of the main source files that helps QEMU communicate with KVM. VM ioctls can be considered as a basic class of ioctls of KVM API, which consists of query and set attributes that affect an entire virtual machine, for example, memory layout. In Kernel Space, KVM module holds a kernel dirty bitmap (one bit per page), which is known as *kvm_dirty_log*, and it is updated once a write-operation occurs. In this way, the corresponding bit is set to "1", meaning that the page is dirty. During the iterative phase of live migration, QEMU grabs the dirty bitmap from kernel space to keep track of the pages in the guest memory area that have been modified since the previous request of such data [29]. This snapshot of memory is saved in a kvm_dirty_log struct in userspace. A function called *kvm_physical_sync_dirty_bitmap()* in the *kvm-all.c* file is responsible for the above functionality by using the ***kvm_vm_ioctl(*KVM_GET_DIRTY_LOG*)*** system call.

Optimizations of Live Migration

Although we focus on vanilla pre-copy live migration, QEMU/KVM supports many features that compose a powerful tool for migration:

(i) Autoconverge: This feature provides a method for dynamically throttling guest CPUs in order to force the completion of busy guests' migration. As time advances, autoconverge will continuously increase the amount of guest CPU throttling until the write speed of guest is slow enough to allow the transition from stage 2 to stage 3 and pre-copy migration to finish.

(ii) XBZRLE [31]: Xor Binary Zero Run-Length-Encoding (XBZRLE) compression reduces the traffic sent over the network when migrating VMs with write-intensive workloads, because of sending compressed difference of pages between successive iterations of stages 2.

(iii) RDMA: Remote Direct Memory Access on top of modern high-speed interconnects, like Infiniband devices, ensures a much faster transfer of migration data compared to Ethernet.

(iv) Block Migration: In case of non-shared block storage, this feature allows the migration of storage along with the guest state. Block migration adds load to the network and the storage subsystems due to the huge data size that needs to send across.

(v) Multiple thread (de)compression: Instead of sending the guest memory directly, enabling this feature compresses the RAM page before sending them to the destination. Upon receiving, the data will be decompressed. Because compression consumes additional CPU cycles, the use of multiple thread compression can be helpful when the network bandwidth is very limited and the CPU resource is adequate.

2.4 Workload Characterization

Workload characterization has been addressed in several works that are related to Cloud Computing [32, 33, 34]. Live Migration of Virtual machines is strongly dependent on the number of memory accesses performed by the applications. Similar to the technical report from Sliwko, L. and Getov, V. [35] about transfer cost of VM Live Migration, we distinguish the types of workloads depending on their resource requirements, as shown in figure 2.9, and we focus on memory-intensive applications.

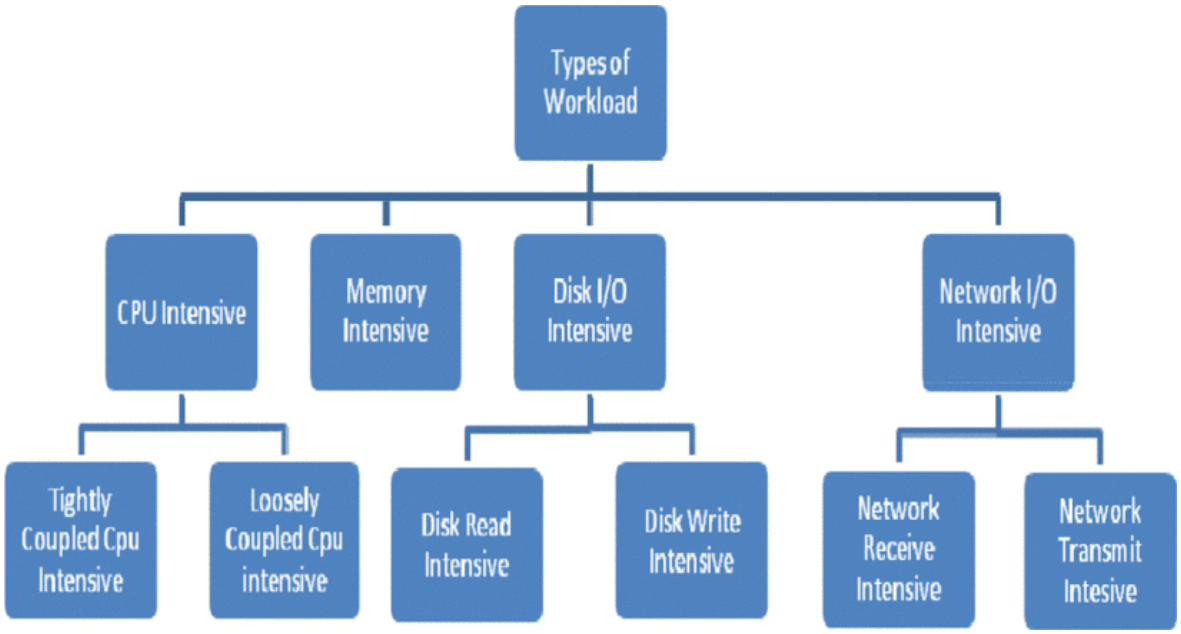


Figure 2.9: Types of Workload

2.4.1 CPU-Intensive

Workloads that the time spent in order to complete is determined principally by the speed of the CPU can be described as CPU-intensive. In such cases, the number of instructions being executed by the processor during a given period is high and the processor utilization may reach 100% usage for many seconds. Upgrading the CPU or optimizing code will improve the overall computer performance.

2.4.2 Memory-Intensive

Applications of this type perform extensive memory access spending most of its execution time on reading and writing data. The main performance bottleneck is the lack of memory(memory-bound). This indicates that more memory is needed or applications need to be managed in a more effective manner.

2.4.3 I/O-Intensive

Many workloads can be described as I/O-intensive if they spend a significant amount of time waiting for Input/Output operations to be completed. We typically associate it with disk(Disk I/O), but networking(Network I/O) or communication in general is common too. A program that looks through a huge file for some data might become I/O bound, since the bottleneck is the reading of the data from disk. The CPU must stall its operation while waiting for data to be loaded or unloaded from main memory or secondary storage.

Chapter 3

Evaluation of Pre-Copy Live Migration

The aspects of VM live migration performance have been well studied in literature [35, 36, 37, 38] and in this chapter we try to verify them by presenting the results of various tests on live migrating a Virtual Machine using QEMU/KVM pre-copy algorithm. We examine the performance of different kind of workloads running in VMs when migrating between hosts with Gigabit Ethernet interconnection. Similar to Li et al. [37], we explore the influences of the configured parameters maximum bandwidth and downtime limit along with the VM properties on the live migration performance (total migration time and downtime). Based on the experiment results, we try to find the minimum *downtime-limit* for each workload that ensures the successful completion of migration. We also make an attempt to figure out the memory write intensive applications after examining the factors that affect the performance of live migration. As Ibrahim et al. presented [39], this type of workloads is difficult to migrate because rate limits and downtime should be optimally configured in conjunction with knowledge of the application memory behavior in order to make accurate migration decisions while satisfying the SLAs. Finally, we analyze the process followed and the challenges we faced while trying to monitor the memory related features of live migration. At each step we describe our goal and demonstrate the data collected from each tool as well as their limitations.

3.1 Testbed

The setup established for evaluation consists of two hosts where each node has 2 CPUs with 4 cores, so 8 cores(model name: Intel(R) Xeon(R) CPU E5405 @ 2.00GHz) in total and 8 GB RAM. The live migration process is performed with QEMU(version 2.10.1) deployed on KVM hypervisors since hosts support CPU virtualization extensions. Moreover, the migration requirements mentioned in chapter 2.3.3 are satisfied. The hosts are connected via Gigabit Ethernet(GbE) on the same LAN and a localhost TCP transport is used for migration. The guests run *Linux*4.9.0-0.bpo.2-amd64 kernel. For most of the tests the data are collected either for small uniprocessor guests (1 vCPU and 1 GB of RAM) or a moderately sized uniprocessor guest (1 vCPU and 4 GB of RAM). In all cases, the single virtual core is pinned to a real core.

3.2 Benchmarks

In order to obtain an overview of how applications interact with the memory we develop some simple programs in C language and we also use a set of single-threaded benchmarks from Spec2006 suite and the multi-threaded application SPECjbb2005. These workloads used for evaluating the

features of live migration are described below.

3.2.1 Microbenchmarks

We require a deterministic application to measure the behavior of the live migration. For this purpose, we develope a synthetic userspace page modification microbenchmark, called *write_memory_loop*, that sequentially and circularly changes (writes to) the first byte of every memory page. The size of each page is 4KB. The allocated memory, which is equal to the Writable Working Set in this benchmark, is given by user as command-line argument. In our figures, we denote as ***write_memory_loop_xMB*** the *write_memory_loop* application that allocates and constantly dirties x MB of memory with the aforementioned way.

Furthermore, we develope an I/O intensive application that continuously writes to a file, called ***write_file***. We use the sync() function with the intention of ensuring that all pending modifications to filesystem metadata and cached file data are written to the underlying filesystems .

3.2.2 SPEC2006

SPEC2006 suite [40] is a set of single-threaded benchmarks that are widely used to evaluate the performance of computer systems. They are real world applications written in high level, portable, language (C, C++ or FORTRAN) with slight code modifications in order to minimize input/output and thus let the processor, memory and compiler be the factors under evaluation. In order to study the behavior of workloads with different dataset and characteristics, we choose the following applications:

436.cactusADM, ***450.soplex***, ***459.GemsFDTD***, ***471.omnetpp***, ***473.astar***

3.2.3 SPECjbb2005

SPECjbb2005 (Java Server Benchmark) [41] is SPEC's benchmark for evaluating the performance of server side Java. This is achieved by emulating a three-tier client/server system (with emphasis on the middle tier). SPECjbb2005 runs in a single JVM in which threads represent terminals called "warehouses". Each thread independently generates random input before performing active user posting transaction requests within a warehouse. There is neither network nor disk IO in SPECjbb2005. We set the number of threads equal to 8 and we refer to this benchmark as ***SPECjbb*** or equal to 1 thread when it is mentioned as ***SPECjbb*_1thread**

3.2.4 Idle

Idle VMs are virtual machines that remain running even though they are not used. These virtual machines consume host's CPU, memory and storage resources that could be used by other machines. Live migration of idle VMs should be considered as the baseline.

3.3 Evaluation of QEMU/KVM Live Migration Features

In this section we demonstrate the results collected by migrating an extensive number of guests. Our goal is to obtain a great overview of the naive pre-copy algorithm in accordance with the basic

optional features that have been developed for QEMU in order to allow migration to complete. Firstly, we evaluate how the memory size reserved by the Virtual Machine (*VM_Size*) affects the results obtained regardless of the application running in the VM. Afterwards, we tune the existing *downtime-limit* and *max-bandwidth* parameters of QEMU/KVM migration in order to understand their importance under different guest's workloads. Moreover, we present the importance of the moment that migration is triggered as a consequence of the varying memory consumption of the application throughout its runtime. As a final step, we estimate the minimum *downtime-limit* for each workload with the aim of maximizing the likelihood that migration can complete with a small actual downtime.

When the *max-bandwidth* reaches the physical bandwidth (1 Gbps), we notice that the throughput may fluctuate in some cases because of the usage of the same physical link for other non-migration related actions. It is suggested not to consume all the bandwidth during migration, thus in our tests we set the *max-bandwidth* equal to 800Mbps which is the 80% of physical bandwidth. Greater *max-bandwidth* values might lead to the instability of the network and degrade the performance quality.

As already stated, the pre-copy algorithm may never converge when the guest dirties the memory faster than the rate that source host can transmit the dirty pages to the destination during the iterative phase. We avoid such cases by cancelling migrations that do not reach the stop-and-copy phase after a certain timeout equal to 40 seconds. The parameter of timeout was arbitrary selected and refers to the maximum acceptable total migration time. This case is represented in our graphs with a zero value or a relatively small negative bar.

Finally, in order to ensure the data precision each of the presented experiment results are obtained via running benchmarks three times on the same configuration and for each test the values are averaged. For all tests, the application runs for a certain time(*running_time*) in the Virtual Machine before the migration is triggered. Except the *VM_Size* evaluation, all other tests are performed with a Virtual Machine with 1G memory size.

3.3.1 Memory Resource Reservation (*VM_Size*)

Firstly, we evaluate the impacts of the memory resource allocated to the guest Virtual Machine at the boot time. We refer to this feature as *VM_Size* and we perform various tests while increasing the virtual machine memory size from 512M[1] to 4G[2] using an *idle* guest. As figure 3.1 shows, the total migration time increases as the virtual machine memory size becomes larger and the downtime also increases but with a slow growth rate. This is expected because the existence of large virtual machine memory provokes more amount of dirty data required to transfer and thereby the downtime and total migration time are prolonged. Since *idle* VM is the baseline of our results, each benchmark that performs memory operations will have similar behavior regarding the increase of *VM_Size*, but with greater values of total time and downtime compared to idle VM.

[1] M or MiB for mebibytes (2^{20} or blocks of 1,048,576 bytes)
[2] G or GiB for gibibytes (2^{30} or blocks of 1,073,741,824 bytes)

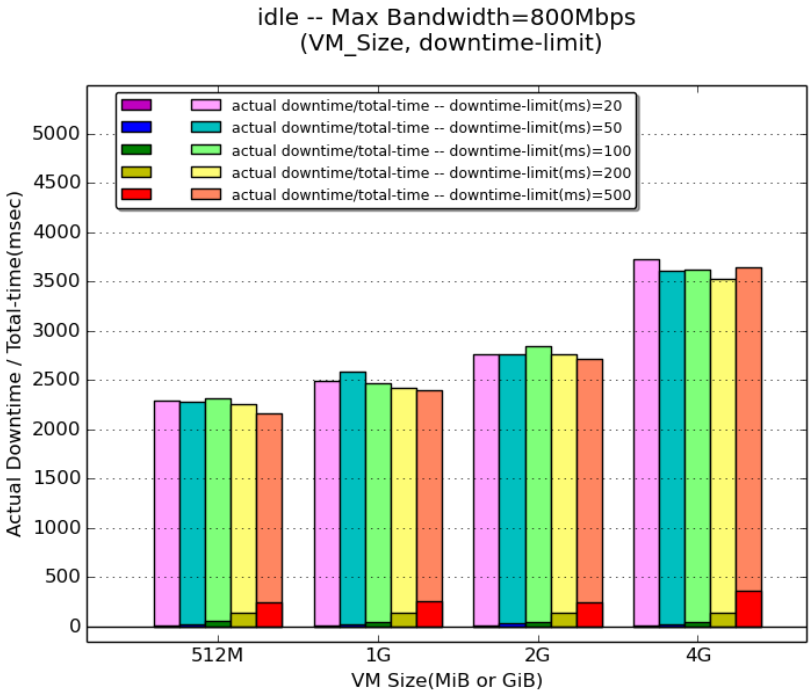


Figure 3.1: Impact of *VM_Size* in Live Migration

3.3.2 Maximum acceptable downtime (*downtime-limit*) evaluation

As already mentioned in chapter 2.3.4, the *downtime-limit* parameter controls how long a black-out period is permitted during the switchover. QEMU pre-copy algorithm measures the achievable network transfer rate and compares it to the amount of the remaining RAM to determine if it can be transferred within the configured downtime window. Therefore, assuming a stable bandwidth for τηε workload, there are *downtime-limit* values that guarantee that the guest will always suffer from the maximum downtime blackout.

The results presented in figure 3.2 show that the VM running the *write_memory_loop_256MB* microbenchmark with *downtime-limit* values lower than 2500 msec never completes the migration from one host to another when the network transfer rate is up to 800Mbps(*max-bandwidth*). The missing values from 2000 to 2500 msec might result in successful migration in some cases, but this is not guaranteed as the available bandwidth may fluctuate around the given *max-bandwidth* limit.

Another important observation from live migration results in figure 3.2 is the fact that QEMU/KVM algorithm targets to optimize the total time of migration within the given *downtime-limit*. Thus, when we increase the maximum acceptable downtime, the actual total-time is decreasing while the actual downtime is becoming higher but adhering to the *downtime-limit* threshold as shown for values from 2500ms to 5000ms. Further increase of *downtime-limit* value does not provide any enhancement as the actual downtime remains at about 4000msec and is almost equal to total migration time.

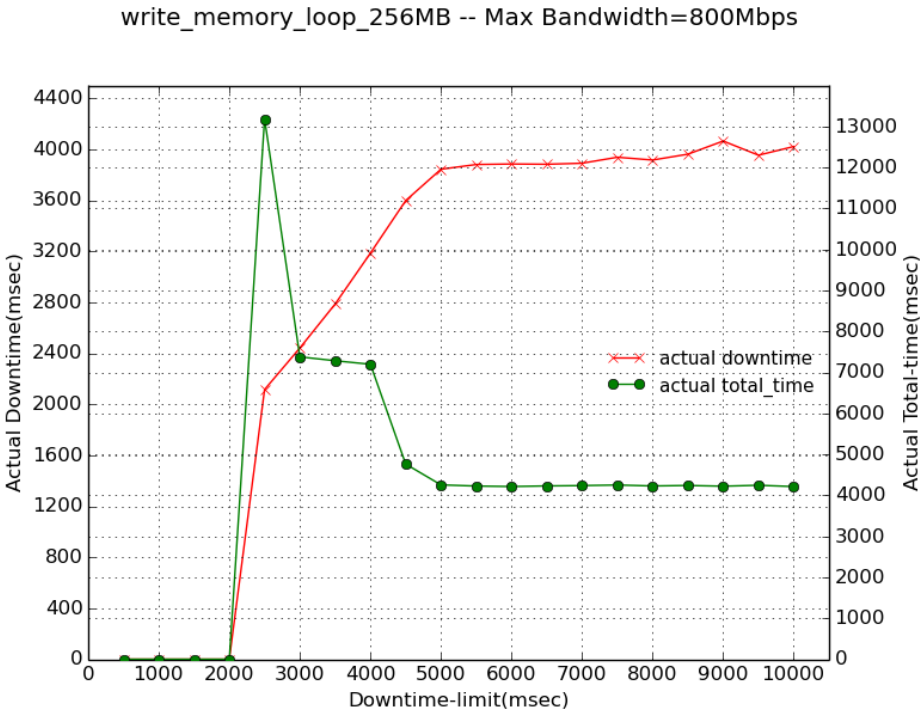


Figure 3.2: Impact of *downtime-limit* in Live Migration

3.3.3 Maximum available bandwidth (*max-bandwidth*) evaluation

In the next test we evaluate the influence of the available bandwidth for transferring the required data for migration. We start from setting the *max-bandwidth* parameter equal to 200 Mbps until we reach the upper bound of 1000 Mbps with a step of 200 Mbps.

The results are presented in figure 3.3 with a VM running the *write_memory_loop_256MB* microbenchmark and for each maximum bandwidth we also configure the *downtime-limit* parameter. It seems that the pair (*max-bandwidth*, *downtime-limit*) is important for a migration to successfully converge. In case of low *max-bandwidth* values, such as 200 Mbps or 400 Mbps, the migration fails even for large maximum acceptable downtimes. In these cases the dirty pages are generated faster than the network transfer capacity. As the bandwidth increases, we can observe that for some *downtime-limit* values the dirty page rate can be covered by the network transfer rate. Thus, if one aims to maximize the chances of getting a successful migration, the first attempt should be to maximize the network bandwidth available to the migration operation. Afterwards, as we presented in the previous figure 3.2, with respect to the SLA with the cloud customer we can tune the *downtime-limit* in order to achieve the optimal total migration time.

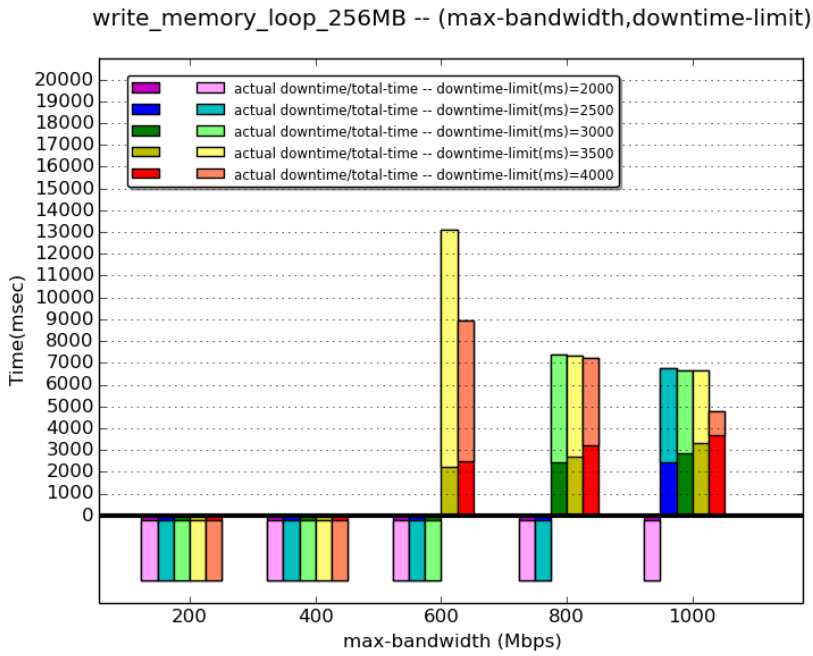


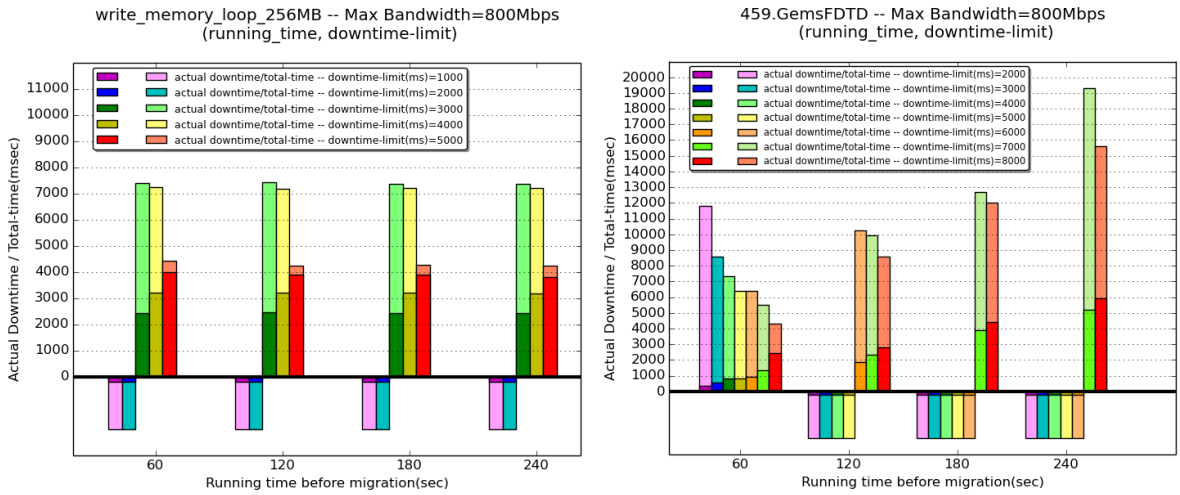
Figure 3.3: Impact of *max-bandwidth* in Live Migration

3.3.4 Evaluation of the moment triggering live migration

The utilization of resources by the same application may vary during its execution according to its execution phase or load (e.g. in the case of a web service). As a result, we perform some tests for evaluating the influence of the exact moment that migration is triggered since most real-world applications dynamically change their memory accesses during runtime. In this test we start the migration after 60,120,180 and 240 seconds of runtime of the application. If migration is not completed after 40 seconds, then it is cancelled by the host and it is considered unsuccessful.

As presented in figure 3.4a the precise moment of the start of migration of *write_memory_loop* microbenchmark results in totally the same actual total-time and downtime for all the *downtime-limit* values. This behavior can be explained as the microbenchmark has a constant dirty page rate and the same amount of data need to be transferred in each iteration. However, as figure 3.4b indicates, for the exact configuration of migration parameters, the results produced by real world workloads, such as *459.GemsFDTD* from the SPEC suite, may differ when the migration is being started at different moments in time. For example, the beginning of migration after 240 sec of runtime leads to 5 out of 7 unsuccessful migrations compared to runtime of 60sec. For the completed migrations with *downtime-limit* 7000msec and 8000msec, the actual total-time was 3.5 times larger while the actual downtime 2.5-3.5 times larger. A possible explanation is the fact that during the runtime there are moments that the dirty pages are generated very quickly and other moments that are generated in lower rate, e.g. when they perform writes with data from the same page.

Since the results of *running_time*=60 and *running_time*=120 differentiate significantly in some cases, we handle them as if they were separate workloads that produce their own migration results.



(a) Microbenchmark **(b)** Real-world application

Figure 3.4: Impact of the exact moment migration is triggered

3.3.5 Minimum *downtime-limit* for successful migration

As a last step of our evaluation we try to find the minimum *downtime-limit* value that leads to a successful live migration. Consequently, we perform a wide range of migrations in order to find the optimal value of *downtime-limit* while a workload runs in a VM and the migration is triggered after *running_time*=60 and *running_time*=120. Aiming to obtain the most accurate results, all tests are executed 5 times and the actual total-time and downtime values for completed migrations are averaged.

As presented in figure 3.5 for each workload there are 3 different categories of results:

- Unsuccessful migration: For the configured *downtime-limit* of y axis values, the migration is always cancelled after 40 seconds because of not reaching the condition for convergence[3].
- Not always successful migration: For the configured *downtime-limit* of y axis values, the migration is not completed within the acceptable migration time of 40 seconds for at least 1 out of 5 migration attempts.
- Successful migration: For the configured *downtime-limit* of y axis values, the migration is successfully completed within the acceptable migration time of 40 seconds for all(5/5) the migrations performed. Above the points of this category, we remark the actual total-time and actual downtime measured values in the form of X / Y .

In the following table 3.1 we present the minimum *downtime-limit* required for a migration to complete. As we can notice, for most of the benchmarks of SPEC suite, the minimum acceptable downtime for successful migration differs between the runtimes tested.

[3] The *Iterative Phase* of source-to-destination memory transfer will converge when the set of memory pages that has not been transferred yet will be small enough that it can be transferred all at once within a chosen maximum downtime threshold

Benchmark	*downtime-limit*(msec)	
	rt60	**rt120**
idle	20	20
write_file	20	20
write_memory_loop_16MB	200	200
write_memory_loop_32MB	400	400
write_memory_loop_64MB	700	700
write_memory_loop_128MB	1400	1400
write_memory_loop_256MB	2600	2600
write_memory_loop_512MB	5200	5200
SPECjbb (1 thread)	600	600
SPECjbb (8 threads)	2400	2400
436.cactusADM	20	20
450.soplex	80	150
459.GemsFDTD	2800	6000
471.omnetpp	900	900
473.astar	1800	200

Table 3.1: Minimum *downtime-limit* for successful migration (*max-bandwidth*=800Mbps)

Based on the overall results presented in this chapter as well as the workload characterization in chapter 2.4 we try to remark some of our benchmarks as memory write intensive. Many of our workloads seem to have relatively large value of minimum *downtime-limit* for a successful migration because of extensive memory accesses. Furthermore, keeping in mind that the default *downtime-limit* of QEMU/KVM is 300ms, values such as 2000msec are comparatively much higher. Thus, the benchmarks *write_memory_loop* (especially for memory sizes greater than 128MB), *SPECjbb* and *459.GemsFDTD* can be characterized as **memory-intensive**.

3.4 Memory-related Live Migration Features

After having executed a various amount of live migrations with different applications, as shown in this chapter, we have an overview of the aspects that are involved in the process of live migration and especially those related to the memory behavior of the application running in VM. We focus on the following features:

(i) The entire memory of a VM is transferred from one host to another, so the total allocated memory reserved by the VM(**VM_Size**) affects the migration.

(ii) The **Dirty Page Rate**, which is defined as the amount of memory that changed its status from clean to dirty per second during a specific interval. Since in the pre-copy algorithm the source host iteratively sends the modified pages to the destination host, this feature is required in order to check if the condition for convergence is met in *Iterative Phase* of live migration.

(iii) The total number of pages dirtied during the whole migration process, which is defined as Writable Working Size (**WWS**) and is equal to the upper bound of the unique pages dirtied.

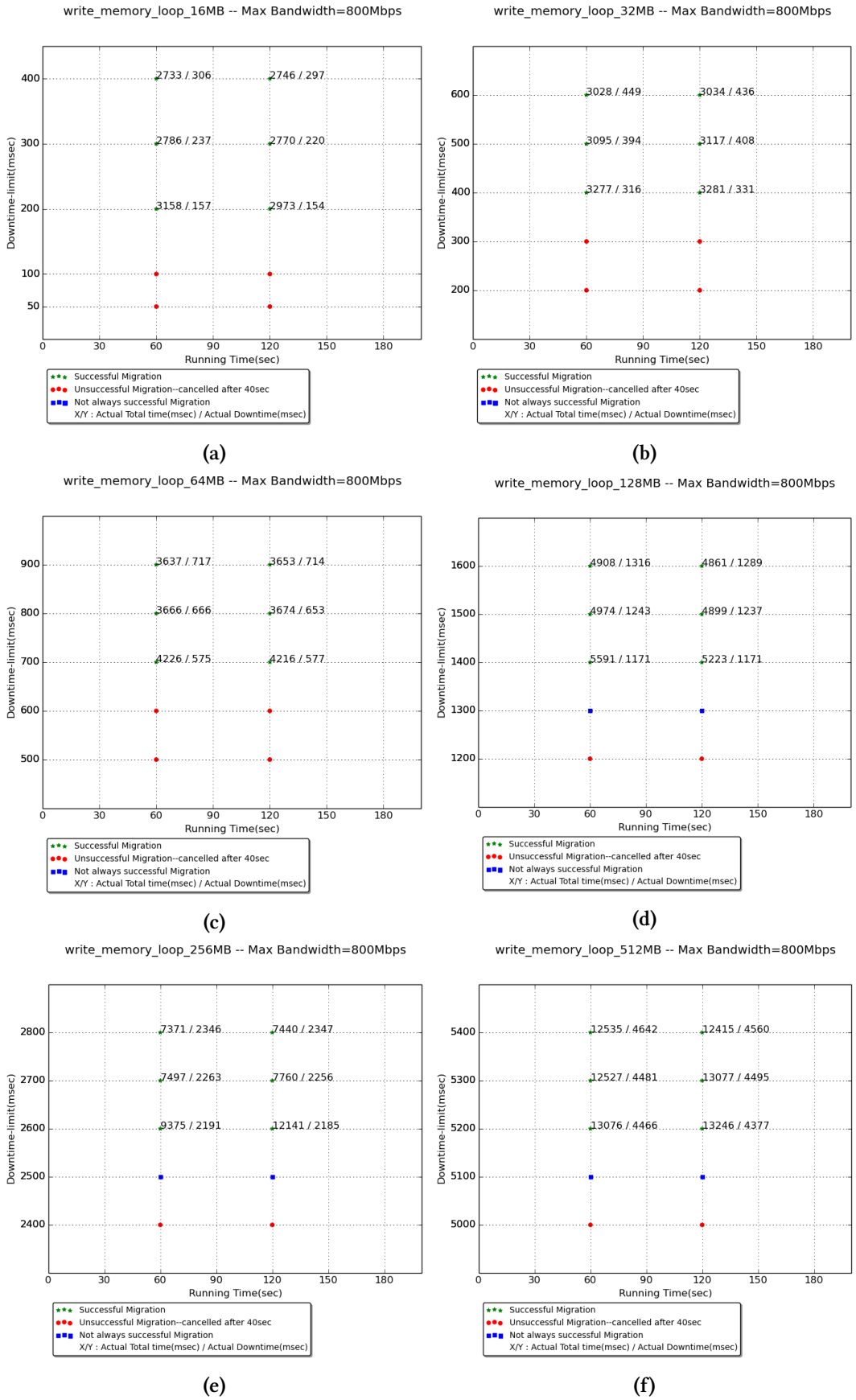


Figure 3.5: Minimum *downtime-limit* for successful Live Migration

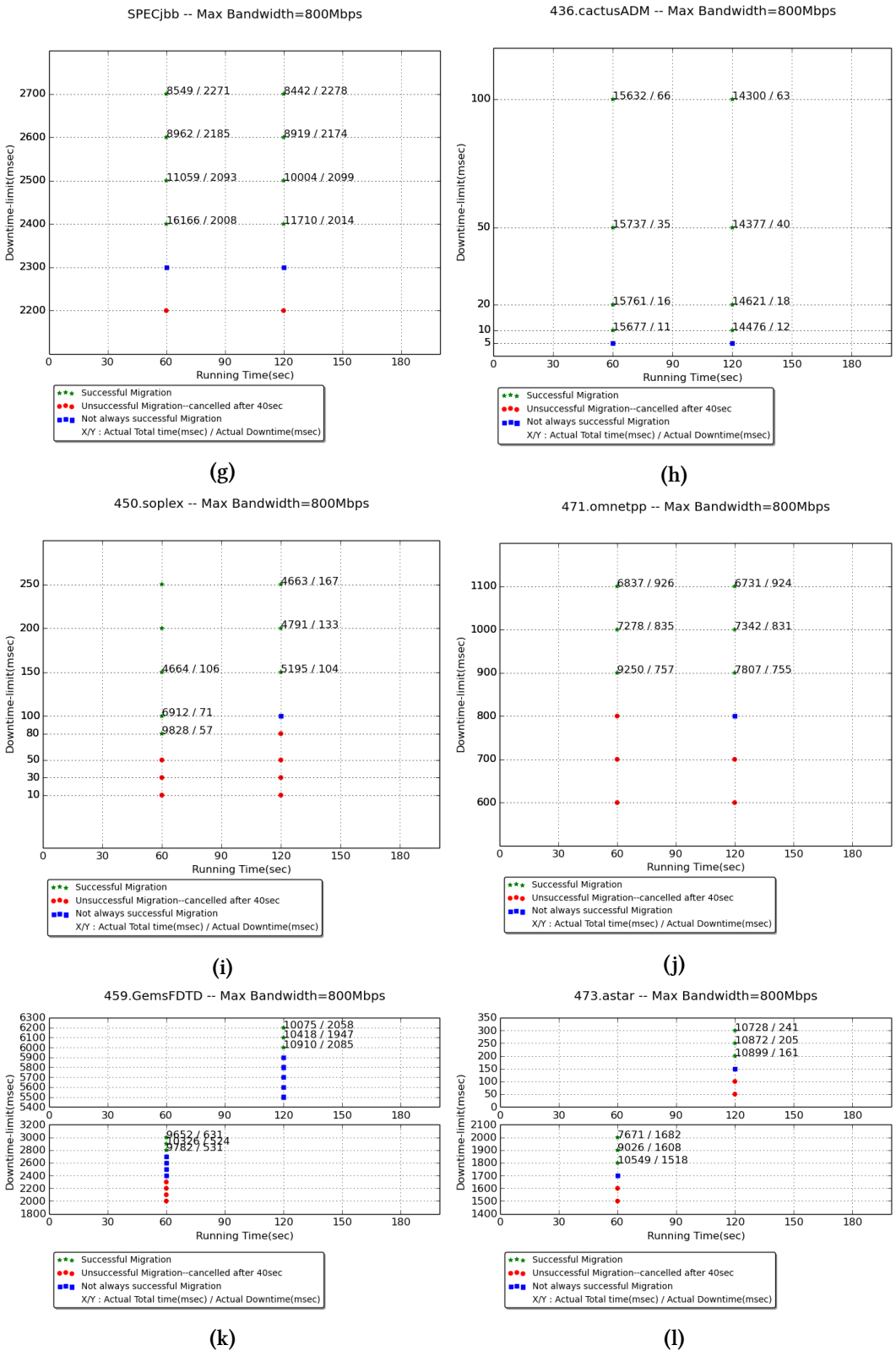


Figure 3.5: Minimum *downtime-limit* for successful Live Migration (cont.)

3.5 Memory Monitoring Tools

In order to identify the memory related features that affect migration, and mainly the Dirty Page Rate, the WWS and the *VM_Size*, we use some tools provided from Linux Operating System (OS), such as *perf* and *iowait* counters. In this way, we successfully distinguish I/O intensive. However, resource-demanding applications cannot be ordered using the data collected from these tools. We also unsuccessfully try to predict application's memory behavior using simple synthetic programs.

3.5.1 Cache misses and MPKI using Performance Counters

A first approach is to collect and analyze performance and trace data using Performance Counters provided by Linux. *Perf* [42, 43, 44] is a profiler tool with simple CLI that exposes to the user information collected from performance counters. There are a lot of events that can be traced via the *perf* command. Broadly speaking, these events can be **software events** such as context switches and page faults, or **hardware events** that originate from the processor itself, like cache misses (all levels), or number of clock cycles. The number and the kind of performance events are issued from Performance Monitoring Units (PMUs) hardware. Performance counters support two usage modes, counting and sampling. In counting mode, the counters are initialized and provide the total number of events during a certain interval. ("*perf stat*" command). In sampling mode, the counters are reported as aggregated samples. Each sample is accumulated upon an interrupt occurs after certain number of events ("*perf report*" command). The specific underlying events may differ between processors and this is a main concern regarding their use.

Firstly, we investigate whether the *page-fault* counter could give us an overview of the number of writes performed by the workload running in a Virtual Machine. We use *perf* in counting mode because we want to measure all the desired events and not just a sample of them. Moreover, we add as a parameter of "*perf stat*" command the interval (*-I msecs, –interval-print msecs*) in order to specify how often the performance counters are printed. At the beginning of each interval, the counters are reset. The results collected regarding the page-fault events provide no information on our workload's memory accesses. The only useful data derive from the execution of the "*perf stat*" command alongside with the application in the VM. The sum of the page-fault counters at the beginning of the execution could give us an estimation of the application's WWS. However, our goal is to measure the guest's memory writes from hypervisor without affecting its operation at all. We should keep in mind that a real page fault happens when the entry is missing (or is otherwise protected) in the primary page tables. Such faults are reflected back to the OS which knows how to handle them properly. Faults that are caused by the OS manipulating page tables or by the absence of an entry in the shadow page tables are serviced by the hypervisor.

As a next step, we try to prove if memory writes of application are associated with the number of memory accesses that could not be served by any of the caches. This is the Performance Counter (PC) *cache-misses* that *perf* tool provides, and it can be related to the pages transferred from the main memory upon request. This hardware event can be combined with "*instructions*" counter, which is also collected from "*perf stat*", in order to count the MPKI (Misses per Kilo Instructions). Supposing that we gather the required counters every second for n samples, MPKI can be calculated with the following equation:

$$MPKI = \frac{\sum_i^n cache\text{-}misses[i]}{\sum_i^n instructions[i]} \cdot 1000$$

As it can be seen from figure 3.6, there is no actual correlation between MPKI and memory writes, as memory-intensive applications can have low value of MPKI (e.g. *write_memory_loop_xMB* applications) and vice versa, large MPKI values do not mean workload with low dirty page rate (e.g. *450.soplex_rt60* or *450.soplex_rt120*). This metric can only give us an indication how well the cache is working; the lower the ratio the better. In addition, the configuration of the cache may differ between processor models. Depending on the architecture, the software versions and system configuration, the PMU hardware may not be supported by *perf* utility in some KVM guest machines. These issues make the direct comparison of event counters between processors difficult.

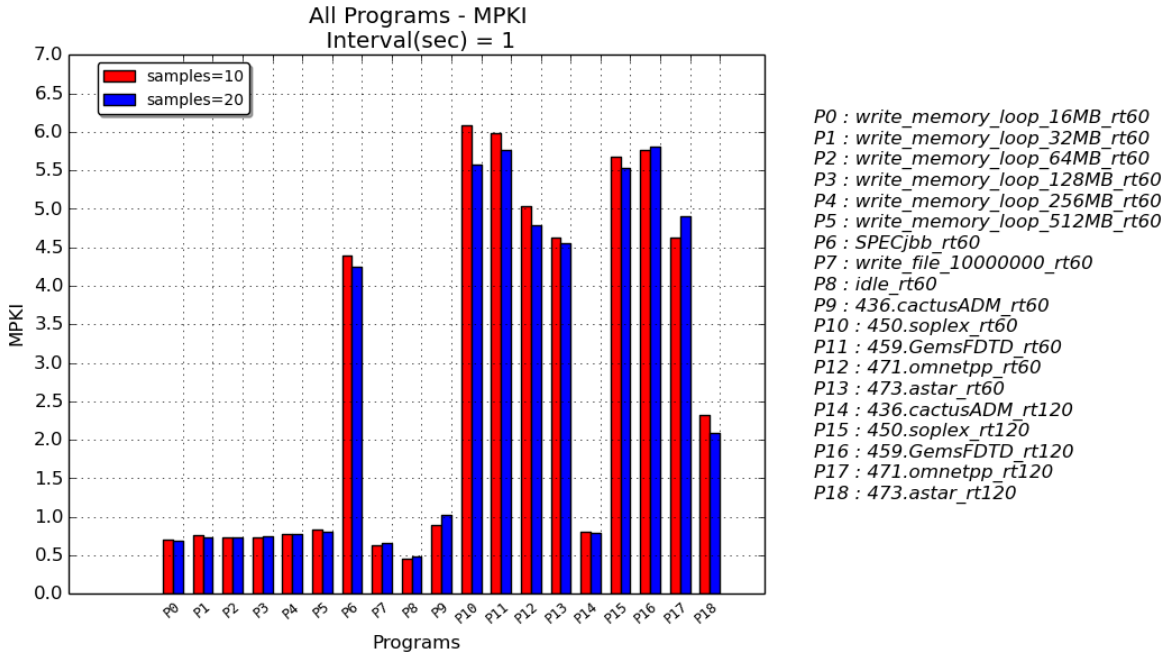


Figure 3.6: MPKI using *cache-misses* from *perf* tool

3.5.2 IOwait

IOwait, as defined by man page of *iostat* [45] command, is the percentage of time that the CPU or CPUs were idle during an outstanding disk I/O request. Therefore, iowait means that from the CPU point of view, no tasks were runnable, but at least one I/O was in progress. In general, a CPU can be in one of four states: user, sys, idle, or iowait. The kernel tracks each of these states using counters. On each clock interrupt, the kernel checks the CPU state and increments the appropriate counter. The user can check the counters in */proc/stat* file.

Since migration's convergence is affected primarily by the dirty page rate, I/O-intensive workloads have almost zero downtime. IOwait measurement can be extremely useful during profiling period as we can easily detect this kind of applications and hence get an overview of their expected downtime when migration occurs. However, this tool cannot provide information for memory-intensive workloads which is the main category that affects migration process. Furthermore, this

counter might contain the time spent for network traffic that affects the accuracy of measured values.

3.5.3 Tuning dirty page rate with microbenchmarks

The last approach takes care of the implementation of some new microbenchmarks, simple synthetic programs that allow us to control the allocated memory and the dirty page rate by calling the sleep function after specific amount of memory writes. We try to predict the memory usage patterns of a wider range of applications. For some benchmarks, despite the fact that the dirty page rate is approached, we do not get the expected migration results. By tuning the dirty page rate, the total-time and the transferred bytes comes close, but the actual downtime of migration is much lower than the one of the real workloads. During sleep time, no data are written in RAM and this is the reason for the fast convergence of migration algorithm. Another disadvantage of this approach is that most of real-world applications do not access memory in the same way during their execution and the dirty page rate can alter significantly from time to time. These microbenchmarks are not able to predict this behavior.

Chapter 4

Design and Implementation of BitmapTrace mechanism in QEMU/KVM

The aforementioned approaches described in previous section (3.5) could not be applied for memory-intensive applications in order to obtain a clear view of the memory footprint that would help us estimate migration times to decide on the migration order. Therefore, we implement a module in QEMU/KVM called BitmapTrace which efficiently profiles the required features. In this chapter, we discuss the design and implementation of this mechanism and then we evaluate its parameters by running tests with various workloads (3.2). We also aim at identifying the optimal parameter combinations that ensure an adequate profiling period. Finally, we analyze the performance overhead introduced by BitmapTrace.

4.1 Design

The existing Linux monitoring tools do not provide us with enough information to form an opinion about the memory behavior of workloads. Consequently, we implement a mechanism, called BitmapTrace, as part of QEMU (version 2.10.1) using the code of KVM migration feature, but with minor modifications. We intend to perform a "shadow" migration that takes a snapshot of the dirty bitmap by getting advantage of *KVM_GET_DIRTY_LOG* ioctl() before migration. This module is responsible for creating and maintaining the working-set behavior that records the memory footprint. While VM is executing, we store the new dirty pages that are produced at periodic intervals. Each iteration lasts for *"period"* time, where the number of iterations and the period are defined by the user. These pages are stored in an array with size equal to the number of iterations in order to have a snapshot of the memory dirtying behavior of a VM. When BitmapTrace finishes the memory profiling the results are logged to a file for future use.

For our research purpose, the functionality of the BitmapTrace module outweighs the existing Linux tools mentioned in chapter 3.5. Specifically, iteration zero of the final results provides us with the total number of pages allocated for the Virtual Machine (***VM_Size***), which is equal to the page-faults counter of *perf* results. This value indicates that in the first iteration the whole memory is considered dirty. Furthermore, I/O-intensive workloads can be easily tracked, because the new dirty pages remain consistent per iteration and equal to the minimum value, which is almost the same one as idle VM. Therefore, by gathering the logs issued from BitmapTrace, we can have a detailed snapshot of applications' memory **Dirty Page Rate** per iteration for a certain profiling period. BitmapTrace mechanism can also compute the **WWS** when the *"period"* argument given as input is a relatively large value, e.g. 3000 msec or greater. As we present this case further down, the profiling

window per iteration is long enough and the unique pages dirtied in this interval are logged before clearing the dirty bitmap and moving to the next iteration for recalculating the new dirty pages.

4.2 Implementation

Since live migration feature relies on recording dirty memory pages so that they can be resent if they change their state during migration process, we leverage KVM's ioctl(*KVM_GET_DIRTY_LOG*) system call in order to keep a memory footprint of the new dirty pages in the interval between two consecutive moments, which is equal to *"period"* time. The total profiling period is computed by multiplying the number of *"iterations"* by the *"period"* time. Both are specified by the user when the BitmapTrace mechanism is triggered by the relevant HMP or QMP command. In general, the BitmapTrace module reuses many features of QEMU's migration code and especially the part of creating, clearing and synchronizing the migration's dirty bitmap, which is held at the *ram.c* file. This explains why the core of our module was implemented inside this file.

QEMU migration uses RAMState struct to track the state of RAM during migration and many members of this struct are related to the QEMUFile and RAMBlocks. As already mentioned in section 2.3.4, the former represents a buffer that dirty memory pages are sent before being transferred to the destination through the socket connection, and the latter are in a global list called RAMList, which, except RAMBlocks, holds the dirty memory bitmaps [46] as well. Since we are only interested in tracking the number of memory writes for a given (iterations, period) pair, we create a BitmapTraceState struct similar to RAMState but we keep only the members related to dirty pages and bitmap. Moreover, members of RAMState that deal with migration optimizations are out of scope of the current implementation. The BitMapTraceState struct is defined below.

```
1       struct BitmapTraceState {
2           int state;
3           FILE *pFile;
4           int64_t current_period;
5           int64_t current_iteration;
6           int64_t iterations;
7           int64_t time_last_bitmap_sync;     /*last time we did a full
    bitmap_sync*/
8           uint64_t num_dirty_pages_period;  /*number of dirty pages since
    start_time(per iteration)*/
9           uint64_t bitmapTrace_dirty_pages; /*number of dirty bits in the
    bitmap*/
10          uint64_t *dirtyPages_array;
11          QemuMutex bitmap_mutex;           /*protects modification of the
    bitmap*/
12          QemuThread thread;
13
14          bool bitmapTrace_thread_running;
15
16          /* The last error that occurred */
17          Error *error;
18        };
19
```

Listing 4.1: BitmapTraceState Struct

The BitmapTrace mechanism is executed from a separate thread, which is responsible for logging the results into a file and the total process can be divided into three phases – ***bitmapTrace_init***, ***bitmapTrace_dirty_bitmap_sync***(in *bitmap_trace_thread* function) and ***bitmap_trace_thread_end***. Once the appropriate HMP or QMP command is given in QEMU, *bitmapTrace_init()* creates the BitmapTraceState struct and initializes its members. A new thread executes the *bitmap_trace_thread()* function which, after marking all RAM dirty similar to initial stage of migration algorithm, divides the total profiling period into equal sized intervals. In each interval the *bitmapTrace_dirty_bitmap_sync()* function is called which reuses the QEMU's migration code for capturing the dirty bitmap from KVM through ioctl(*KVM_GET_DIRTY_LOG*) system call. Hence, we identify the dirtied memory regions per iteration and store them in *dirtyPages_array* which, in the final step of *bitmap_trace_thread()*, is exposed to QEMU monitor as well as to a file whose path is given by the user (*"filename"* parameter). Finally, *bitmap_trace_thread_end* marks the end of logging period and deallocates the memory used by BitmapTraceState stuct.

BitmapTrace implementation code is available in https://github.com/dmtrskal/QEMU-BitmapTrace.

4.3 Basic Usage

The arguments for using BitmapTrace's commands are the following:

- state: trace enabled or disabled (on/off) (**mandatory**)
- filename: name of the file where the dirty pages per iteration will be logged (**mandatory**)
- iterations: number of times the new dirty pages will be logged. The min and max values are 3 and 100000 respectively. The default value is 3, if the "iterations" parameter is not stated.
- period: time interval in milliseconds between two consecutive iterations. The min and max values are 10 and 100000 respectively. The default value is 10, if the "period" parameter is not stated.

QMP command example:

(i) Verify that QEMU started in KVM mode as well as QMP on a TCP socket, so that telnet can be used :
$ qemu [...] -enable-kvm [...] -qmp tcp:localhost:4444,server,nowait

(ii) Run telnet :
$ telnet localhost 4444

(iii) We should see QMP's greeting banner :
{"QMP": {"version": {"qemu": {"micro": 0, "minor": 6, "major": 1}, "package": ""}, "capabilities": []}}

(iv) Issue the *qmp_capabilities* command, so that QMP enters command mode :
{ "execute": "qmp_capabilities" }

(v) We can now issue commands. So, to enable BitmapTrace mechanism, we should issue *bitmap-trace* with the appropriate arguments, as shown below:
{ "execute": "bitmap-trace", "arguments": { "state": true, "filename": "/home/user/dirty_pages_profiling.log", "iterations": 10, "period": 1000 } }

(vi) Once BitmapTrace is finished successfully, results are displayed on HMP and are logged to filename specified.

QEMU Monitor (or HMP) command example:

(i) Verify that QEMU started in KVM mode :
$ qemu [...] -enable-kvm

(ii) With *"help bitmap-trace"*, we can get details about the command:
(qemu) help bitmap-trace
bitmap-trace state filename iterations period – Enable/Disable the bitmap tracking mechanism

filename: file where the dirty pages per iteration will be logged

iterations: number of times the dirty pages will be logged

period: time interval in milliseconds between two consecutive iterations

(iii) Enable BitmapTrace mechanism, by issuing *"bitmap-trace"* with the appropriate arguments:
(qemu) bitmap-trace on "/home/user/dirty_pages_profiling.log" 10 1000

(iv) Once BitmapTrace is finished successfully, results are displayed on HMP and are logged to filename specified.

4.4 Evaluation of BitmapTrace mechanism

In the following subsections we evaluate the BitmapTrace mechanism. We try to measure the WWS of each application and then we tune the mechanism's parameters in order to identify the optimal values that guarantee an adequate profiling period. Our tests are implemented using all benchmarks described in chapter 3.2. BitmapTrace is enabled after 60 seconds runtime for all applications, while for SPEC benchmarks the mechanism is not only enabled after 60 seconds but also after 120 seconds runtime. They are used in the same way as live migration tests and each case is denoted as *rt60* or *rt120*.

The memory size of each Virtual Machine (*VM_Size*) used in our test is 1G, so at the end of the first iteration of BitmapTrace 266450 pages are dirtied, similar to pre-copy migration algorithm where the whole memory is considered dirty at the beginning. However, this value is not depicted in our figures for consistency reasons due to the large difference with the other measured values.

4.4.1 WWS Estimation

Tuning the *"period"* parameter of BitmapTrace mechanism enables the host to evaluate the Writable Working Set(WWS) of the workload running in the VM. We perform various tests for our benchmarks with relatively large values for *"period"*(1000,3000,5000,8000,10000 msec) while we

kept the *"iterations"* equal to 10. The new dirty pages per iteration for each *"period"* are displayed in figure 4.1 . Despite the fact that the profiling time varies in each case, this test is intended to find the WWS of its application. As we can see, *"period"* values higher than or equal to 3000 msec provide the required information about the WWS for almost all workloads, as greater values of *"period"* return the same results. Furthermore, it is evident that the WWS of *write_memory_loop_xMB* microbenchmarks is doubled from 128MB to 256MB and from 256MB to 512MB. I/O-intensive workloads do not consume any memory as expected and their WWS is like *idle*'s VM. Finally, we can observe the influence of the moment that migration is started(*running_time*) as the WWS differs significantly for some benchmarks, such as *436.cactusADM* and *473.astar*.

4.4.2 Tuning *period* parameter

As a second evaluation of our trace mechanism, we try to tune the *"period"* parameter in order to find an optimal value for tracking the memory footprint of the guest during the whole profiling period. Due to the observations of previous section about WWS, we do not want a large *"period"* as the new dirty pages will have the same value after each iteration which will be equal to the WWS of the application. Thus, as a first evaluation, the required *"period"* should be lower than 3000 msec. We perform various tests where the *"period"* has the following values: 10,50,100,200,500,1000,1500,2000 or 2500 msec and the *"iterations"* parameter is configured so that the total profiling period is 20000msec in each case. The results for each benchmark are presented in figure 4.2 and we can observe that low *"period"* values, such as 10,20,100,200 msec restrict the applications' memory write rate, as larger intervals between the iterations allow the benchmarks to perform more memory accesses. On the other hand, for memory "hungry" applications such as *write_memory_loop_xMB* and *459.GemsFDTD*, choosing large *"period"* values(2000 or 2500 msec) may result in new dirty pages per iteration equal to the WWS, as depicted in figures 4.1, which should be avoided. Therefore, we suggest choosing *"period"* values that neither conceal the information provided about the actual memory behavior of the of the application nor return the WWS after each iteration.

In addition, since BitmapTrace mechanism performs a "shadow" migration using most part of QEMU's migration code, we should have in mind that QEMU resets the dirty pages counters at the first time after 1000msec when performing a bitmap sync. The corresponding part of code from QEMU's *migration/ram.c* file is shown in listing 4.2.

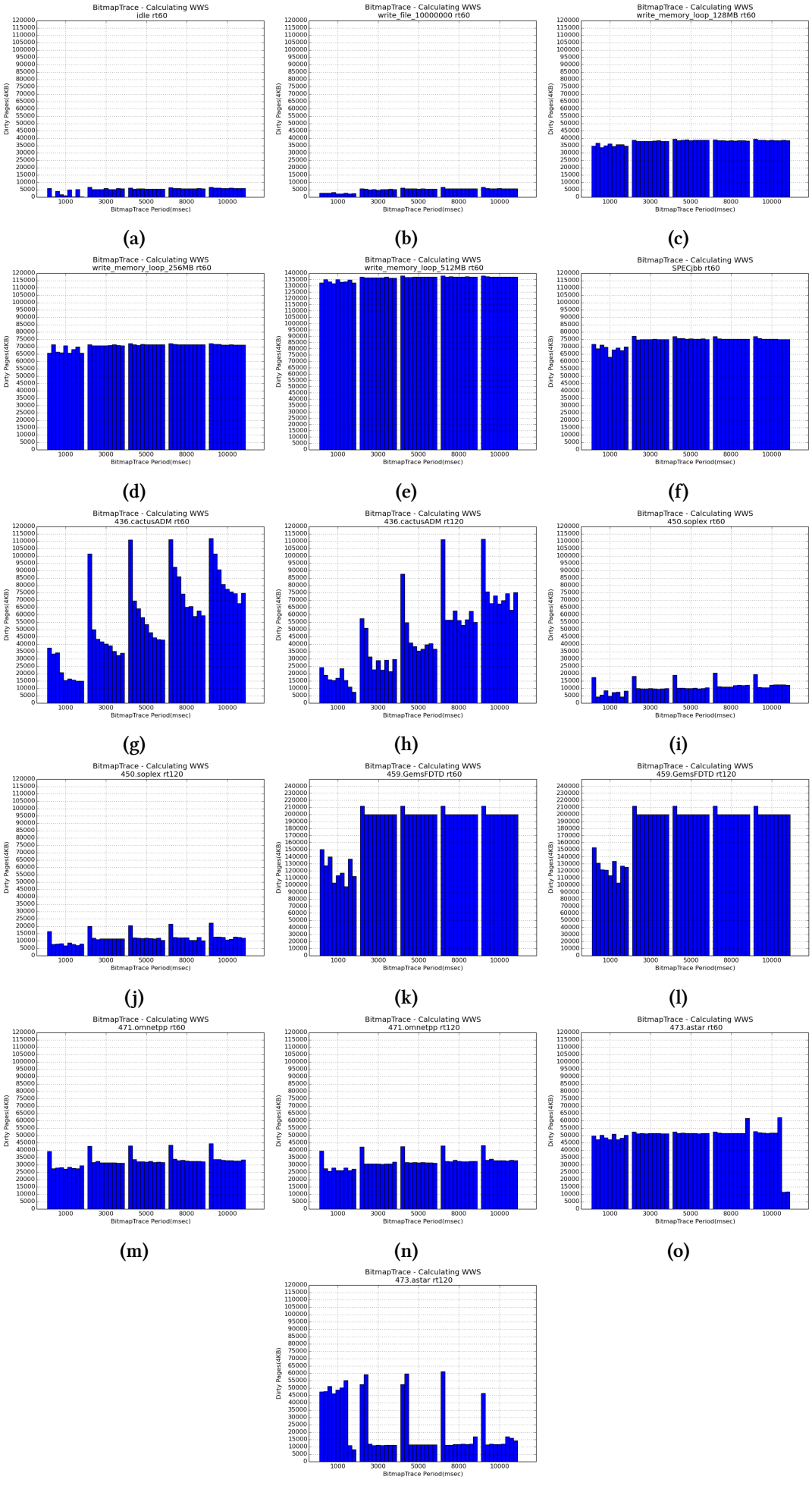


Figure 4.1: WWS Evaluation

```
1   static void migration_bitmap_sync(RAMState *rs)
2   {
3       RAMBlock *block;
4       int64_t end_time;
5       uint64_t bytes_xfer_now;
6
7       ram_counters.dirty_sync_count++;
8
9       if (!rs->time_last_bitmap_sync) {
10          rs->time_last_bitmap_sync = qemu_clock_get_ms(QEMU_CLOCK_REALTIME);
11      }
12
13      trace_migration_bitmap_sync_start();
14      memory_global_dirty_log_sync();
15
16      qemu_mutex_lock(&rs->bitmap_mutex);
17      rcu_read_lock();
18      RAMBLOCK_FOREACH(block) {
19          migration_bitmap_sync_range(rs, block, 0, block->used_length);
20      }
21      rcu_read_unlock();
22      qemu_mutex_unlock(&rs->bitmap_mutex);
23
24      trace_migration_bitmap_sync_end(rs->num_dirty_pages_period);
25
26      end_time = qemu_clock_get_ms(QEMU_CLOCK_REALTIME);
27
28      /* more than 1 second = 1000 millisecons */
29      if (end_time > rs->time_last_bitmap_sync + 1000) {
30          /* calculate period counters */
31          ram_counters.dirty_pages_rate = rs->num_dirty_pages_period * 1000
32              / (end_time - rs->time_last_bitmap_sync);
33          bytes_xfer_now = ram_counters.transferred;
34          ...
35
36          /* reset period counters */
37          rs->time_last_bitmap_sync = end_time;
38          rs->num_dirty_pages_period = 0;
39          rs->bytes_xfer_prev = bytes_xfer_now;
40
41          ...
42  }
43
```

Listing 4.2: *migration_bitmap_sync* function in QEMU migration code

Therefore, the suggested *"period"* values for using BitmapTrace mechanism are 1000 or 1500msec and these values will be used in the next sections as well as in our tests in chapter 5.

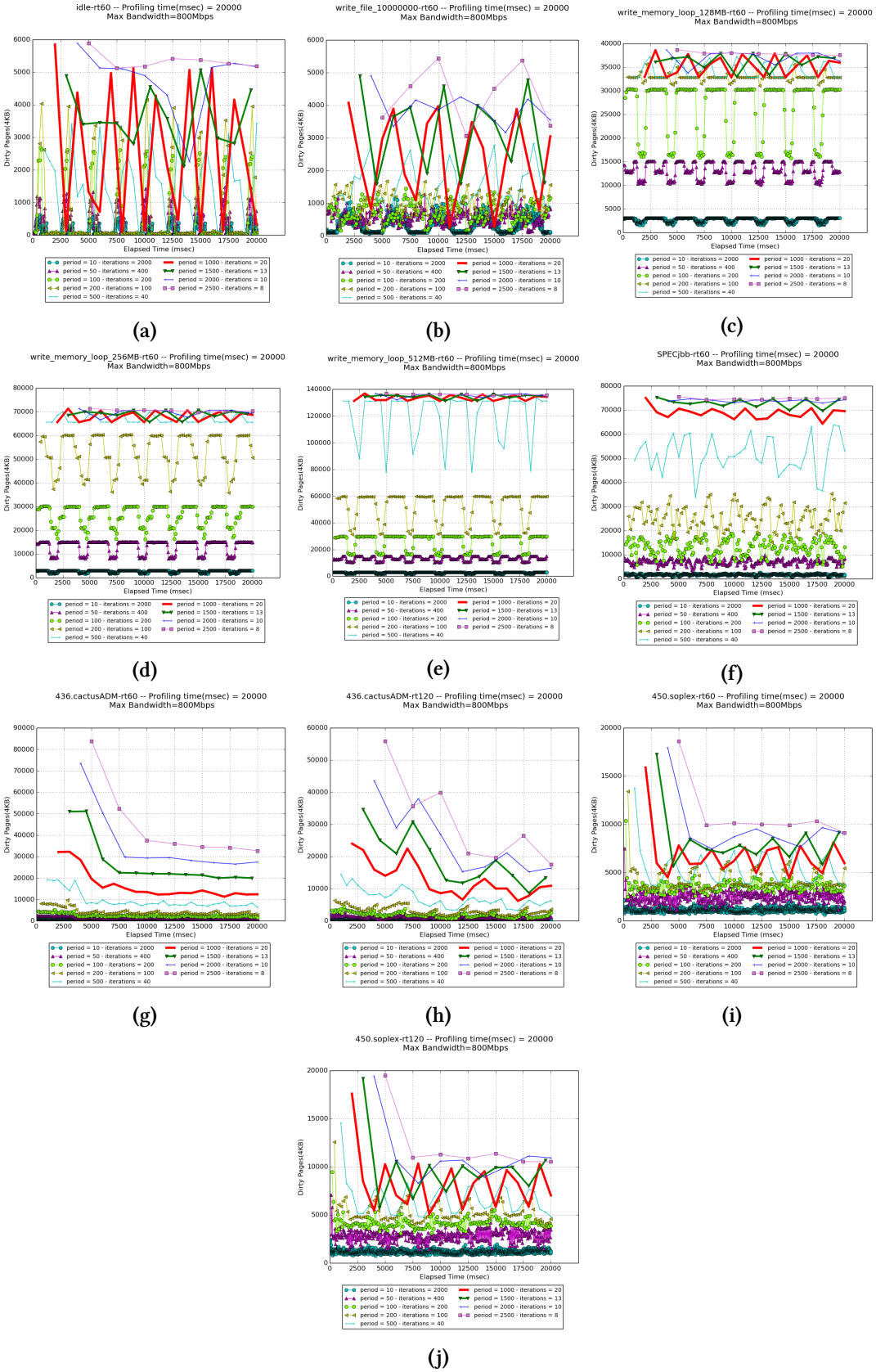


Figure 4.2: Tuning *"period"* parameter of BitmapTrace

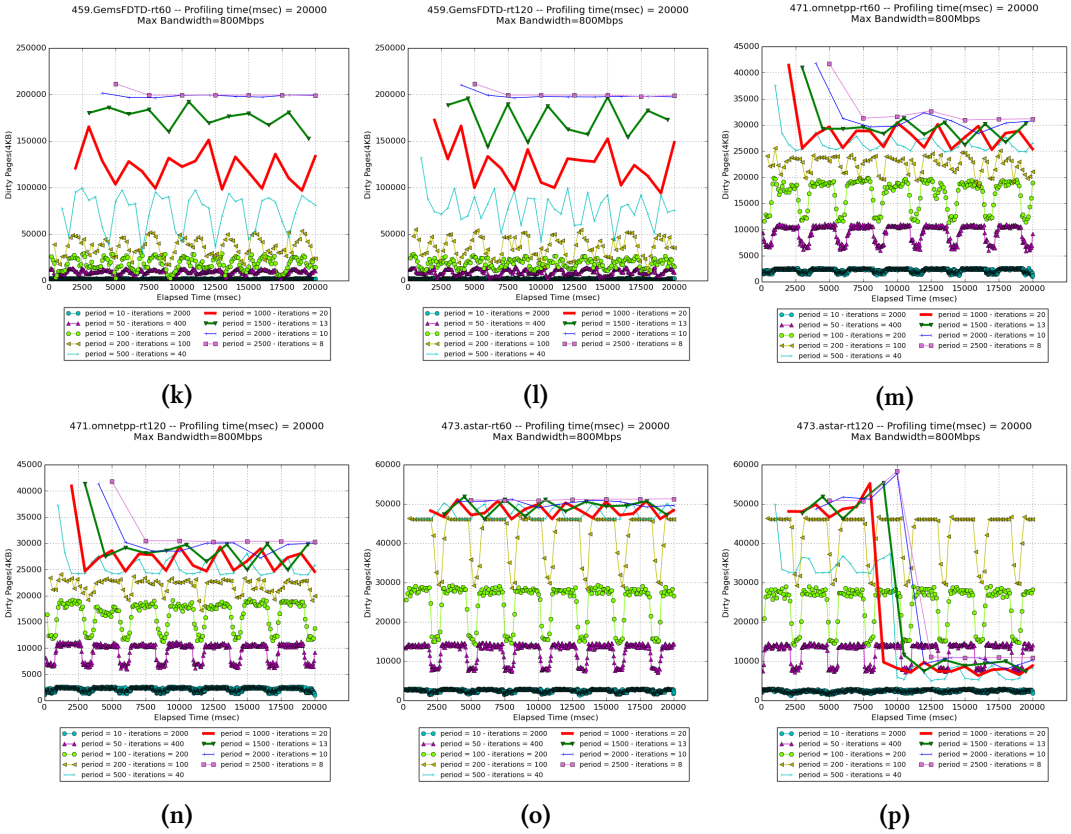


Figure 4.2: Tuning *"period"* parameter of BitmapTrace (cont.)

4.4.3 Tuning *iterations* parameter

In our final evaluation tests of the BitmapTrace mechanism we study the influence of the *"iterations"* parameter. As we can see from figures 4.3 the number of iterations determines the total number of samples that are collected, and thus defines the total profiling time for a certain *"period"* value. Greater number of *"iterations"* results in obtaining a larger memory snapshot. We can observe that even for 5 iterations we can get a great overview about the memory behavior of the guest. However, selecting a small number of iterations might not be sufficient enough for making the proper conclusions regarding the memory footprint of real-world workloads. This is presented in *473.astar_rt120* application (figure 4.3p) which suddenly slows down its dirtying rate. Therefore, we suggest that 10 iterations with period 1000msec is tolerable in order to track applications for a total profiling window of 10 seconds. When the *VM_Size* is larger than 1GB the total profiling period may need to be increased.

Given our analysis about *"period"* in previous section, we select the 1000ms and 1500ms values for our current tests. Especially in the graphs with 1000ms *"period"*, one can notice the **Dirty Page Rate** of each workload throughout the profiling period, as it depicts the new dirty pages per 1000ms. Furthermore, in most cases it is observed that the memory write pattern is almost the same for the *"period"* 1000ms and 1500ms. However, in some instances, such as *SPECjbb*, *436.cactusADM* and *459.GemsFDTD*, the line for 1500ms *"period"* is offset vertically. A possible explanation is that the application is memory "hungrier" than the allowed portion of memory that is dirtied for the selected *"period"*.

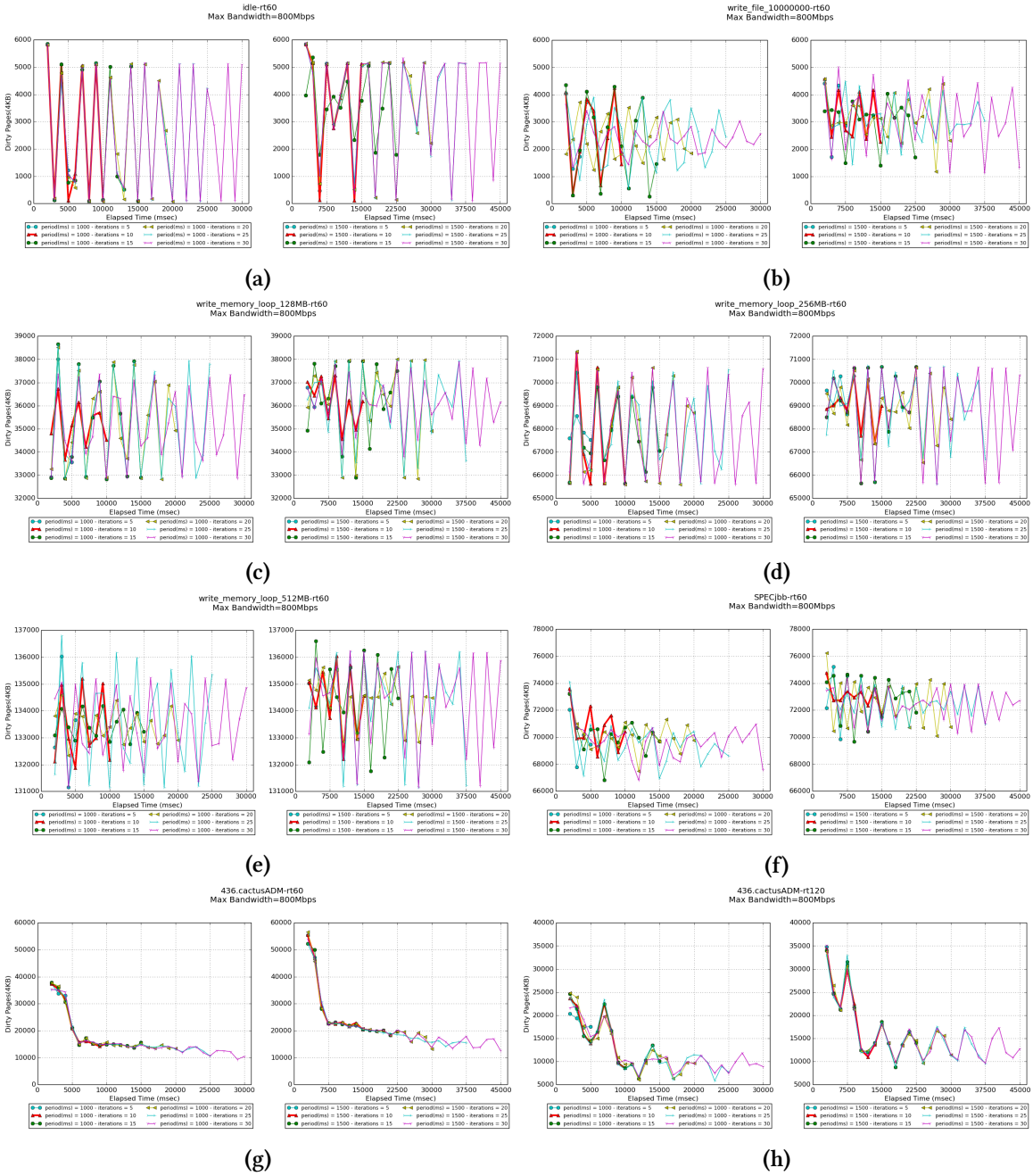


Figure 4.3: Tuning *"iterations"* parameter of BitmapTrace

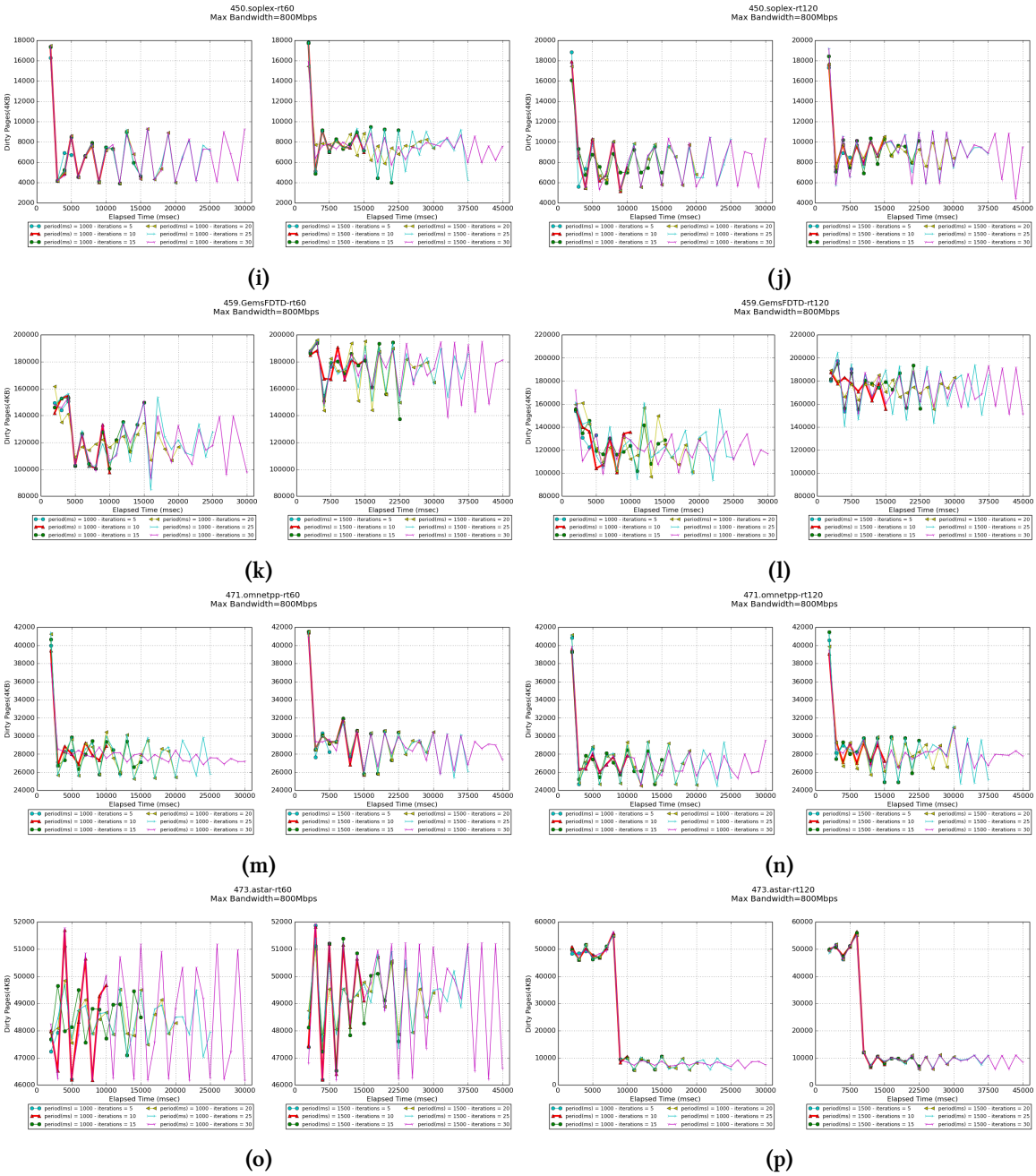


Figure 4.3: Tuning *"iterations"* parameter of BitmapTrace (cont.)

4.4.4 Performance Overhead

In order to analyze the guest's performance during profiling, we decide to evaluate its behavior when faced with the worst case scenario. Thus, we modify the *write_memory_loop_xMB* microbenchmark in order to tune its execution time as well as how many times the allocated memory segment(xMB) is touched. Specifically speaking, we set the allocated memory equal to 512MB. Furthermore, we use a simple microbenchmark that doesn't perform any memory writes. The following tests are performed while the BitmapTrace mechanism is either triggered or deactivated:

(i) For the suggested pair (*iterations,period*)=(10,1000) we run the memory-intensive benchmark for different number of memory writes and execution time. (Table 4.1)

(ii) For a combination of (*iterations,period*) values we run the memory-intensive benchmark for a specific number of memory writes and execution time. (Table 4.2)

(iii) For the suggested pair (*iterations,period*)=(10,1000) we run the memory-intensive benchmark and the simple (without memory writes) benchmark for a specific number of memory writes and execution time. (Table 4.3)

We measure the total execution time of the application using the *time* [47] command provided by Linux. As we can observe from the following tables, BitmapTrace mechanism can be considered a lightweight monitoring tool as it only introduces an offset of 5-7 seconds to the completion time of a memory-intensive application when the suggested values for *"iterations"* and *"period"*, as introduced in sections 4.4.2, 4.4.3, are used. This extra execution time is minimal if we consider that the workload is executed in the VM for hours. In case of applications which do not interact with the memory, the overhead introduced by the BitmapTrace mechanism is almost zero, as presented in table 4.3.

As presented in table 4.2, when the number of iterations is increased the overhead is slightly increased. This is expected, because the times that the kernel dirty bitmap is fetched (ioctl(*KVM_GET_DIRTY_LOG*)) for the calculation of the dirty pages is increased.

BitmapTrace (*iterations,period*)	Number of memory writes	Execution Time (msec)	BitmapTrace Overhead (msec)
No BitmapTrace	(10000)	175556	-
	(20000)	348883	-
	(30000)	522520	-
(10,1000)	(10000)	180570	5014
	(20000)	353771	4888
	(30000)	529954	7434

Table 4.1: Runtime overhead of application dirtying 512MB of memory when triggering BitmapTrace mechanism for (*"iterations"*, *"period"*)=(10,1000)

BitmapTrace (*iterations*,*period*)	Execution Time (msec)	BitmapTrace Overhead (msec)
No BitmapTrace	184200	-
(5,1000)	185845	1645
(10,1000)	188708	4508
(15,1000)	190597	6397
(5,1500)	186120	1920
(10,1500)	188952	4752
(15,1500)	190675	6475

Table 4.2: Runtime overhead of application dirtying 512MB of memory when triggering BitmapTrace mechanism for different (*"iterations"*, *"period"*) values

BitmapTrace (*iterations*,*period*)	Application type	Number of completed loops	BitmapTrace Performance Overhead (%)
No BitmapTrace	memory-intensive(512MB)	10248	-
	simple(without memory writes)	2523475150	-
(10,1000)	memory-intensive(512MB)	9982	2,59
	simple(without memory writes)	2522351699	0,04

Table 4.3: Performance overhead of different applications with execution time=3min when triggering BitmapTrace mechanism

Chapter 5

Machine Learning Approach for VM Migration Candidates

Machine learning algorithms build a mathematical model of sample data, known as ”training data”, in order to make predictions on new data with the same structure. We leverage the approach of supervised learning for generating models that predict the minimum *downtime-limit* for successful live migration and the migration's times (downtime, total-time) based on the memory footprint of the applications right before the migration is triggered. Cloud Service Providers need to facilitate the optimal migration decision making between various choices so that they minimize the migration cost and respect the Service-Level Agreements (SLAs) with the customers. We form some operation policies and later, we employ these models in a live migration framework to automatically sort the virtual machines that run on a single host and select the best candidates for migration.

5.1 Model Building

The main goal of our work is to predict the migration order of the Virtual Machines running on a single host. The modeling approach should operate transparently to the running applications. The target metric of the model is the minimum *downtime-limit* that ensures the pre-copy live migration algorithm will converge and thus, the migration process will complete. As a further approach to our problem, we build separate models that try to predict the actual downtime and actual total time accordingly. Despite the fact that the accuracy of these models is lower than expected, they can be used to approximately estimate these performance metrics of live migration.

5.1.1 Building the Data Set

The features that will be selected for the training data of our model play a major role in the prediction accuracy. All metrics depend on both the benchmarks' characteristics and the platform's architecture. The training data for the machine learning models have been generated by monitoring the guest under various workloads (see 3.2) with the BitmapTrace mechanism as well as performing a large number of live migrations of VMs with different configuration parameters. The testbed is similar to chapter 3.1 where the maximum available bandwidth (*max-bandwidth*) is configured to 800Mbps and the VM memory size is 1G.

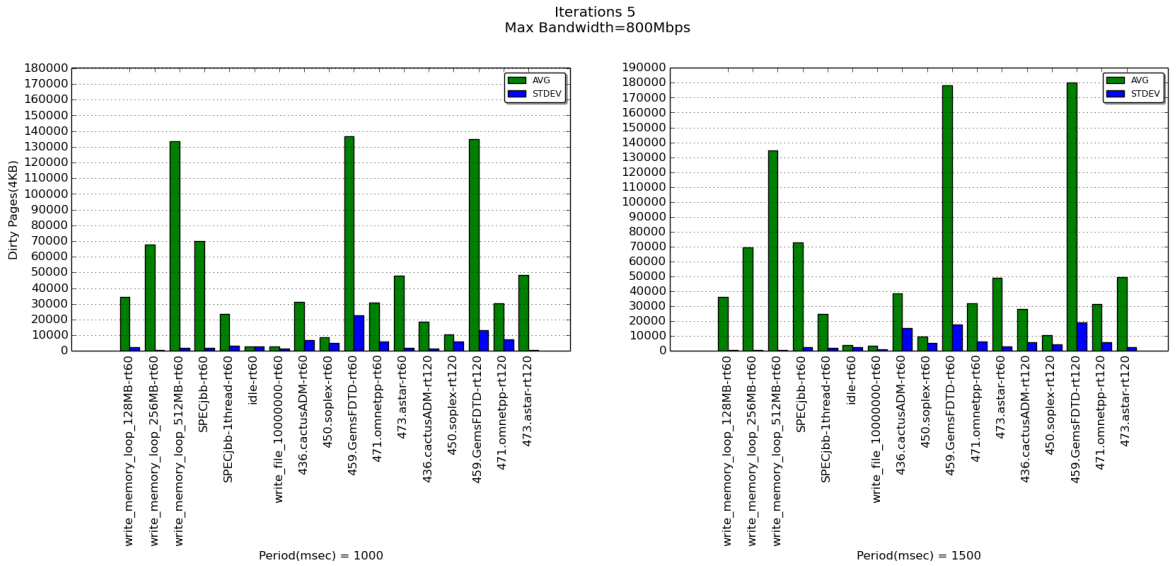
Regarding the first model (*min_success_downtime-limit* model) we consider as features the average (AVG)[1] and the standard deviation (STDEV)[2] of the dirty pages of the guest for a certain

[1] For a data set, the average, also called the arithmetic mean or mathematical expectation, is the central value of a discrete set of numbers: specifically, the sum of the values divided by the number of values.

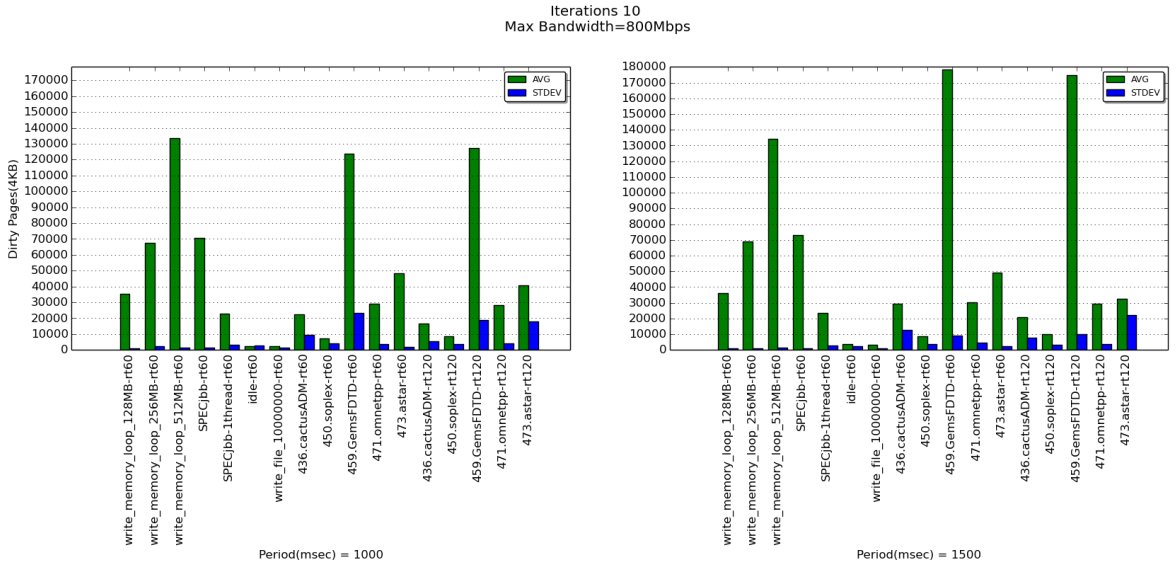
[2] The standard deviation is a statistic that measures the dispersion of a dataset relative to its mean and is calculated as the square root of the variance by determining the variation between each data point relative to the mean.

profiling period. The dirty pages are monitored by the BitmapTrace mechanism with the suggested (*iterations,period*) pairs after the evaluation, as described in chapter 4.4.

The average (AVG) and the standard deviation (STDEV) values of our collected samples for each iteration are calculated using the Python's NumPy [48] package and the results are demonstrated in figure 5.1.

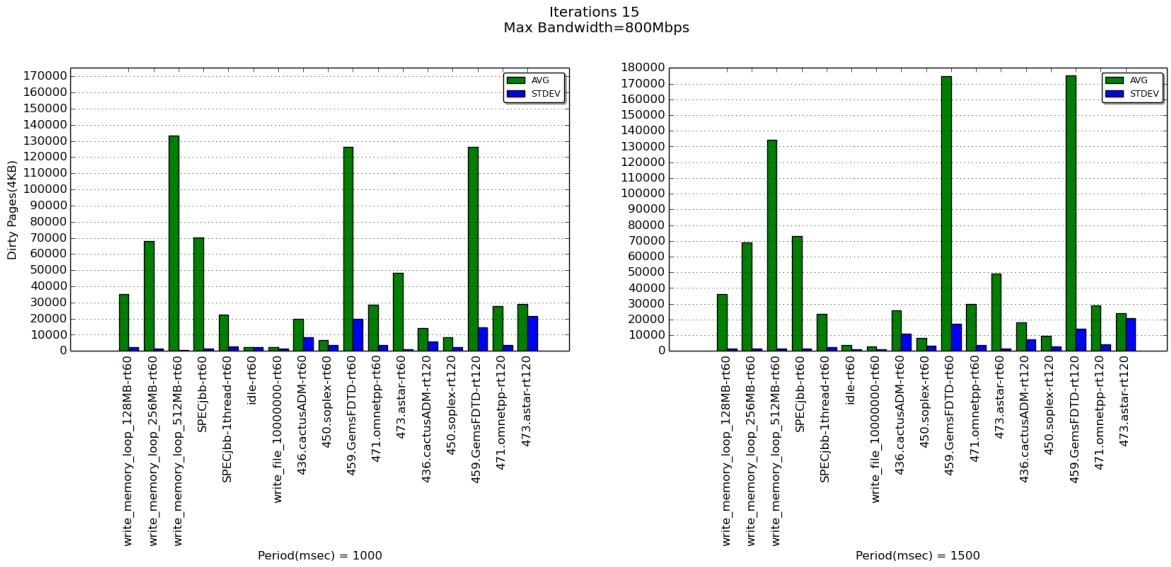


(a)

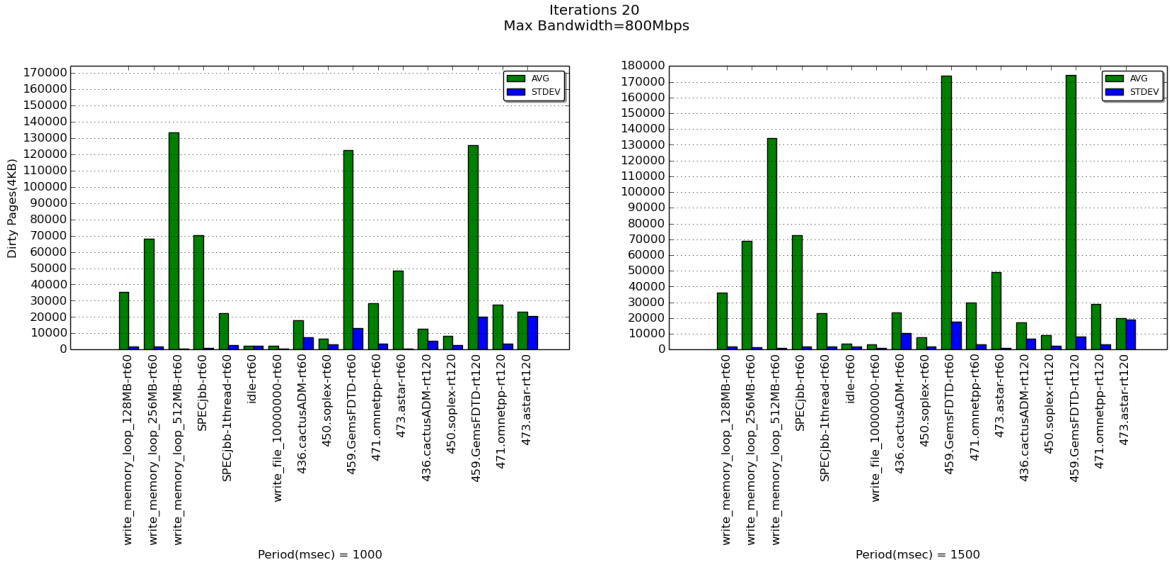


(b)

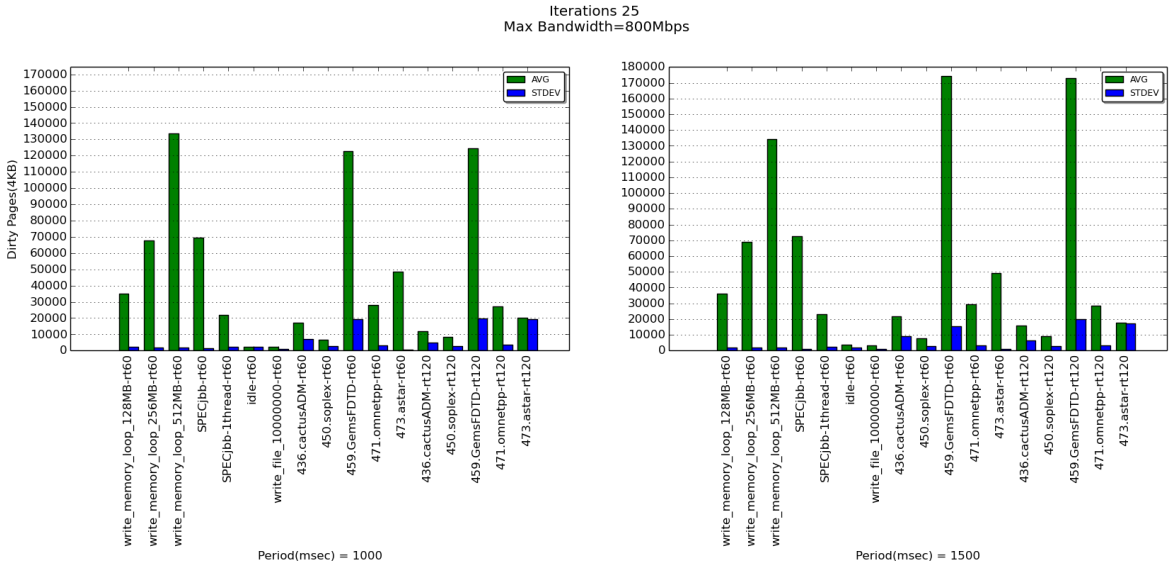
Figure 5.1: AVG and STDEV Dirty Pages during profiling period



(c)



(d)



(e)

Figure 5.1: AVG and STDEV Dirty Pages during profiling period (cont.)

The target metric is the minimum *downtime-limit* value that leads to a successful live migration using QEMU/KVM. Using this model, we aim to decide if pre-copy algorithm will converge for the specific migration configuration and application's memory behavior. We present again the results for each benchmark, after our analysis described in chapter 3.3.5.

Benchmark	***downtime-limit*(msec)**	
	rt60	**rt120**
idle	20	20
write_file	20	20
write_memory_loop_16MB	200	200
write_memory_loop_32MB	400	400
write_memory_loop_64MB	700	700
write_memory_loop_128MB	1400	1400
write_memory_loop_256MB	2600	2600
write_memory_loop_512MB	5200	5200
SPECjbb (1 thread)	600	600
SPECjbb (8 threads)	2400	2400
436.cactusADM	20	20
450.soplex	80	150
459.GemsFDTD	2800	6000
471.omnetpp	900	900
473.astar	1800	200

Table 5.1: Minimum *downtime-limit* for successful migration (*max-bandwidth*=800Mbps)

The above observations can be combined in order to reach some useful conclusions:

- As far as memory-intensive applications are concerned, we observe that the AVG Dirty Pages value is fairly high and the STDEV is either close to zero (e.g. *write_memory_loop_xMB_rt60*, *SPECjbb* and *473.astar_rt60*) or is a small proportion of AVG (e.g. *459.GemsFDTD_rt60* and *459.GemsFDTD_rt120*).
- I/O intensive application *write_file* has the minimum calculated AVG and STDEV values equal to *idle*'s VM values, which means that no memory accesses are performed in both cases.
- When the ratio STDEV/ AVG is higher than 0.5, then it seems that the STDEV of dirty pages predominates and it is the main factor that determines the minimum acceptable *downtime-limit* ensuring a successful migration (e.g. *436.cactusADM_rt60*, *436.cactusADM_rt120* and *473.astar_rt120*). A high standard deviation indicates that the data points are spread out over a wider range of values. Thus, migration algorithm can benefit from occasionally low dirty page rates and can reach to the state of convergence.
- When the AVG values are approximately equal but the STDEV values differ, applications with the higher STDEV have lower minimum *downtime-limit* for successful migration. For instance, in figure 5.1b, when period=1000ms the AVG value of *436.cactusADM_rt60* and *SPECjbb-1thread* is 20000 dirty pages approximately. However, STDEV of *436.cactusADM_rt60* is much higher which results in its low minimum *downtime-limit*=20ms while *SPECjbb-1thread* has minimum *downtime-limit*=600ms.

- SPEC benchmarks that represent real-world applications have higher values of STDEV compared to *write_memory_loop_xMB* microbenchmarks whose STDEV is almost zero, as expected, because of constantly dirtying memory. This variation of STDEV values of real-world benchmarks can be interpreted as different write memory pattern of each benchmark.

For the rest prediction models(*downtime* and *total_time* models) except the AVG and STDEV of dirty pages, the *downtime-limit* is also given as input feature. As presented in chapter 3.3.2, QEMU/KVM pre-copy live migration algorithm targets to minimize the total migration time. Therefore, the *downtime-limit* parameter is vital for the algorithm's convergence and the measured actual downtime is in inverse proportion to the total time of migration. Specifically, the training set consists of data obtained after executing various migrations with our benchmarks while tuning the *downtime-limit* parameter. Before migration is triggered, we profile their memory accesses with the BitmapTrace mechanism and then we calculate the AVG and STDEV of dirty pages. The actual total-time and actual downtime are logged after each live migration. In order to ensure that the data covers a wide range of the parameter space, the *downtime-limit* values range from the *min_success_downtime-limit* to large enough values that let the VM migrate instantly with total migration time almost equal to the actual downtime.
The above features and target metrics are referred as Historical Data for each model.

5.1.2 Model Generation

Given the analysis in section 5.1.1, the AVG and STDEV features taken from the dirty pages indicate that they are linearly correlated with the *min_success_downtime-limit*. Furthermore, the correlation of these features with the actual downtime also seems to be strong. Since migration's total time is also dependent on the link bandwidth, which is considered fixed in our case, the correlation with AVG and STDEV dirty pages is expected to be less clear. However, we examine the linear relation of these metrics and their degree of dependency will be evaluated in the next section. Thus, we select the Linear Regression technique which fits the given samples using a straight line with minimal error.

We use Python's scikit-learn v0.20 [49] toolikt for building and evaluating our regression models. We use the Support Vector Regression (SVR) [50] which is a regression technique for complex data points that exhibit a linear or non-linear relationship between the features and the target value. This relationship is specified by the kernel type used in the algorithm. We tune the parameters with exhaustive search for the SVR estimator using scikit-learn GridSearchCV [51] and based on the best score we select the following parameter values:

- For the *min_success_downtime-limit* model we use the 'linear' kernel with penalty parameter C=1.0 and epsilon=0.1 (default).
- For the *downtime* and *total_time* models we use the 'linear' kernel with penalty parameter C=0.1 and epsilon=0.1 (default).

The penalty parameter C is used to maintain regularization. Thus, it is also known as regularization parameter and represents misclassification or error term. The misclassification or error term specifies how much error is bearable in the SVM optimization. In this way, we can control the

trade-off between decision boundary and misclassification term. When the C parameter is small, the classifier can maximize the margin between most of the points, resulting in misclassifying a few points because of the low penalty value.

5.2 Model Evaluation

In this section we discuss the way we train our models and evaluate their accuracy. For this purpose, we used supervised machine learning. We expect the machine learning models to be as accurate as possible when predicting on new data (for which the target variable is unknown). The Historical Data is split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data in the future. We use the test dataset (or subset) to test our model's prediction on this subset. During this process, the main goal is to avoid overfitting[3] and underfitting[4].

In order to evaluate the performance of our machine learning models we perform K-fold cross-validation (CV). As figure 5.2 displays, in k-fold cross-validation the original sample is randomly (shuffle=True) partitioned into k subsamples. Out of the k subsamples, a single subsample is retained as the validation data for testing the model and the remaining k − 1 subsamples are used as training data. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the validation data. The k results from the folds can then be averaged to produce a single estimation. The advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once.

After tuning the value K in combination with parameter tuning using GridsearchCV, we set the number of splits equal to three (3-fold cross-validation) in case of *min_success_downtime-limit* model and five for the *downtime* and *total_time* models (5-fold cross-validation) since the data set is larger in the last cases. The data set is first split into three (acc. five) equal-sized subsets. Each subset serves as the test set once while the union of the remaining two (acc. four) forms the training data set. The reported values represent the average of the 3 (acc. 5) evaluations. The mean accuracy of K-fold cross-validation after fitting the model and computing the score three (acc. five) consecutive times is presented in table 5.3.

The time required to train each proposed model in case of the suggested pair (iterations,period) = (10,1000) is displayed in table 5.2. In a data center, re-training of the models is desirable occasionally, as the models should reflect the current set of workloads running on each VMs. The computational overhead of a few minutes every few days seems acceptable.

Model	***min_success_downtime-limit***	***downtime***	***total_time***
Training Time(msec)	384	1686	543

Table 5.2: Model Training time in case of (iterations,pediod) = (10,1000)

In order to evaluate the prediction accuracy of our model we use the r2_score [52] and mean_absolute_error [53] functions from sklearn.metrics [54] package with the SVR model. The results are presented in

[3] Overfitting means that model we trained has been trained "too well" and is fit too closely to the training dataset. This model will be very accurate on the training data but will probably be inaccurate on untrained or new data.

[4] When a model is underfitted, it means that the model does not fit the training data and therefore misses the trends in the data. It also means the model cannot be generalized to new data.

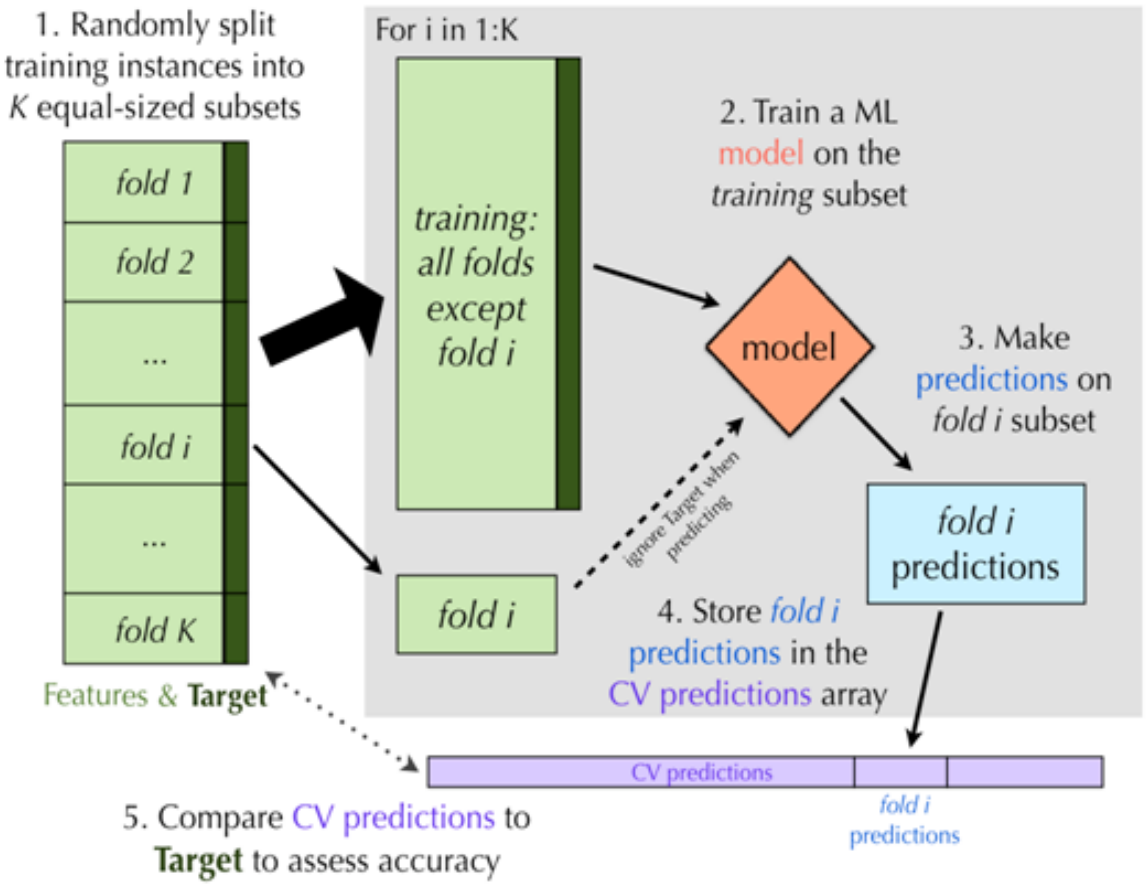


Figure 5.2: Flowchart of the K-fold cross-validation

table 5.3 . The coefficient of determination (CoD, or R^2) measures how well the prediction fits the measured value. An R^2 of 1 indicates a perfect fit of the prediction to the measured target value. Low R^2 values, such as 20% and 25% for the *total_time* appear because when setting large values of *downtime-limit* the actual total migration time remains equal to the downtime. It seems that the linear model cannot predict this behavior which causes low R^2 values. The Mean Absolute Error (MAE) represents the average divergence of the predicted value to the actual value in absolute units of the metric (msec). In each step of cross-validation, the predicted value of test set is compared to the value of training set and the mean absolute of the k iterations is calculated.

Model	(iterations,period)	Model Accuracy(%)	Prediction Accuracy(%)	MAE(ms)
min_success_downtime-limit	(10,1000)	88	89	271
	(10,1500)	87	88	222
downtime	(10,1000)	85	86	301
	(10,1500)	69	70	403
total_time	(10,1000)	26	25	1828
	(10,1500)	20	20	2036

Table 5.3: Accuracy and MAE of the *min_success_downtime-limit*, *downtime* and *total_time* prediction models

The whole process for generating and evaluating our regression models can be summarized in figure 5.3 :

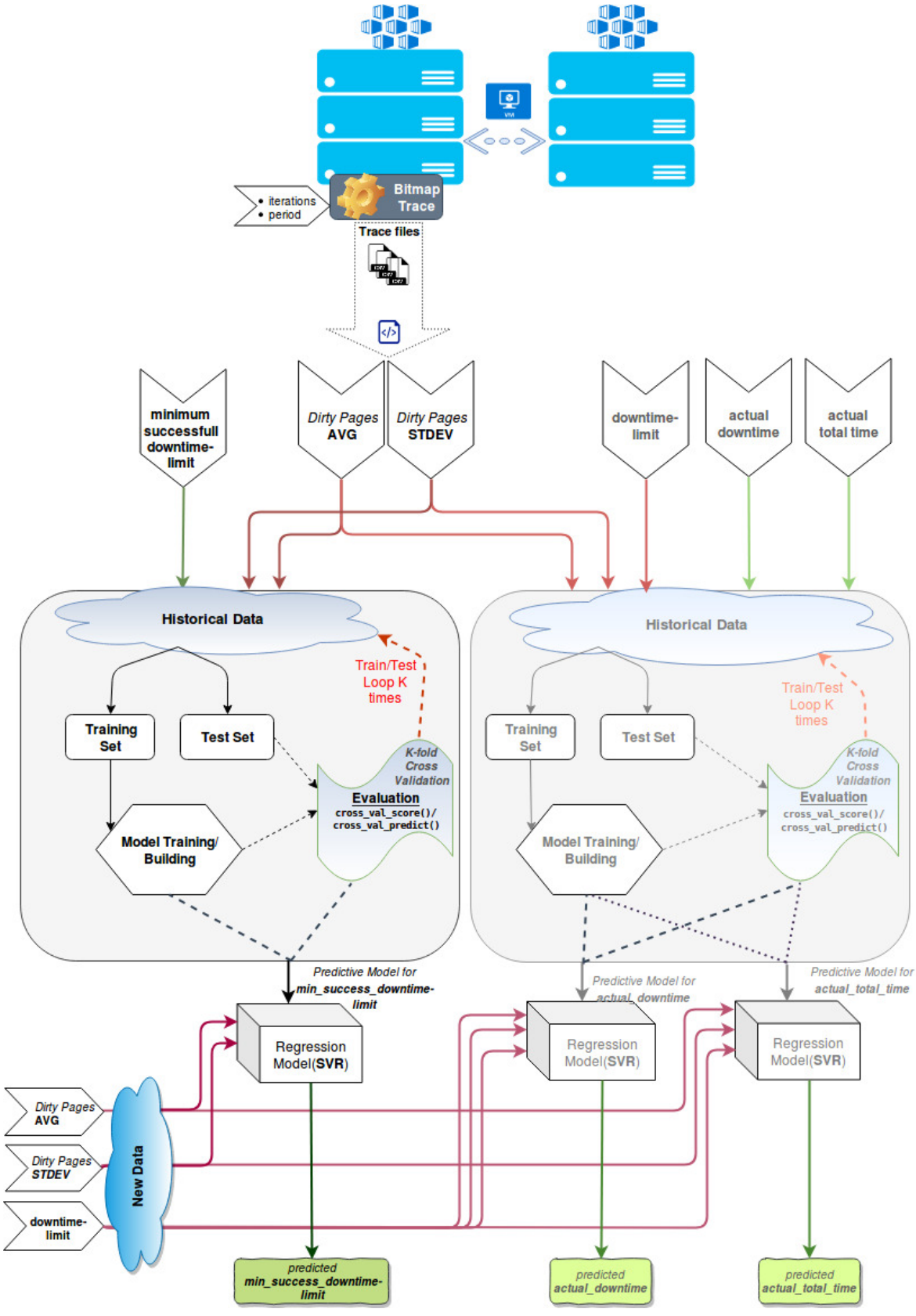


Figure 5.3: General prediction models workflow

5.3 Operation Policies for VM Live Migration Candidate Selection

Given the characteristics of the VM's memory footprint, which can be gathered using Bitmap-Trace mechanism as well as the existing SLAs, we sort the running Virtual Machines in the source host in order to decide which of them (k out of N) will be selected for migration to the destination host. Moreover, we follow the operation policies described below in order to ensure the successful live migration of the selected VM candidates.

(i) Each VM live migration should finish within a specific time window. We cancel live migrations that do not converge within a window of 40 seconds (timeout) for a specified *downtime-limit*.

(ii) Do not allow a migration not to happen. After tracking the memory footprint of the running VMs, we suggest the minimum *downtime-limit* that ensures a successful migration within a given timeout based on prediction techniques.

(iii) Satisfy SLA constraints to the maximum possible. A data center should ensure the availability of the provided services whenever they are required. This means that the downtime of each Virtual Machine during migration should not exceed the *max_downtime* which is agreed by the SLA.

(iv) Minimize the aggregated total time of migrating k out of N VMs from the source host to the destination host. This is the total time from starting migration of the first VM to finishing migration of the last one.

(v) The Virtual Machines are sorted based on the *Relative Downtime Error*. This metric is defined as:

$$\textit{Relative Downtime Error} = \frac{\textit{max\_downtime} - \overbrace{\textit{min\_success\_downtime-limit}}^{predicted}}{\textit{max\_downtime}}$$

Virtual Machines with high *Relative Downtime Error* are the best candidates for migration. Negative values of this metric indicate that the chosen *max_downtime*(based on SLA) does not ensure successful migration with the current memory access behavior of the workload running in the VM.

Large values of the *Relative Downtime Error* are expected to migrate fast enough, which actually means low actual total-time. In case of sequential migration order, we first migrate the Virtual Machines with low *Relative Downtime Error* values, as the *max_downtime* of these VMs is close to the predicted minimum *downtime-limit* for successful migration. Since the memory behavior of the workloads changes dynamically from time to time, we desire the migration of these borderline VMs right after their memory profiling in order to avoid a possible increase of dirty page rate which will affect the predicted migration result. In case of parallel migration, the order does not matter, since all the k selected VMs are simultaneously migrated to the destination host.

In case of two or more VMs have the same *Relative Downtime Error* value, we first choose the VM with the minimum AVG/STDEV value. As we analyzed in section 5.1.1, this ratio determines the convergence of the pre-copy migration algorithm.

(vi) The configured *downtime-limit* for the k-selected VMs for migration is equal to the *max_downtime* if the *Relative Downtime Error* value is positive or equal to the predicted *min_success_downtime-limit* if the *Relative Downtime Error* value is negative. Based on the sorting criteria described above, we choose the VMs with the minimum penalty that is incurred from violating the maximum acceptable downtime agreed in SLA.

5.4 Evaluation of VM Live Migration Candidate Selection

The proposed model was integrated into a VM live migration scheduling framework which automatically sorts the VM candidates for migration based on their application's memory footprint when the *max_downtime* is provided as SLA constraint. It also performs migration of k out of N Virtual Machines under the operation policies described in section 5.3. The framework is evaluated through the execution of three different scenarios. In all cases we consider that eight (8) VMs are co-located in the source host and four (4) of them should migrate. The testbed described in chapter 3.1 is used and each VM, with memory size equal to 1G, is pinned to a separate core.

A vital operation policy decision is whether the selected VMs will be migrated in parallel or sequentially. With a parallel migration the chances that neither guest will ever be able to complete are increased as the bandwidth is shared between the migrating Virtual Machines. Considering the physical bandwidth (1 Gbps) restriction in our testbed and the importance of the the successful completion of each VM live migration, the selected VMs are migrated sequentially. The maximum available bandwidth (*max-bandwidth*) for each migration is configured to 800Mbps. Moreover, the migration sequence does not matter for the current research. For example, migrating sequentially the virtual machines VM1, VM2, VM3, VM4 is considered the same as migrating VM4, VM3, VM2, VM1. However, as indicated in 5.3(v), the Virtual Machines with low *Relative Downtime Error* values, i.e. the *max_downtime* is close to the predicted minimum *downtime-limit* for successful migration, are firstly migrated in order to avoid possible increases of the dirty page rate of their workload.

Since migration order is out of our concerns, the possible combinations for selecting k out of N VM candidates for live migration can be calculated with the formula C(N,k) = $\frac{n!}{k!(n-k)!}$. In our scenarios N=8 and k=4, so there are C(N,k)=70 possible combinations. We present our experimental results compared to random selecting method of VMs for migration which is the simplest way for dealing with load balancing issues in a data center. However, this method may incur SLA violations and subsequent penalties for some of the selected VMs as well as ineffective network bandwidth utilization if the pre-copy migration algorithm does not converge.

In our experiments, each scheme is tested for 10 times and the presented aggregated total times are averaged. In all scenarios, the profiling of each VM is triggered after 30 seconds of execution time. Then, the migration is automatically initiated right after the profiling period, which lasts 10 seconds ((*iterations,period*)=(10,1000)). The decision making is based on our *min_success_downtime-limit* model. Furthermore, in each scenario we present the predicted aggregated total time which is based on *total_time* prediction model. Although the predicted and actual times differ a few seconds, the predicted value can be used if the Cloud Service Provider would like an estimation of the aggregated total migration time before triggering the migration.

5.4.1 VMs with same application and without SLA violations

VM: Application	max_downtime(msec) (SLA)
VM1: *write_memory_loop_256MB*	3800
VM2: *write_memory_loop_256MB*	3600
VM3: *write_memory_loop_256MB*	2600
VM4: *write_memory_loop_256MB*	3400
VM5: *write_memory_loop_256MB*	3200
VM6: *write_memory_loop_256MB*	2800
VM7: *write_memory_loop_256MB*	4000
VM8: *write_memory_loop_256MB*	3000

Table 5.4: Live Migration Scheduling scenario with VMs running the same application and without SLA violations

In this scenario, each VM runs the *write_memory_loop_256MB* microbenchmark that sequentially and circularly writes to the allocated memory. The only difference between the running virtual machines is the *max_downtime* as displayed in table 5.4. All the selected values are greater or equal to the *min_success_downtime-limit* for this application, which is calculated 2600msec when migration is initiated after 60 or 120 seconds of application's execution time. In case of model prediction scheduling, the predicted *min_success_downtime-limit* is about 2490msec for each VM. The values with the error bar (equal to MAE of *min_success_downtime-limit* model) are presented in figure 5.4. The selected VMs from our framework and their sequential migration order, as far as the *Relative Downtime Error* is considered, are : VM4, VM2, VM1, VM7 which are the best candidates with the minimum migration cost. In table 5.5 we show that the aggregated total time compared to the random choice of VMs is reduced by 3,75% (~1 second less).

Scheduling Scenario	Actual Aggregated Total Time	Predicted Aggregated Total Time
Model Prediction	29487	28840
Random	30636	-

Table 5.5: Aggregated total time with VMs running the same application and without SLA violations

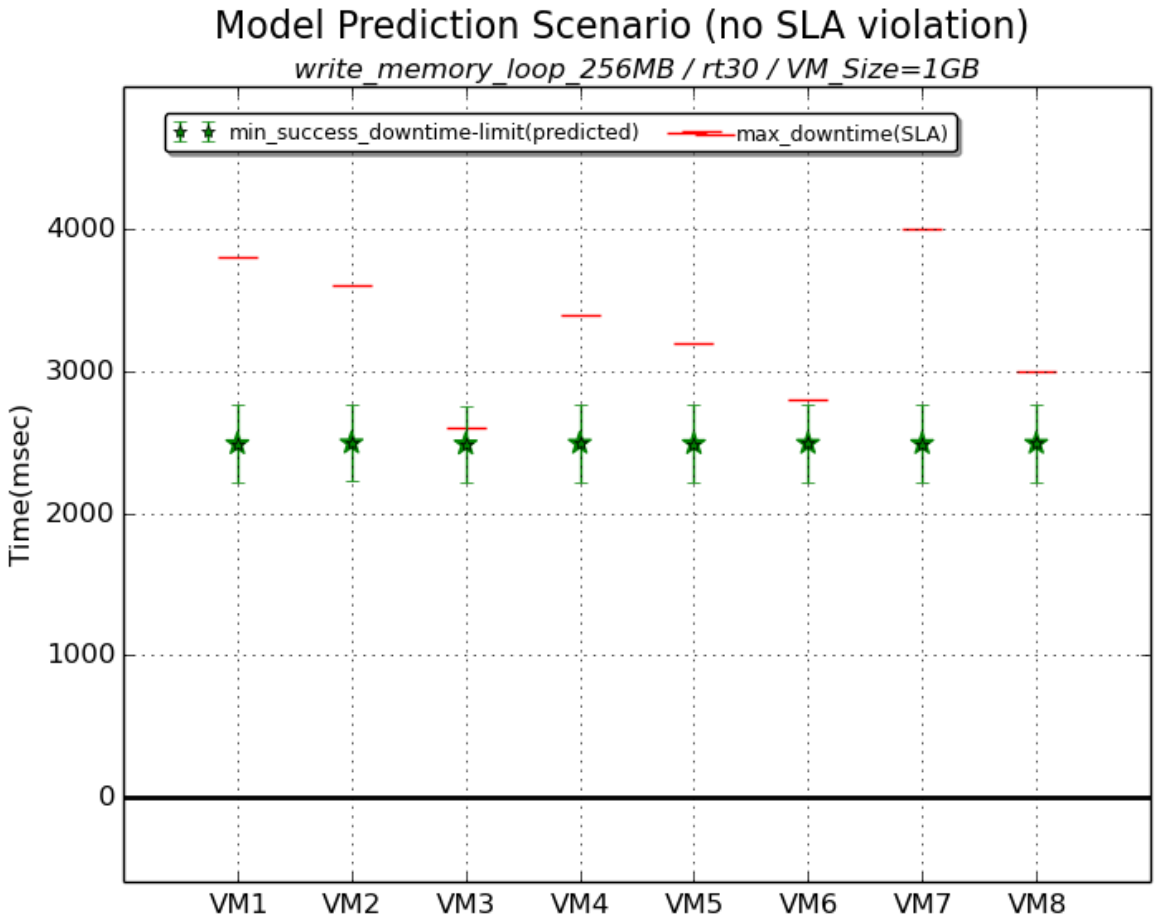


Figure 5.4: *max_downtime* and *predicted min_success_downtime-limit* in case of VMs with same application and without SLA violations

5.4.2 VMs with same application and SLA violations

VM: Application	**max_downtime(msec) (SLA)**
VM1: *write_memory_loop_256MB*	1400
VM2: *write_memory_loop_256MB*	2000
VM3: *write_memory_loop_256MB*	2600
VM4: *write_memory_loop_256MB*	2400
VM5: *write_memory_loop_256MB*	1600
VM6: *write_memory_loop_256MB*	2800
VM7: *write_memory_loop_256MB*	2200
VM8: *write_memory_loop_256MB*	1800

Table 5.6: Live Migration Scheduling scenario with VMs running the same application and SLA violations

Similar to the previous scenario, each VM runs the *write_memory_loop_256MB* microbenchmark that sequentially and circularly writes to the allocated memory. The only difference between the running virtual machines is the allowed *max_downtime*. In this case six out of eight *max_downtime* values are less than 2600msec which is the *min_success_downtime-limit* (Table 5.6) for *write_memory_loop_256MB*

when migration is initiated after 60 or 120 seconds of application's execution time. In this way, some VMs will violate the SLA's *max_downtime* during migration. In model prediction scheme, the predicted *min_success_downtime-limit* is about 2490msec again for each VM. The values with the error bar (equal to MAE of *min_success_downtime-limit* model) are presented in figure 5.5. Since we need to migrate 4 VMs, our framework selects the two VMs with *max_downtime* greater or equal to *min_success_downtime-limit* and the other two selected VMs are those with the minimum SLA violation rate. Therefore, VM7, VM4, VM3, VM6 are sequentially migrated. In a random choice scheduling scheme, the four selected VMs may violate the SLA constraints. In addition, the value of the *downtime-limit* parameter should be randomly chosen without any previous knowledge for the application of each VM.

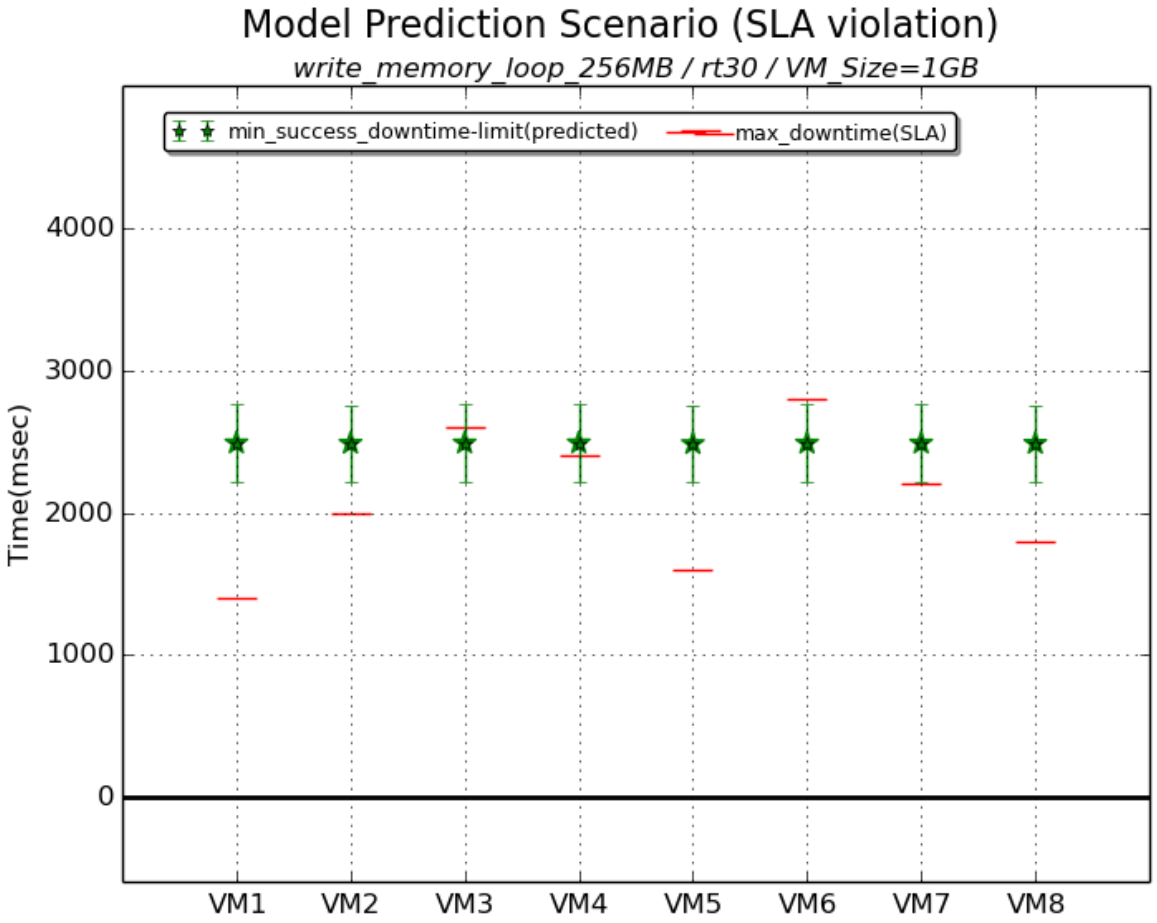


Figure 5.5: *max_downtime* and *predicted min_success_downtime-limit* in case of VMs with same application and SLA violations

5.4.3 VMs with different application and without SLA violations

VM: Application	max_downtime(msec) (SLA)
VM1: *SPECjbb* (1 thread)	900
VM2: *write_file*	40
VM3: *450.soplex*	500
VM4: *write_memory_loop_512MB*	6000
VM5: *471.omnetpp*	2000
VM6: *436.cactusADM*	100
VM7: *473.astar*	2000
VM8: *459.GemsFDTD*	5000

Table 5.7: Live Migration Scheduling scenario with VMs running different application and without SLA violations

Scheduling Scenario	Actual Aggregated Total Time	Predicted Aggregated Total Time
Model Prediction	23368	21513
Random	29440	-

Table 5.8: Aggregated total time with VMs running different application and without SLA violations

The migration scheduling scenarios described in sections 5.4.2, 5.4.1 do not represent real-world scenarios. In a data center, each VM executes a different application. Thus, we try to simulate a scenario where each VM runs a different application, as presented in table 5.7. The selected *max_downtime* value for each VM is greater than the *min_success_downtime-limit* calculated when the migration is initiated after 60 or 120 seconds of application's execution time (Table 5.1). The predicted *min_success_downtime-limit* values with the error bar (equal to MAE of *min_success_downtime-limit* model) in case of model prediction scheduling scheme are displayed in figure 5.6. Since each VM executes a different workload, the *min_success_downtime-limit* differs between them after 30 seconds runtime. The best VM candidates for migration and their sequential migration order, as far as the *Relative Downtime Error* is considered, are : VM8, VM5, VM3, VM7. We can find that the migration cost can be significantly reduced with the guide of our model, as presented in table 5.8. The aggregated total time compared to the random choice of VMs is reduced by 20,63% (~6 seconds less), which also causes reduction in network traffic and energy consumption.

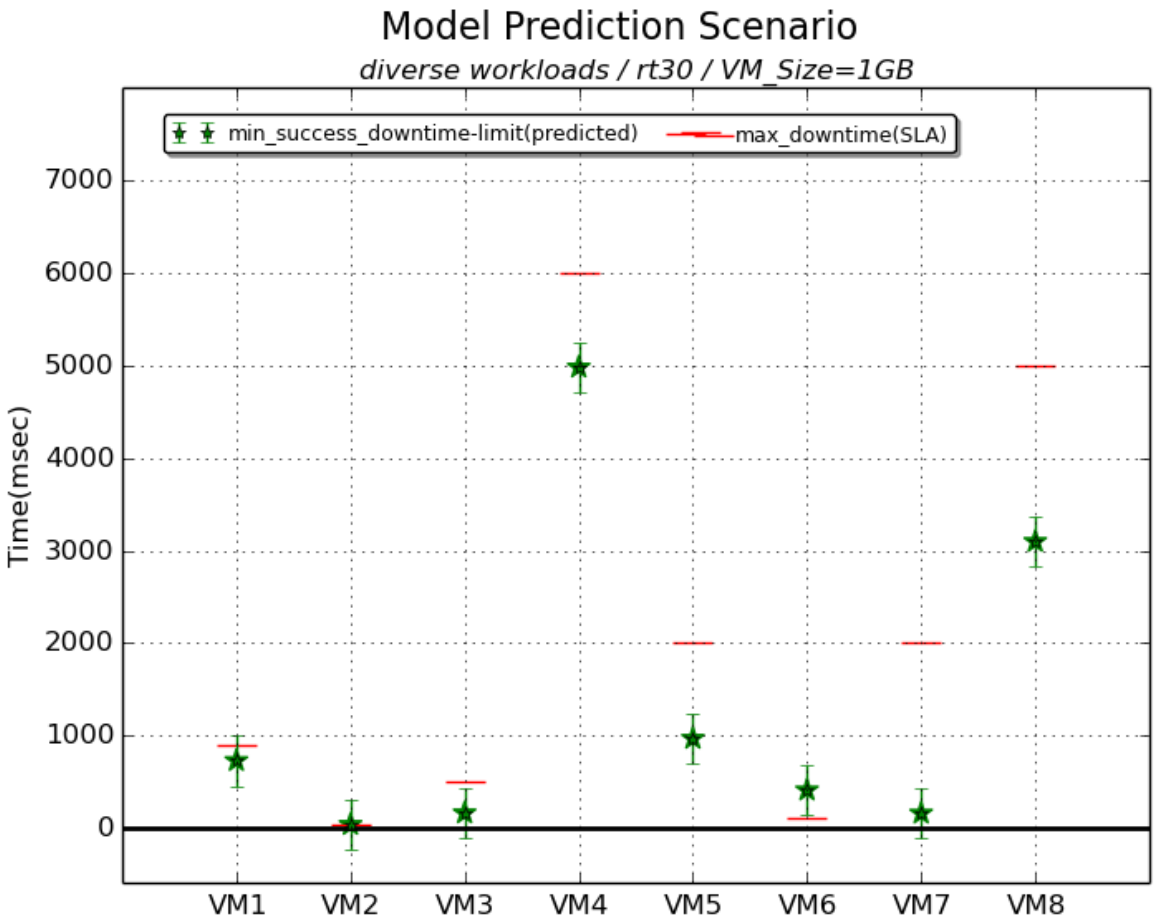


Figure 5.6: *max_downtime* and *predicted min_success_downtime-limit* in case of Ms with different application and without SLA violations

Chapter 6

Related Work

In this section, we discuss the earlier work that has been done in the area of live migration and especially the approaches to predict the factors that influence the migration process as well as the prediction techniques for multiple VM live migration. Zhang et al. [55] present a survey for the existing strategies and optimization mechanisms for VM live migration from the perspective of the three challenges it faces: memory data migration, storage data migration, and network connection continuity. They also sum up the studies which focus on understanding the relationship between migration performances and influence factors.

Live migration was first demonstrated by Clark et al. [19] who try to improve the pre-copy live migration algorithm by tracking the memory pages that are dirtied very frequently, i.e WWS, and propose a solution to reduce the amount of dirty pages that are sent to destination during the iterative phase. They leverage XEN's paravirtualization feature for recording the WWS of each process in the migrated VM, and limiting the maximum page faults for each process. In further, to make a trade-off between migration convergence and network bandwidth utilization, a dynamic rate-limiting approach is designed by them to control the migration bandwidth. Zhang et al. [56] design a monitoring system called MigVisor which first invokes the Pattern-Collector (PC) prediction and, if the PC prediction returns negative results, it invokes the Dry Run (DR) module to predict the VM live migration behaviour with an optimized memory compression scheme. These modules are integrated in QEMU/KVM and the working-set pattern (WSP) prediction model is used to determine the least convergent bandwidth (minimum bandwidth for a successful Live migration) similar to Clark's work. In our research, we design and implement the BitmapTrace mechanism for QEMU/KVM which, similar to PC module, periodically calls the *KVM_GET_DIRTY_LOG* function in order to acquire the latest status of the kernel's dirty pages. Furthermore, we set a static value for maximum available bandwidth and we tune the *downtime-limit* parameter in order to make pre-copy algorithm converge with respect to SLA's *max_downtime*. In case of a non-convergent failure that is driven by DR prediction, a long execution time to simulate the failure process may cost more CPU time to complete the prediction as it simulates the process of compressing and copying dirty pages in addition to PC module's overhead. On the other hand, our framework targets on multiple VM migration, so non-convergent VM candidates will not be selected due to their high dirty page rate in accordance to *max_downtime*.

Several prediction models have been developed in order to predict migration's performance metrics. Liu et al. [57] present a model able to predict the time an iterative pre-copy live migration takes to complete, based on measuring the page transfer rate and the page dirtying rate. Information about dirty pages is gathered by adapting the Xen hypervisor. They have evaluated migration delay,

downtime, network traffic and energy consumption for various applications running inside a migrated VM. However, quality of service metrics, e.g., response times and service levels, are not part of their evaluation. Akoush et al. [58] analyze the relationships between the important parameters that affect migration with emphasis on the Xen virtualization platform and highlight how migration performance can vary considerably depending on workload. In addition, they design two simulation models to predict the performance of pre-copy migration pattern, AVG (average page dirty rate) and HIST (history based page dirty). The AVG model is used to predict the migration performance of the VM which has a constant memory dirtying rate, while the HIST model is designed for a VM which has similar memory behaviors between different runs. The latter model is also used by Patel et al. [59] in a framework where time series prediction techniques are developed using historical analysis of past data. They propose two different regression models of time series. The first model is developed using a statistical probability based regression model and it is based on ARIMA (autoregressive integrated moving average) model. The second one is developed using statistical learning based regression model and it uses SVR (support vector regression) model. In contrary to the aforementioned works, our model is based on the AVG dirty page rate as input feature in combination with STDEV of dirty pages, because applications with high values of STDEV may result in migrations with short downtimes. We avoid a model based on historical data as pre-copy live migration highly depends on the dirtying memory behavior of the workload at the moment that is triggered. Nathan et al. [60] analyze in depth twelve existing performance models for pre-copy and propose a new model for pre-copy with page-skipping optimization. They monitor the unique pages dirtied, i.e. the pages that will be transferred in each iteration after the optimization of skip technique, and the number of skipped pages in each iteration. All the above models are tested on Xen to compute downtime, total number of pages transferred, and total migration time, while our model predicts the minimum *downtime-limit* that ensures a successful migration based on the vanilla pre-copy live migration of QEMU/KVM.

As far as multiple VM migration is considered, several approaches have been examined with the goal of improving the utilization of datacenters. Lu et al. [1] presented the OpenStack version of the Generic SLA Manager, alongside its strategies for VM selection and allocation during live migration of VMs. By combining IaaS (OpenStack-SLAM) and PaaS (OpenShift) as a use case they efficiently manage VM live migration with VM placement strategies, when a multi-domain SLA pricing and penalty model is involved. Ye et al. [61] propose to reserve resources (CPU cycles and memory space) for migration. Based on their experimental discoveries, three optimization methods, optimization in the source machine, parallel migration of multiple virtual machines, and workload-aware migration strategy, are proposed to improve the migration efficiency. Considering parallel migration, they found that it is better than sequential migration when enough resources are available for migration, otherwise, parallel migration is worse as we also denote in our work due to Gigabit Ethernet network limitations.

Finally, Jo et al. [62] propose a live migration framework similar to ours whose prediction model reduces the number of SLA and operator-specified constraint violations significantly. Their machine learning-based model is used to predict key metrics (total VM migration time, total amount of data transferred, VM downtime, performance degradation of the VM, and CPU and memory usage on the physical hosts) of different live migration algorithms, such as pre-copy, post-copy, CPU throt-

tling, delta compression and data compression. Compared to their large dataset of input features, our regression prediction model takes only AVG and STDEV of dirty pages as input parameters and predicts with high accuracy the minimum *downtime-limit* for convergence of the vanilla pre-copy algorithm without any optimizations. Our framework aims to automatically select the best VM candidate between the guests running on the same host, while Jo et al. employ their model in a framework in order to automatically select the proper live migration algorithm for the migration of a single VM.

Chapter 7

Conclusions and Future Work

In the present work, we propose a VM live migration framework based on prediction techniques to accurately select the VM candidates for migration that minimize the aggregated total time. We dynamically select the Virtual Machines for migration based on the current memory footprint of the workload running on them and the *max_downtime* that should be guaranteed (SLA constraint) to each of them. In order to obtain the required snapshot of the memory dirtying behavior of the VM for a certain profiling window, we implement a module in QEMU/KVM called BitmapTrace. This mechanism performs a "shadow" migration and logs the dirty pages per iteration by tracking the dirty bitmap through calling the *KVM_GET_DIRTY_LOG* ioctl() before migration is triggered. The average(AVG) and the standard deviation(STDEV) of the acquired samples are given as input parameters to a model which predicts the minimum *downtime-limit* that should be configured for ensuring that the pre-copy live migration algorithm will successfully converge. Especially, migrating memory-intensive applications for load balancing reasons is tricky regarding the *downtime-limit* that should be selected. Thus, we tackle the problem of migrating VMs with *downtime-limit* that may never complete in accordance with meeting the objectives of the operation policies in a cloud computing environment. Furthermore, we avoid wasting networking and computing resources on the VM migration as well as the associated system performance degradation caused by wasting these resources.

Several enhancements regarding the current capabilities of BitmapTrace mechanism including our proposed live migration framework could be discussed. We demonstrate some potential directions for future work :

- **Support more live migration techniques**: Current implementation of BitmapTrace mechanism supports only the pre-copy live migration algorithm. However, as presented in chapter 2, QEMU/KVM supports post-copy live migration technique as well some optimizations, such as auto-converge (CPU throttling) and XBZRLE (delta-compression) which could be integrated in our module.

- **Increase the data set with extra *max-bandwidth* and *VM_Size* values**: Currently the input features(AVG and STDEV of dirty pages) of our model are calculated when the maximum available bandwidth(*max-bandwidth*) is configured to 800Mbps and the VM memory size is 1G. As future work, we intend to monitor VMs with different memory sizes and estimate the *min_success_downtime-limit* for a greater amount of configured *max-bandwidth* values. For each (*max-bandwidth*, *VM_Size*) pair, our model needs to be retrained in order to accurately predict the *min_success_downtime-limit*.

- **Improve total-time accuracy and predict more target metrics**: Accurate predictions about migration durations (downtime and total-time) are secondary target of our work. This is the reason why the model and prediction accuracy of the *total_time* model were not further investigated. Since it needs further improvement for the given input features, we aim to take extra measurements or even try other regression models to make more accurate estimations. Moreover, we could extend our model by predicting more target metrics, such as the amount of the network traffic.

- **Support parallel live migration**: The testbed used for the evaluation of our framework is restricted by the network bandwidth capacity as the available physical bandwidth is 1Gbps. As a result, during a parallel migration the bandwidth is shared between the selected VMs for migration, which increases the chances that none of the guests will ever be able to complete. We intend to evaluate our framework in high throughput networks like 10Gbps or InfiniBand which are increasingly used nowadays by advanced data centers.

- **Predict which destination host should be selected**: The current implementation of our live migration framework assumes that the destination host is free from running workloads. In real cloud computing environments, many destinations exist and the framework should choose which host is the most appropriate for each VM that will be migrated.

Abbreviations

AVG	Average
CLI	Command-Line Interface
CV	Cross-Validation
HMP	Human Monitor Interface
I/O(or IO)	Input/Output
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
MAE	Mean Absolute Error
MPKI	Misses Per Kilo Instructions
OS	Operating System
PaaS	Platform as a Service
PC	Performance Counter
PMU	Performance Monitoring Unit
QEMU	Quick Emulator
QMP	QEMU Machine Protocol
SaaS	Software as a Service
SLA	Service-Level Agreement
STDEV	Standard Deviation
VM	Virtual Machine
VMM	Virtual Machine Monitor
WWS	Writable Working Set
XBZRLE	Xor Binary Zero Run-Length-Encoding

Bibliography

[1] K. Lu, R. Yahyapour, P. Wieder, C. Kotsokalis, E. Yaqub, and A. I. Jehangiri. QoS-Aware VM Placement in Multi-domain Service Level Agreements Scenarios. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 661–668, June 2013.

[2] Stefan Frey, Christoph Reich, and Claudia Lüthje. Key Performance Indicators for Cloud Computing SLAs. 2013.

[3] Peter M. Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, Gaithersburg, MD, United States, 2011.

[4] Amazon, Amazon Elastic Compute Cloud (Amazon EC2). https://aws.amazon.com/ec2/. [Online, accessed April 2019].

[5] Google, Google App Engine. https://cloud.google.com/. [Online, accessed April 2019].

[6] Microsoft, Microsoft Azure. https://azure.microsoft.com/. [Online, accessed April 2019].

[7] Wikipedia: Java Virtual Machine. https://en.wikipedia.org/wiki/Java_virtual_machine. [Online, accessed April 2019].

[8] QEMU Wiki. https://wiki.qemu.org/Main_Page. [Online, accessed April 2019].

[9] Wikipedia: Tiny Code Generator(TCG). https://en.wikipedia.org/wiki/QEMU#Tiny_Code_Generator. [Online, accessed April 2019].

[10] QEMU Machine Protocol(QMP) Wiki. https://wiki.qemu.org/Documentation/QMP. [Online, accessed April 2019].

[11] QEMU Monitor. https://en.wikibooks.org/wiki/QEMU/Monitor. [Online, accessed April 2019].

[12] libvirt: The virtualization API. https://libvirt.org/. [Online, accessed April 2019].

[13] Avi Kivity Qumranet, Yaniv Kamay Qumranet, Dor Laor Qumranet, Uri Lublin Qumranet, and Anthony Liguori. KVM: The Linux virtual machine monitor. *Proceedings Linux Symposium*, 15, 01 2007.

[14] KVM. https://www.linux-kvm.org/page/Main_Page. [Online, accessed April 2019].

[15] VirtIO. https://www.linux-kvm.org/page/Virtio. [Online, accessed April 2019].

[16] XEN Wiki. https://wiki.xen.org/wiki/Xen_Project_Software_Overview. [Online, accessed April 2019].

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. XEN and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[18] Wikipedia: VMware ESXi. https://en.wikipedia.org/wiki/VMware_ESXi. [Online, accessed April 2019].

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, page 273–286. USENIX Association, 2005.

[20] Petter Svärd, Benoit Hudzia, Steve Walsh, Johan Tordsson, and Erik Elmroth. Principles and Performance Characteristics of Algorithms for Live VM Migration. *SIGOPS Oper. Syst. Rev.*, 49(1):142–155, January 2015.

[21] QEMU. https://www.qemu.org/. [Online, accessed April 2019].

[22] XEN Project. https://www.xenproject.org/. [Online, accessed April 2019].

[23] VMware. https://www.vmware.com/. [Online, accessed April 2019].

[24] Jeffrey M. Galloway, Gabriel Loewen, and Susan V. Vrbsky. Performance Metrics of Virtual Machine Live Migration. *2015 IEEE 8th International Conference on Cloud Computing*, pages 637–644, 2015.

[25] Red Hat: KVM Live Migration. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/virtualization/chap-virtualization-kvm_live_migration. [Online, accessed April 2019].

[26] QEMU Wiki: Migration with shared storage. https://wiki.qemu.org/Documentation/Migration_with_shared_storage. [Online, accessed April 2019].

[27] Wikipedia: Network File System(NFS). https://en.wikipedia.org/wiki/Network_File_System. [Online, accessed April 2019].

[28] Wikipedia: Network Time Protocol(NTP). https://en.wikipedia.org/wiki/Network_Time_Protocol. [Online, accessed April 2019].

[29] Red Hat Blog: Live Migrating QEMU-KVM Virtual Machines. https://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines/. [Online, accessed April 2019].

[30] QEMU 2.10.1 Migration Documentation. https://git.qemu.org/?p=qemu.git;a=blob;f=docs/devel/migration.txt;h=1b940a829b14504f28de043426eff50843dff35f;hb=7851197b812b383ae1208c5d86391c5179c8209d. [Online, accessed April 2019].

[31] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. volume 46, pages 111–120, 07 2011.

[32] Pooja and Asmita Pandey. Impact of memory intensive applications on performance of cloud virtual machine. pages 1–6, 03 2014.

[33] Xinkui Zhao, Jianwei Yin, Zuoning Chen, and Sheng He. Workload Classification Model for Specializing Virtual Machine Operating System. pages 343–350, 06 2013.

[34] Lars Cromley. Introduction to High Performance Computing (HPC). https://www.2ndwatch.com/blog/introduction-to-high-performance-computing-hpc/, July 2017. [Online, accessed April 2019].

[35] L. Sliwko and Vladimir Getov. Transfer Cost of Virtual Machine Live Migration in Cloud Systems. Technical report, 15 New Cavendish St., London W1W 6UW, U.K., 2017.

[36] Osama Alrajeh, Matt Forshaw, and Nigel Thomas. Performance of Virtual Machine Live Migration with Various Workloads. September 2016.

[37] Jianxin Li, Jieyu Zhao, Yi Li, Lei Cui, Bo Li, Lu Liu, and John Panneerselvam. iMIG: Toward an Adaptive Live Migration Method for KVM Virtual Machines. *Comput. J.*, 58:1227–1242, 2015.

[38] Daniel Berrange. Analysis of techniques for ensuring migration completion with KVM. https://www.berrange.com/posts/2016/05/12/analysis-of-techniques-for-ensuring-migration-completion-with-kvm/, May 2016. [Online, accessed April 2019].

[39] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. Optimized pre-copy live migration for memory intensive applications. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2011.

[40] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[41] SPECjbb2005. https://www.spec.org/jbb2005/index.html. [Online, accessed April 2019].

[42] Perf Wiki. https://perf.wiki.kernel.org/index.php. [Online, accessed April 2019].

[43] Saurabh Badhwar. Performance profiling with perf. https://fedoramagazine.org/performance-profiling-perf/, August 2016. [Online, accessed April 2019].

[44] Paul Drongowski. PERF. https://sandsoftwaresound.net/perf/, May 2015. [Online, accessed April 2019].

[45] iostat(1) Linux man page. https://linux.die.net/man/1/iostat. [Online, accessed April 2019].

[46] Stefan Hajnoczi. QEMU Internals: How guest physical RAM works. http://blog.vmsplice.net/2016/01/qemu-internals-how-guest-physical-ram.html, January 2016. [Online, accessed April 2019].

[47] time(1) Linux man page. https://linux.die.net/man/1/time. [Online, accessed April 2019].

[48] Travis Oliphant. NumPy: A guide to NumPy. http://www.numpy.org/. [Online, accessed April 2019].

[49] scikit-learn - Machine Learning in Python. https://scikit-learn.org/stable/. [Online, accessed April 2019].

[50] scikit-learn - Support Vector Regression(SVR. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html. [Online, accessed April 2019].

[51] scikit-learn - GridSearchCV. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. [Online, accessed April 2019].

[52] scikit-learn - $R\hat{2}$ (coefficient of determination) regression score function. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html. [Online, accessed April 2019].

[53] scikit-learn - Mean absolute error regression loss. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html. [Online, accessed April 2019].

[54] API Reference - scikit-learn : sklearn.metrics: Metrics. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics. [Online, accessed April 2019].

[55] Fei Zhang, Guang Ming Liu, Xiaoming Fu, and Ramin Yahyapour. A Survey on Virtual Machine Migration: Challenges, Techniques, and Open Issues. *IEEE Communications Surveys & Tutorials*, 20:1206–1243, 2018.

[56] Jinshi Zhang, Eddie Dong, Jian Li, and Haibing Guan. MigVisor Accurate Prediction of VM Live Migration Behavior Using a Working-Set Pattern Model. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 30–43, New York, NY, USA, 2017. ACM.

[57] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and Energy Modeling for Live Migration of Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 171–182, New York, NY, USA, 2011. ACM.

[58] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper. Predicting the Performance of Virtual Machine Migration. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 37–46, Aug 2010.

[59] Minal Patel, Sanjay Chaudhary, and Sanjay Garg. Machine Learning Based Statistical Prediction Model for Improving Performance of Live Virtual Machine Migration. *Journal of Engineering*, 2016:Article ID 3061674, 9 pages, 04 2016.

[60] Senthil Nathan, Umesh Bellur, and Purushottam Kulkarni. Towards a Comprehensive Performance Model of Virtual Machine Live Migration. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 288–301, New York, NY, USA, 2015. ACM.

[61] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang. Live Migration of Multiple Virtual Machines with Resource Reservation in Cloud Computing Environments. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 267–274, July 2011.

[62] Changyeon Jo, Youngsu Cho, and Bernhard Egger. A Machine Learning Approach to Live Migration Modeling. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 351–364, New York, NY, USA, 2017. ACM.